HANDBOOK OF ALGORITHMS FOR PHYSICAL DESIGN AUTOMATION

EDITED BY
CHARLES J. ALPERT
DINESH P. MEHTA
SACHIN S. SAPATNEKAR

CRC Press
Taylor & Francis Group
AN AUERBACH BOOK

HANDBOOK OF ALGORITHMS FOR PHYSICAL DESIGN AUTOMATION

HANDBOOK OF ALGORITHMS FOR PHYSICAL DESIGN AUTOMATION

EDITED BY
CHARLES J. ALPERT
DINESH P. MEHTA
SACHIN S. SAPATNEKAR

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
AN AUERBACH BOOK

Auerbach Publications
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Auerbach is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works
Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number-13: 978-0-8493-7242-1 (Hardcover)

Library of Congress Cataloging-in-Publication Data

Handbook of algorithms for physical design automation / edited by Charles J. Alpert, Dinesh P. Mehta, Sachin S. Sapatnekar.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-8493-7242-1
ISBN-10: 0-8493-7242-9
1. Integrated circuit layout--Mathematics--Handbooks, manuals, etc. 2. Integrated circuit layout--Data processing--Handbooks, manuals, etc. 3. Integrated circuits--Very large scale integration--Design and construction--Data processing--Handbooks, manuals, etc. 4. Algorithms. I. Alpert, Charles J. II. Mehta, Dinesh P. III. Sapatnekar, Sachin S., 1967- IV. Title.

TK7874.55.H36 2009
621.3815--dc22 2008014182

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the Auerbach Web site at
http://www.auerbach-publications.com

Dedications

To the wonderful girls in my life:
Cheryl, Candice, Ciara, and Charlie

Charles J. Alpert

To the memory of my grandparents:
Nalinee and Gajanan Kamat, Radha and Shreenath Mehta

Dinesh P. Mehta

To Ofelia and Arunito

Sachin S. Sapatnekar

Contents

Editors

Charles J. Alpert (Chuck) was born in Bethesda, Maryland, in 1969. He received two undergraduate degrees from Stanford University in 1991 and his doctorate from the University of California, Los Angeles, California in 1996, in computer science. Upon graduation, Chuck joined IBM's Austin Research Laboratory where he currently manages the Design Productivity Group, whose mission is to develop design automation tools and methodologies to improve designer productivity and reduce design cost. Chuck has over 100 conference and journal publications and has thrice received the best paper award from the ACM/IEEE Design Automation Conference. He has been active in the academic community, serving as chair for the Tau Workshop on Timing Issues and the International Symposium on Physical Design. He also serves as an associate editor of *IEEE Transactions on Computer-Aided Design*. He received the Mahboob Khan Mentor Award in 2001 and 2007 for his work in mentoring. He was also named the IEEE fellow in 2005.

Dinesh P. Mehta received his BTech in computer science and engineering from the Indian Institute of Technology, Bombay, India, in 1987; his MS in computer science from the University of Minnesota, Minneapolis, Minnesota, in 1990; and his PhD in computer science from the University of Florida, Gainesville, Florida, in 1992. He was on the faculty at the University of Tennessee Space Institute, Tullahoma, Tennessee from 1992 to 2000, where he received the Vice President's Award for Teaching Excellence in 1997. He was a visiting professor at Intel's Strategic CAD Labs in 1996 and 1997. He has been on the faculty in the mathematical and computer science departments at the Colorado School of Mines, Golden, Colorado since 2000, where he is a professor and currently also serves as department head. He is a coauthor of *Fundamentals of Data Structures in C++* and a coeditor of *Handbook of Data Structures and Applications*. His publications and research interests are in VLSI design automation, and applied algorithms and data structures. He is a former associate editor of the *IEEE Transactions on Circuits and Systems-I*.

Sachin S. Sapatnekar received his BTech from the Indian Institute of Technology, Bombay, India in 1987; his MS from Syracuse University, New York, in 1989; and his PhD from the University of Illinois at Urbana–Champaign, Urbana, Illinois, in 1992. From 1992 to 1997, he was an assistant professor in the Department of Electrical and Computer Engineering at Iowa State University, Ames, Iowa. Since then, he has been on the faculty of the Department of Electrical and Computer Engineering at the University of Minnesota, Minneapolis, Minnesota, where he is currently the Robert and Marjorie Henle Professor. He has published widely in the area of computer-aided design of VLSI circuits, particularly in the areas of timing, layout, and power. He has held positions on the editorial board of the *IEEE Transactions on CAD* (he is currently the deputy editor-in-chief), the *IEEE Transactions on VLSI Systems*, and the *IEEE Transactions on Circuits and Systems II*. He has served on the technical program committee for various conferences, as a technical program co-chair for Design Automation Conference (DAC), and as a technical program and general chair for both the IEEE/ACM Tau Workshop and the ACM International Symposium on Physical Design. He is a recipient of the NSF Career Award, three best paper awards at DAC, and one at International Conference on Computer Design (ICCD), and the Semiconductor Research Corporation Technical Excellence award. He is a fellow of the IEEE.

Contributors

Saurabh N. Adya
Synopsys, Inc.
Sunnyvale, California

Ameya R. Agnihotri
Magma Design Automation
San Jose, California

Christoph Albrecht
Cadence Research Laboratories
Berkeley, California

Charles J. Alpert
IBM Corporation
Austin, Texas

Kia Bazargan
Department of Electrical and Computer Engineering
University of Minnesota
Minneapolis, Minnesota

Murat Becer
CLK Design Automation
Littleton, Massachusetts

Ulrich Brenner
Research Institute for Discrete Mathematics
University of Bonn
Bonn, Germany

Yao-Wen Chang
Department of Electrical Engineering and Graduate Institute of Electronics Engineering
National Taiwan University
Taipei, Taiwan

Charlie Chung-Ping Chen
Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan

Tung-Chieh Chen
Graduate Institute of Electronics Engineering
National Taiwan University
Taipei, Taiwan

Scott Y.L. Chin
Electrical and Computer Engineering
University of British Columbia
Vancouver, British Columbia, Canada

Minsik Cho
Electrical and Computer Engineering Department
University of Texas
Austin, Texas

Nathalie Chan King Choy
Electrical and Computer Engineering
University of British Columbia
Vancouver, British Columbia, Canada

Chris Chu
Department of Electrical and Computer Engineering
Iowa State University
Ames, Iowa

Jason Cong
Computer Science Department
University of California
Los Angeles, California

Yan Feng
Cadence Design Systems
San Jose, California

Jon Frankle
Cadence Design Systems
San Jose, California

Aki Fujimura
Direct 2 Silicon
San Jose, California

Joseph Ganley
Synopsys, Inc.
Vienna, Virginia

Puneet Gupta
Department of Electrical Engineering
University of California
Los Angeles, California

Bill Halpin
Synopsys, Inc.
Sunnyvale, California

Asmus Hetzel
Magma Design Automation, Inc.
San Jose, California

Nathaniel Hieter
IBM Corporation
East Fishkill, New York

Miloš Hrkić
Magma Design Automation
Austin, Texas

Jiang Hu
Department of Electrical and Computer Engineering
Texas A & M University
College Station, Texas

Shiyan Hu
Department of Electrical and Computer Engineering
Michigan Technology University
Houghton, Michigan

Yehea I. Ismail
Electrical Engineering and Computer Science Department
Northwestern University
Evanston, Illinois

Andrew B. Kahng
Electrical and Computer Engineering and Computer Science and Engineering
University of California
San Diego, California

Andrew Kennings
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

Vishal Khandelwal
Synopsys, Inc.
Hillsboro, Oregon

Cheng-Kok Koh
School of Electrical and Computer Engineering
Purdue University
West Lafayette, Indiana

Dorothy Kucar
IBM Corporation
Yorktown Heights, New York

Zhuo Li
IBM Corporation
Austin, Texas

John Lillis
Department of Computer Science
University of Illinois
Chicago, Illinois

Frank Liu
IBM Corporation
Austin, Texas

Zhi-Quan Luo
Department of Electrical and Computer Engineering
University of Minnesota
Minneapolis, Minnesota

Patrick H. Madden
Computer Science Department
Binghamton University
Binghamton, New York

Igor L. Markov
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, Michigan

Dinesh P. Mehta
Department of Mathematical and Computer Sciences
Colorado School of Mines
Golden, Colorado

Arjen Mets
IBM Corporation
East Fishkill, New York

Joydeep Mitra
Electrical and Computer Engineering Department
University of Texas
Austin, Texas

Gi-Joon Nam
IBM Corporation
Austin, Texas

Sani Nassif
IBM Corporation
Austin, Texas

Ralph H.J.M. Otten
Eindhoven University of Technology
Eindhoven, the Netherlands

Muhammet Mustafa Ozdal
Intel Corporation
Hillsboro, Oregon

David Z. Pan
Electrical and Computer Engineering Department
University of Texas
Austin, Texas

Min Pan
Cadence Design Systems, Inc.
San Jose, California

Rajendran Panda
Freescale Semiconductor, Inc.
Austin, Texas

Evanthia Papadopoulou
IBM Corporation
Yorktown Heights, New York

Kara K.W. Poon
Electrical and Computer Engineering
University of British Columbia
Vancouver, British Columbia, Canada

Ruchir Puri
IBM Corporation
Yorktown Heights, New York

Lakshmi Reddy
IBM Corporation
East Fishkill, New York

Haoxing Ren
IBM Corporation
Austin, Texas

Gabriel Robins
Department of Computer Science
University of Virginia
Charlottesville, Virginia

Jarrod A. Roy
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, Michigan

Sanghamitra Roy
Department of Electrical and Computer Engineering
University of Wisconsin–Madison
Madison, Wisconsin

Jeffrey S. Salowe
Cadence Design Systems
San Jose, California

Kambiz Samadi
Department of Electrical and Computer Engineering
University of California
San Diego, California

Sachin S. Sapatnekar
Electrical and Computer Engineering Department
University of Minnesota
Minneapolis, Minnesota

Prashant Saxena
Synopsys, Inc.
Hillsboro, Oregon

Louis K. Scheffer
Cadence Design Systems
San Jose, California

Franklin M. Schellenberg
Mentor Graphics Corporation
San Jose, California

Rupesh S. Shelar
Intel Corporation
Hillsboro, Oregon

Joseph R. Shinnerl
Tabula, Inc.
Santa Clara, California

Ankur Srivastava
Department of Electrical and Computer Engineering
University of Maryland
College Park, Maryland

Haihua Su
Magma Design Automation, Inc.
Austin, Texas

Susmita Sur-Kolay
Advanced Computing and Microelectronics Unit
Indian Statistical Institute
Kolkata, India

William Swartz
InternetCAD.com
Dallas, Texas

Cliff C. N. Sze
IBM Corporation
Austin, Texas

Steve Teig
Tabula, Inc.
Santa Clara, California

Louise Trevillyan
IBM Corporation
Yorktown Heights, New York

Paul G. Villarrubia
IBM Corporation
Austin, Texas

Kristofer Vorwerk
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

Jens Vygen
Research Institute for Discrete Mathematics
University of Bonn
Bonn, Germany

Ting-Chi Wang
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan

Steven J.E. Wilton
Electrical and Computer Engineering
University of British Columbia
Vancouver, British Columbia, Canada

Martin D.F. Wong
Department of Electrical and Computer Engineering
University of Illinois at Urbana–Champaign
Urbana, Illinois

Xiaojian Yang
Synopsys, Inc.
Sunnyvale, California

Evangeline F.Y. Young
Department of Computer Science and Engineering
Chinese University of Hong Kong Shatin
Hong Kong, China

Alexander Zelikovsky
Department of Computer Science
Georgia State University
Atlanta, Georgia

Hai Zhou
Department of Electrical Engineering and Computer Science
Northwestern University
Evanston, Illinois

Vladimir Zolotov
IBM Corporation
Yorktown Heights, New York

Part I

Introduction

1 Introduction to Physical Design

Charles J. Alpert, Dinesh P. Mehta, and Sachin S. Sapatnekar

CONTENTS

1.1 INTRODUCTION

The purpose of VLSI physical design is to embed an abstract circuit description, such as a netlist, into silicon, creating a detailed geometric layout on a die. In the early years of semiconductor technology, the task of laying out gates and interconnect wires was carried out manually (i.e., by hand on graph paper, or later through the use of layout editors). However, as semiconductor fabrication processes improved, making it possible to incorporate large numbers of transistors onto a single chip (a trend that is well captured by Moore's law), it became imperative for the design community to turn to the use of automation to address the resulting problem of scale. Automation was facilitated by the improvement in the speed of computers that would be used to create the next generation of computer chips resulting in their own replacement! The importance of automation was reflected in the scientific community by the formation of the Design Automation Conference in 1963 and both the International Conference on Computer-Aided Design and the *IEEE Transactions on Computer-Aided Design* in 1983; today, there are several other conferences and journals on design automation.

While the problems of scale have been one motivator for automation, other factors have also come into play. Most notably, improvements in technology have resulted in the invalidation of some critical assumptions made during physical design: one of these is related to the relative delay between gates and the interconnect wires used to connect gates to each other. Initially, gate delays dominated interconnect delays to such an extent that interconnect delay could essentially be ignored when computing the delay of a circuit. With technology scaling causing feature sizes to shrink by a factor of 0.7 every 18 months or so, gates became faster from one generation to the next, while wires became more resistive and slower. Early metrics that modeled interconnect delay as proportional to the length of the wire first became invalid (as wire delays scale quadratically with their lengths) and then valid again (as optimally buffered interconnects show such a trend). New signal integrity effects began to manifest themselves as power grid noise or in the form of increased crosstalk as wire cross-sections became "taller and thinner" from one technology generation to the next. Other problems came into play: for instance, the number of buffers required on a chip began to show trends that increased at alarming rates; the delays of long interconnects increased to the range of several clock cycles; and new technologies emerged such as 3D stacked structures with multiple layers of

active devices, opening up, literally and figuratively, a new dimension in physical design. All of these have changed, and are continuing to change, the fundamental nature of classical physical design.

A major consequence of interconnect dominance is that the role of physical design moved upstream to other stages of the design cycle. Synthesis was among the first to feel the impact: traditional 1980s-style logic synthesis (which lasted well into the 1990s) used simplified wire-load models for each gate, but the corresponding synthesis decisions were later unable to meet timing specifications, because they operated under gross and incorrect timing estimates. This realization led to the advent of physical synthesis techniques, where synthesis and physical design work hand in hand. More recently, multicyle interconnects have been seen to impact architectural decisions, and there has been much research on physically driven microarchitectural design.

These are not the only issues facing the designer. In sub-90 nm technologies, manufacturability issues have come to the forefront, and many of them are seen to impact physical design. Traditionally, design and manufacturing inhabited different worlds, with minimal handoffs between the two, but in light of * issues related to subwavelength lithography and planarization, a new area of physical design has opened up, where manufacturability has entered the equation. The explosion in mask costs associated with these issues has resulted in the emergence of special niches for field programmable gate arrays (FPGAs) for lower performance designs and for fast prototyping; physical design problems for FPGAs have their own flavors and peculiarities.

Although there were some early texts on physical design automation in the 1980s (such as the ones by Preas/Lorenzetti and Lengauer), university-level courses in VLSI physical design did not become commonplace until the 1990s when more recent texts became available. The field continues to change rapidly with new problems coming up in successive technology generations. The developments in this area have motivated the formation of the International Symposium on Physical Design (ISPD), a conference that is devoted solely to the discipline of VLSI physical design; this and other conferences became the major forum for the learning and dissemination of new knowledge. However, existing textbooks have failed to keep pace with these changes. One of the goals of this handbook is to provide a detailed survey of the field of VLSI physical design automation with a particular emphasis on state-of-the-art techniques, trends, and improvements that have emerged as a result of the dramatic changes seen in the field in the last decade.

1.2 OVERVIEW OF THE PHYSICAL DESIGN PROCESS

Back when the world was young and life was simple, when Madonna and Springsteen ruled the pop charts, interconnect delays were insignificant and physical design was a fairly simple process. Starting with a synthesized netlist, the designer used floorplanning to figure out where big blocks (such as arrays) were placed, and then placement handled the rest of the logic. If the design met its timing constraints before placement, then it would typically meet its timing constraints after placement as well. One could perform clock tree synthesis followed by routing and iterate over these process in a local manner.

Of course, designs of today are much larger and more complex, which requires a more complex physical design flow. Floorplanning is harder than ever, and despite all the algorithms and innovations described here, it is still a very manual process. During floorplanning, the designers plan their I/Os and global interconnect, and restrict the location of logic to certain areas, and of course, the blocks (of which there are more than ever). They often must do this in the face of incomplete timing data. Designers iterate on their floorplans by performing fast physical synthesis and routing congestion estimation to identify key problem areas.

Once the main blocks are fixed in location and other logic is restricted, global placement is used to place the rest of the cells, followed by detailed placement to make local improvements. The placing of cells introduces long wires that increase delays in unexpected places. These delays are then reduced

* Pun unintended.

by wire synthesis techniques of buffering and wire sizing. Iteration between incremental placement and incremental synthesis to satisfy timing constraints today takes place in a single process called physical synthesis. Physical synthesis embodies just about all traditional physical design processes: floorplanning, placement, clock tree construction, and routing while sprinkling in the ability to adapt to the timing of the design. Of course, with a poor floorplan, physical synthesis will fail, so the designer must use this process to identify poor block and logic placement and plan global interconnects in an iterative process.

The successful exit of physical synthesis still requires post-timing-closure fix-up to address noise, variability, and manufacturability issues. Unfortunately, repairing these can sometimes force the designer back to earlier stages in the flow.

Of course, this explanation is an oversimplification. The physical design flow depends on the size of the design, the technology, the number of designers, the clock frequency, and the time to complete the design. As technology advances and design styles change, physical design flows are constantly reinvented as traditional phases are removed or combined by advances in algorithms (e.g., physical synthesis) while new ones are added to accommodate changes in technology.

1.3 OVERVIEW OF THE HANDBOOK

This handbook consists of the following ten parts:

1. Introduction: In addition to this chapter, this part includes a personal perspective from Ralph Otten, looking back on the major technical milestones in the history of physical design automation. A discussion of physical design objective functions that drive the techniques discussed in subsequent parts is also included in this part.
2. Foundations: This part includes reviews of the underlying data structures and basic algorithmic and optimization techniques that form the basis of the more sophisticated techniques used in physical design automation. This part also includes a chapter on partitioning and clustering. Many texts on physical design have traditionally included partitioning as an integral step of physical design. Our view is that partitioning is an important step in several stages of the design automation process, and not just in physical design; therefore, we decided to include a chapter on it here rather than devote a full handbook part.
3. Floorplanning: This identifies relative locations for the major components of a chip and may be used as early as the architecture stage. This part includes a chapter on early methods for floorplanning that mostly viewed floorplanning as a two-step process (topology generation and sizing) and reviews techniques such as rectangular dualization, analytic floorplanning, and hierarchical floorplanning. The next chapter exclusively discusses the slicing floorplan representation, which was first used in the early 1970s and is still used in a lot of the recent literature. The succeeding two chapters describe floorplan representations that are more general: an active area of research during the last decade. The first of these focuses on mosaic floorplan representations (these consider the floorplan to be a dissection of the chip rectangle into rooms that will be populated by modules, one to each room) and the second on packing representations (these view the floorplan as directly consisting of modules that need to be packed together). The penultimate chapter describes recent variations of the floorplanning problem. It explores formulations that more accurately account for interconnect and formulations for specialized architectures such as analog designs, FPGAs, and three-dimensional ICs. The final chapter in this part describes the role of floorplanning and prototyping in industrial design methodologies.
4. Placement: This is a classic physical design problem for which design automation solutions date back to the 1970s. Placement has evolved from a pure wirelength-driven formulation to one that better understands the needs of design closure: routability, white space distribution, big block placement, and timing. The first chapter in this part overviews how the placement

problem has changed with technology scaling and explains the new types of constraints and objectives that this problem must now address.

There has been a renaissance in placement algorithms over the last few years, and this can be gleaned from the chapters on cut-based, force-directed, multilevel, and analytic methods. This part also explores specific aspects of placement in the context of design closure: detailed placement, timing, congestion, noise, and power.

5. Net Layout and Optimization: During the design closure process, one needs to frequently estimate the layout of a particular net to understand its expected capacitance and impact on timing and routability. Traditionally, maze routing and Steiner tree algorithms have been used for laying out a given net's topology, and this is still the case today. The first two chapters of this part overview these fundamental physical design techniques.

   Technology scaling for transistors has occurred much faster than for wires, which means that interconnect delays dominate much more than for previous generations. The delays due to interconnect are much more significant, thus more care needs to be taken when laying out a net's topology. The third chapter in this part overviews timing-driven interconnect structures, and the next three chapters show how buffering interconnect has become an absolutely essential step in timing closure. The buffers in effect create shorter wires, which mitigate the effect of technology scaling. Buffering is not a simple problem, because one has to not only create a solution for a given net but also needs to be cognizant of the routing and placement resources available for the rest of the design. The final chapter explores another dimension of reducing interconnect delay, wire sizing.
6. Routing Multiple Signal Nets: The previous part focused on optimization techniques for a single net. These approaches need conflict resolution techniques when there are scarce routing resources. The first chapter explores fast techniques for predicting routing congestion so that other optimizations have a chance to mitigate routing congestion without having to actually perform global routing. The next two chapters focus on techniques for global routing: the former on the classic rip-up and reroute approach and the latter on alternative techniques like network flows. The next chapter discusses planning of interconnect, especially in the context of global buffer insertion. The final chapter addresses a very important effect from technology scaling: the impact of noise on coupled interconnect lines. Noise issues must be modeled and mitigated earlier in the design closure flows, as they have become so pervasive.
7. Manufacturability and Detailed Routing: The requirements imposed by manufacturability and yield considerations place new requirements on the physical design process. This part discusses various aspects of manufacturability, including the use of metal fills, and resolution-enhancement techniques and subresolution assist features. These techniques have had a major impact on design rules, so that classical techniques for detailed routing cannot be used directly, and we will proceed to discuss the impact of manufacturability considerations on detailed routing.
8. Physical Synthesis: Owing to the effects that have become apparent in deep submicron technologies, wires play an increasingly dominant role in determining the circuit performance. Therefore, traditional approaches to synthesis that ignored physical design have been supplanted by a new generation of physical synthesis methods that integrate logic synthesis with physical design. This part overviews the most prominent approaches in this domain.
9. Designing Large Global Nets: In addition to signal nets, global nets for supply and clock signals consume a substantial fraction of on-chip routing resources, and play a vital role in the functional correctness of the chip. This part presents an overview of design techniques that are used to route and optimize these nets.
10. Physical Design for Specialized Technologies: Although most of the book deals with mainstream microprocessor or ASIC style designs, the ideas described in this book are largely

applicable to other paradigms such as FPGAs and to emerging technologies such as 3D integration. These problems require unique solution techniques that can satisfy these requirements. The last part overviews constraints in these specialized domains, and the physical design solutions that address the related problems.

1.4 INTENDED AUDIENCE

The material in this book is suitable for researchers and students in physical design automation and for practitioners in industry who wish to be familiar with the latest developments. Most importantly, it is a valuable complete reference for anyone in the field and potentially for designers who use design automation software.

Although the book does lay the basic groundwork in Part I, this is intended to serve as a quick review. It is assumed that the reader has some background in the algorithmic techniques used and in physical design automation. We expect that the book could also serve as a text for a graduate-level class on physical design automation.

NOTE ABOUT REFERENCES

The following abbreviations may have been used to refer to conferences and journals in which physical design automation papers are published.

ASPDAC	Asian South Pacific Design Automation Conference
DAC	Design Automation Conference
EDAC	European Design Automation Conference
GLSVLSI	Great Lakes Symposium on VLSI
ICCAD	International Conference on Computer-Aided Design
ICCD	International Conference on Computer Design
ISCAS	International Symposium on Circuits and Systems
ISPD	International Symposium on Physical Design
IEEE TCAD	*IEEE Transactions on the Computer-Aided Design of Integrated Circuits*
IEEE TCAS	*IEEE Transactions on Circuits and Systems*
ACM TODAES	*ACM Transactions on the Design Automation of Electronic Systems*
IEEE TVLSI	*IEEE Transactions on VLSI Systems*

2 Layout Synthesis: A Retrospective

Ralph H.J.M. Otten

CONTENTS

2.1 THE FIRST ALGORITHMS (UP TO 1970)

Design automation has a history of over half a century if we look at its algorithms. The first algorithms were not motivated by design of electronic circuits. Willard Van Orman Quine's work on simplifying truth functions emanated from the philosopher's research and teaching on mathematical logic. It produced a procedure for simplifying two-level logic that remained at the core of logic synthesis for decades (and still is in most of its textbooks). Closely involved in its development were the first pioneers in layout synthesis: Sheldon B. Akers and Chester Y. Lee. Their work on switching networks, both combinational and sequential, and their representation as binary decision programs came from the same laboratory as the above simplification procedure, and preceded the landmark 1961 paper on routing.

2.1.1 Lee's Router

What Lee [1] described is now called a grid expansion algorithm or maze runner, to set it apart from earlier independent research on the similar abstract problem: the early paper of Edsger W. Dijkstra on shortest path and labyrinth problems [2] and Edward F. Moore's paper on shortest paths through a maze [3] were already written in 1959. But in Lee's paper the problem of connecting two points on a grid with its application to printed circuit boards was developed through a systematization of the intuitive procedure: identify all grid cells that can be reached in an increasing number of steps until the target is among them, or no unlabeled, nonblocked cells are left. In the latter case, no such path exists. In the former case, retracing provides a shortest path between the source and the target (Figure 2.1).

The input consists of a grid with blocked and nonblocked cells. The algorithm then goes through three phases after the source and target have been chosen, and the source has been labeled with 0:

1. Wave propagation in which all unlabeled, nonblocked neighbors of labeled cells are labeled one higher than in the preceding wave.
2. Retracing starts when the target has received a label and consists of repeatedly finding a neighboring cell with a lower label, thus marking a shortest path between the source and the target.
3. Label clearance prepares the grid for another search by adding the cells of the path just found to the set of blocked cells and removing all labels.

The time needed to find a path is $\mathcal{O}(L^2)$ if L is the length of the path. This makes it worst case $\mathcal{O}(N^2)$ on an $N \times N$ grid (and if each cell has to be part of the input, that is any cell can be initially blocked, it is a linear-time algorithm). Its space complexity is also $\mathcal{O}(N^2)$. These complexities were

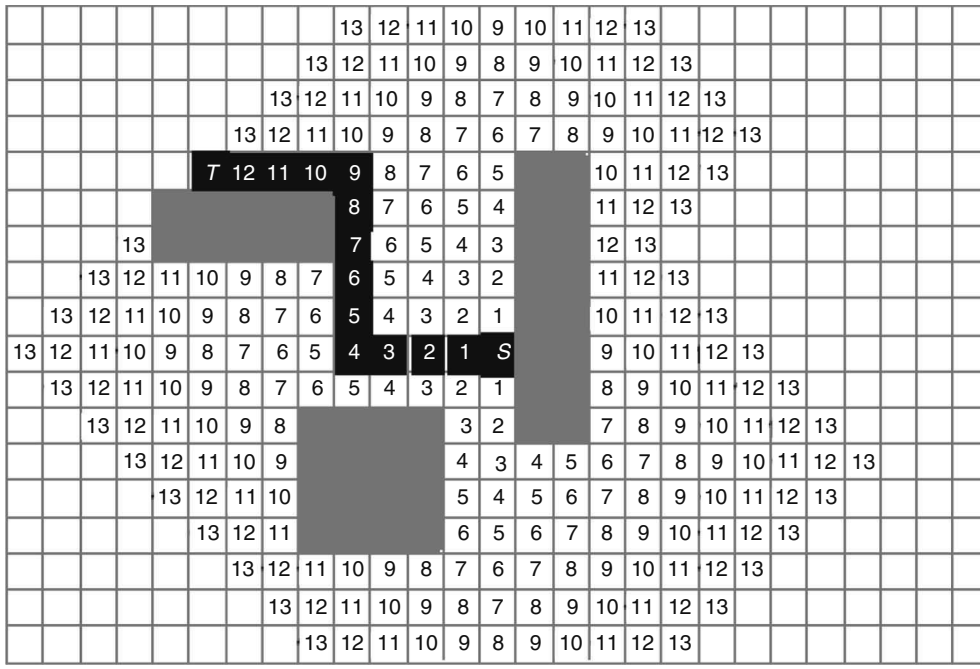


FIGURE 2.1 Wave propagation and retracing. Waves are sets of grid cells with the same label. The source S gets label 0. The target T gets the length of the shortest path as a label (if any). Retracing is not unique in general.

soon seen as serious problems when applied to real-world cases. Some relief in memory use was found in coding the labels: instead of labeling each explored cell with its distance to the source, it suffices to record that number modulo 3, which works for any path search on an unweighted graph. Here, however, the underlying structure is bipartite, and Akers [4] observed that wave fronts with a label sequence in which a certain label bit is twice on and then twice off (i.e., 1, 1, 0, 0, 1, 1, 0, …) suffice. Trivial speedup techniques were soon standard in maze running, mostly aimed at reducing the wave size. Examples are designating the most off-center terminal as the source, starting waves from both terminals, and limiting the search to a box slightly larger than the minimum containing the terminals.

More significant techniques to reduce complexity were discovered in the second part of the decade. There are two techniques that deserve a mention in retrospective. The first technique, line probing, was discovered by David W. Hightower [5] and independently by Koichi Mikami and Kinya Tabuchi [6]. It addressed both the memory and time aspects of the router's complexity. The idea is for each so-called base point to investigate the perpendicular line segments that contain the base point and extend those segments to the first obstacles on their way. The first base points are the terminals and their lines are called trial lines of level 0. Mikami and Tabuchi choose next as base points all grid points on the lines thus generated. The trial lines of the next level are the line segments perpendicular to the trial line containing their base point. The process is stopped when lines originating from different terminals intersect. The algorithm guarantees a path if one exists and it will have the lowest possible number of bends. This guarantee soon becomes very expensive, because all possible trial lines of the deepest possible level have to be examined. Hightower therefore traded it for more efficiency in the early stages by limiting the base points to the so-called escape points, that is, only the closest grid point that allows extension beyond the obstacle that blocked the trial line of the previous level. Line expansion, a combination of maze running and line probing, came some ten years later [7], with the salient feature of producing a path whenever one existed, though not necessarily with the minimum number of bends.

The essence of line probing is in working with line segments for representing the routing space and paths. Intuitively, it saves memory and time, especially when the search space is not congested. The complexity very much depends on the data structures maintained by the algorithm. The original papers were vague about this, and it was not until the 1980s that specialists in computational geometry could come up with a rigorous analysis [8]. In practice, line probers were used for the first nets with distant terminals. Once the routing space gets congested, more like a labyrinth where trial lines are bound to be very short, a maze runner takes over.

The second technique worth mentioning is based on the observation that from a graph theoretical point of view, Lee's router is just a breadth-first search that may take advantage of special features like regularity and bipartiteness. But significant speed advantage can be achieved by including a sense of direction in the wave propagation phase, preferring cells closer to the target. Frank Rubin [9] implements such an idea by sorting the cells in the wavefront with a key representing the grid distance to the target. It shifts the character of the algorithm from breadth-first to depth-first search.

This came close to what was developed simultaneously, but in the field of artificial intelligence: the A* algorithm [10]. Here the search is ordered by an optimistic estimate of the source–target pathlength through the cell. The sum of the number of steps to reach that cell (exactly as in the original paper of Lee) plus the grid distance to the target (as introduced by Rubin) is a satisfactory estimate, because the result can never be more than that estimate. This means that it will find the shortest route, while exploring the least number of grid cells. See Chapter 23 for a more detailed description of maze routing.

Lee's concept combined with A* is still the basis of modern industrial routers. But many more issues than just the shortest two-pin net have to be considered. An extension to multiterminal nets is easy (e.g., after connecting two pins, take the cells on that route as the initial wavefront and find the shortest path to another terminal, etc.), but it will not in general produce the shortest connecting tree (for this the Steiner problem on a grid has to be solved, a well-known NP-hard problem, which is

discussed in Chapter 24). Routing in many wiring layers can also straightforwardly be incorporated by adopting a three-dimensional grid. Even bipartiteness is preserved, but looses its significance because of preferences in layers and usually built-in resistance against creating vias. The latter and some other desirable features can be taken care of by using other cost functions than just distance and tuning these costs for satisfactory results. Also a net ordering strategy has to be determined, mostly to achieve close to full wire list completion. And taking into account sufficient effects of modern technology (e.g., cross talk, antenna phenomena, metal fill, lithography demands) makes router design a formidable task, today even more than in the past. This will be the subject of Chapters 34 through 36 and 38.

2.1.2 Assignment and Placement

Placement is initially seen as an assignment problem where n modules have to be assigned to at least n slots. The easiest formulation associated a cost with every module assignment to each slot, independent of other assignments. The Hungarian method (also known as Munkres' algorithm [11]) was already known and solved the problem in polynomial time. This was however an unsatisfactory problem formulation, and the cost function was soon replaced by

$$\sum_i a_{i,p(i)} + \sum_{i,j} c_{i,j} d_{p(i),p(j)}$$

where

$d_{p(i),p(j)}$ is the distance between the slots assigned to modules i and j
$a_{i,p(i)}$ is a cost associated with assigning module i to slot $p(i)$
$c_{i,j}$ is a weight factor (e.g., the number of wires between module i and j) penalizing the distance between the modules i and j

With all $c_{i,j}$ equal to zero, it reduces to the assignment problem above and with all a equal to zero, it is called the quadratic assignment problem that is now known to be NP hard (the traveling salesperson problem is but a special case).

Paul C. Gilmore [12] soon provided (in 1962) a branch-and-bound solution to the quadratic assignment problem, even before that approach had got this name. In spite of its bounding techniques, it was already impractical for some 15 modules, and was therefore unable to replace an earlier heuristic of Leon Steinberg [13]. He used the fact that the problem can be easily solved when all $c_{i,j} = 0$, in an iterative technique to find an acceptable solution for the general problem. His algorithm generated some independent sets (originally all maximal independent sets, but the algorithm generated independent sets in increasing size and one can stop any time). For each such set, the wiring cost for all its members for all positions occupied by that set (and the empty positions) was calculated. These numbers are of course independent of the positions of the other members of that set. By applying the Hungarian method, these modules were placed with minimum cost. Cycling through these independent sets continues until no improvement is achieved during one complete cycle. Steinberg's method was repeatedly improved and generalized in 1960s.*

Among the other iterative methods to improve such assignments proposed in these early years were force-directed relaxation [14] and pairwise interchange [15]. In the former method, two modules in a placement are assumed to attract each other with a force proportional to their distance. The proportionality constant is something like the weight factor $c_{i,j}$ above. As a result, a module is subjected to a resultant force that is the vector sum of all attracting forces between pairs it is involved in. If modules could move freely, they would move to the lowest energy state of the system. This

* Steinberg's 34-module/36-slot example, the first benchmark in layout synthesis, is only recently optimally solved for Euclidean norm, almost 40 years after its publication in 1961. The wirelength was 4119.74. The best result of the 1960s was by Frederick S. Hiller (4475.28).

is mostly not a desirable assignment because many modules may opt for the same slot. Algorithms therefore are moved one module at a time to a position close to the zero-tension point

$$\left(\frac{\sum_i c_{Mi}x_i}{\sum_i c_{Mi}}, \frac{\sum_i c_{Mi}y_i}{\sum_i c_{Mi}}\right)$$

Of course, if there is a free slot there, it can be assigned to it. If not, the module occupying it can be moved in the same way if it is not already at its zero-tension point. Numerous heuristics to start and restart a sequence of such moves are imaginable, and kept the idea alive for the decennia to come, only to mature around the year 2000 as can be seen in Chapter 18.

A simple method to avoid occupied slots is pairwise interchange. Two modules are selected and if interchanging their slot positions improves the assignment, the interchange takes place. Of course only the cost contribution of the signal nets involved has to be updated. However, the pair selection is not obvious. Random selection is an option, ordering modules by connectedness was already tried before 1960, and using the forces above in various ways quickly followed after the idea got in publication. But a really satisfactory pair selection was not shown to exist.

The constructive methods in the remainder of that decade had the same problem. They were ad-hoc heuristics based on a selection rule (the next module to be placed had to have the strongest bond with the ones already placed) followed by a positioning rule (such as pair linking and cluster development). They were used in industrial tools of 1970s, but were readily replaced by simulated annealing when that became available. But one development was overlooked, probably because it was published in a journal not at all read by the community involved in layout synthesis. It was the first analytic placer [16], minimizing in one dimension

$$\sum_{i,j=1}^{n} c_{ij}\left[p(i) - p(j)\right]^2$$

with the constraints $\mathbf{p^T p} = 1$ and $\sum_i p(i) = 0$, to avoid the trivial solution where all components of $\mathbf{p}$ are the same. That is, an objective that is the weighted sum of all squared distances. Simply rewriting that objective in matrix notation yields

$$2\mathbf{p^T A p}$$

where $\mathbf{A} = \mathbf{D} - \mathbf{C}, \mathbf{D}$ being the diagonal matrix of row sums of $\mathbf{C}$. All eigenvalues of such a matrix are nonnegative. If the wiring structure is connected, there will be exactly one eigenvalue of $\mathbf{A}$ equal to 0 (corresponding to that trivial solution), and the eigenvector associated with the next smallest eigenvalue will minimize the objective under the given constraints. The minimization problem is the same for the other dimension, but to avoid a solution where all modules would be placed on one line we add the constraint that the two vectors must be orthogonal. The solution of the two-dimensional problem is the one where the coordinates correspond with the components of the eigenvectors associated with second and third smallest eigenvalues.

The placement method is called Hall placement to give credit to the inventor Kenneth M. Hall. When applied to the placement of components on chip or board, it corresponds to the quadratic placement problem. Whether this is the right way to formulate the wire-length objective will be extensively discussed in Chapters 17 and 18, but it predates the first analytic placer in layout synthesis by more than a decade!

2.1.3 Single-Layer Wiring

Most of the above industrial developments were meant for printed circuit boards (in which integrated circuits with at most a few tens of transistors are interconnected in two or more layers) and backplanes

(in which boards are combined and connected). Integrated circuits were not yet subject to automation. Research, both in industry and academia, started to get interesting toward the end of the decade. With only one metal layer available, the link with graph planarity was quickly discovered. Lots of effort went into designing planarity tests, a problem soon to be solved with linear-time algorithms. What was needed, of course, was planarization: using technological possibilities (sharing collector islands, small diffusion resistors, multiple substrate contacts, etc.) to implement a circuit using a planarized model. Embedding the planar result onto the plane while accounting for the formation of isolated islands, and connecting the component pins were the remaining steps [17].

Today the constraints of those early chips are obsolete. Extensions are still of some validity in analogue applications, but are swamped by a multitude of more severe demands. Planarization resurfaced when rectangular duals got attention in floorplan design. Planar mapping as used in these early design flows started a whole new area in graph theory, the so-called visibility graphs, but without further applications in layout synthesis.*

The geometry of the islands provided the first models for rectangular dissections and their optimization, and for the compaction algorithms based on longest path search in constraint graphs. These graphs, originally called polar graphs and illustrated in Figure 2.3, were borrowed† from early works in combinatorics (how to dissect rectangles into squares?) [20]. They enabled systematic generations of all dissection topologies, and for each such topology a set of linear equations as part of the optimization tableau for obtaining the smallest rectangle under (often linearized) constraints. The generation could not be done in polynomial time of course, but linear optimization was later proven to be efficient.

A straightforward application of Lee's router for single-layer wiring was not adequate, because planarity had to be preserved. Its ideas however were used in what was a first form of contour routing. Contour routing turned out to be useful in the more practical channel routers of the 1980s.

2.2 EMERGING HIERARCHIES (1970–1980)

Ten years of design automation for layout synthesis produced a small research community with a firm basis in graph theory and a growing awareness of computational complexity. Stephen Cook's famous theorem was not yet published and complexity issues were tackled by bounding techniques, smart speedups, and of course heuristics. Ultimately, and in fact quite soon, they proved to be insufficient. Divide-and-conquer strategies were the obvious next approaches, leading to hierarchies, both uniform requiring few well-defined subproblems and pluriform leaving many questions unanswered.

2.2.1 Decomposing the Routing Space

A very effective and elegant way of decomposing a problem was achieved by dividing the routing space into channels, and solving each channel by using a channel router. It found immediate application in two design styles: standard cell or polycell where the channels were height adjustable and channel routing tried to use as few tracks as possible (Figure 2.2 for terminology), and gate arrays where the channels had a fixed height, which meant that channel router had to find a solution within a given number of tracks. If efficient minimization were possible, the same algorithm would suffice, of course. The decision problems, however, were shown to be NP complete.

The classical channel-routing problem allows two layers of wires: one containing the pins at grid positions and all latitudinal parts (branches), exactly one per pin, and one containing all longitudinal parts (trunks), exactly one for each net. This generates two kinds of constraints: nets with overlapping intervals need different tracks (these are called horizontal constraints), and wires that have pins at the same longitudinal height must change layer before they overlap (the so-called vertical constraints).

* In this context, they were called horvert representations [18].

† The introduction of polar graphs in layout synthesis [19] was one on the many contributions that Tatsuo Ohtsuki gave to the community.

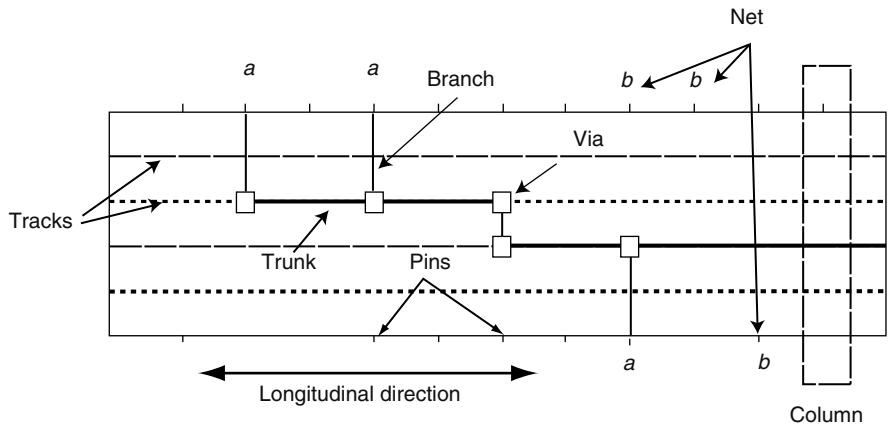


FIGURE 2.2 Terminology in channel routing.

The problem does not always have a solution. If the vertical constraints form cycles, then the routing cannot be completed in the classical model. Otherwise a routing does exist, but finding the minimum number of tracks is NP hard [21].

In the absence of vertical constraints, the problem can be solved optimally in almost linear time by a pretty simple algorithm [22], originally owing to Akihiro Hashimoto and James Stevens, that is known as the left-edge algorithm.* Actually there are two simple greedy implementations both delivering a solution with the minimum number of tracks. One is filling the tracks one by one from left to right each time trying the unplaced intervals in sequence of their left edges. The other places the intervals in that sequence in the first available track that can take it. In practice, the left-edge algorithm gets quite far in routing channels, in spite of possible vertical constraints. Many heuristics therefore started with left-edge solutions.

To obtain a properly wired channel in two layers, the requirements that latitudinal parts are one-to-one with the pins and that each net can have only one longitudinal part are mostly dropped by introducing doglegs.† Allowing doglegs enables in practice always a two-layer routing with latitudinal and longitudinal parts never in the same layer, although in theory problems exist that cannot be solved. It has been shown that the presence of a single column without pins guarantees the existence of a solution [23]. Finding the solution with the least number of tracks remains NP hard [24].

Numerous channel routers have been published, mainly because it was a problem that could be easily isolated. The most effective implementation, without the more or less artificial constraints of the classical problem and its derivations, is the contour router of Patrick R. Groeneveld [25]. It solves all problems although in practice not many really difficult channels were encountered. In modern technologies, with a number of layers approaching ten, channel routing has lost its significance.

2.2.2 Netlist Partitioning

Layout synthesis starts with a netlist, that is, an incidence structure or hypergraph with modules as nodes and nets as hyperedges. The incidences are the pins. These nets quickly became very large,

* It is often referred to as an algorithm for coloring an interval graph. This is not correct, because an interval representation is assumed to be available. It is, however, possible to color an interval graph in polynomial time. One year after the publication of the left-edge algorithm, Yanakakis Gavril gave such an algorithm for chordal graphs of which interval graphs are but a special case.

† Originally, doglegs were only allowed at pin positions. The longitudinal parts might be broken up in several longitudinal segments. The dogleg router of that paper was probably never implemented and the presented result was edited. The paper became nevertheless the most referenced paper in the field because it presented the benchmark known as the Deutsch difficult example. Every channel router in the next 20 years had to show its performance when solving that example.

in essence following Moore's law of exponential complexity growth. Partitioning was seen as the way to manage complex design. Familiarity with partitioning was already present, because the first pioneers were involved in or close to teams that had to make sure that subsystems of a logic design could be built in cabinets of convenient size. These subsystems were divided over cards, and these cards might contain replaceable standard units. One of these pioneers, Uno R. Kodres, who had already provided in 1959 an algorithm for the geometrical positioning of circuit elements [26] in a computer, possibly the first placement algorithm in the field, gave an excellent overview of these early partitioners [27]. They started with one or more seed modules for each block in the partitioning. Then, based once more on a selection rule, blocks are extended by assigning one module at a time to one block. Many variations are possible and were tried, but all these early attempts were soon wiped out by module migration methods, and first by the one of Brian W. Kernighan and Shen Lin [28]. They started from a balanced two-partition of the netlist, that is, division of all modules into two nonoverlapping blocks of approximately equal size. The quality of that two-partition was measured in the number of nets connecting modules in both blocks, the so-called cutsize. This number was to be made as low as possible. This was tried in a number of iterations. For each iteration, the gain of swapping two modules, one from each block, was calculated, that is, the reduction in cutsize as a consequence of that swap. Gains can be positive, zero, or negative. The pairs are unlocked and ordered from largest to smallest gain. In that order each unlocked pair is swapped, locked to prevent it from moving back, and its consequence (new blocks and updated gains) is recorded. When all modules (except possibly one) are locked the best cutsize encountered is accepted. A new iteration can take place if there is a positive gain left.

Famous as it is, the Kernighan–Lin procedure left plenty of room for improvement. Halfway in the decade, it was proven that the decision problem of graph partition was NP complete, so the fact that it mostly only produced a local optimum was unavoidable, but the limitations to balanced partitions and only two-pin nets had to be removed. Besides a time-complexity of $O(n^3)$ for an n-module problem was soon unacceptable. The repair of these shortcomings appeared in a 1982 paper by Charles M. Fiduccia and Robert M. Mattheyses [29]. It handled hyperedges (and therefore multipin nets), and instead of pair swapping it used module moves while keeping bounds on balance deviations, possibly with weighted modules. More importantly, it introduced a bucket data structure that enabled a linear-time updating scheme. Details can be found in Chapter 7.

At the same time, one was not unaware of the relation between partitioning and eigenvalues. This relation, not unlike the theory behind Hall's placement [16], was extensively researched by William E. Donath and Alan J. Hoffman [30]. Apart from experiments with simulated annealing (not very adequate for the partitioning problem in spite of the very early analogon with spin glasses) and using migration methods for multiway partitioning, it would be well into the 1990s before partitioning was carefully scrutinized again.

2.2.3 Mincut Placement

Applying partitioning in a recursive fashion while at the same time slicing the rectangular silicon estate in two subrectangles according to the area demand of each block is called mincut placement. The process continues until blocks with known layouts or suitable for dedicated algorithms are obtained. The slicing cuts can alternate between horizontal and vertical cuts, or have the direction depend on the shape of the subrectangle or the area demand. Later, also procedures performing four-way partition (quadrisection) along with dividing in four subrectangles were developed. A strict alternation scheme is not necessary and many more sophisticated cut-line sequences have been developed. Melvin A. Breuer's paper [31] on mincut placement did not envision deep partitioning, but large geometrically fixed blocks had to be arranged in a nonoverlapping configuration by positioning and orienting. Ulrich Lauther [32] connected the process with the polar graph illustrated in Figure 2.3. The mincut process by itself builds a series-parallel polar graph, but Lauther also defined three local operations, to wit mirroring, rotating, and squeezing, that more or less preserved the relative positions.

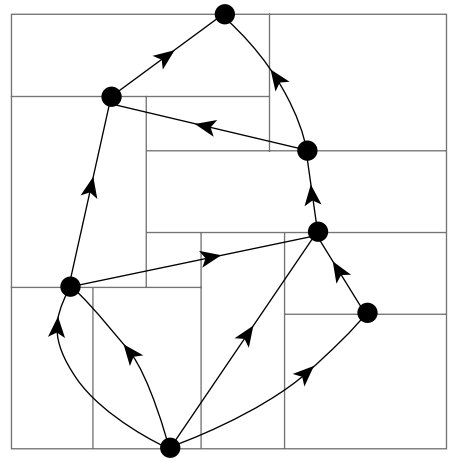


FIGURE 2.3 Polar graph of a rectangle dissection.

The first two are pretty obvious and do not change the topology of the polar graph. The last one, squeezing, does change the graph and might result in a polar graph that is not series parallel.

The intuition behind mincut placement is that if fewer wires cross the first cut lines, there will be fewer long connections in the final layout. An important drawback of the early mincut placers, however, is that they treat lower levels of partitioning independent from the blocks created earlier, that is, without any awareness of the subrectangles to which connected modules were assigned. Modules in those external blocks may be connected to modules in the block to be partitioned, and be forced unnecessarily far from those modules. Al Dunlop and Kernighan [33] therefore tried to capture such connectivities by propagating modules external to the block to be partitioned as fixed terminals to the periphery of that block. This way their connections to the inner modules are taken into account when calculating cutsizes. Of course, now the order in which blocks are treated has an impact on the final result.

2.2.4 Chip Fabrication and Layout Styles

Layout synthesis provides masks for chip fabrication, or more precisely, it provides data structures from which masks are derived. Hundreds of masks may be needed in a modern process, and with today's feature sizes, optical correction is needed in addition to numerous constraints on the configurations. Still, layout synthesis is only concerned with a few partitions of the Euclidean plane to specify these masks.

When all masks are specific to producing a particular chip, we speak of full-custom design. It is the most expensive setup and usually needs high volume to be cost effective. Generic memory always was in that category, but certain application specific designs also qualified. Even in the early 1970s, the major computer seller of the day saw the advantage of sharing masks over as many as possible different products. They called it the master image, but it became known ten years later as the gate-array style in the literature. Customization in these styles was limited to the connection layers, that is, the layers in which fixed rows of components were provided with their interconnect. Because many masks were never changed in a generation of gate-array designs, these were known as semi-custom designs. Wiring was kept in channels of fixed width in early gate arrays.

Another master-image style was developed in the 1990s that differed from gate arrays by not leaving space for wires between the components. It was called sea-of-gates, because the unwired chip was mostly nothing else than alternating rows of *p*-type and *n*-type metal oxide semiconductor (MOS)-transistors. Contacts with the gates were made on either side of the row, although channel

contacts were made between the gates. A combination of routers was used to achieve this over-the-cell routing. The routers were mostly based on channel routers developed for full-custom chips.

Early field programmable gate arrays predated (and survived) the sea-of-gates approach, which never became more than niche in the cost-profit landscape of the chip market. It allows individualization away from the chip production plant by establishing or removing small pieces of interconnect.

Academia believed in full-custom, probably biased by its initial focus on chips for analogue applications. Much of their early adventures in complete chip design for digital applications grew out of the experience described in Section 2.1.3 and were encouraged by publications from researchers in industry such as Satoshi Goto [34], and Bryan T. Preas and Charles W. Gwyn [35]. Rather than a methodology, suggested by the award-winning paper in 1978, it established a terminology. Macrocell layout and general-cell assemblies in particular remained for several years names for styles without much of a method behind it.

Standard-cell (or polycell) layout was a full-custom style that lent itself to automation. Cells with uniform height and aligned supply and clock lines were called from a library to form rows in accordance with a placement result. Channel routing was used to determine the geometry of the wires in between the rows. The main difference with gate-array channels was that the width was to be determined by the algorithm. Whereas in gate-array styles, the routers had to fit all interconnect in channels of fixed width, the problem in standard-cell layouts was to minimize the number of tracks, and whatever the result, reserve enough space on the chip to accommodate them.

2.3 ITERATION-FREE DESIGN

By 1980, industrial tools had developed in what was called spaghetti code, depending on a few people with inside knowledge of how it had developed from the initial straightforward idea sufficient for the simple examples of the early 1970s, into a sequence of patches with multiple escapes from where it could end up in almost any part of the code. In the meantime, academia were dreaming of compiling chips. Carver A. Mead and Lynn (or Robert) Conway wrote the seminal textbook [36] on very large scale integration between 1977 and 1979, and, although not spelled out, the idea of (automatically) deriving masks from a functional specification was born shortly after the publication in 1980. A year later, David L. Johannsen defended his thesis on silicon compilation.

2.3.1 Floorplan Design

From the various independent algorithms for special problems grew the layout synthesis as constrained optimization: wirelength and area minimization under technology design rules. The target was functionality with acceptable yield. Speed was not yet an issue. Optimum performance was achieved with multichip designs, and it would take another ten years before single-chip microprocessors would come into their ball park.

The real challenge in those days was the phase problem between placement and routing. Obviously, placement has a great impact on what is achievable with routing, and can even render unroutable configurations. Yet, it was difficult to think about routing without coordinates, geometrical positions of modules with pins to be connected. The dream of silicon compilation and designs scalable over many generations of technology was in 1980 not more than a firm belief in hierarchical approaches with little to go by apart from severe restrictions in routing architecture.* A breakthrough came with the introduction of the concept of floorplans in the design trajectory of chips by Ralph H.J.M. Otten [37]. A floorplan was a data structure capturing relative positions rather than fixed

* There was an exception: when in 1970 Akers teamed up with James M. Geyer and Donald L. Roberts [38] and tried grid expansion to make designs routable. It consisted of finding cuts of horizontal and vertical segments of only conductor areas in one direction and conductor free lines in the other. Furthermore, the cutting segment in the conductor area should be perpendicular to all wires cut. The problems that it created were an early inspiration for slicing.

coordinates. In a sense, floorplan design is a generalization of placement. Instead of manipulating fixed geometrical objects in a nonoverlapping arrangement in the plane, floorplan design treats modules as objects with varying degrees of flexibility and tries to decide on their position relative to the position of others.

In the original paper, the relative positions were captured by a point configuration in the plane. By a clever transformation of the netlist into the so-called dutch metric, an optimal embedding of these points could be obtained. The points became the centers of rectangular modules with an appropriate size that led to a set of overlapping rectangles when the point configuration was more or less fit in the assessed chip footprint. The removal of overlap was done by formulating the problem as a mathematical program.

Other data structures than Cartesian coordinates were proposed. A significant related data structure was the sequence pair of Hiroshi Murata, Kunihiro Fujiyoshi, Shigetoshi Nakatake, and Yoji Kajitani in 1997 [39]. Before that, a number of graphs, including the good old-polar graphs from combinatorial theory, were used and especially around the year 2000 many other proposals were published. Chapters 9 through 11 will describe several floorplan data structures.

The term floorplan design came from house architecture. Already in 1960s, James Grason [40] tried to convert preferred neighbor relationships into rectangles realizing these relations. The question came down to whether a given graph of such relations had a rectangular dual. He characterized such graphs in a forbidden-graph theorem. The algorithms he proposed were hopelessly complex, but the ideas found new following in the mid-1980s. Soon simple, necessary, and sufficient conditions were formulated, and Jayaram Bhasker and Sartaj Sahni produced in 1986 a linear-time algorithm for testing the existence of a rectangular dual and, in case of the affirmative, constructing a corresponding dissection [41].

The success of floorplanning was partially due to giving answers that seemed to fit the questions of the day like a glove: it lent itself naturally to hierarchical approaches* and enabled global wiring as a preparation for detailed routing that took place after the geometrical optimization of the floorplan. It was also helped by the fact that the original method could reconstruct good solutions from abstracted data in extremely short computation times even for thousands of modules. The latter was also a weakness because basically it was the projection of a multidimensional Euclidean space with the exact Dutch distances onto the plane of its main axes. Significant distances perpendicular to that plane were annihilated.

2.3.2 Cell Compilation

Hierarchical application of floorplanning ultimately leads to modules that are not further dissected. They are to be filled with a library cell, or by a special algorithm determining the layout of that cell depending on specification and assessed environment. The former has a shape constraint with fixed dimensions (sometimes rotatable). The latter is often macrocells with a standard-cell layout style. They lead to staircase functions as shape constraints where a step corresponds to a choice of the number of rows.

In the years of research toward silicon compilers, circuit families tended to grow. The elementary static complementary metal oxide semiconductor (CMOS)-gate has limitations, specifically in the number of transistors in series. This limits the number of distinct gates severely. The new circuit techniques allowed larger families. Domino logic, for example, having only a pull-down network determining its function, allows much more variety. Single gates with up to 60 transistors have been used in designs of the 1980s. This could only be supported if cells could be compiled from their functional specification.

The core of the problem was finding a linear-transistor array, where only transistors sharing contact areas could be neighbors. This implied that the charge or discharge network needed a topology of an Euler graph. In static cmos, both networks had to be Eulerian, preferably with the same sequence

* Many even identified floorplanning with hierarchical layout design, clearly an undervaluation of the concept.

of input signals controlling the gate. The problem even attracted a later fields medallist in the person of Curtis T. McMullen [42], but the final word came from the thesis of Robert L. Maziasz [43], a student of John P. Hayes. Once the sequence was established, the left-edge algorithm could complete the network, if the number of tracks would fit on the array, which was a mild constraint in practice; but an interesting open question for research is to find an Euler path leading to a number of tracks under a given maximum.

2.3.3 Layout Compaction

Area minimization was considered to be the most important objective in layout synthesis before 1990. It was believed that other objectives such as minimum signal delay and yield would benefit from it. A direct relation between yield and active area was not difficult to derive and with gate delay dominating the overall speed performance, chips usually came out faster than expected. The placement tools of the day had the reputation of using more chip area than needed, a belief that was based mainly on the fact that manual design often outperformed automatic generation of cell layouts. This was considered infeasible for emerging chip complexities, and it was felt that a final compaction step could only improve the result. Systematic ways of taking a complete layout of a chip and producing a smaller design-rule correct chip, while preserving the topology, therefore became of much interest.

Compaction is difficult (one may see it as the translation of topologies in the graph domain to mask geometries that have to satisfy the design rules of the target technology). Several concepts were proposed to provide a handle on the problem: symbolic layout systems, layout languages, virtual grids, etc. At the bottom, there is the combinatorial problem of minimizing the size of a complicated arrangement of many objects in several related and aligned planes. Even for simple abstractions the two-dimensional problem is complex (most of them are NP hard). An acceptable solution was often found in a sequence of one-dimensional compactions, combined with heuristics to handle the interaction between the two dimensions (sometimes called $1\frac{1}{2}$-compaction). Many one-dimensional compaction routines are efficiently solvable, often in linear time. The basis is found in longest-path problem, already popular in this context during 1970s. Compaction is discussed in several texts on VLSI physical design such as those authored by Majid Sarrafzadeh and Chak-Kuen Wong [44], Sadiq M. Sait and Habib Youssef [45], and Naveed Sherwani [46], but above all in the book of Thomas Lengauer [47].

2.3.4 Floorplan Optimization

Floorplan optimization is the derivation of a compatible (i.e., relative positions of the floorplan are respected) rectangle dissection, optimal under a given contour score e.g., area and perimeter that are possibly constrained, in which each undissected rectangle satisfies its shape constraint. A shape constraint can be a size requirement with or without minima imposed on the lengths of its sides, but in general any constraint where the length of one side is monotonically nonincreasing with respect to the length of the other side.

The common method well into the 1980s was to capture the relative positions as Kirchhoff equations of the polar graph. This yields a set of linear equalities. For piecewise linear shape constraints that are convex, a number of linear inequalities can be added. The perimeter can then be optimized in polynomial time. For nonconvex shape constraints or nonlinear objectives, one had to resort to branch-and-bound or cutting-plane methods: for general rectangle dissections with nonconvex shape constraints the problem is NP hard. Larry Stockmeyer [48] proved that even a pseudo-polynomial algorithm does not exist when $P \neq NP$.

The initial success of floorplan design was, beside the facts mentioned in Section 2.3.1, also due to a restraint that was introduced already in the original paper. It was called slicing because the geometry of compatible rectangle dissection was recognizable by cutting lines recursively slicing completely through the rectangle. That is rectangles resulting from slicing the parent rectangle could

either be sliced as well or were not further dissected. This induces a tree, the slicing tree, which in an hierarchical approach that started with a functional hierarchy produced a refinement: functional submodules remained descendants of their supermodule.

More importantly, many optimization problems were tractable for slicing structures, among which was floorplan optimization. A rectangle dissection has the slicing property iff its polar graph is series parallel. It is straightforward to derive the slicing tree from that graph. Dynamic programming can then produce a compatible rectangle dissection, optimal under any quasi-concave contour score, and satisfying all shape constraints [49]. Also labeling a partition tree with slicing directions can be done optimally in polynomial time if the tree is more or less balanced and the shape constraints are staircase functions as Lengauer [50] showed. Together with Lukas P.P.P. van Ginneken, Otten then showed that floorplans given as point configurations could be converted to such optimal rectangle dissections, compatible in the sense that slices in the same slice respect the relative point positions [51]. The complexity of that optimization for N rectangles was however $\mathcal{O}(N^6)$, unacceptable for hundreds of modules. The procedure was therefore not used for more than 30 modules, and was reduced to $\mathcal{O}(N^3)$ by simple but reasonable tricks. Modules with more than 30 modules were treated as flexible rectangles with limitations on their aspect ratio.

2.3.5 Beyond Layout Synthesis

It cannot be denied that research in layout synthesis had an impact on optimization in other contexts and optimization in general. The left-edge algorithm may be rather simple and restricted (it needs an interval representation), simulated annealing is of all approaches the most generic. A patent request was submitted in 1981 by C. Daniel Gelatt and E. Scott Kirkpatrick, but by then its implementation (MCPlace) was already compared (by having Donald W. Jepsen watching the process at a screen and resetting temperature if it seemed stuck in local minimum) against IBM's warhorse in placement (APlace) and soon replaced it [52]. Independent research by Vladimir Cerny [53] was conducted around the same time. Both used the metropolis loop from 1953 [54] that analyzed energy content of a system of particles at a given temperature, and used an analogy from metallurgy were large crystals with few defects were obtained by annealing, that is, controlled slow cooling.

The invention was called simulated annealing but could not be called an optimization algorithm because of many uncertainties about the schedule (begin temperature, decrements, stopping criterion, loop length, etc.) and the manual intervention. The annealing algorithm was therefore developed from the idea to optimize the performance within a given amount of elapsed CPU time to be used [55]. Given this one parameter, the algorithm resolved the uncertainties by creating a Markov chain that enhanced the probability of a low final score.

The generic nature of the method led to many applications. Further research, notably by Sara A. Solla, Gregory B. Sorkin, and Steve R. White, showed that, in spite of some statements about its asymptotic behavior, annealing was not the method of choice in many cases [56]. Even the application described in the original paper of 1983, graph partitioning, did not allow the construction of a state space suitable for efficient search in that way. It was also shown however that placement with wirelength minimization as objective lent itself quite well, in the sense that even simple pairwise interchange produced a space with the properties shown to be desirable by the above researchers. Carl Sechen exploited that fact and with coworkers he created a sequence of releases of the widely used timberwolf program [57], a tool based on annealing for placement. It is described in detail in Chapter 16.

It is not at all clear that simulated annealing performs well for floorplan design where sizes of objects differ in orders of magnitude. Yet, almost invariably, it is the method of choice. There was of course the success of Martin D.F. Wong and Chung Laung (Dave) Liu [58] who represented the slicing tree in polish notation and defined a move set on it (that move set by the way is not unbiased, violating a requirement underlying many statements about annealing). Since then the community has been flooded with innovative representations of floorplans, slicing and nonslicing, each time

resorting to annealing as the underlying design engine, without attention for configuring the state space. Nevertheless, it is the structure of the local minima,* determined by the move set that is crucial for a reliable application of annealing.

2.4 CLOSURE PROBLEMS

The introduction of the fruits of design automation of the 1980s in industry generated mostly distrust and disbelief among designers. No longer was it simply computer-aided design limited to liberating them from routine, but tedious tasks that were reliably performed for them with predictable results. Modules were absorbed, duplicated, split, and spread, and signal nets had disappeared. The whole structure of a design might have changed beyond recognition after a single run of retiming. In many places, designers felt insecure and the introduction of new tools hampered production rather than boosting it.

Layout synthesis got its share of this skepticism. One of its pioneers phrased it as layout is on its way out. Yet, there was a solid background in algorithms and heuristics, and a better understanding of the problem and its context. Many of the original approaches were revisited, improved and, above all, compared with others on the basis of a common meaningful set of benchmarks. No longer was it acceptable to publish yet another heuristic for a well-known subtask of the layout synthesis problem with some self-selected examples to suggest effectiveness and efficiency. This book provides ample evidence that tool making for layout synthesis matured after 1995. The perfection and adaptation of these tools for the ongoing evolution of silicon technologies is the major achievement of the 1990s.

Beside reliable tools supported by rigorous proofs and unbiased comparison, additional shifts were needed. The field developed over three decades from translating intuition into (interactive) procedures, over formulation of well-defined optimization problems, toward integral trajectories without global iteration. This was feasible as long as there was a dominant objective: get it on a chip that is manufacturable. Wire-length minimization served as such an objective. There was some intuition that short wires were good for area, speed, and power, but they were not a target. In the pioneering stages, this was surprisingly successful: most designs were faster than expected and power was not yet a problem.

By 1990, this was no longer enough. Certainly, speed became an important performance characteristic, and it was forseeable that it would not stay the only additional one. It inspired formulations where one characteristic was optimized under constraints for the other characteristics. They were called closure problems: one aspect was to be guaranteed (closed), while others were handled as well as possible.

2.4.1 WIRING CLOSURE

The research aiming at silicon compilation can therefore be viewed as wiring closure. The phase problem of placement and routing where unroutable placements might occur, and, with the increased chip complexity, not easily repaired, was solved by introducing restrained floorplans. Floorplans only captured relative positions, but combined with efficient optimization, they could provide enough information to perform global routing. The technology of that era, which allowed not more than two wiring layers and therefore kept routing separate from the so-called active area, benefited from the decomposition of the wiring space into channels. Global wiring was therefore in essence assigning wires to channels.

When adopting slicing as a restraint, a number of conflicts can be avoided. An important property of slicing structures is that detailed routing can be done with a single algorithm: channel routing.

* Sara A. Solla, Gregory B. Sorkin, and Steve R. White [56] proposed a measure for the ultrametricity of the space of local minima and the barriers between them. A good state space should be close to ultrametric. The proposed measure was the correlation between the heights of the higher two barriers in every triple of minima. Placement of equal-sized objects score close to 1, although partitioning typically ends up with 0.6.

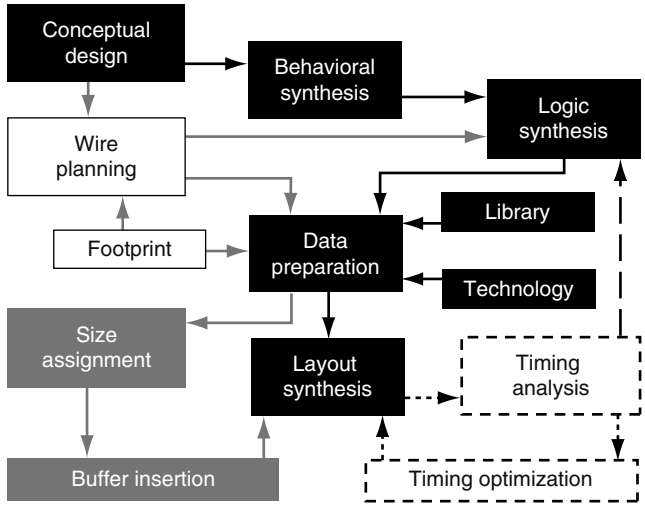


FIGURE 2.4 Modern flows in design automation.

That is, there is a sequence in which channels can be routed with fully specified longitudinal pin positions. Switch boxes are not necessary. Moreover, channels are just rectangles between the slices, with a shape constraint based on the information provided by the global router (almost all channels can be routed close to density, and density can be estimated when there is a good idea of where the pins of the nets are going to be).

Shapes only have to be assessed, and the assessment can be updated whenever more information comes available. Once the cell assemblers (among which channel routers) are called, the shapes become final. Floorplan optimization can then be called to convert the floorplan into a placement with exact coordinates. Slicing floorplans can be optimized quickly. Therefore, there is no harm in calling it whenever convenient. Thus, iteration-free synthesis was enabled. Figure 2.4 shows in the black boxes a generic iteration-free flow. The essence is that each block makes its decisions once and the flow never goes back to it. In the reality of the 1980s, slicing guaranteed wiring when used in a (possibly hierarchical) floorplanning context. With more wiring layers available, the premise of this solution was no longer valid.

2.4.2 Timing Closure

The first reaction to demands concerning speed were to include timing analysis in the tool set. After producing the geometry of the layout, the netlist got extended with parasitics and other network elements to determine its performance. The dynamical tuning of net weights was never a good idea because of convergence problems. Soon wire-load models were developed to obtain more precise estimates and indications where the critical paths were. Transistor sizing, buffer insertion, and fanout tree construction could then improve timing without changing the logic. It did not take long before logic resynthesis entered the scene, using these load models to make another netlist with hopefully better timing properties. In Figure 2.4, this is indicated by the dashed boxes and arrows.

All these measures introduced iteration, the latter even global iteration over almost the complete trajectory. And still it could only produce just another local optimum, not likely to be global. In other words, if timing demands were not met, one was never sure whether that was because the technology of the day could not provide it or the tool set simply did not find it. With Moore's law of unforgiving push behind the chip market, this was not satisfactory.

A paradigm shift was needed and was found by adopting a delay model for gates that had roots in a paper by Ivan E. Sutherland and Robert F. Sproull [59], and was justified in practice by Joel

Grodstein, Eric Lehman, Heather Harkness, William J. Grundmann, and Yoshinori Watanabe [60]. The key observation was that the size of a gate with constant delay varies linearly with the load. Writing this down for a network (not necessarily acyclic) leads to solving a set of linear equations for gate sizes (a so-called Leontieff system), which can be iteratively updated with the sizes to adjust the capacitive loads, and will surely converge if there are no limitations on the gate sizes [61]. Timing could be guaranteed (if feasible) for networks that could be modeled with lumped capacitances, which was true for all networks within the scope of the logic synthesis tools in those days.

With timing no longer the more or less arbitrary outcome of an optimization with area/wire length as its objective, the uncertainty shifted to area. Buffer insertion kept a spot in the flow, but now no longer for improving speed. Buffers can only slow down a path in a network sized by the Leontieff method. Timing can only be kept within the specification when buffers are inserted in noncritical paths with enough slack. That might be beneficial, because it may save area, but it would never make the network faster. In addition, the flow became iteration free again as can be seen in Figure 2.4 with only the black and grey boxes included.

2.4.3 Wire Planning

The complexity of chips in the meantime had developed from a state in which delay was mainly caused by capacitive loads, predominantly gate capacitances, to a situation where most of the global delay was in the wires. Whereas the Sutherland model maintains it salient property as long as resistance between the gate and its load could be neglected, it was of little value when performance critically depends on the distributed resistance and capacitance of wires on today's chips. A delay model published shortly after the Second World War [62], named Elmore delay after the author, was the basis of much of the research on performance of designs in silicon in the second half of the decade of the 1990s. It was pretty accurate for point-to-point connections when it predicted that the delay of long wires depended quadratically on their lengths. It could also be used in combination with the buffer models of Takayasu Sakurai [63] to show that when optimally segmented, the delay became linear in its length (regardless of the size of these buffers). The length of these segments did depend on the layer (or rather on the resistance and capacitance per unit length). An interesting observation however is that the delay of a segment in an optimally segmented and buffered wire (of course also an optimum size can be determined for the buffers) does not depend on the layer: it depends on the properties of the transistors in the buffer. This implies that the delay is known as soon as the process is chosen in which the buffers are going to be made [61].

These theoretical facts open new possibilities for design automation of the backend, and a wealth of opportunities for research. A lot of assumptions are quite idealistic: there is not always place for a buffer at its optimal position, derivations are usually for homogeneous wires, connections are trees in general, etc. But the two observations of delay in length and segment delay independent of layer enable a scenario for wire planning:

1. Assign global wires as connection between modules that synthesis can cope with and therefore so small that buffering does not help in speedup.
2. For given chip performance do time budgeting with convex time-size trade-offs for the modules.
3. Synthesize netlists for the modules with function and delay for all gates.
4. Size the gates for constant delay.

In Figure 2.4, the scenario is depicted (exclude the dashed boxes and arrows) and shows that no global iterations are implied. An initial footprint has to be chosen though, and convex trade-offs (enabling efficient area minimization under timing bounds) have to be available (or extracted). Only after time budgeting is it clear whether the design will fit in the chosen floorplan. Timing closure for large chips is not yet fully solved.

2.5 WHAT DID WE LEARN?

The present goal of design automation has to be design closure, that is, how to specify a function to be implemented on a chip, feed it to an electronic design automation (EDA) tool, and get, without further interaction, a design that meets all requirements concerning functionality, speed, size, power, yield, and other costs. It is the obvious quest of industry and the natural evolution from the sequence of closure problems of the past decennium. Instead of focusing on trade-offs between two or three performance characteristics whenever such a closure problem surfaces such as how to achieve wireability in placement of components or modules on a chip, how to allocate resources to optimize schedules, or how to ensure timing convergence with minimal size, a more general approach should be taken that in principle accounts for all combinations of performance characteristics [64].

No doubt the best algorithms developed in layout synthesis in the last 15 years will be key ingredients and will get due attention in this book. Today's practice of offering rigorous background and thorough evaluation, preferably using well-established benchmarks, will be exemplified.

REFERENCES

1. C.Y. Lee, An algorithm for path connections and its applications, *IRE Transactions on Electronic Computers*, EC-10(3): 346–365, September 1961.
2. E.W. Dijkstra, A note on two problems in connexion with graphs, *Numerische Mathematik*, 1: 269–271, 1959.
3. E.F. Moore, Shortest path through a maze, *Annals of the Computation Laboratory of Harvard University*, 30: 285–292, 1959.
4. S.B. Akers, A modification of Lee's path connection algorithm, *IEEE Transactions on Electronic Computers*, EC-16(1): 97–98, February 1967.
5. D.W. Hightower, A solution to line-routing problems on the continuous plane, in *Proceedings of the 6th Design Automation Workshop*, Las Vegas, NV, pp. 1–24, 1969.
6. K. Mikami and K. Kabuchi, A computer program for optimal routing of printed circuit connectors, *IFIPS Proceedings*, H47: 1475–1478, 1968.
7. W. Heyns, W. Sansen, and H. Beke, A line-expansion algorithm for the general routing problem with a guaranteed solution, in *Proceedings of the 17th Design Automation Conference*, Minneapolis, MN, pp. 243–249, 1980.
8. T. Asano, M. Sato, and T. Ohtsuki, Computational geometry algorithms, Chapter 9, in *Layout Design and Verification*, ed. T. Ohtsuki, North-Holland, Amsterdam, the Netherlands, pp. 295–347, 1986.
9. F. Rubin, The lee path connection algorithm, *IEEE Transactions on Computers*, C-23(9): 907–914, September 1974.
10. P.E. Hart, N.J. Nilsson, and B. Raphael, A formal basis for the heuristic determination of minimum cost paths, *IEEE Transactions on Systems Science and Cybernetics*, SSC4(2): 100–107, 1968.
11. J. Munkres, Algorithms for the assignment and transportation problems, *Journal of the Society of Industrial and Applied Mathematics*, 5(1): 32–38, March 1957.
12. P.C. Gilmore, Optimal and suboptimal algorithms for the quadratic assignment problem, *Journal of the Society of Industrial and Applied Mathematics*, 10(2): 305–313, June 1962.
13. L. Steinberg, The backboard wiring problem: A placement algorithm, *Society of Industrial and Applied Mathematics Reviews*, 3(1): 37–50, January 1961.
14. C.J. Fisk, D.L. Caskey, and L.L. West, ACCEL: Automated circuit card etching layout, *Proceedings of the IEEE*, 55(11): 1971–1982, November 1967.
15. M. Hanan and J.M. Kurtzberg, A review of the placement and quadratic assignment problems, *Society of Industrial and Applied Mathematics Reviews*, 14(2): 324–342, April 1972.
16. K.M. Hall, An r-dimensional quadratic placement algorithm, *Management Science*, 17(3): 219–229, November 1970.
17. M.C. van Lier and R.H.J.M. Otten, Automatic IC layout: The model and technology, *IEEE Transactions on Circuits and Systems*, 22(11): 845–855, November 1975.
18. R.H.J.M. Otten and J.G. van Wijk, Graph representations in interactive layout design, in *Proceedings IEEE International Symposium Circuits and Systems*, New York, NY, pp. 914–918, 1978.

19. T. Ohtsuki, N. Sugiyama, and H. Kawanishi, An optimization technique for integrated circuit layout design, in *Proceedings of the ICCST-Kyoto*, Kyoto, Japan, pp. 67–68, September 1970.
20. R.L. Brooks, C.A.B. Smith, A.H. Stone, and W.T. Tutte, The dissection of rectangles into squares, *Duke Mathematical Journal*, 7: 312–340, 1940.
21. A.S. Lapaugh, Algorithms for integrated circuit layout, PhD thesis, MIT, Boston, MA, November 1980.
22. A. Hashimoto and J. Stevens, Wire routing by optimizing channel assignment within large apertures, in *Proceedings of the 8th Design Automation Workshop*, Atlantic City, NJ, pp. 155–169, 1971.
23. P.R. Groeneveld, Necessary and sufficient conditions for the routability of classical channels, *Integration, the VLSI Journal*, 16: 59–74, 1993.
24. T.G. Szymanski, Dogleg channel routing is NP-complete, *IEEE Transactions on CAD of Integrated Circuits and Systems*, 4(1): 31–41, January 1985.
25. P.R. Groeneveld, H. Cai, and P. Dewilde, A contour-based variable-width gridless channel router, in *Proceedings of the International Conference on Computer-Aided Design*, San José, CA, pp. 374–377, 1987.
26. U.R. Kodres, Geometrical positioning of circuit elements in a computer, in *AIEE Fall General Meeting*, Chicago, ILL, No. 59-1172, October 1959.
27. U.R. Kodres, Partitioning and card selection, Chapter 4, in *Design Automation of Digital Systems*, Vol. 1, ed. M.A. Breuer, Prentice Hall, Englewood Cliffs, NJ, pp. 173–212, 1972.
28. B.W. Kernighan and S. Lin, An efficient heuristic procedure for partitioning graphs, *Bell System Technical Journal*, 49(2): 291–307, February 1970.
29. C.M. Fiduccia and R.M. Mattheyses, A linear time heuristic for improving network partitions, in *Proceedings of the 19th Design Automation Conference*, Las Vegas, NV, pp. 175–181, 1982.
30. W.E. Donath and A.J. Hoffman, Algorithms for partitioning of graphs and computer logic based on eigenvectors of connection matrices, *IBM Technical Disclosure Bulletin*, 15: 938–944, 1972.
31. M.A. Breuer, A class of min-cut placement algorithms, in *Proceedings of the 14th Design Automation Conference*, New Orleans, LA, pp. 284–290, 1977.
32. U. Lauther, A min-cut placement algorithm for general cell assemblies based on a graph representation, in *Proceedings of the 16th Design Automation Conference*, San Diego, CA, pp. 1–10, 1979.
33. A.E. Dunlop and B.W. Kernighan, A procedure for placement of standard-cell VLSI circuit, *IEEE Transactions on CAD of Integrated Circuits and Systems*, CAD-4 (1): 92–98, January 1985.
34. S. Goto, An efficient algorithm for the two-dimensional placement problem in electrical circuit layout, *IEEE Transactions on Circuits and Systems*, CAS-28 (1): 12–18, January 1981.
35. B.T. Preas and C.W. Gwyn, Methods for hierarchical automatic layout of custom LSI circuit masks, in *Proceedings of the 15th Design Automation Conference*, pp. 206–212, 1978.
36. C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA, 1980.
37. R.H.J.M. Otten, Automatic floorplan design, in *Proceedings of the 19th Design Automation Conference*, Las Vegas, NV, pp. 261–267, 1982.
38. S.B. Akers, J.M. Geyer, and D.L. Roberts, IC mask layout with a single conductor layer, in *Proceedings of the 7th Workshop on Design Automation*, San Francisco, CA, pp. 7–16, 1970.
39. H. Murata, F. Fujiyoshi, S. Nakatake, and Y. Kajitani, VLSI module placement based on rectangle packing by the sequence-pair, *IEEE Transactions on Computer-Aided Design*, 15: 1518–1524, December 1996. (ICCAD 1995).
40. J. Grason, A dual linear graph representation for space-filling location problems of the floor plan type, in *Emerging Methods in Environmental Design and Planning*, ed. G.T. Moore, Proceedings of the Design Methods Group, 1st International Conference, Cambridge, MA, pp. 170–178, 1968.
41. J. Bhasker and S. Sahni, A linear algorithm to find a rectangular dual of a planar triangulated graph, *Algorithmica*, 3: 247–278, 1988.
42. C.T. McMullen and R.H.J.M. Otten, Minimum length linear transistor arrays in MOS, in *IEEE International Symposium on Circuits and Systems*, Kyoto, Japan, pp. 1783–1786, 1988.
43. R.L. Maziasz and J.P. Hayes, *Layout Minimization of CMOS Cells*, Kluwer Academic Publishers, Boston, MA; Dordrecht, The Netherlands; London, U.K., 1992.
44. M. Sarrafzadeh and C.-K. Wong, *An Introduction to VLSI Physical Design*, McGraw-Hill Companies Inc., Hightstown, NJ, 1996.
45. S.M. Sait and H. Youssef, *VLSI Physical Design Automation*, McGraw-Hill Companies Inc./IEEE Press, Hightstown, NJ, 1995.

46. N. Sherwani, *Algorithms for VLSI Physical Design Automation*, Kluwer Academic Publishers, Boston, MA; Dordrecht, The Netherlands; London, U.K., 1995.
47. T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, John Wiley & Sons, New York; Berlin, Germany, 1990.
48. L. Stockmeyer, Optimal orientations of cells in slicing floorplan designs, *Information and Control*, 57(2–3): 91–101, May/June 1983.
49. R.H.J.M. Otten, Efficient floor plan optimization, in *Proceedings of International Conference on Computer Design*, Portchester, NY, pp. 499–503, October–November 1983.
50. T. Lengauer and R. Müller, The complexity of floorplanning based on binary circuit partitions, Technical Report 46, Department of Mathematics and Computer Science, University of Paderborn, Paderborn, Germany, 1986.
51. L.P.P.P. van Ginneken and R.H.J.M. Otten, Optimal slicing of plane point placements, *Proceedings of the Conference on European Design Automation*, Glascow, U.K., pp. 322–326, 1990.
52. S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi, Optimization by simulated annealing, *Science*, 220(4598): 671–680, 1983.
53. V. Cerny, A thermodynamical approach to the travelling salesman problem: An efficient simulation algorithm, *Journal of Optimization Theory and Applications*, 45: 41–51, 1985.
54. N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller, Equations of state calculations by fast computing machines, *Journal of Chemical Physics*, 21(6): 1087–1092, 1953.
55. R.H.J.M. Otten and L.P.P.P. van Ginneken, *The Annealing Algorithm*, Kluwer Academic Publishers, Boston, MA; Dordrecht, The Netherlands; London, U.K., 1989.
56. S.A. Solla, G.B. Sorkin, and S.R. White, Configuration space analysis for optimization problems, in *Disordered Systems and Biological Organization*, eds. E. Bienenstock et al., NATO ASI Series, F20, Springer Verlag, Berlin, Germany, pp. 283–293, 1986.
57. C. Sechen and A.L. Sangiovanni-Vincentelli, The timberwolf placement and routing package, *IEEE Journal of Solid-State Circuits*, 20: 510–522, 1985.
58. D.F. Wong and C.L. Liu, A new algorithm for floorplan design, in *Proceedings of the 23rd Design Automation Conference*, Las Vegas, NV, pp. 101–107, 1986.
59. I.E. Sutherland and R.F. Sproull, Logical effort: Designing for speed on the back of an envelope, in *Proceedings of the 1991 University of California Santa Cruz Conference on Advanced Research in VLSI*, ed. C. Sequin, MIT Press, Santa Cruz, CA, pp. 1–16, 1991.
60. J. Grodstein, E. Lehman, H. Harkness, W.J. Grundmann, and Y. Watanabe, A delay model for logic synthesis of continuously-sized networks, in *Proceedings of the International Conference on Computer-Aided Design*, San Francisco, CA, pp. 458–462, 1995.
61. R.H.J.M. Otten, A design flow for performance planning: New paradigms for iteration free synthesis, in *Architecture Design and Validation Methods*, ed. E. Böerger, Springer, Berlin, Heidelberg, Germany; New York, pp. 89–139, 2000.
62. W.C. Elmore, The transient analysis of damped linear networks with particular regard to wideband amplifiers, *Journal of Applied Physics*, 19(1): 55–63, January 1948.
63. T. Sakurai, Approximation of wiring delay in MOSFET LSI, *IEEE Journal of Solid-State Circuits*, 18(4): 418–426, August 1983.
64. M.C.W. Geilen, T. Basten, B.D. Theelen, and R.H.J.M. Otten, An algebra of Pareto points, *Fundamenta Informaticae*, 78(1): 35–74, 2007.

3 Metrics Used in Physical Design

Frank Liu and Sachin S. Sapatnekar

CONTENTS

Physical design consists of a number of steps that attempt to optimize one or more specified design objectives, under one or more design constraints. This optimization is based on predictors and metrics that measure the value of the circuit property. These metrics must be computationally efficient, so that they may be embedded in the inner loop of an optimizer and may be called repeatedly during optimization, and yet have sufficient accuracy that is commensurate with the needs of the specific stage of physical design. In this chapter, we overview several metrics that may be used in objective and constraint functions in physical design, used to measure circuit properties such as timing, noise, power, and temperature. It should be noted that although area is also a metric used in optimization, area metrics are generally quite simple, and are not covered in this chapter.

3.1 TIMING

For most of today's VLSI designs, a dominant portion is synchronous in nature. In a synchronous design, a main clock signal is required to coordinate the operation of various logic blocks across the chip. A highly simplified view of a logic block is shown in Figure 3.1. The block consists of a cluster of combinational circuits, surrounded by the input and output latch banks, which may, e.g., be

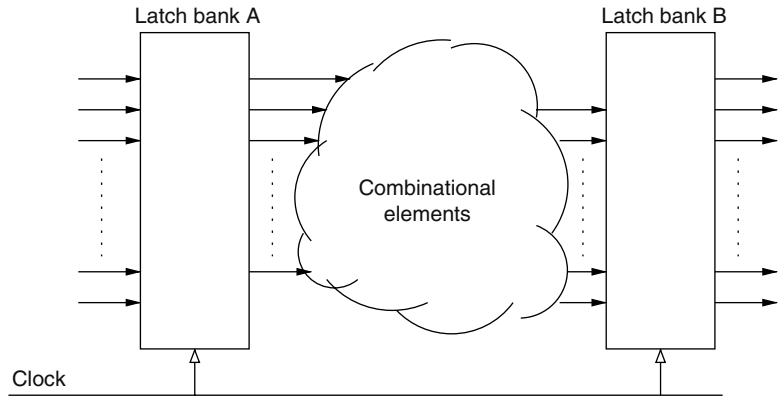


FIGURE 3.1 An illustrative timing diagram of a sequential circuit.

level clocked or edge triggered. A clock signal synchronizes the operations of the latch banks. The input latch bank provides primary inputs, which are computational results of the previous stage, to the combinational cluster, and the results of the logic computation are stored (or latched) by the output latch bank. Because the two latch banks open at a fixed interval, which is determined by the frequency of the clock signal, the time the combinational cluster takes to complete logic computation has to meet this constraint. In a modern VLSI design, the circumstance is much more complicated, but the general principle still holds.

It is quite likely that the combinational cluster will be constructed by the instances of logic gates from a predefined library. The timing performance of the combinational block is a strong function of the physical design, such as the placement of the gates, the routing of signal wires, as well as the sizing of the transistors. Therefore, any of these physical design optimizations must be guided by fast timing evaluators.

In this chapter, we briefly introduce the timing metrics commonly used in physical design. We first review the classic Elmore delay and slew metric, followed by more advanced fast timing estimation metrics. Finally, we review the fundamentals of static timing analysis of combinational circuits.

3.1.1 Elmore Delay and Slew Metrics

The dynamic behavior of an interconnect structure can be described by a system of ordinary differential equations. From a physical design point of view, this behavior can be characterized by two quantities: delay and slew (or rise/fall time), as depicted in Figure 3.2. This section outlines techniques for calculating these two quantities efficiently, with the given parameters of the interconnect structure.

3.1.1.1 Elmore Delay

The Elmore delay was first proposed by W. C. Elmore in 1948 [1], but did not receive much attention for over three decades. It was not until the 1980s, when the wire delays on an integrated circuit became nonnegligible, that it was rediscovered by Rubenstein et al. [2], and today, it is still the most popular timing metric in physical optimization. The reason for its popularity can be attributed not only to its simplicity but also to other important characteristics such as additivity, which we discuss later.

We will proceed under the reasonable assumption that an interconnect structure can be modeled as a set of lumped RLC segments, and we represent the impulse response of a specific node voltage in the circuit by $h(t)$. If we denote the Laplace transformation of $h(t)$ as $H(s)$, we can expand it into a Taylor series at $s = 0$:

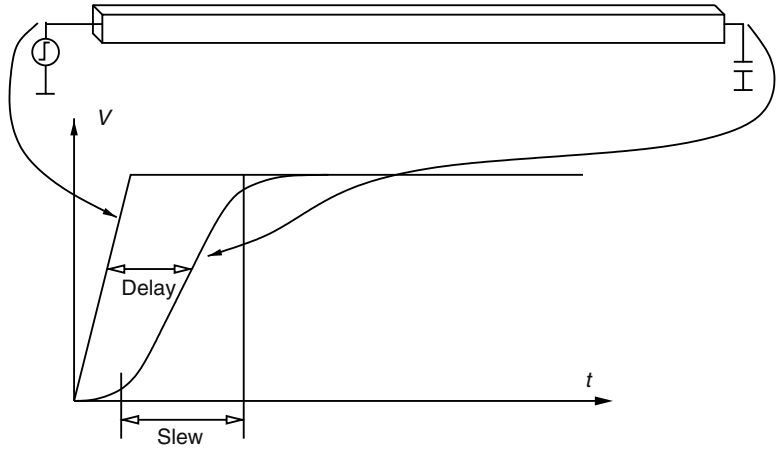


FIGURE 3.2 Delay and slew of a wire segment.

$$H(s) = \int_0^\infty h(t)\, \mathrm{e}^{-st}\mathrm{d}t = \sum_{k=0}^{\infty} \frac{(-1)^k}{k!} s^k \int_0^\infty t^k h(t)\mathrm{d}t \tag{3.1}$$

Therefore,

$$H(s) = m_0 + m_1 s + m_2 s^2 + m_3 s^3 + \cdots \tag{3.2}$$

where

$$m_k = \frac{(-1)^k}{k!} \int_0^\infty t^k h(t)\mathrm{d}t \quad \text{for} \quad k = 0, 1, 2, \ldots \tag{3.3}$$

The coefficients of the Taylor expansion is commonly known as the (circuit) moments.

For an RC circuit without resistive path to ground, the impulse response $h(t)$ satisfies the following conditions:

$$\begin{cases} h(t) \geq 0, & \forall\, t \\ \int_0^\infty h(t)\mathrm{d}t = 1 \end{cases} \tag{3.4}$$

In probability theory, any continuous real function that satisfies Equation 3.4 is a probability density function (PDF). The integral of a PDF is defined as a cumulative density function:

$$S(t) = \int_0^t h(\tau)\mathrm{d}\tau \tag{3.5}$$

This corresponds to the step response in circuit analysis (Figure 3.3).

Several characteristics are commonly used to describe a statistical distribution. The first is the mean, which is defined as

$$\mu = \int_0^\infty t h(t)\mathrm{d}t \tag{3.6}$$

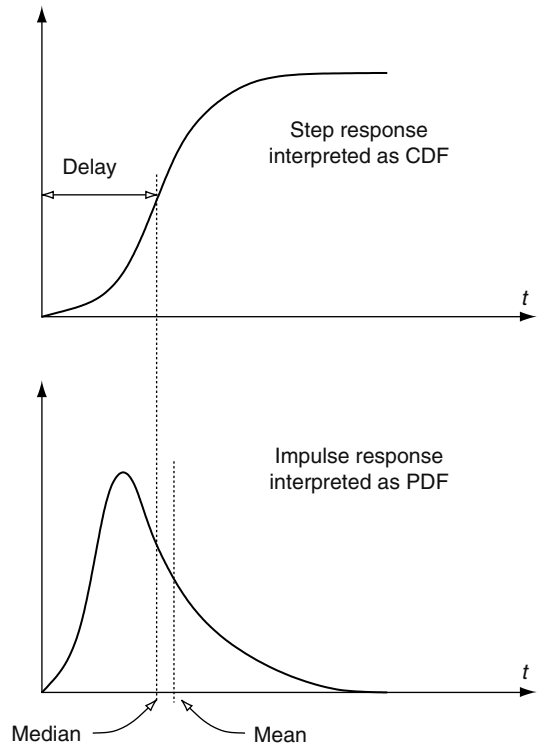


FIGURE 3.3 Elmore delay: approximating the median with the mean.

Another important characteristic is the median, which is defined as the halfway point on a PDF curve:

$$\int_0^M h(t)\mathrm{d}t = \frac{1}{2} \tag{3.7}$$

The similarity between the impulse response of an RC tree and a statistical PDF is quite clear. Observe that the commonly used 50 percent delay point in circuit analysis actually corresponds to the median of the underlying distribution. This is the keen observation of Elmore in 1948. Moreover, he also made the proposal that as the median was difficult to calculate, one could use the mean, which is much easier to calculate, as an approximation of median:

$$M \approx \mu = -m_1 = \int_0^\infty t\, h(t)\mathrm{d}t \tag{3.8}$$

3.1.1.2 Elmore Delay for RC Trees

For an RC tree (i.e., an RC network with no direct resistive path to ground), the calculation of Elmore delay can be carried out quite efficiently. In such a case, the Elmore delay between any two nodes can be expressed as

$$\mu = \sum R_i \cdot \sum_{\text{downstream}} C_j \tag{3.9}$$

where

R_i is the traversal of the resistors on the unique path between two nodes

C_j permutes all the capacitance seen from resistor R_i

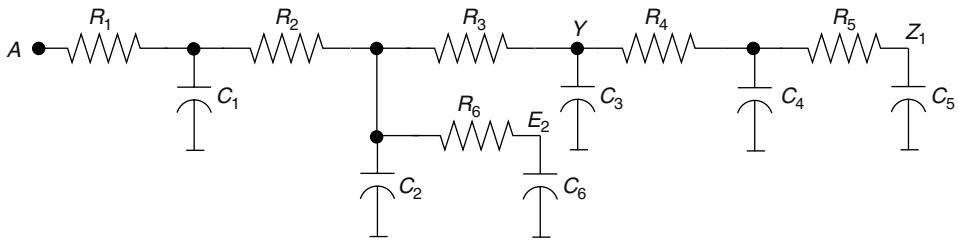


FIGURE 3.4 An example of RC tree to illustrate the process of calculating Elmore delay.

For the simple example shown in Figure 3.4, the Elmore delay from root node A and fan-out node $Z1$ can be calculated by traversing the unique resistive path from $Z1$ to A:

$$\begin{aligned}\mathrm{ED}_{A\to Z1} = {} & R_5C_5 + R_4(C_4 + C_5) + R_3(C_3 + C_4 + C_5) \\ & + R_2(C_2 + C_3 + C_4 + C_5 + C_6) \\ & + R_1(C_1 + C_2 + C_3 + C_4 + C_5 + C_6)\end{aligned}$$

The Elmore delay has a nice property: it is additive. In other words, for two nodes A and C on a branch, if node B lies between A and C, we can write:

$$\mathrm{ED}_{A\to C} = \mathrm{ED}_{A\to B} + \mathrm{ED}_{B\to C}$$

For the example shown in Figure 3.4, we can easily verify that

$$\begin{aligned}\mathrm{ED}_{A\to Y} = {} & R_3(C_3 + C_4 + C_5) + R_2(C_2 + C_3 + C_4 + C_5 + C_6) \\ & + R_1(C_1 + C_2 + C_3 + C_4 + C_5 + C_6) \\ \mathrm{ED}_{Y\to Z1} = {} & R_5C_5 + R_4(C_4 + C_5)\end{aligned}$$

Thus,

$$\mathrm{ED}_{A\to Z1} = \mathrm{ED}_{A\to Y} + \mathrm{ED}_{Y\to Z1}$$

The Elmore delay of an RC tree has another important property: it can be proven to be the upper bound of the true 50 percent circuit delay under any input excitation [3]. In other words, if a particular RC net is optimized based on the Elmore delay, its real delay is guaranteed to be better. Empirically it has been shown that although the Elmore delay is the upper bound, the error can be quite substantial in some cases, especially for those nodes close to the driving point. The accuracy for far-end nodes (those close to the sink pins) is much better. Note that this property only applies to RC trees, and it does not hold for nontree circuits, e.g., meshes.

The Elmore delay can also be calculated for distributed circuits. For a uniform wire at the length of L, with a unit resistance R, a unit capacitance C, and a loading capacitance C_L, it can be shown that the Elmore delay at the far-end of the wire is

$$\mathrm{ED} = \frac{1}{2}RL(CL + C_\mathrm{L})$$

3.1.1.3 Elmore Delay for Nontrees

For a nontree RC network, the calculation of Elmore delay is more involved. The simple traversal algorithm for tree-like structures is no longer valid. Instead, we can formulate the circuit into the

modified nodal analysis (MNA) formulation and solve for the moments. In this case, a linear circuit can be formulated as

$$\mathbf{G}x(t) + \mathbf{C}\frac{\mathrm{d}}{\mathrm{d}t}x(t) = \mathbf{B}u(t)$$

where

G is the conductance matrix
C is the capacitance matrix
matrix **B** specifies where the excitations are applied

The entries in unknown vector $x(t)$ consists of node voltages, branch currents of voltage sources, as well as branch currents of inductors. $u(t)$ is the external time-varying excitation. The Laplace transformation of the MNA formulation is

$$\mathbf{G}X(s) + s\mathbf{C}X(s) = \mathbf{B}U(s)$$

The first circuit moment is

$$m_1 = -\mathbf{G}^{-1}\mathbf{C}\mathbf{G}^{-1}\mathbf{B}$$

Therefore, the Elmore delay at a particular node can be calculated by selecting the corresponding entry in the vector of the first moment:

$$ED_i = \mathbf{e}_i^{\mathrm{T}}\mathbf{G}^{-1}\mathbf{C}\mathbf{G}^{-1}\mathbf{B}$$

where vector $\mathbf{e}_i$ is the selection vector with all entries zero except at the ith location.

Computationally, only one LU factorization of the conductance matrix **G** is required in the above calculation, and the rest of calculation is merely forward–backward substitution of the prefactorized matrix as well as matrix–vector multiplication, which can be carried out quite efficiently.

It is also worth pointing out that the above procedure is the general description of the Elmore delay calculation for any linear circuit. Thus, it can be used to calculate the Elmore delay of an RC tree as well. However, due to its special topology, the LU factorization of an RC tree can be carried out without explicit formulation of the conductance and capacitance matrices, and a closed-form formula, described earlier, for the Elmore delay can be obtained. More details on how to construct the MNA matrices and the calculation of Elmore delay for a general circuit can be found in Ref. [4].

3.1.1.4 Elmore Slew

In his original paper, Elmore refereed to slew as the gyration. If we follow the probability interpretation of signal transition, it can be shown that just as the delay corresponds to the median of the PDF function, the slew corresponds to the variance of the PDF function. A first-order estimate of variance is the second central moment, which is defined as

$$\sigma^2 = m_1^2 - 2m_2$$

In practice, because quite often slew is defined as the difference of delay between 10 percent and 90 percent delay points, the above metric needs to be scaled accordingly.

$$\text{Slew} = \frac{8}{10}\sqrt{m_1^2 - 2m_2}$$

Note that we need the second circuit moment to calculate the slew. In general, it can be shown that the second circuit moment can be calculated in MNA formulation as

$$m_2 = \mathbf{G}^{-1}\mathbf{C}\mathbf{G}^{-1}\mathbf{C}\mathbf{G}^{-1}\mathbf{B}$$

In practice, the factorized matrix **G** during m_1 calculation can be reused to calculate m_2. Therefore, the added computational complexity is only a few matrix–vector multiplications and

backward/forward substitutions, which are usually much cheaper than matrix factorization itself. For RC trees, the matrix does not need to be explicitly formulated and factorized at all. The path-tracing algorithm used in m_1 calculation can be applied as well. More details can be found in Ref. [4].

3.1.1.5 Limitations of Elmore Delay

As we have discussed earlier, the Elmore delay has a few very nice properties when applied on RC trees. They are

- Easy to calculate
- Proven to be the upper bound for any node under any input excitation
- Additive along the signal path

During physical design, most on-chip signal wires can be modeled as trees, therefore, the Elmore delay has been quite popular and has been implemented in many physical design algorithms.

However, the Elmore metric also has some limitations, especially in terms of accuracy. Empirically it has been shown that even for RC trees, the accuracy of Elmore delay can be over ten times off at certain nodes, especially for the nodes close to the driving point. The reason for this inaccuracy can be explained as follows: the essence of Elmore delay is to use mean to approximate median for a particular PDF. Such an approximation is only accurate when the PDF is unimodal and has zero skew, e.g., the PDF is symmetric. For an RC tree, this is only true for far-end nodes. For the near-end nodes (the ones which are close to the driving point), the skewness of the impulse response (which we interpreted as a PDF) is quite large. As a consequence, the approximation used in Elmore delay becomes inaccurate.

3.1.2 Fast Timing Metrics

The essence of Elmore delay is the probability interpretation of the impulse response of a linear circuit. This allows the signal response to be approximated by using a structured continuous function as the template, thus making it possible to quickly extract delay and slew metrics. In the derivation of Elmore delay, it is assumed that the underlying PDF function is symmetric. A natural extension of the idea is to remove this assumption: we can use an asymmetric PDF and hopefully the accuracy can be improved. In the first proposed method [5], the gamma distribution function was used as the template function. Later on, other distribution functions are proposed to be the template function, including the Weibull [6] and lognormal [7] functions. Another benefit of these extended approaches is that we are no longer limited to the 50 percent delay point. Once the parameters of the function template are known, we can calculate any percentile delay point. The price we have to pay to get better accuracy is that more moments are needed. Besides, all of these fast delay metrics cannot be proved to be the upper bound of the true delay, although empirically it has been shown that overall they are more accurate.

3.1.2.1 PRIMO and H-Gamma

The idea of PRIMO [5] was to approximate the circuit impulse response as the PDF function of a gamma distribution. Because only two parameters are needed to determine a gamma distribution, these two parameters can be easily determined by applying the moment-matching principle. Once the coefficients of the gamma distribution are known, we do not need to approximate the median with the mean. Instead, we can directly calculate the median, which corresponds to the 50 percent delay. Later, an improved version of gamma fitting was introduced in H-gamma [8]. Here, we only describe H-gamma.

The gamma statistical distribution is defined on support $x > 0$, with the PDF defined as

$$f(x;k,\theta) = \frac{\theta^k x^{k-1}\, \mathrm{e}^{-\theta x}}{\Gamma(k)}$$

where $\Gamma(k)$ is the gamma function:

$$\Gamma(k) = \int_0^\infty x^{k-1}\, \mathrm{e}^{-x}\, \mathrm{d}x$$

Each gamma distribution is uniquely determined by two parameters, k and θ, and both of them have to be positive. The mean and the variance of a gamma distribution are

$$\text{mean} = \frac{k}{\theta}$$
$$\text{variance} = \frac{k}{\theta^2}$$

To derive H-gamma, we can rewrite the impulse response of a circuit node as

$$\begin{aligned} Y(s) &= m_0 + m_1 s + m_2 s^2 + m_3 s^3 + \cdots \\ &= m_0 + m_1 s \left(1 + \frac{m_2}{m_1} s + \frac{m_3}{m_1} s^2 + \cdots\right) \end{aligned}$$

The series in parenthesis is referred as the normalized homogeneous function. In H-gamma, the normalized homogeneous function is fit into the PDF of a gamma distribution by matching the first two moments. The results are

$$\frac{k}{\theta} = -\frac{m_2}{m_1}$$
$$\frac{k}{\theta^2} = 2\left(\frac{m_3}{m_1}\right) - \left(\frac{m_2}{m_1}\right)^2$$

Once two parameters k and θ are calculated, we can approximate the step response as

$$y(t) \approx 1 + m_1 \frac{\theta^k t^{k-1}\, \mathrm{e}^{-\theta t}}{\Gamma(k)}$$

The delay at any percentile point ϕ can be calculate by setting the left-hand-side of the above equation to ϕ and solve for t. Unfortunately, this process requires a nonlinear iteration method such as Newton–Raphson because this equation cannot be explicitly solved.

To address this issue, the nonlinear iteration process can be simplified to a table look-up procedure by scaling time t with θ, and k with $-m_1$. The scaled response approximation can be shown to be

$$y_{\lambda,k}(x) = 1 - \frac{\lambda x^{k-1}\, \mathrm{e}^{-x}}{\Gamma(k)}$$

For any percentile ϕ, a two-dimensional table needs to be preconstructed with λ and k as the input and x as the output. The final delay is then calculated by scaling x with θ: $t = x/\theta$. Empirically it has been shown that H-gamma metric has good accuracy for both near and far-end nodes. One reason for its accuracy is particularly due to the fact that three moments are used to calculate the delay at each node.

3.1.2.2 Weibull-Based Delay

Another proposed delay metric uses Weibull distribution as the underlying function template. The advantage of using the Weibull distribution is that the percentile points are very easy to calculate. A Weibull distribution is defined on the support of $t > 0$ and is determined by two parameters:

$$f(x\colon \alpha, \beta) = \alpha \beta^{-\alpha} x^{\alpha-1}\, \mathrm{e}^{-(x/\beta)^\alpha}$$

Both parameters, α and β, must be positive. The mean and variance of a Weibull distribution is

$$\text{Mean} = \beta\Gamma(1+\theta)$$
$$\text{Variance} = \beta^2[\Gamma(1+2\theta) - \Gamma^2(1+\theta)]$$

Unlike the gamma distribution, in which the distribution parameters can be easily calculated from moments, the Weibull distribution requires iterative evaluation of gamma functions. To simplify the process, it is proposed that a look-up table be precharacterized. The look-up table requires the first two circuit moments as inputs and it returns the parameter θ:

r	$\text{Log}_{10}(r)$	θ
0.63096	−0.2	0.48837
0.79433	−0.1	0.76029
1.00000	+0.0	1.00000
1.25892	+0.1	1.22371
1.58489	+0.2	1.43757
1.99526	+0.3	1.64467
2.51189	+0.4	1.84678
3.16228	+0.5	2.04507
3.98107	+0.6	2.24031
5.01187	+0.7	2.43305
6.30957	+0.8	2.62371
7.94328	+0.9	2.81262
10.00000	+1.0	3.00000
12.58925	+1.1	3.18607
15.84893	+1.2	3.37098

where $r = m_2/m_1^2$. Note that it is recommended to use $\log_{10}(r)$ value in the interpolation. Once θ is known, the other parameter, β, is calculated by using the following equation:

$$\beta = \frac{-m_1}{\Gamma(1+\theta)}$$

Although an evaluation of the gamma function is again needed, the following table can be used to avoid the evaluation:

x	Gamma(x)
1.0	1.00000
1.1	0.95135
1.2	0.91817
1.3	0.89747
1.4	0.88726
1.5	0.88623
1.6	0.89352
1.7	0.90864
1.8	0.93138
1.9	0.96176
2.0	1.00000

The table only covers the data range between 1 and 2, and the following recursive property of the gamma function can be used to calculate other x:

$$\Gamma(x+1) = x\Gamma(x) \quad \forall\, x > 1$$

Once α and β are known, the delay at any percentile ϕ can be calculated as

$$t_\phi = \beta \left(\ln \frac{1}{1-\phi} \right)^\theta$$

In particular, the 50 percent delay point can be calculated as

$$t_{0.5} = \beta[\ln(2)]^\theta \approx \beta \cdot (0.693)^\theta$$

3.1.2.3 Lognormal Delay

Another delay metric uses lognormal distribution for probability interpretation of response signal [7]. The lognormal distribution is determined by two parameters μ and σ. Its PDF is defined as

$$f(x;\mu,\sigma) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\left\{ \frac{[\ln(x)-\mu]^2}{2\sigma^2} \right\}$$

Similar to Weibull-based delay, the first two circuit moments are matched to the moments of the distribution to calculate μ and σ. Once they are known, the delay can be calculated by calculating the median of the lognormal distribution. After simplification, it turns out that the 50 percent delay metric is a closed form of the two circuit moments:

$$t_{0.5} = \frac{m_1^2}{\sqrt{2m_2}}$$

The lognormal distribution can also be used to provide a closed-form slew metric. Because slew metric is equivalent to the difference of two delay points (e.g., 10 percent and 90 percent delay), the accuracy requirement is higher. In some cases, especially for the near-end nodes, metrics based on two moments may not be sufficiently accurate. To achieve the balance between the accuracy and complexity, a three-piece approach was proposed, based on the value of $r = m_1/\sqrt{m_2}$:

- $r \le 0.35$:

$$\text{Slew}_{12} = \frac{m_1^2}{\sqrt{2m_2}} \left(e^{kS\sqrt{2}} - e^{-kS\sqrt{2}} \right)$$

  where $S = \sqrt{\ln(2m_2/m_1^2)}$, and the value of k depends on the definition of slew and is explained later.
- $r \ge 1$

$$\text{Slew}_{23} = \sqrt{\frac{2m_2 - m_1^2}{z(z-1)} \left(e^{k\sqrt{2\ln(z)}} - e^{-k\sqrt{2\ln(z)}} \right)}$$

  where $z = (y - 1/y)^2 + 1$ and $y = \sqrt[3]{(\gamma + \sqrt{4+\gamma^2})/2}$, where $\gamma = (-6m_3 + 6m_1m_2 - 2m_1^3)/(2m_2 - m_1^2)^{3/2}$ and k is the function of slew ratio.
- $0.35 < r < 1$

$$\text{Slew} = \left(\frac{20}{13}r - \frac{7}{13} \right) \text{slew}_{23} + \frac{20}{13}(1-r)\,\text{slew}_{12}$$

The value k is the scaling factor needed to reflect difference in terms of slew definition. It is calculated based on the table below:

Slew Definition	k
10/90	0.9063
20/80	0.5951
25/75	0.4769
30/70	0.3708

3.1.3 Fundamentals of Static Timing Analysis

As discussed earlier in this section, a sequential circuit consists of combinational elements and sequential elements and can be represented as a set of combinational blocks that lie between latches. This subsection presents methods that compute the delay of a combinational logic block.

A combinational logic circuit can be represented as a timing graph $G = (V, E)$, where the elements of V, the vertex set, are the logic gates in the circuit and the primary inputs and outputs of the circuit. A pair of vertices, u and $v \in G$, are connected by a directed edge $e(u, v) \in E$ if there is a connection from the output of the element represented by vertex u to the input of the element represented by vertex v. A simple logic circuit and its corresponding graph are illustrated in Figure 3.5a and b, respectively. In this section, we present techniques that are used for the static timing analysis of digital combinational circuits. The word "static" alludes to the fact that this timing analysis is carried out in an input-independent manner, and purports to find the worst-case delay of the circuit over all possible input combinations. The method is often referred to as CPM (critical path method). The computational efficiency of CPM has resulted in its widespread use, even though it has some limitations.

The CPM-based algorithm, applied to a timing graph $G = (V, E)$, can be summarized by the pseudocode shown below:

```
Algorithm CRITICAL_PATH_METHOD
Q = Ø;
for all vertices i ∈ V
   n_visited_inputs [i] = 0;
/* Add a vertex to the tail of Q if all inputs are ready */
for all primary inputs i
   /* Fanout gates of i */
   for all vertices j such that (i → j) ∈ E
      if (++n_visited_inputs[j] == n_inputs[j]) addQ(j,Q);
while (Q ≠ Ø) {
   g = top(Q);
   remove (g,Q);
   compute_delay[g]
   /* Fanout gates of g */
   for all vertices k such that (g → k) ∈ E
      if (++n_visited_inputs[k] == n_inputs[k]) addQ(k,Q);
}
```

The procedure is best illustrated by means of a simple example. Consider the circuit in Figure 3.6, which shows an interconnection of blocks. Each of these blocks could be as simple as a logic gate

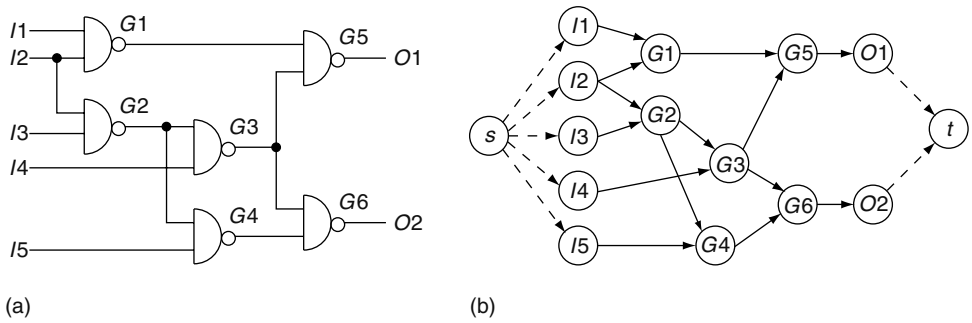


FIGURE 3.5 (a) An example combinational circuit and (b) its timing graph. (From Sapatnekar, S. S., *Timing*, Kluwer Academic Publisher, Boston, MA, 2004. With permission.)

or could be a more complex combinational block, and is characterized by the delay from each input pin to each output pin. For simplicity, this example will assume that for each block, the delay from any input to the output is identical. Moreover, we will assume that each block is an inverting logic gate such as a NAND or a NOR, as shown by the "bubble" at the output. The two numbers, d_r/d_f, inside each gate represent the delay corresponding to the delay of the output rising transition, d_r, and that of the output fall transition, d_f, respectively. We assume that all primary inputs are available at time zero, so that the numbers "0/0" against each primary input represent the worst-case rise and fall arrival times, respectively, at each of these nodes. The critical path method proceeds from the primary inputs to the primary outputs in topological order, computing the worst-case rise and fall arrival times at each intermediate node, and eventually at the outputs of a circuit.

A block is said to be ready for processing when the signal arrival time information is available for all of its inputs; in other words, when the number of processed inputs of a gate g, `n_visited_inputs[g]`, equals the number of inputs of the gate, `n_inputs[g]`. Notationally, we refer to each block by the symbol for its output node. Initially, because the signal arrival times are known only at the primary inputs, only those blocks that are fed solely by primary inputs are ready for processing. In the example, these correspond to the gates i, j, k, and l. These are placed in a queue `Q` using the function `addQ`, and are processed in the order in which they appear in the queue.

In the iterative process, the block at the head of the queue `Q` is taken off the queue and scheduled for processing. Each processing step consists of

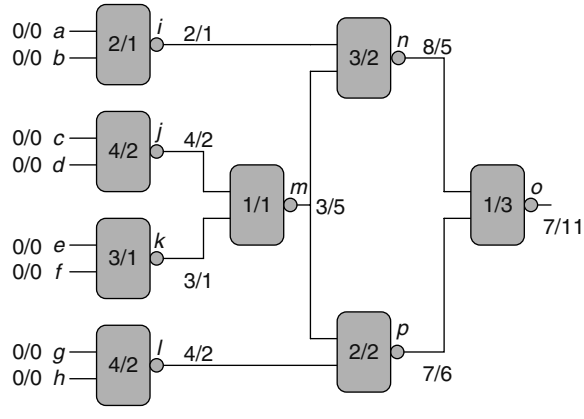


FIGURE 3.6 An example illustrating the application of the CPM on a circuit with inverting gates. The numbers within the gates correspond to the rise delay/fall delay of the block, and the bold numbers at each block output represent the rise/fall arrival times at that point. The primary inputs are assumed to have arrival times of zero, as shown. (From Sapatnekar, S. S., *Timing*, Kluwer Academic Publisher, Boston, MA, 2004. With permission.)

- Finding the latest arriving input to the block that triggers the output transition (this involves finding the maximum of all worst-case arrival times of inputs to the block), and then adding the delay of the block to the latest arriving input time, to obtain the worst-case arrival time at the output. This is represented by function `compute_delay` in the pseudocode.
- Checking all of the block that the current block fans out to, to find out whether they are ready for processing. If so, the block is added to the tail of the queue using function `addQ`.

The iterations end when the queue is empty. In the example, the algorithm is executed as follows:

Step 1: In the initial step gates, i, j, k, and l are placed on the queue because the input arrival times at all of their inputs are available.
Step 2: Gate i, at the head of the queue, is scheduled. Because the inputs transition at time 0, and the rise and fall delays are 2 and 1 units, respectively, the rise and fall arrival times at the output are computed as $0 + 2 = 2$ and $0 + 1 = 1$, respectively. After processing i, no new blocks can be added to the queue.
Step 3: Gate j is scheduled, and the rise and fall arrival times are similarly found to be 4 and 2, respectively. Again, no additional elements can be placed in the queue.
Step 4: Gate k is processed, and its output rise and fall arrival times are computed as 3 and 1, respectively. After this computation, we see that all arrival times at the input to gate m have been determined. Therefore, it is deemed ready for processing, and is added to the tail of the queue.
Step 5: Gate l is now scheduled, and the rise and fall arrival times are similarly found to be 4 and 2, respectively, and no additional elements can be placed in the queue.
Step 6: Gate m, which is at the head of the queue, is scheduled. Because this is an inverting gate, the output falling transition is caused by the latest input rising transition, which occurs at time $\max(4, 3) = 4$. As a consequence, the fall arrival time at m is given by $\max(4, 3) + 1 = 5$. Similarly, the rise arrival time at m is $\max(2, 1) + 1 = 3$. At the end of this step, both n and p are ready for processing and are added to the queue.
Step 7: Gate n is scheduled, and its rise and fall arrival times are calculated as $\max(1, 5)+3 = 8$ and $\max(2, 3) + 2 = 5$ respectively.
Step 8: Gate p is now processed, and its rise and fall arrival times are found to be $\max(5, 2) + 2 = 7$ and $\max(3, 4) + 2 = 6$, respectively. This sets the stage for adding gate o to the queue.
Step 9: Gate o is scheduled, and its rise and fall arrival times are $\max(5, 6) + 1 = 7$ and $\max(8, 7) + 3 = 11$, respectively. The queue is now empty and the algorithm terminates.

The worst-case delay for the entire block is therefore $\max(7, 11) = 11$ units.

Because there are many paths in a combinational block, it is important to identify the path (or paths) on which the worst-case delay of the whole block is achieved for physical design optimization. The critical path, defined as the path between an input and an output with the maximum delay, can be easily found by using a traceback method. We begin with the block whose output is the primary output with the latest arrival time: this is the last block on the critical path. Next, the latest arriving input to this block is identified, and the block that causes this transition is the preceding block on the critical path. The process is repeated recursively until a primary input is reached.

In the example, we begin with Gate o at the output, whose falling transition corresponds to the maximum delay. This transition is caused by the rising transition at the output of gate n, which must therefore precede o on the critical path. Similarly, the transition at n is affected by the falling transition at the output of m, and so on. By continuing this process, the critical path from the input to the output is identified as being caused by a falling transition at either input c or d, and then progressing as follows: rising $j \rightarrow$ falling $m \rightarrow$ rising $n \rightarrow$ falling o.

3.2 NOISE

Coupling noise is yet another unwanted side effect of the scaling in deep submicron technology, and its impact can be reduced through physical design transformations. The effect arises due to geometric scaling, which requires the wires to be narrower and the spacing between adjacent wires smaller. On the other hand, because the chip size is also getting larger (in terms of multiples of the minimum feature size), it is necessary to reduce wire resistance by increasing the aspect ratio of the wire cross section. The compounded effect is the increase of the coupling capacitance between adjacent signal wires.

When two interconnect networks are capacitively coupled, usually the one with the stronger driving gate is referred to as the aggressor, while the one with the weaker driver is called the victim. It is quite possible that an aggressor can affect multiple victims, and a victim can have more than one aggressors. For simplicity, we only discuss the case with one aggressor and one victim. These ideas can be easily extended to more general cases. The application domain for this analysis is in noise-aware routing. For scalable methods that can be applied to full-chip noise analysis, the reader is referred to Chapter 34.

When the aggressor switches, if the victim is quiet, then the coupling will generate a glitch on the victim wire. If the glitch is sufficiently large and occurs within a certain timing window, the (erroneous) glitch can be latched into a memory storage element and cause a logic error. If the victim is also switching, then depending on the polarities of the signals and the corresponding switching windows, the signal on the victim wire can be slowed down or sped up, which may cause timing violations. Although very elaborate algorithms are available to estimate the coupling effects between the signal wires, (see Refs. [9,10]), it is highly desirable to correct the problem at its root, i.e., during the physical design phase.

The exact amount of noise injected to the victim net from the aggressor is a function of circuit topologies and values of both aggressor and victim nets, as well as the properties of the signal. To accurately estimate the noise, every component of the coupled network is required, which is not realistic during physical design. Fortunately, there is a simple noise metric equivalent to Elmore delay in timing [11].

In Ref. [11], it is assumed that the excitations in the aggressor net are infinite ramps, which are signals whose first derivatives are zero before $t = 0$, and constant afterward. From the circuit analysis point of view, the coupling capacitors act like differentiators. Thus, the coupling node voltage will come to a steady state, whose level can be used as an indicator of the coupling effect. For a circuit of general topology, the noise metric must be solved with circuit analysis techniques, which involves the construction of MNA formulations and solving of the matrices. The MNA matrices of a coupled circuit can be written as

$$\begin{bmatrix} \mathbf{G}_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{G}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} + \begin{bmatrix} \mathbf{C}_{11} & \mathbf{C}_c \\ \mathbf{C}_c^{\mathbf{T}} & \mathbf{C}_{22} \end{bmatrix} \begin{bmatrix} \dot{\mathbf{x}}_1 \\ \dot{\mathbf{x}}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{B}_1 \\ \mathbf{0} \end{bmatrix} \mathbf{u}$$

In the above equation, the first partition is the aggressor net while the second partition is the victim net. The submatrices $\mathbf{G}_{11}$ and $\mathbf{C}_{11}$ represent the conductance and capacitance of the aggressor net; and $\mathbf{G}_{22}$ and $\mathbf{C}_{22}$ represent the conductances and capacitances of the victim net; while $\mathbf{C}_c$ represents the coupling between the two nets. According to Ref. [11], simple algebraic manipulations can be employed to estimate the worst case as

$$\mathbf{V}_{2,\max} = \mathbf{G}_{22}^{-1}\mathbf{C}_c\mathbf{G}_{11}^{-1}\mathbf{B}_1\,\dot{\mathbf{u}}$$

Note that the worst-case noise is only a function of the resistances of the victim and aggressor nets, as well as the coupling capacitances. It is not a function of the self-capacitances of the two nets (under the assumption that the input is an infinite ramp).

If both aggressor and victim nets have tree-like topologies, then the above noise metric can be calculated with a simple graph traversal, which is similar to the Elmore delay calculation of tree-like RC networks. To illustrate the procedure, we can rewrite the worst-case noise metric as

$$\begin{aligned} \mathbf{I_c} &= \mathbf{C_c G_{11}^{-1} B_1}\, \dot{\mathbf{u}} \\ \mathbf{V_{2,max}} &= \mathbf{G_{22}^{-1} I_c} \end{aligned} \tag{3.10}$$

Because the excitation in the aggressor net is an infinite ramp, the term $\mathbf{I_c}$ represents the coupling current injected into the victim net. If the aggressor net is properly connected, then it can be shown using simple circuit arguments [11] that the injected current is simply $\mathbf{C_c \dot{u}}$. The calculation of the voltage in the victim net can then be carried out using a procedure similar to Elmore delay propagation, except that we traverse the tree from the root to the leaf nodes. To illustrate the procedure, we give a simple example shown in Figure 3.7.

Because the noise is not a function of the self-capacitance of either the aggressor or the victim, it is not drawn in the diagram. In the first step of the calculation, the equivalent current injections from the aggressor net is calculated, which correspond to evaluating the first equation in Equation 3.10. Because there is no direct resistive path to ground and there is only one independent voltage source in the aggressor, it is trivial to show that

$$\begin{aligned} I_1 &= C_1 \cdot \dot{u} \\ I_2 &= C_2 \cdot \dot{u} \\ I_3 &= C_3 \cdot \dot{u} \end{aligned}$$

We then replace the coupling capacitors of the victim with those current sources. Because the root of the tree is grounded, we calculate the worst-case noise by a graph traversal, from root to leaves:

$$\begin{aligned} I_{B,\max} &= R_1(C_1 + C_2 + C_3)\dot{u} \\ I_{D,\max} &= R_1(C_1 + C_2 + C_3)\dot{u} + R_2(C_2 + C_3)\dot{u} \\ I_{E,\max} &= R_1(C_1 + C_2 + C_3)\dot{u} + R_2(C_2 + C_3)\dot{u} + R_3(C_3)\dot{u} \\ I_{F,\max} &= R_1(C_1 + C_2 + C_3)\dot{u} + R_2(C_2 + C_3)\dot{u} \end{aligned}$$

Although Devgan's metric is easy to calculate, its accuracy is limited. For fast transitions, in particular, the metric evaluates to a physically impossible value that exceeds the supply voltage. Note that the evaluation is still correct, as Devgan's noise metric only guarantees an upper bound on the noise; however, the accuracy in such cases is clearly limited.

To improve accuracy, a further improvement of the static noise metric was proposed in Ref. [12], which extended the idea of Devgan's metric through the use of more than one moment. In addition,

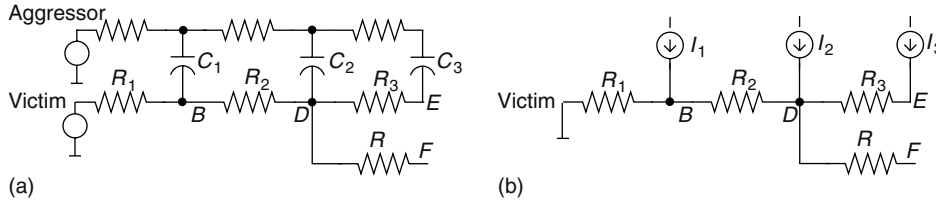


FIGURE 3.7 An example of worst-case noise calculation, showing (a) the original circuit and (b) the equivalent circuit when coupling capacitors are replaced by injected noise current sources. (From Sapatnekar, S. S., *Timing*, Kluwer Academic Publisher, Boston, MA, 2004. With permission.)

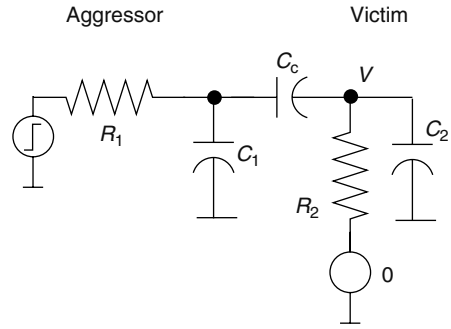


FIGURE 3.8 Circuit diagram for the derivation of the transient noise peak.

the aggressor excitation was assumed to be a first-order exponential function rather than an infinite ramp. To obtain a closed-form noise metric, the response at the victim net was calculated using moment-matching approach.

Another type of noise metric that takes the transient approach is the work in Ref. [13]. Instead of assuming that the excitation at the aggressor is an infinite ramp, it is assumed that the aggressor excitation is a step signal. However, the circuit topology is highly simplified so that a close-form noise metric can be derived. The aggressor–victim pair is simplified as shown in Figure 3.8 [13], where R_1 is the total resistance of the aggressor and R_2 is the total resistance of the victim. Note that the victim is grounded by a zero-valued voltage source. It then can be proved that the noise peak at node V is

$$X_V = \frac{1}{1 + \frac{C_2}{C_c} + \frac{R_1}{R_2}\left(1 + \frac{C_1}{C_c}\right)}$$

An alternative two-stage π model is presented in the metric in Ref. [14].

For long global nets, there is also the possibility that two nets are inductively coupled, especially when the the aggressor and victim nets are in parallel, such as in a bus structure. The analysis and estimation of the inductively coupling is much more involved. Because inductive coupling mostly occurs in selected cases, quite often detailed circuit analysis is affordable. In many cases, various types of shielding are implemented to minimize the inductive coupling effect [15,16].

3.3 POWER

With the decreasing transistor channel lengths and increasing die sizes, power dissipation has become a major design constraint. There are three major components of power dissipation: the dynamic power, the short-circuit power, and the static power. Historically, the dynamic power and short-circuit power have been the subject of many studies. In recent years, as the complimentary metal oxide semiconductor (CMOS) devices rapidly approach the fundamental scaling limit, static power has become a major component of the total power consumption. We discuss these components separately in the subsequent subsections.

3.3.1 Dynamic Power

For a CMOS circuit, its states are represented by the charges stored at various metal oxide semiconductor field effect transistors (MOSFETs). When the circuit is operating, the change of the circuit states is realized by charging and discharging of these transistors. This charge/discharge operation can be illustrated in the simple circuit shown in Figure 3.9. When the circuit is in quiescence, inverters INV1, INV2, and INV3 store 1, 0, and 1, respectively. When a falling transition occurs at the input

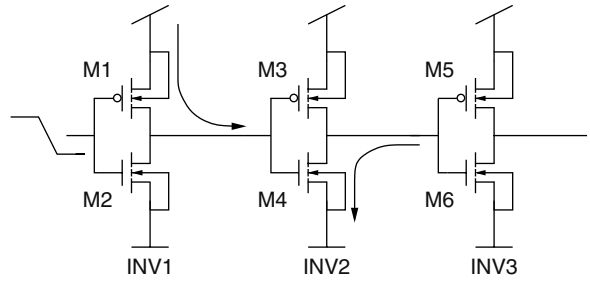


FIGURE 3.9 A simple circuit to illustrate charging and discharging of the gate capacitors.

of INV1, the state of INV2 changes from 0 to 1, by charging the gate capacitor of MOSFET M3 and M4 via transistor M1. In the meantime, the state of INV3 changes from 1 to 0, which is achieved by discharging the gate capacitor of M5 and M6 to ground (via M4).

Consider gate INV1, whose output is being charged from low to high. During this transition, the output parasitic capacitance is charged, and some energy is dissipated in the (nonlinear) positive-channel metal oxide semiconductor (PMOS) transistor resistance. It can be shown that for a single transition, each of these components equals $\frac{1}{2}C_L V_{DD}^2$, where C_L is the load capacitance at the output, and this energy is supplied by the V_{DD} source. In a subsequent cycle, when the output of INV1 discharges, all of the dynamic energy dissipated in the negative-channel metal oxide semiconductor (NMOS) transistor comes from the capacitor C_L, and none comes from V_{DD}. Therefore, for every high-to-low-to-high transition, the energy dissipated in a single cycle can be calculated as

$$P_{\text{dynamic}} = C_{\text{gate}} V_{DD}^2 \tag{3.11}$$

where C_{gate} is the total capacitance of all gate capacitors involved. If a rising transition occurs next, the energy stored at the gate capacitance of M3 and M4 is simply dissipated to ground.

This concept can be generalized from inverters to arbitrary gates, and the essential idea and the formula remain valid. If the clock frequency is f and the gate switches on every clock transition, then the number of transitions is multiplied by f. The power dissipated can then be calculated as the energy dissipated per unit time. In general, though, a gate does not switch on every single clock transition, and if α is the probability that a gate will switch during a clock transition, the dynamic power of the gate can be estimated as

$$P_{\text{dynamic}} = \alpha C_{\text{gate}} V_{DD}^2 f \tag{3.12}$$

Here, α is referred to as the switching factor for the gate.

The total power of the circuit can be computed by summing up Equation 3.12 over all gates in the circuit. For a circuit in which the final signal value settles to V_{DD}, as is the case for static CMOS logic, the above calculation is accurate; simple extensions are available for other logic circuits (such as pass transistor logic) where the signal value does not reach V_{DD} [17].

The value of α is dependent on the context of the gate in the circuit. For tree-like structures, this computation is straightforward, but for general circuits with reconvergent fanout, it is quite difficult to accurately calculate the switching factors [18]. Nevertheless, numerous heuristic approaches are available, and are widely used.

From the physical design point of view, usually the supply voltage V_{DD} is determined by the technology and the switching factor α is determined by the logic synthesis. Therefore, only gate capacitance C_{gate} can be optimized during the physical optimization phase. As shown in Equation 3.12, the smaller the overall gate capacitance, the smaller the dynamic power. The minimization of the overall gate size (while maintaining the necessary timing constraints) is actually the same objective

of many physical design algorithms. Therefore, an optimal solution of those algorithms is also the optimal solution in terms of dynamic power.

Another approach to reduce dynamic power is to avoid unnecessary toggling of the devices. This is possible for state-storage devices such as latches and flip-flops. A carefully designed clock gating scheme can greatly reduce dynamic power. However, usually these techniques are beyond the scope of physical design flows.

3.3.2 Short-Circuit Power

The mechanism of short-current power can be illustrated in the simple example shown in Figure 3.9. For INV1, when the falling transition occurs at the input, the PMOS device M1 switches from off to on, while the NMOS device M2 switches from on to off. Because of the intrinsic delays of the MOSFET devices as well as the loading effect of the gate capacitance of INV2, the switching cannot occur instantaneously. For a short period during the transition, both M1 and M2 are partially on, thus providing a direct path between the power supply and the ground. Certain amount of power is dissipated by this short-circuit current, which is also referred as the shoot-through current.

The short-circuit power has strong dependence on the capacitive load and the input signal transition time. Although accurate circuit simulation can be applied to calculate the short-circuit power, such an approach is prohibitively expensive. A more realistic approach is to estimate the short-circuit power via empirical equations. Some analysis techniques are proposed in Refs. [19,20]. However, according to many reports, the short-circuit current only accounts for between 5 and 10 percent of total power consumption in a well-designed circuit.

3.3.3 Static Power

In a digital circuit, MOSFET devices function as switches to realize certain logic functions. Ideally, we would like these switches to be completely off when the the controlling gate is off. However, MOSFET devices are far from ideal. Even when the circuit is not operating, the MOSFET devices are "leaking" current between terminals. Although each transistor only leaks a small amount of current, the overall full chip leakage can be substantial due to the sheer number of transistors.

There are two major components of leakage current: subthreshold leakage current and gate tunneling current [21]. These two components are illustrated in Figure 3.10. The subthreshold leakage current (I_1 in Figure 3.10) is the leakage current between the drain and source node when the device is in the off state (the voltage between the gate and source terminal is zero). Historically, in 0.25 μm and higher technology nodes, the subthreshold leakage was small enough to be negligible (several orders of magnitude smaller than the on-current). However, the traditional scaling requires the reduction of supply voltage V_{DD}, along with the reduction of the channel length. As a consequence, the threshold voltage must be scaled accordingly to maintain the driving capability of the MOSFET

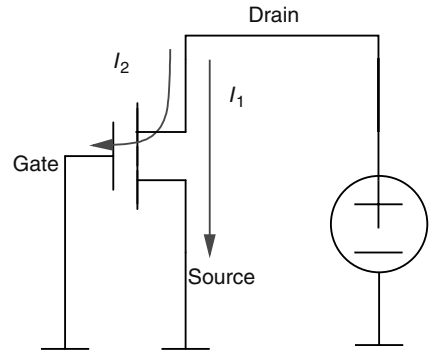


FIGURE 3.10 Two major components of the leakage current.

device. The smaller threshold voltage causes large increase of subthreshold leakage current, so that this is a significant factor in nanometer technologies.

The second component of the static power is the gate tunneling current (I_2 in Figure 3.10), which is also the consequence of scaling. As the device dimensions are reduced, the gate oxide thickness also has to be reduced. An unwanted consequence of thinner gate oxide thickness is the increased gate tunneling leakage current.

Many factors can affect the amount of subthreshold leakage current, including many device and environmental variables. An expression for the subthreshold leakage current density, i.e., the current per unit transistor area, is given by Ref. [22]:

$$J_{\text{sub}} = \frac{W}{L_{\text{eff}}}\mu\sqrt{\frac{q\epsilon_{\text{si}}N_{\text{cheff}}}{2\phi_s}}\upsilon_{\text{T}}^2 \exp\left(\frac{V_{\text{gs}} - V_{\text{th}}}{\eta\upsilon_{\text{T}}}\right)\left[1 - \exp\left(\frac{-V_{\text{ds}}}{\upsilon_{\text{T}}}\right)\right] \tag{3.13}$$

The details of the parameters in the above equation can be found in Ref. [22]. Here we would like to mention a few points:

- Term $\upsilon_{\text{T}} = kT/q$ is the thermal voltage, where k is the Boltzmann's constant, q is the electrical charge, and T is the junction temperature. From the equation, we can see that the leakage is an exponential function of the junction temperature T.
- Symbol V_{th} represents the threshold voltage. It can be shown that for a given technology, V_{th} is a function of the effective channel length L_{eff}. Therefore, subthreshold leakage is also an exponential function of effective channel length.
- Drain-to-source voltage, V_{ds}, is closely related to supply voltage V_{DD}, and has the same range in static CMOS circuits. Therefore, subthreshold leakage is an exponential function of the supply voltage.
- Threshold voltage V_{th} is also affected by the body bias V_{BS}. In a bulk CMOS technology, because the body node is always tied to ground for NMOS and V_{DD} for PMOS, the body bias conditions for stacked devices are different, depending on the location of the off device on a stack (e.g., top of the stack or bottom of the stack). As a result, the subthreshold leakage current can quite vary when different input vectors are applied to a gate with stacks.

For the gate tunneling current, a widely used model is the one provided in Ref. [23]:

$$J_{\text{tunnel}} = \frac{4\pi m^* q}{h^3}(kT)^2\left(1 + \frac{\gamma kT}{2\sqrt{E_B}}\right)\exp\left(\frac{E_{\text{F0,Si/SiO}_2}}{kT}\right)\exp\left(-\gamma\sqrt{E_B}\right) \tag{3.14}$$

where

T is the operating temperature

$E_{\text{F0,Si/SiO}_2}$ is the Fermi level at the Si/SiO_2 interface

m^* depends on the the underlying tunneling mechanism

Parameters k and q are defined as above, and h is Planck's constant: all of these are physical constants. The term $\gamma = 4\pi t_{\text{OX}}\sqrt{2m_{\text{OX}}}/h$, where t_{ox} is the oxide thicknes, and m_{ox} is the effective electron mass in the oxide. Besides physical constants and many technology-dependent parameters, it is quite clear that the gate-tunneling leakage depends on the gate oxide thickness and the operating temperature. The former is a strong dependence, but the latter is more complex: over normal ranges of operating temperature, the variations in gate leakage are roughly linear. In comparison with subthreshold leakage, which shows exponential changes with temperature, these gate leakage variations are often much lower. More details about this model can be found in Ref. [23].

One possible solution to mitigate the negative impact of gate current is to use material with higher dielectric constants (so-called high-k material) in junction with metal gates [24]. In many

current technologies, the gate leakage component is non-negligible. Recently, some progress has been reported on the development of high-k material. If successfully deployed, the new technology can reduce gate tunneling leakage by at least an order of magnitude, and at least postpone the point at which gate leakage becomes significant.

Owing to the power consumption limit dictated by the air-cooling technique widely accepted by the industry and market, power consumption, especially static power, has become a major design constraint. In addition to the advancements in manufacturing technology and material science, several circuit level power reduction techniques also have implications on the physical design flow. They include power gating, V_{th} (or effective channel length) assignment, input vector assignment, or any combination of these methods. More details on these topics can be found in Refs. [21,25–27].

3.4 TEMPERATURE

One of the primary effects of increased power dissipation is that it can lead to a higher on-chip operating temperature. High chip temperature is not only a performance issue but also a reliability and cost issue. High channel temperature affects MOSFET device performance by reducing the threshold voltage V_{th} and the mobility. If V_{DD} is unchanged, the lowered threshold voltage usually leads to larger driving current, while reduced mobility leads to smaller driving current. For a normal design with an increase of 100°C, the effect is dominated by mobility reduction, thus higher temperature leads to smaller overall driving capability [28], although inverse temperature dependence, where the speed of a gate increases with temperature, is also seen [29]. For interconnect networks, higher wire temperature will cause larger interconnect delay because metal has positive temperature coefficients. For example, for every 10°C increase, the resistivity of copper will increase by approximately 3 percent. On the reliability side, at elevated temperature, the metal molecules are more prone to electromigration, negative temperature bias instability (NBTI) [30–32], and time-dependent oxide breakdown (TDDB) [33]. Thus, temperature is always an important factor in reliability analysis. From a cost point of view, the cost of a heat sinking solution increases steeply with the total power dissipation of the chip. Air-cooled technologies are the cheapest option, but these can achieve only a certain level of cooling; beyond this level, all available options are substantially more expensive, and in today's commercial world, they are not viable for consumer products.

For many high-performance microprocessors, due to the large size of the die and large power dissipation, it is common to observe a temperature differential of 30°C–50°C between regions with high switching activity levels (e.g., a processor core) and those with low activity levels (e.g., memory). Potentially, these large spatial distributions can cause functional failures.

Before describing the flow of thermal analysis, we briefly describe how heat is dissipated from today's IC product. Figure 3.11 shows a highly simplified cross section of a typical IC product. Most

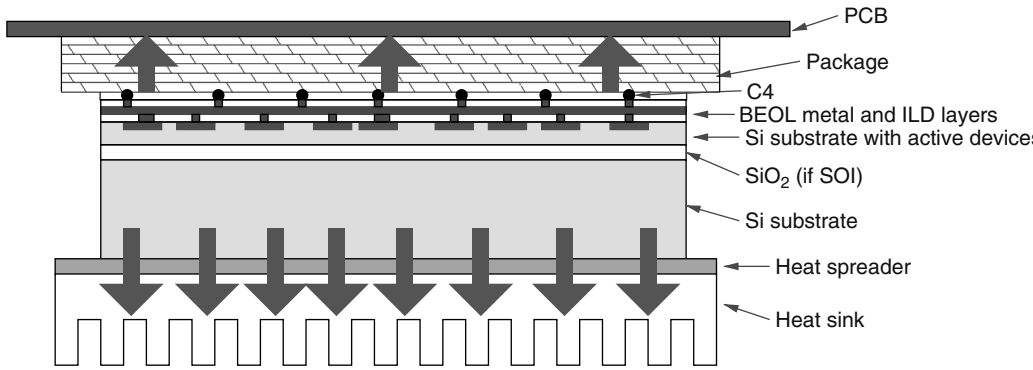


FIGURE 3.11 Simplified cross section to illustrate heat transfer from an IC chip.

high-performance IC designs use C4 technology for I/O and power delivery (versus the cheaper wire-bond technology that is used for lower performance parts). Hundreds to thousands of lead C4's are placed on top of the metal layer, and are connected to the printed circuit board (PCB) via the package. On the substrate side, a heat spreader is mounted next to the die, which is connected to a heat sink. The whole structure actually is "flipped" upside down so that a heat sink is on the top (thus C4 technology is also called flip-chip technology). The heat can be dissipated from both the heat sink side and the C4 side. However, because the heat sink has much smaller thermal resistivity, majority of the heat is dissipated from the heat sink.

There are three major mechanisms for heat transfer: conduction, convection, and radiation [34]. Convection occurs when heat is transferred by fluid movement (e.g., air or water). Radiation is the mechanism when the heat is transferred by photons of light in the spectrum. For modern IC products, convection and radiation only occur at interface of the heat sink, while almost all on-chip heat transfer is through conduction. The heat transfer at the heat sink interface is often described as a macromodel. For on-chip thermal analysis, cooling issues related to the heat sink are often decoupled from on-chip analysis by assuming it to be at the ambient temperature. Therefore, we only focus on conduction in this section.

The fundamental physics law governing heat conduction is the Fourier's law. If uniform material is assumed, it can be described as

$$\nabla^2 T(\mathbf{r}) + \frac{g(\mathbf{r})}{k_{\mathbf{r}}} = \frac{\rho c}{k_{\mathbf{r}}} \frac{\partial T}{\partial t}$$

where

k is the thermal conductivity at the particular location
ρ is the density of the material
c is the specific heat capacity
g is the volume power density, which is also location dependent

Usually the problem is formulated in three-dimensional space, therefore $\mathbf{r}$ is a three-dimensional array $\mathbf{r} = (x, y, z)$. Because the time constant of on-chip temperature change is usually in the order of milliseconds, while the operating frequency of electric signal is in the picoseconds range, it is often assumed that the thermal dissipation is a steady-state problem. Under this assumption, the heat diffusion equation can be simplified as

$$\nabla^2 T(\mathbf{r}) = -\frac{g(\mathbf{r})}{k_{\mathbf{r}}} \tag{3.15}$$

To solve the above three-dimensional thermal equation, appropriate boundary conditions need to be established. Because many layers of materials are involved and they all have different thermal conductivities, also due to the fact that power density distribution is uneven across the die, usually relatively fine spatial discretization is needed. Overall, it is difficult to solve the problem analytically. Instead a numerical method is applied.

Like other partial differential equations, the heat diffusion problem can be solved using the finite difference method [35,36], the finite element method, or the boundary element method [37]. A commonly used method is the finite difference method. Because $\nabla^2 T(\mathbf{r}) = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}$, if we discretized the space in 3D space, the term $\frac{\partial^2 T}{\partial x^2}$ can be approximated by

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i+1,j,k} - 2T_{i,j,k} + T_{i-1,j,k}}{\Delta x^2}$$

where i, j, and k are the indexes in the x, y, and t directions, respectively. After some algebraic manipulation, the steady-state thermal diffusion problem can be formulated into the matrix form:

$$\mathbf{GT} = \mathbf{P}$$

where
G is the thermal conductance matrix
unknown vector **T** is the steady-state temperature at all mesh points

Depending on the resolution required, the size of the problem can be quite large. The problem can be solved by applying a direct solver or using iterative techniques [35,38,39].

Like the finite difference method, the finite element method also results in a matrix of the type

$$K\mathbf{T} = \mathbf{P} \tag{3.16}$$

although in this case, the left-hand side coefficient matrix is denser (but still qualifies as a sparse matrix). The T variables here are node temperatures in the discretization, and the elements of K can be set up using element stamps. The finite element method essentially uses a polynomial fit within each grid cell, and the element stamps represent this fit. In the finite element parlance, the left-hand side matrix, K, is referred to as the global stiffness matrix. Stamps for boundary conditions can similarly be derived. Conductive boundary conditions simply correspond to fixed temperatures; because these parameters are no longer variables, they can be eliminated and the quantities moved to the right-hand side so that K is nonsingular.

As discussed earlier, a change in temperature will change the threshold voltage and mobility of a MOSFET device [28]. Usually, but not always, an elevated temperature causes the reduction of the overall driving strength of the MOSFET device. However, as the voltage supply V_{DD} gets close to 1-V range, the reduction of mobility may not offset the increase of driving capability due to the lowering of threshold voltage V_{th}. In other words, the higher temperature causes the transistors to have stronger driving capability, which in turn make the temperature increase further. Moreover, the subthreshold leakage increases exponentially with temperature, so that a small change in the temperature can result in a large change in the static power. When this happens, a positive feedback loop is formed between temperature and transistor driving capability. The issue is especially troublesome during "burn-in" testing, when the finished product is stress-tested under a higher supply voltage and an increased ambient temperature. During testing, the phenomenon is often referred as thermal runaway. Once this happens, the usual outcome is the complete destruction of the product. Fortunately, so far there have been no reports that thermal runaway happens for products operating under normal conditions, but nevertheless, thermal effects can cause parts to deviate from their prescribed power and timing specifications.

ACKNOWLEDGMENT

Part of Section 3.1.3 has been published in *Timing*, authored by Sachin Sapatnekar, by Kluwer Academic Publishers in 2004 [40]. (Used with kind permission of Springer Science and Business Media.)

REFERENCES

1. W. C. Elmore. The transient response of damped linear networks with particular regard to wideband amplifiers. *Journal of Applied Physics*, 19:55–63, January 1948.
2. J. Rubenstein, P. Penfield, and M. A. Horowitz. Signal delay in RC tree networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 202–211, July 1983.
3. R. Gupta, B. Tutuianu, and L. T. Pileggi. The Elmore delay as a bound for RC trees with generalized input signals. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(1):95–104, January 1997.

4. L. T. Pillage and R. A. Rohrer. Asymptotic wavelform evaluation for timing analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(4):352–366, April 1990.
5. R. Kay and L. Pileggi. PRIMO: Probability interpretation of moments for delay calculation. In *Proceedings of the ACM/IEEE Design Automation Conference*, San Francisco, CA, pp. 463–468, 1998.
6. F. Liu, C. V. Kashyap, and C. J. Alpert. A delay metric for RC circuits based on the Weibull distribution. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, San Jose, CA, pp. 620–624, 2002.
7. C. J. Alpert, F. Liu, C. V. Kashyap, and A. Devgan. Close-form delay and skew metrics made easy. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(12):1661–1669, December 2004.
8. T. Lin, E. Acar, and L. Pileggi. H-gamma: An RC delay metric based on a gamma distribution approximation of the homogeneous response. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, San Jose, CA, pp. 19–25, 1998.
9. K. L. Shepard, V. Narayanan, and R. Rose. Harmony: Static noise analysis of deep submicron digital integrated circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(8):1132–1150, August 1999.
10. P. Chen, D. A. Kirkpatrick, and K. Keutzer. Miller factor for gate-level coupling delay calculation. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, San Jose, CA, pp. 68–74, 2000.
11. A. Devgan. Efficient coupled noise estimation for on-chip interconnects. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, San Jose, CA, pp. 147–153, 1997.
12. M. Kuhlmann and S. S. Sapatnekar. Exact and efficient crosstalk estimation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(7):858–866, July 2001.
13. A. Vittal and M. Marek-Sadowska. Crosstalk reduction for VLSI. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(3):290–298, March 1997.
14. J. Cong, D. Z. Pan, and P. V. Srinivas. Improved crosstalk modeling for noise-constrained interconnect optimization. In *Proceedings of the Asia/South Pacific Design Automation Conference*, Yokohama, Japan, pp. 373–378, 2001.
15. L. He and K. M. Lepak. Simultaneous shield insertion and net ordering for capacitive and inductive coupling minimization. In *Proceedings of the ACM International Symposium on Physical Design*, San Diego, CA, pp. 55–60, 2000.
16. Y. Massoud, S. Majors, J. Kawa, T. Bustami, D. MacMillen, and J. White. Managing on-chip inductive effects. *IEEE Transactions on VLSI Systems*, 10(6):789–798, December 2002.
17. N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*, 2nd edn. Addison-Wesley, Reading, MA, 1993.
18. F. Najm. A survey of power estimation techniques in VLSI circuits. *IEEE Transactions on VLSI Systems*, 2(4):446–455, December 1994.
19. A. Hirata, H. Onodera, and K. Tamaru. Estimation of short-circuit power dissipation for static CMOS gates. *IEICE Transactions on Fundamentals of Electronics*, E00-A(1):304–311, January 1995.
20. K. Nose and T. Sakurai. Analysis and future trend of short-circuit power. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(9):1023–1030, September 2000.
21. K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand. Leakage current mechanisms and leakage reduction techniques in deep-micrometer CMOS circuits. *Proceedings of the IEEE*, 91(2):305–327, February 2003.
22. S. Mukhopadhyay, A. Raychowdury, K. Roy, and C. Kim. Accurate estimation of total leakage in nanometer-scale bulk CMOS circuits based on device geometry and doping profile. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(3):363–381, March 2005.
23. K. Bowman, L. Wang, X. Tang, and J. D. Meindl. A circuit-level perspective of the optimum gate oxide thickness. *IEEE Transactions on Electron Devices*, 48(8):1800–1810, August 2001.
24. B. H. Lee, L. Kang, W. J. Qi, R. Nieh, Y. Jeon, K. Onishi, and J. C. Lee. Ultrathin hafnium oxide with low leakage and excellent reliability for alternative gate dielectric application. In *Technical Digest of International Electron Devices Meeting (IEDM)*, Washington, D.C., pp. 133–136, 1999.
25. F. Gao and J. P. Hayes. Exact and heuristic approach to input vector control for leakage power reduction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(11):2564–2571, November 2006.

26. S. Mutoh et al. 1-V power supply high-speed digital circuit technology with multithreshold voltage CMOS. *IEEE Journal of Solid-State Circuits*, 30(8):847–854, August 1995.
27. D. Lee, D. Blaauw, and D. Sylvester. Static leakage reduction through simultaneous υ_t/t_{ox} and state assignment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(7):1014–1029, July 2005.
28. K. Kanda, K. Nose, H. Kawaguchi, and T. Sakurai. Design impact of positive temperature dependence on drain current in sub-1-V CMOS VLSIs. *IEEE Journal of Solid-State Circuits*, 36(10):1559–1564, October 2001.
29. V. Gerousis. Design and modeling challenges for 90 nm and 50 nm. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, San Jose, CA, pp. 353–360, 2003.
30. D. K. Schroder. Negative bias temperature instability: Road to cross in deep submicron silicon semiconductor manufacturing. *Journal of Applied Physics*, 94(1):1–18, July 2003.
31. M. A. Alam. A critical examination of the mechanics of dynamic NBTI for pMOSFETs. In *IEEE International Electronic Devices Meeting*, Washington, D.C., pp. 14.4.1–14.4.4, 2003.
32. S. V. Kumar, C. H. Kim, and S. S. Sapatnekar. An analytical model for negative bias temperature instability (NBTI). In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, San Jose, CA, pp. 493–496, 2006.
33. A. M. Yassine, H. E. Nariman, M. McBride, M. Uzer, and K. R. Olasupo. Time dependent breakdown of ultrathin gate oxide. *IEEE Transactions on Electron Devices*, 47(7):1416–1420, July 2000.
34. J. H. Lienhard and J. H. Lienhard. *A Heat Transfer Textbook*, 3rd edn. Phlogiston Press, Cambridge, MA, 2005.
35. Y. Cheng and S. M. Kang. A temperature-aware simulation environment for reliable ULSI chip design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(10):1211–1220, October 2000.
36. T. -Y. Wang and C. C. -P. Chen. 3-D thermal-ADI: A linear-time chip level transient thermal simulator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(12):1434–1445, December 2002.
37. Y. Zhan, B. Goplen, and S. Sapatnekar. Electrothermal analysis and optimization techniques for nanoscale integrated circuits. In *Proceedings of the Asia/South Pacific Design Automation Conference*, Yokohama, Japan, pp. 219–222, 2006.
38. H. Qian, S. Nassif, and S. Sapatnekar. Random walks in a supply network. In *Proceedings of the ACM/IEEE Design Automation Conference*, Anaheim, CA, pp. 93–98, 2003.
39. P. Li, L. T. Pileggi, M. Ashehi, and R. Chandra. IC thermal simulation and modeling via efficient multigrid-based approaches. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(9):1763–1776, September 2006.
40. S. Sapatnekar, *Timing*, Kluwer Academic Publishers, Boston, MA, 2004.

Part II

Foundations

4 Basic Data Structures

Dinesh P. Mehta and Hai Zhou

CONTENTS

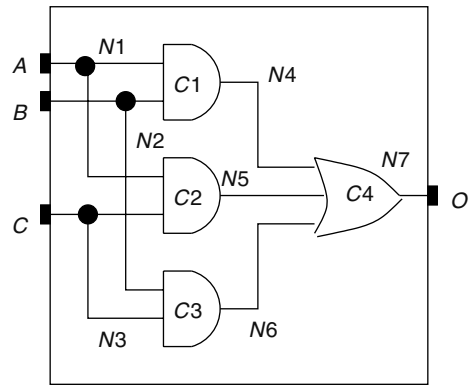
4.1 INTRODUCTION

Physical design automation may be viewed as the process of converting a circuit into a geometric layout. We distinguish between three categories of data structures for the purpose of organizing this chapter:

1. Data structures used to represent the input to physical design: the circuit or the netlist
2. Data structures used during the physical design process
3. Data structures used to represent the output of physical design: the layout

4.2 INPUT DATA STRUCTURES

A circuit consists of components and their interconnections. Each component contains logic that implements some functionality. It also has pins (or terminals) with which it communicates with other components. The entire circuit also needs to be able to communicate with the rest of the world and does so through the use of external pins. An interconnection connects (or makes electrically



Net 1: (*A*, *C*1.in1, *C*2.in1)
Net 2: (*B*, *C*1.in2, *C*3.in1)
Net 3: (*C*, *C*2.in2, *C*3.in2)
Net 4: (*C*1.out, *C*4.in1)
Net 5: (*C*2.out, *C*4.in2)
Net 6: (*C*3.out, *C*4.in3)
Net 7: (*C*4.out, *O*)

FIGURE 4.1 Circuit and its netlist.

equivalent) a set of two or more pins. These pins may be associated with the components or may be external pins. Each interconnection is called a net. The circuit is described by a list of all nets, the netlist. Figure 4.1 shows a simple example, where the components are simple logic gates. Components do not necessarily have to be logic gates. A component could be more complex. For example, it could be a multiplier that was manually designed or designed by some other tool. The chip corresponding to a circuit can itself be a component in a larger circuit.

The mathematical structure that comes closest to representing a circuit is the hypergraph. A hypergraph consists of a set of vertices and a set of hyperedges, where each hyperedge connects a set of $k \geq 2$ vertices. (When $k = 2$ for each edge, the hypergraph reduces to the more familiar graph.) A hypergraph approximates a circuit in that each vertex is mapped to a component and each hyperedge corresponds to a net. Even so, the hypergraph is not a complete representation of a circuit:

1. Components may have associated physical attributes. For example, if the component is a rectangle, its height and width will be provided; locations of pins on the rectangle may also be provided.
2. Nets have an associated direction, which play a role during routing. Consider Net 1 in Figure 4.1 that interconnects three terminals. Pin *A* is the source of the signal and *C*1.in1 and *C*2.in1 are the sinks.
3. Nets connect pins, but hyperedges connect components. You could fix this by having vertices model pins rather than components, but then you lose the property that some pins are associated with a single component. If this component is moved, all of its pins must move with it.

The number of mathematical and algorithmic tools available for hypergraphs is small relative to that for graphs. So, it is unlikely that there is much to be gained even if the hypergraph was a complete representation. As a result, a netlist is sometimes represented by a graph. This is not unreasonable because it turns out that the vast majority of nets are indeed two-terminal nets. There is no well-defined way to convert a net with more than two terminals into one or more graph edges. One approach is to add an edge between every pair of terminals in the net. A netlist converted into a graph is often represented by a connectivity matrix. A matrix element in position $[i][j]$ denotes the number of nets that connect modules i and j.*

The netlist of Figure 4.1 is a complete description of a circuit. It may be read from a circuit file, parsed and used to populate an internal data structure. This internal data structure is the starting point of the physical design process. How should this internal data structure be organized? It

* This is actually a multigraph and not a graph because many edges are permitted between a pair of vertices.

seems obvious that at a minimum, the data structure should consist of a list of nets, where each net object contains a list of pins. Should there also be a list of components where each component object also contains a list of pins? Should each component contain a list of nets that are incident on it? Is it necessary to instantiate a pin object? If so, should it contain pointers to the component and net to which it belongs? The answer to these questions depend on what kinds of queries will be posed to the data structure by the particular physical design (PD) tool. One size does not fit all.

4.3 DATA STRUCTURES USED DURING PD

There are too many data structures in this category to describe in this chapter. Fortunately, the vast majority of these are traditional data structures such as arrays, linked lists, search trees, hash tables, and graphs. We do not discuss these structures as they are typically covered in an undergraduate data structures text (e.g., Ref. [1]). Graph algorithms are covered in Chapter 5. Below, we sample some advanced data structures that have either been specifically designed with PD applications in mind or have found widespread application in PD.

4.3.1 Floorplanning Data Structures

Several innovative data structures (representations) have been developed for floorplanning. We defer a discussion of these data structures to the floorplanning section of the handbook, where they are discussed in considerable detail (see Chapters 9 through 11).

4.3.2 Geometric Data Structures

Each stage of physical design automation has a significant geometric aspect, with the possible exception of partitioning that is more of a graph-theoretic problem. The computational geometry literature [2] describes a number of geometric data structures. The benefit of using geometric data structures is that a query has a better time complexity than it would on a simple data structure such as an array or a linked list. Implementing geometric data structures can be time consuming, but they may be found in algorithmic or geometric libraries [3,4]. A practitioner should weigh their benefits against the simplicity of arrays and linked lists. Examples of geometric data structures include interval trees, range trees, segment trees, kd trees, and priority search trees. Voronoi diagrams and Delaunay triangulations may also be viewed as geometric data structures. Some of these structures can be extended to higher dimensions although this comes at the cost of simplicity and time complexity. Two or three dimensions are usually sufficient for physical design applications. These data structures are often used in conjunction with the planesweep algorithm technique. Describing all of these data structures is beyond the scope of this chapter. Instead, we pick two, the interval tree and kd tree, and describe these briefly to give the reader a flavor of how they work.

4.3.2.1 Interval Trees

Most physical designs can be represented as a set of axis-parallel rectangles. The boundaries of these rectangles can be viewed as intervals. One common operation needed on these intervals is to find a subset of them that intersect with a perpendicular line. If such a query only happens a limited number of times, it can be efficiently processed by a sweep-line algorithm in $O(n \log n)$ time. However, when such queries need to be done repeatedly, it is better to preprocess the intervals and store them in a data structure that can answer the queries more efficiently. The interval tree is a structure that can be built in $O(n \log n)$ time and then answers the query in $O(\log n + k)$ time, where k is the number of intervals intersecting the perpendicular line.

Even though an interval lies on a line that is a one-dimensional space, it is actually a two-dimensional datum because it has two independent parameters. An interval starting at a and ending at b is represented by $[a, b]$. It is not possible to have a total order over the set of intervals. The idea of

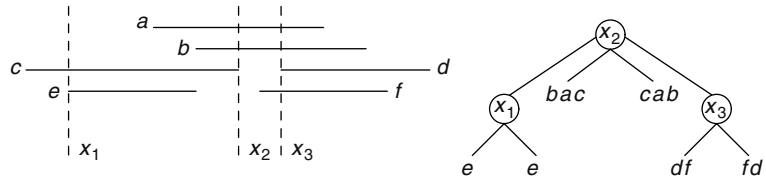


FIGURE 4.2 Set of intervals and its interval tree.

the interval tree is to partition the set of intervals into three groups based on a given point x: intervals to the left of the point $L(x)$, intervals to the right of the point $R(x)$, and intervals overlapping with the point $C(x)$. The subsets $L(x)$ and $R(x)$ of intervals can be recursively represented. The subset $C(x)$ also needs to be organized for the queries. Even though $C(x)$ could include all the intervals in the original set, organizing them is much simpler: they can be ordered both on their left points and on their right points. If the query point $q < x$, only the left points of $C(x)$ need to be checked in increasing order; if $q > x$, only the right points of $C(x)$ need to be checked in decreasing order. To balance $L(x)$ and $R(x)$, thus to have a short tree, it is desired to use the median of all the endpoints as x. Figure 4.2 shows an interval tree for a set of intervals, where the intervals in $C(x)$ are organized in two lists according to their left and right points.

The following result can be easily proved based on the above discussion.

Theorem 1 *For a given set of n intervals, an interval tree can be constructed in $O(n \log n)$ time; with it, a query on the intervals containing a given point can be answered in $O(\log n + k)$ time, where k is the number of covering intervals.*

Applications of interval trees may be found in Refs. [5–7].

4.3.2.2 kd Trees

The query facilitated by a kd tree can be viewed as the reverse of that by an interval tree. In one dimension, a set of points are given and a query by an interval wants to find all the points in it. If the queries happen a limited number of times, they can be efficiently processed by linear scans of the points in $O(n)$ time. When queries need to be done frequently, a sorted array or a binary tree can be built by preprocessing, and a query can be done in $O(\log n + k)$ time where k is the number of points on the interval.

A kd tree is simply an extension of this binary tree to higher dimension space. It first partitions all the points into two groups of almost the same size along one dimension, and then recursively partitions the groups along other dimensions. It follows the same order of dimensions for further partitionings. Figure 4.3 shows a kd tree for a set of points on a plane (two-dimensional space)

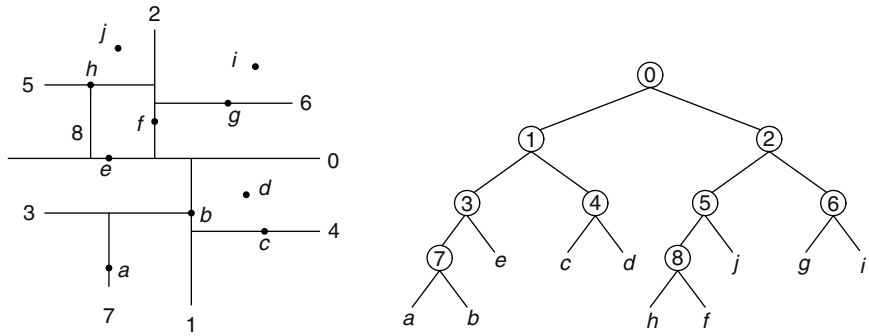


FIGURE 4.3 Set of points on the plane and its kd tree.

```
Algorithm KdTreeQuery(v, R)
if v is a leaf
    then output the point if it is in R
    else {
         if left(v) is fully contained in R
              then output points in left(v)
              else if left(v) intersects R
                   then KdTreeQuery (left(v),R)
         // similar code for right (v) omitted
    }
```

FIGURE 4.4 Range query algorithm on a kd tree.

with a horizontal partitioning followed by a vertical one. The algorithm for building a kd tree is straightforward, based on recursive bipartitioning of the points along one dimension. Its runtime is in $O(n \log n)$. Given an orthogonal range, a query on a kd tree will give all the points within the range. The range query algorithm is just a simple extension of the interval query on binary trees and it is described in Figure 4.4.

Theorem 2 *A kd tree for n points can be built in $O(n \log n)$ time; a query with an axis-parallel range can be performed in $O(n^{1-1/d} + k)$ where $d > 1$ is the dimension and k is the number of points within the range. In a two-dimensional plane, a query takes $O(\sqrt{n} + k)$ time.*

An application of the kd tree may be found in Ref. [8].

4.3.3 Spanning Graphs: A Global Routing Data Structure

Given a set of n points in a plane, a spanning tree is a set of edges that connects all n points and contains no cycles. When each edge is weighted using some distance metric, the minimum spanning tree is a spanning tree whose sum of edge weights is minimum. If Euclidean distance (L_2) is used, it is called the Euclidean minimum spanning tree; if rectilinear distance (L_1) is used, it is called the rectilinear minimum spanning tree (RMST). The RMST is often used as a starting point for constructing a Steiner tree, which is used extensively in global routing (see Chapter 24).

The usual approach for constructing a minimum spanning tree is to first define a complete weighted graph on the set of points and then to construct a spanning tree on it, for example, by running Kruskal's algorithm (see Chapter 5). Given a set of points V, an undirected graph $G = (V, E)$ is called a spanning graph if it contains a minimum spanning tree. The cardinality of a graph is its number of edges. The complete graph has a cardinality of $\Theta(n^2)$, which is expensive. For the L_2 metric, the Delaunay triangulation, a spanning graph of cardinality $O(n)$, can be constructed in $\Theta(n \log n)$ time. However, this approach does not work for the L_1 metric as the Delaunay triangulation may be degenerate. Zhou et al. [9] describe a rectilinear spanning graph of cardinality $O(n)$ that can be constructed in $O(n \log n)$ time [9]. Its use in the construction of a Steiner tree is described in Ref. [10]. We sketch the salient features of this data structure below.

Minimum spanning tree algorithms use two properties to infer the inclusion and exclusion of edges in a minimum spanning tree:

1. Cut property states that an edge of smallest weight crossing any partition of the vertex set into two parts belongs to a minimum spanning tree.
2. Cycle property states that an edge with largest weight in any cycle in the graph can be safely deleted.

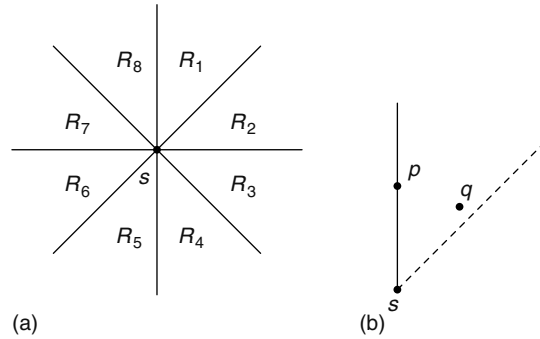


FIGURE 4.5 Octal partition of the plane.

Define the octal partition of the plane with respect to s as the partition induced by the two rectilinear lines and the two 45° lines through s, as shown in Figure 4.5a. Here, each of the regions R_1 through R_8 includes only one of its two bounding half line as shown in Figure 4.5b.

Lemma 1 *Given a point s in the plane, each region R_i, $1 \leq i \leq 8$, of the octal partition has the property that for every pair of points $p, q \in R_i$, $\|pq\| < \max(\|sp\|, \|sq\|)$.*

Here $\|sp\|$ is the L_1-distance between s and p. Consider the cycle on points s, p, and q and suppose $\|sp\| < \|sq\|$. From the cycle property, edge sq can safely be excluded from the spanning graph. This can be extended to excluding edges from s to all points in R_1, except for the nearest one.

A property of the L_1-metric is that the contour of equidistant points from s forms a line segment in each region. In regions R_1, R_2, R_5, and R_6, these segments are captured by an equation of the form $x + y = c$; in regions R_3, R_4, R_7, and R_8, they are described by the form $x - y = c$. This property is used to devise a planesweep algorithm to construct the spanning graph. For each point s, we need to find its nearest neighbor in each octant. We illustrate how to efficiently compute the nearest neighbor in R_1 for each point. Other octants are similarly processed. For the R_1 octant, a sweep line is moved along all points in increasing order of $x + y$. During the sweep, we maintain an active set consisting of points whose nearest neighbors in R_1 are yet to be discovered. When a point p is processed, we identify all points in the active set that have p in their R_1 regions. Suppose s is such a point in the active set. Because points are scanned in increasing $x + y$, p must be the nearest point to s in R_1. Therefore, we add edge sp to the spanning graph and delete s from the active set. After processing these active points, we also add p to the active set. Each point is added and deleted at most once from the active set. The runtime for the sweep is $O(n \log n)$. Each point s has an edge to its nearest neighbor in each octant. This gives a spanning graph of cardinality $\Theta(n)$.

4.3.4 Max-Plus Lists

Max-plus lists are applicable to slicing floorplans [11], technology mapping [12], and buffer insertion [13] problems. Consider a list where each item consists of a pair of elements (m, p). Each item represents a possible solution to an optimization problem that seeks to minimize both m and p (e.g., m and p could represent the height and width of a chip). Solution j is said to be redundant with respect to solution i if $i.m \leq j \cdot m$ and $i \cdot p \leq j \cdot p$ because it is no better than i on either attribute. Consider a list of three solutions: $S_1 = (5, 4)$, $S_2 = (4, 6)$, and $S_3 = (5, 5)$. S_3 is redundant wrt S_1. Neither S_1 nor S_2 is redundant wrt any of the other solutions. Redundant elements are discarded from the list.

Consider an ordered list $A = [(A_1 \cdot m, A_1 \cdot p), \ldots, (A_q \cdot m, A_q \cdot p)]$ such that $A_i \cdot m > A_j \cdot m \wedge A_i \cdot p < A_j \cdot p$ for any $i < j$. Such an ordering of solutions is always possible if redundant solutions are not present in the list. Our example list of three elements above can be rewritten as [(5,4), (4,6)].

These lists arise in the context of dynamic programming, which tries to find an optimal solution to a problem by first finding optimal solutions to subproblems and then merging them to find an optimal solution to the larger problem. Each list represents possible optimal solutions to a subproblem. Merging them gives us a list of possible optimal solutions to the bigger problem.

We next define the list merge. Given two ordered lists A and B as defined above with q and r elements, respectively, compute another list C such that each element c of C is obtained by combining an element a of A with an element b of B using the max-plus operation as follows:

$$c.m = \max(a \cdot m, a \cdot m)$$
$$c.p = a \cdot p + b \cdot p$$

Redundant solutions are not permitted in C. Thus, C only contains the irredundant combinations among the qr possible combinations of elements in A and B. Let the size of C be s.

To illustrate the rationale for the max-plus operation to combine elements, consider two rectangles with dimensions $h_1 \times w_1$ and $h_2 \times w_2$. Suppose one rectangle is stacked on top of the other and we wish to determine the dimensions of the smallest bounding box that encloses both rectangles. The height of this bounding box is the sum of the heights of the two rectangles while its width is the maximum of the two rectangle widths; that is, the max plus operation. In buffer insertion, the two quantities are delay (maximum operation) and downstream capacitance (plus operation).

Stockmeyer [11] proposed an algorithm to perform the list merge in time $O(q + r)$. However, when the merge tree is skewed, it takes r^2 time to combine all the lists even though the total number of items in C is r. Stockmeyer's algorithm is inefficient when the two lists have very different lengths. An extreme case is when a single item is being merged with a big list. In this case, the algorithm reduces to a linear time search to find the location of an element in a sorted list. Balanced binary search trees [14] were used to represent each list so that a search can be done in $O(\log r)$ time. In addition, to avoid updating the p values individually, the update was annotated on a node for the rooted subtree. Shi's algorithm is faster when the merge tree is skewed, with $O(r \log r)$ time relative to Stockmeyer's $O(r^2)$ time. However, Shi's algorithm is complicated and much slower when the merge tree is balanced.

To summarize, the merge of two candidate lists using balanced binary search trees can only speed up the merge of two candidate lists of very different lengths (unbalanced situation), but not the merge of two candidate lists of similar lengths (balanced situation).

Figure 4.6 illustrates the best data structure for maintaining solutions in each of the two extreme cases: the balanced situation requires a linked list that can be viewed as a totally skewed tree; the unbalanced situation requires a balanced binary tree. However, most cases in reality are between these extremes, where neither data structure is the best. The max-plus list is an efficient data structure for the merge operation [15]. As shown in Figure 4.6, it can adapt to the structure of the merge tree: it becomes a linked list in balanced situations and behaves like a balanced binary tree in unbalanced situations. The merge algorithm based on max-plus list has the same asymptotic time complexity as that used in Refs. [14,16] but is easier to implement and more efficient in practice [15].

The max-plus list is based on the skip list [17]. Because a max-plus list is similar to a linked list, its merge operation is just a simple extension of Stockmeyer's algorithm. During each iteration of Stockmeyer's algorithm, the current item with the maximal m value in one list is finished, and the

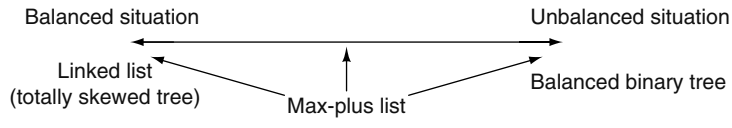


FIGURE 4.6 Flexibility of max-plus list.

new item is equal to the finished item with its p value incremented by the p value of the other current item. The idea of the max-plus list is to finish a sublist of more than one item at one iteration. Assume that $A_i \cdot m > B_j \cdot m$, we want to find a $i \leq k \leq a$ such that $A_k \cdot m \geq B_j \cdot m$ but $A_{k+1} \cdot m < B_j \cdot m$. These items $A_i, \ldots, A_k$ are finished and put into the new list after their p values are incremented by $B_j \cdot p$. The speedup over Stockmeyer's algorithm comes from the fact that this sublist is processed (identified and updated) in a batch mode instead of item by item. The forward pointers in a max-plus list are used to skip items when searching for the sublist, and an adjust field is associated with each forward pointer to record the incremental amount on the skipped items. Each item is defined by the following C code:

```
struct maxplus_item{
   int level; /* the level*/
   float m, p; /* the two values*/
   float *adjust;
   struct maxplus_item **forward; /*forward pointers*/
}
```

The size of adjust array is equal to the level of this item, and adjust[i] means that the p values of all the items jumped over by forward[i] should add a value of adjust[i].

Two skip lists with sizes q and $r (q \leq r)$ can be merged in $O(q + q \log r/q)$ expected time [18]. This quantity is proportional to the number of jump operations performed on the skip list. Max-plus lists are merged in a similar manner, except that the adjust field need to be updated. The complexity is also proportional to the number of jump operations. However, it can be shown that the number of jump operations in a maxplus merge is within a constant facor of the number of jumps in an ordinary skip list. Thus, the expected complexity of a max-plus merge is identical to that of a skip-list merge, which is the same as that of a balanced binary search tree.

4.4 LAYOUT DATA STRUCTURES

Transistors and logic gates are manufactured in layers on silicon wafers. Silicon's conductivity can be significantly improved by diffusing n- and p-type dopants into it. This layer of the chip is called the diffusion (diff) layer. The source and drain of a transistor are formed by separating two n-type regions with a p-type region (or vice versa) and its gate is formed by sandwiching a silicon dioxide (an insulator) layer between the p-type region and a layer of polycrystalline silicon (a conductor). Because polycrystalline silicon (poly) is a conductor, it is also used for short interconnections (wires). Although poly conducts electricity, it is not sufficient to complete all the interconnections in one layer. Modern chips usually have several layers of aluminum (metal), a conductor, separated from each other by insulators on top of the poly layer. These make it possible for the gates to be interconnected as specified in the design. Note that a layer of material X (e.g., poly) does not mean that there is a monolithic slab of poly over the entire chip area. The poly is only deposited where gates or wires are needed. The remaining areas are filled with insulating materials and for our purposes may be viewed as being empty. In addition to the layers as described above, it is necessary to have a mechanism for signals to pass between layers. This is achieved by contacts (to connect poly with diffusion or metal) and vias (to connect metal on different layers).

A layout data structure stores and manipulates the rectangles on each layer. Some important high-level operations that a layout data structure must support are design-rule checking, layout compaction, and parasitic extraction. Design rules specify geometric constraints on the layout so that the patterns on the processed wafer preserve the topology of the designs. An example of a design rule is that the width of a wire must be greater than a specified minimum. If this constraint is violated, it is possible that for the wire to be discontinuous because of errors in the fabrication process. Additional design rules for CMOS technology may be found in Ref. [19, p. 142]. Capacitance, resistance, and

inductance are commonly referred to as parasitics. After a layout has been created, the parasitics must be computed to verify that the circuit will meet its performance goals. The parasitics are computed from the geometry of the layout. For example, the resistance of a rectangular slab of metal is $\frac{\rho l}{tw}$, where ρ is the resistivity of the metal and l, w, and t are the slab's length, width, and thickness, respectively. See Ref. [19, Chapter 4] for more examples. Compaction tries to make the layout as small as possible without violating any design rules. Reducing chip area dramatically reduces cost per chip. (The cost of a chip can grow as a power of five of its area [20].) Two-dimensional compaction is NP-hard, but one-dimensional compaction can be carried out in polynomial time. Heuristics for two-dimensional compaction often iteratively interleave one-dimensional compactions in the x- and y-directions. For more details, see Ref. [21].

4.4.1 Corner Stitching

In a layout editor, a user manually designs the layout, by inserting rectangles of the appropriate dimensions at the appropriate layer. The MAGIC system [22] developed at U.C. Berkeley includes a layout editor. The corner-stitching data structure was proposed by Ousterhout [23] to store nonoverlapping rectilinear circuit components in MAGIC. The data structure is obtained by partitioning the layout area into horizontally maximal rectangular tiles. There are two types of tiles: solid and vacant, both of which are explicitly stored in the corner-stitching data structure. Tiles are obtained by extending horizontal lines from corners of all solid tiles until another solid tile or a boundary of the layout region is encountered. The set of solid and vacant tiles so obtained is unique for a given input. The partitioning scheme ensures that no two vacant or solid tiles share a vertical side. Each tile T is stored as a node that contains the coordinates of its bottom left corner, x_1 and y_1, and four pointers N, E, W, and S. N (respectively, E, W, S) points to the rightmost (respectively, topmost, bottommost, leftmost) tile neighboring its north (respectively, east, west, south) boundary. The x and y coordinates of the top right corner of T are $T.E \rightarrow x_1$ and $T.N \rightarrow y_1$, respectively, and are easily obtained in $O(1)$ time. Figure 4.7 illustrates the corner-stitching data structure.

Corner stitching supports a rich set of operations. These include simple geometric operations like insertion and deletion of rectangles, point finding (search for the tile containing a specified point), neighbor finding (find all tiles that abut a given tile), area searches (do any solid tiles intersect a given rectangular area?), and area enumeration (enumerate all tiles that intersect a given rectangular area). It also supports more sophisticated operations like plowing (move a large piece of a design in a specified direction) and one-dimensional compaction. We describe the point-find operation below to provide the reader with a flavor of the corner-stitching data structure (Figure 4.8). Given a pointer to an arbitrary tile T in the layout, the algorithm seeks the tile in the layout containing the point P.

Figure 4.9 illustrates the execution of the point-find operation on a pathological example. From the start tile T, the while loop of line 5 follows north pointers until tile A is reached. We change

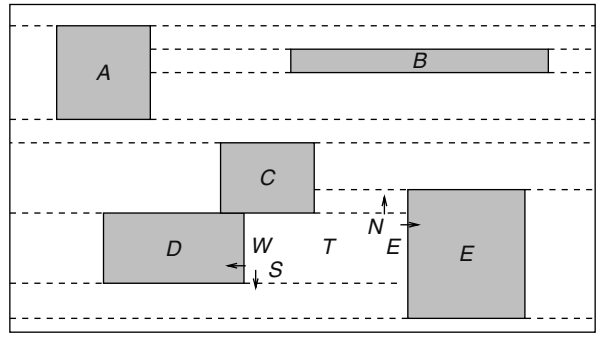


FIGURE 4.7 Corner stitching data structure. Pointers (stitches) are shown for tile T.

```
Algorithm Tile Point_Find (Tile T, Point P)
1. current = T;
2. while (P is not contained in current)
3.   while(P.y does not lie in current's y-range)
4.      if (P.y is above current) current = current → N;
5.      else current = current → S;
6.   while (P.x does not lie in current's x-range)
7.      if (P.x is to the right of current) current = current → E;
8.      else current = current → W;
9. return (current);
```

FIGURE 4.8 Point find in corner stitching.

directions at tile A because its y-range contains P. Next, west pointers are followed until tile F is reached (whose x-range contains P). Notice that the sequence of west moves causes the algorithm to descend in the layout resulting in a vertical position that is similar to that of the start tile. As a result of this misalignment, the outer while loop of the algorithm must execute repeatedly until the point is found (note that the point will eventually be found because the point-find algorithm is guaranteed to converge). *Point_Find* has a worst case complexity is $O(n)$ and its average complexity is $O(\sqrt{n})$. In comparison, a tree-type data structure has an average case complexity of $O(\log n)$. The slow speed may be tolerable in an interactive environment and may be somewhat ameliorated in that it could often take $O(1)$ time because of locality of reference (i.e., two successive points searched for by a user are likely to be near each other requiring fewer steps of the point-find algorithm).

The space requirements of corner stitching must take into account the number of vacant tiles. Mehta [24] shows that the number of vacant tiles is $3n + 1 - k$, where n is the number of solid tiles and k is a quantity that depends on the geometric locations of the tiles.

Expanded rectangles [25] expands solid tiles in the corner-stitching data structure so that each tile contains solid material and the empty space around it. No extra tiles are needed to represent empty space. Marple et al. [26] developed a layout system called tailor that was similar to MAGIC except that it allowed 45° layout. Thus, rectangular tiles are replaced by trapezoidal tiles. Séquin

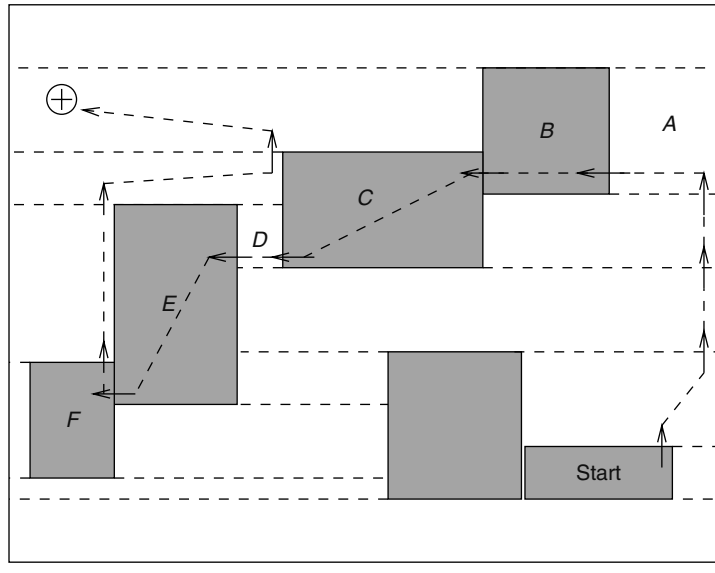


FIGURE 4.9 Illustration of point find operation and misalignment.

and Façanha [27] proposed two generalizations to geometries including circles and arbitrary curved shapes, which arise in microelectromechanical systems. As with its corner-stitching-based predecessors, the layout area is decomposed in a horizontally maximal fashion into tiles. Consequently, tiles have upper and lower horizontal sides. Their left and right sides are represented by parameterized cubic Bezier curves or by composite paths composed of linear, circular, and spline segments. Mehta and Blust [28] extended Ousterhout's corner-stitching data structure to directly represent L-shape and other simple rectilinear shapes without partitioning them into rectangles. This results in a data structure that is topologically different from the other versions of corner stitching described above.

Because, in practice, circuit components can be arbitrary rectilinear polygons, it is necessary to partition them into rectangles to enable them to be stored in the corner-stitching format. MAGIC handles this by using horizontal lines to partition the polygons. Nahar and Sahni [29] studied this problem and presented an algorithm to decompose a polygon that outperforms the standard planesweep algorithm. Lopez and Mehta [30] presented algorithms for the problem of breaking an arbitrary rectilinear polygon into *L*-shapes using horizontal cuts to optimize its memory requirements.

Corner stitching requires rectangles to be non-overlapping. So, an instance of the corner-stitching data structure can only be used for a single layer. However, corner stitching can be used to store multiple layers in the following way. Consider two layers A and B. Superimpose the two layers. This can be thought of as a single layer with four types of rectangles: vacant rectangles, type A rectangles, type B rectangles, and type AB rectangles. Unfortunately, this could greatly increase the number of rectangles to be stored. It also makes it harder to perform insertion and deletion operations. Thus, in MAGIC, the layout is represented by a number of single-layer corner-stitching instances and a few multiple-layer instances when the intersection between rectangles in different layers is meaningful, for example, transistors are formed by the intersection of poly and diffusion rectangles.

4.4.2 Quad Trees and Variants

In contrast to layout editors, industrial layout verification benefits from a more automated approach. This is better supported by hierarchical structures such as the quad tree. The underlying principle of the quad tree is to recursively subdivide the two-dimensional layout area into four quads until a stopping criterion is satisfied. The resulting structure is represented by a tree with a node corresponding to each quad, with the entire layout area represented by the root. A node contains children pointers to the four nodes corresponding the quads formed by the subdivision of the node's quad. Quads that are not further subdivided are represented by leaves in the quad tree.

Ideally, each rectangle is the sole occupant of a leaf node. In general, of course, a rectangle does not fit inside any leaf quad, but rather intersects two or more leaf quads. To state this differently, it may intersect one or more of the horizontal and vertical lines (called bisectors) used to subdivide the layout region into quads. Three strategies have been considered in the literature as to where in the quad tree these rectangles should be stored. These strategies, which have given rise to a number of quad tree variants, are listed below and are illustrated in Figure 4.10:

1. Smallest: Store a rectangle in the smallest quad (not necessarily a leaf quad) that contains it. Such a quad is guaranteed to exist because each rectangle must be contained in the root quad.
2. Single: Store a rectangle in precisely one of the leaf quads that it intersects.
3. Multiple: Store a rectangle in all of the leaf quads that it intersects.

Obviously, if there is only one rectangle in a quad, there is no need to further subdivide the quad. However, this is an impractical (and sometimes impossible) stopping criterion. Most of the quad-tree variants discussed below have auxiliary stopping criteria. Some subdivide a quad until it reaches a specified size related to the typical size of a small rectangle. Others stop if the number of rectangles in a quad is less than some threshold value. Figure 4.11 lists and classifies the quad-tree variants.

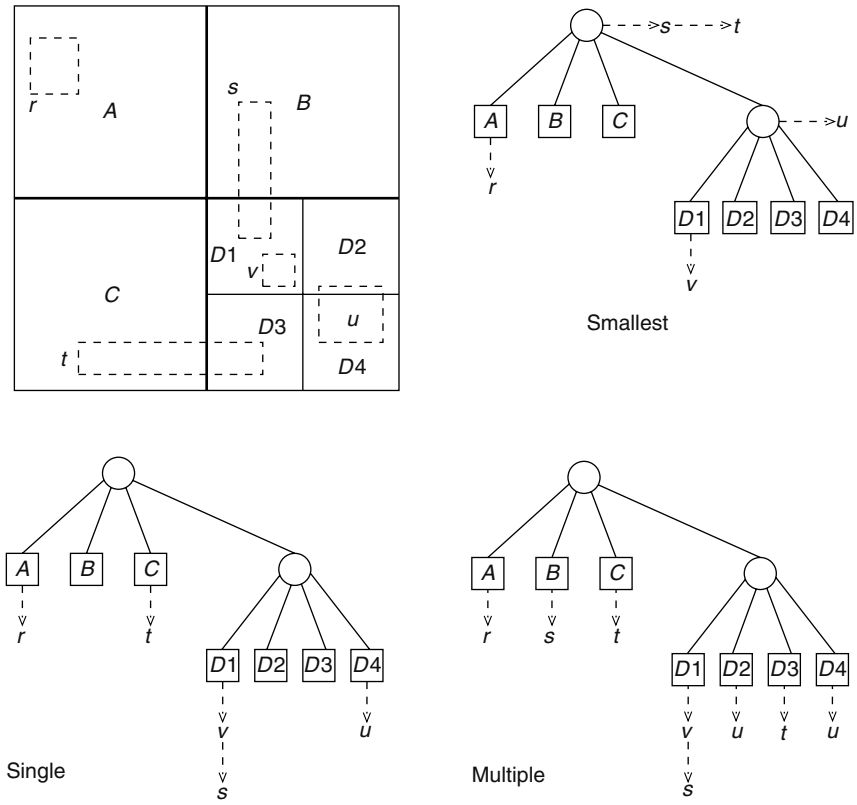


FIGURE 4.10 Quad-tree variations.

4.4.2.1 Bisector List Quad Trees

Bisector list quad trees (BLQT) [31], which was the first quad-tree structure proposed for VLSI layouts, used the smallest strategy. Here, a rectangle is associated with the smallest quad (leaf or nonleaf) that contains it. Any nonleaf quad Q is subdivided into four quads by a vertical bisector and a horizontal bisector. Any rectangle associated with this quad must intersect one or both of the bisectors (otherwise, it is contained in one of Q's children, and should not be associated with Q). The set of rectangles are partitioned into two sets: V, which consists of rectangles that intersect the vertical bisector, and H, which consists of rectangles that intersect the horizontal bisector. Rectangles

Author	Abbreviation	Year of Publication	Strategy
Kedem	BLQT	1982	Smallest
Rosenberg	kd	1985	N/A
Brown	MSQT	1986	Multiple
Weyten et al.	QLQT	1989	Multiple
Pitaksanonkul et al.	BQT	1989	Single
Lai et al.	HV	1993	Smallest
Lai et al.	HQT	1996	Multiple

FIGURE 4.11 Summary of quad-tree variants.

that intersect both bisectors are arbitrarily assigned to one of V and H. These lists were actually implemented using binary trees. The rationale was that because most rectangles in integrated circuit (IC) layouts were small and uniformly distributed, most rectangles will be at leaf quads. A region-search operation identifies all the quads that intersect a query window and checks all the rectangles in each of these quads for intersection with the query window.

4.4.2.2 kd Trees

Rosenberg [32] compared BLQT with kd trees and showed experimentally that kd trees outperformed an implementation of BLQT. Rosenberg's implementation of the BLQT differs from the original in that linked lists rather than binary trees were used to represent bisector lists. It is hard to evaluate the impact of this on the experimental results, which showed that point-find and region-search queries visit fewer nodes when the kd tree is used instead of BLQT. The experiments also show that kd trees consume about 60–80 percent more space than BLQTs.

4.4.2.3 Multiple Storage Quad Trees

In 1986, Brown proposed a variation [33] called multiple storage quad trees (MSQT). Each rectangle is stored in every leaf quad it intersects. (See the quad tree labeled "Multiple" in Figure 4.10.) An obvious disadvantage of this approach is that it results in wasted space. This is partly remedied by only storing a rectangle once and having all of the leaf quads that it intersects contain a pointer to the rectangle. Another problem with this approach is that queries such as region search may report the same rectangle more than once. This is addressed by marking a rectangle when it is reported for the first time and by not reporting rectangles that have been previously marked. At the end of the region-search operation, all marked rectangles need to be unmarked in preparation for the next query. Experiments on VLSI mask data were used to evaluate MSQT for different threshold values and for different region-search queries. A large threshold value results in longer lists of pointers in the leaf quads that have to be searched. On the other hand, a small threshold value results in a quad tree with greater height and more leaf nodes as quads have to be subdivided more before they meet the stopping criterion. Consequently, a rectangle now intersects and must be pointed at by more leaf nodes. A region-search query with a small query rectangle (window) benefits from a smaller threshold because it has to search smaller lists in a handful of leaf quads. A large window benefits from a higher threshold value because it has to search fewer quads and encounters fewer duplicates.

4.4.2.4 Quad List Quad Trees

In 1989, Weyten and De Pauw [34] proposed a more efficient implementation of MSQT called quad list quad trees (QLQT). For region searches, experiments on VLSI data showed speedups ranging from 1.85 to 4.92 over MSQT, depending on the size of the window. In QLQT, four different lists (numbered 0–3) are associated with each leaf node. If a rectangle intersects the leaf quad, a pointer to it is stored in one of the four lists. The choice of the list is determined by the relative position of this rectangle with respect to the quad. The relative position is encoded by a pair of bits xy. x is 0 if the rectangle does not cross the lower boundary of the leaf quad and is 1, otherwise. Similarly, y is 0 if the rectangle does not cross the left boundary of the leaf quad and is 1, otherwise. The rectangle is stored in the list corresponding to the integer represented by this two bit string. Figure 4.12 illustrates the concept. Notice that each rectangle belongs to exactly one list 0. This corresponds to the quad that contains the bottom left corner of the rectangle. Observe, also, that the combination of the four lists in a leaf quad gives the same pointers as the single list in the same leaf in MSQT. The region search of MSQT can now be improved for QLQT by using the following procedure for each quad that intersects the query window. If the query window's left edge crosses the quad, only the quad's lists 0 and 1 need to be searched. If the window's bottom edge crosses the quad, the quad's lists 0 and 2 need to be searched. If the windows bottom left corner belongs to the quad, all four lists must

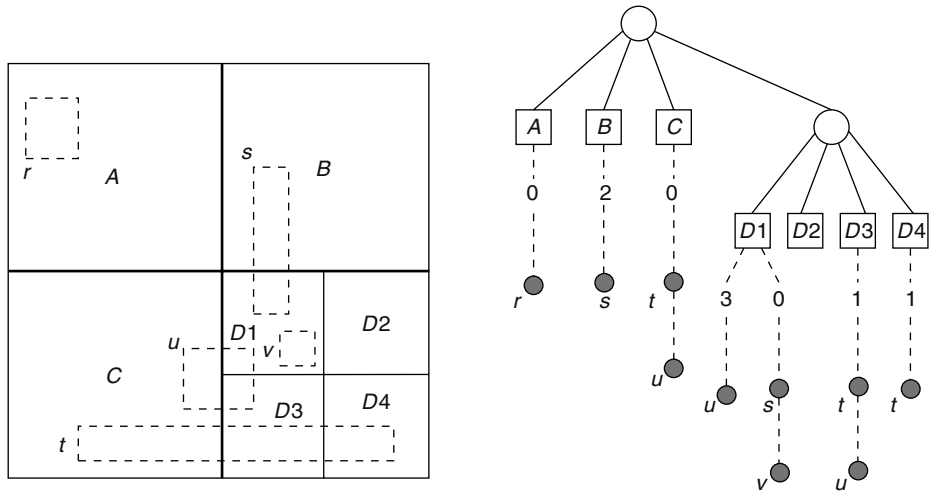


FIGURE 4.12 Leaf quads are A, B, C, $D1$, $D2$, $D3$, and $D4$. The rectangles are r–v. Rectangle t intersects quads C, $D3$, and $D4$ and must appear in the lists of each of the leaf nodes in the quad tree. Observe that t does not cross the lower boundaries of any of the three quads and $x = 0$ in each case. However, t does cross the left boundaries of $D3$ and $D4$ and $y = 1$ in these cases. Thus, t goes into list 1 in $D3$ and $D4$. Because t does not cross the left boundary of C, it goes into list 0 in C. Note that the filled circles represent pointers to the rectangles rather than the rectangles themselves.

be searched. For all other quads, only list 0 must be searched. Thus, the advantages of the QLQT over MSQT are as follows:

1. QLQT has to examine fewer list nodes than MSQT for a region-search query.
2. Unlike MSQT, QLQT does not require marking and unmarking procedures to identify duplicates.

4.4.2.5 Bounded Quad Trees

Later, in 1989, Pitaksanonkul et al. proposed a variation of quad trees [35] that we refer to as bounded quad trees (BQT). Here, a rectangle is only stored in the quad that contains its bottom left corner (see the quad tree labeled "Single" in Figure 4.10). This may be viewed as a version of QLQT that only uses list 0. Experimental comparisons with kd trees show that for small threshold values, quad trees search fewer nodes than kd trees.

4.4.2.6 HV Trees

Next, in 1993, Lai et al. [36] presented a variation that once again uses bisector lists. It overcomes some of the inefficiencies of the original BLQT by a tighter implementation. An HV tree consists of alternate levels of H-nodes and V-nodes. An H-node splits the space assigned to it into two halves with a horizontal bisector, while a V-node does the same by using a vertical bisector. A node is not split if the number of rectangles assigned to it is less than some fixed threshold.

Rectangles intersecting an H-node's horizontal bisector are stored in the node's bisector list. Bisector lists are implemented using cut trees. A vertical cutline divides the horizontal bisector into two halves. All rectangles that intersect this vertical cutline are stored in the root of the cut tree. All rectangles to the left of the cutline are recursively stored in the left subtree and all rectangles to the right are recursively stored in the right subtree. So far, the data structure is identical to Kedem's binary

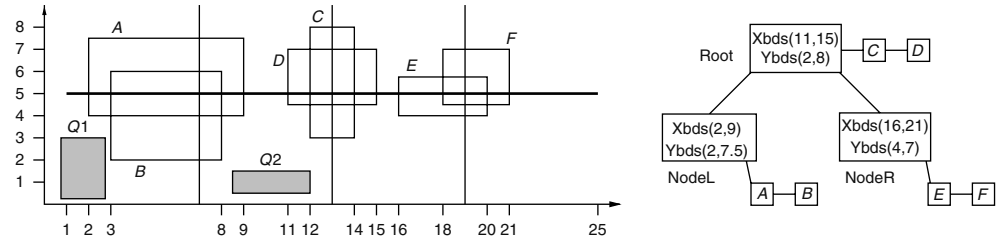


FIGURE 4.13 Bisector list implementation in HVT. All rectangles intersect the thick horizontal bisector line ($y = 5$). The first vertical cutline at $x = 13$ corresponding to the root of the tree intersects rectangles C and D. These rectangles are stored in a linked list at the root. Rectangles A and B are to the left of the vertical cutline and are stored in the left subtree. Similarly, rectangles C and D are stored in the right subtree. The X bounds associated with the root node are obtained by examining the x coordinates of rectangles C and D, while its Y bounds are obtained by examining the y coordinates of all six rectangles stored in the tree. The two shaded rectangles are query rectangles. For $Q1$, the search will start at root, but will not search the linked list with C and D because $Q1$'s right side is to the left of root's lower x bound. The search will then examine nodeL, but not nodeR. For $Q2$, the search will avoid searching the bisector list entirely because its upper side is below root's lower y bound.

tree implementation of the bisector list. In addition to maintaining a list of rectangles intersecting a vertical cutline at the corresponding node n, the HV tree also maintains four additional bounds that significantly improve performance of the region-search operation. The bounds y_upper_bound and y_lower_bound are the maximum and minimum y coordinates of any of the rectangles stored in n or in any of n's descendants. The bounds x_lower_bound and x_upper_bound are the minimum and maximum x coordinates of the rectangles stored in node n. Figure 4.13 illustrates these concepts. Comprehensive experimental results comparing HVT with BQT, kd, and QLQT showed that the data structures ordered from best to worst in terms of space requirements were HVT, BQT, kd, and QLQT. In terms of speed, the best data structures were HVT and QLQT followed by BQT and finally kd.

4.4.2.7 Hinted Quad Trees

In 1997, Lai et al. [37] described a variation of the QLQT that was specifically designed for design-rule checking. Design-rule checking requires one to check rectangles in the vicinity of the query rectangle for possible violations. Previously, this was achieved by employing a traditional region query whose rectangle was the original query rectangle extended in all directions by a specified amount. Region searches start at the root of the tree and proceed down the tree as discussed previously. The hinted quad tree is based on the philosophy that it is wasteful to begin searching at the root, when, with an appropriate hint, the algorithm can start the search lower down in the tree. Two questions arise here: at which node should the search begin and how does the algorithm get to that node? The node at which the design rule check for rectangle r begins is called the owner of r. This is defined as the lowest node in the quad tree that completely contains r expanded in all four directions. Because the type of r is known (e.g., whether it is n-type diffusion or metal), the amount by which r has to be expanded is also known in advance. Clearly, any rectangle that intersects the expanded r must be referenced by at least one leaf in the owner node's subtree. The owner node may be reached by following parent pointers from the rectangle. However, this could be expensive. Consequently, in HQT, each rectangle maintains a pointer to the owner virtually eliminating the cost of getting to that node. Although this is the main contribution of the HQT, there are additional implementation improvements over the underlying QLQT that are used to speed up the data structure. First, the HQT resolves the situation where the boundary of a rectangle stored in the data structure or a query rectangle coincides with that of a quad. Second, HQT sorts the four lists of rectangles stored in each leaf node with one of their x or y coordinates as keys. This reduces the search time at the leaves

and consequently makes it possible to use a higher threshold than that used in QLQT. Experimental results showed that HQT outperforms QLQT, BQT, HVT, and kd on neighbor-search queries by at least 20 percent. However, its build time and space requirements were not as good as some of the other data structures.

ACKNOWLEDGMENT

Section 4.4 was reproduced with permission of Taylor & Francis Group, LLC, from Chapter 52 (Layout Data Structures), in *Handbook of Data Structures and Applications*, Chapman and Hall/CRC Press, edited by Dinesh P. Mehta and Sartaj Sahni.

REFERENCES

1. E. Horowitz, S. Sahni, and D. Mehta. *Fundamentals of Data Structures in C++*, Second Edition. Summit, NJ: Silicon Press, 2007.
2. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*, Second Edition. Berlin, Germany: Springer-Verlag, 2000.
3. K. Mehlhorn and S. Naher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge, United Kingdom: Cambridge University Press, 1999.
4. http://www.cgal.org/.
5. S.C. Maruvada, K. Krishnamoorthy, F. Balasa, and L.M. Ionescu. Red-black interval trees in device-level analog placement. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E86-A (12): 3127–3135, Japan, December 2003.
6. J. Cong, J. Fang, and K.-Y. Khoo. An implicit connection graph maze routing algorithm for ECO routing. *Proceedings of the International Conference on Computer-Aided Design*, San Jose, California, 1999.
7. H.-Y. Chen, Y.-L. Li, and Z.-D. Lin. NEMO: A new implicit connection graph-based gridless router with multi-layer planes and pseudo-tile propagation. *International Symposium on Physical Design*, San Jose, California, 2006.
8. S. Liao, N. Shenoy, and W. Nicholls. An efficient external-memory implementation of region query with application to area routing. *Proceedings of the International Conference on Computer Design*, Freiburg, Germany, 2002.
9. H. Zhou, N. Shenoy, and W. Nicholls. Efficient spanning tree construction without Delaunay triangulation. *Information Processing Letters*, 81(5), 2002.
10. H. Zhou. Efficient steiner tree construction based on spanning graphs. *IEEE Transactions on Computer Aided Design*, 23(5): 704–710, May 2004.
11. L. Stockmeyer. Optimal orientations of cells in slicing floorplan designs. *Information and Control*, 59: 91–101, 1983.
12. K. Keutzer. Dagon: Technology binding and local optimization by dag matching. In *Proceedings of the Design Automation Conference*, Miami Beach, Florida, pp. 617–623, June 1987.
13. L.P.P.P. van Ginneken. Buffer placement in distributed RC-tree networks for minimal Elmore delay. In *Proceedings of the International Symposium on Circuits and Systems*, New Orleans, Louisiana, pp. 865–868, 1990.
14. W. Shi. A fast algorithm for area minimization of slicing floorplans. *IEEE Transactions on Computer Aided Design*, 15: 550–557, 1996.
15. R. Chen and H. Zhou. A flexible data structure for efficient buffer insertion. In *Proceedings of the International Conference on Computer Design*, pp. 216–221, San Jose, CA, October 2004.
16. W. Shi and Z. Li. An $o(n \log n)$ time algorithm for optimal buffer insertion. In *Proceedings of the Design Automation Conference*, pp. 580–585, Anaheim, CA, June 2003.
17. W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6), 1990.
18. W. Pugh. *A Skip List Cookbook*. Technical Report CS-TR-2286.1. College Park, MD: University of Maryland, 1990.
19. N.H.E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*, Second Edition. New York: Addison Wesley, 1993.
20. J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*, Third Edition. New York: Morgan Kaufmann, 2003.

21. D.G. Boyer. Symbolic layout compaction review. In *Proceedings of 25th Design Automation Conference*, Anaheim, California, pp. 383–389, 1988.
22. J. Ousterhout, G. Hamachi, R. Mayo, W. Scott, and G. Taylor. Magic: A VLSI layout system. In *Proceedings of 21st Design Automation Conference*, pp. 152–159, 1984.
23. J.K. Ousterhout. Corner stitching: A data structuring technique for VLSI layout tools. *IEEE Transactions on Computer-Aided Design*, 3(1): 87–100, 1984.
24. D.P. Mehta. Estimating the memory requirements of the rectangular and L-shaped corner stitching data structures. *ACM Transactions on the Design Automation of Electronic Systems*, 3(2), April 1998.
25. M. Quayle and J. Solworth. Expanded rectangles: A new VLSI data structure. In *Proceedings of the International Conference on Computer-Aided Design*, pp. 538–541, 1988.
26. D. Marple, M. Smulders, and H. Hegen. Tailor: A layout system based on trapezoidal corner stitching. *IEEE Transactions on Computer-Aided Design*, 9(1): 66–90, 1990.
27. C.H. Séquin and H. da Silva Façanha. Corner stitched tiles with curved boundaries. *IEEE Transactions on Computer-Aided Design*, 12(1): 47–58, 1993.
28. D.P. Mehta and G. Blust. Corner stitching for simple rectilinear shapes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16: 186–198, February 1997.
29. S. Nahar and S. Sahni. A fast algorithm for polygon decomposition. *IEEE Transactions on Computer-Aided Design*, 7: 478–483, April 1988.
30. M. Lopez and D. Mehta. Efficient decomposition of polygons into L-shapes with applications to VLSI layouts. *ACM Transactions on Design Automation of Electronic Systems*, 1: 371–395, 1996.
31. G. Kedem. The quad-CIF tree: A data structure for hierarchical on-line algorithms. In *Proceedings of the 19th Design Automation Conference*, Washington, pp. 352–357, 1982.
32. J.B. Rosenberg. Geographical data structures compared: A study of data structures supporting region queries. *IEEE Transactions on Computer-Aided Design*, 4(1): 53–67, 1985.
33. R.L. Brown. Multiple storage quad trees: A simpler faster alternative to bisector list quad trees. *IEEE Transactions on Computer-Aided Design*, 5(3): 413–419, 1986.
34. L. Weyten and W. de Pauw. Quad list quad trees: A geometric data structure with improved performance for large region queries. *IEEE Transactions on Computer-Aided Design*, 8(3): 229–233, 1989.
35. A. Pitaksanonkul, S. Thanawastien, and C. Lursinsap. Comparison of quad trees and 4-D trees: New results. *IEEE Transactions on Computer-Aided Design*, 8(11): 1157–1164, 1989.
36. G. Lai, D.S. Fussell, and D.F. Wong. HV/VH trees: A new spatial data structure for fast region queries. In *Proceedings of the 30th Design Automation Conference*, Dallas, Texas, pp. 43–47, 1993.
37. G. Lai, D.S. Fussell, and D.F. Wong. Hinted quad trees for VLSI geometry DRC based on efficient searching for neighbors. *IEEE Transactions on Computer-Aided Design*, 15(3): 317–324, 1996.

5 Basic Algorithmic Techniques

Vishal Khandelwal and Ankur Srivastava

CONTENTS

This chapter provides a brief overview of some commonly used general concepts and algorithmic techniques. The chapter begins by discussing ways of analyzing the complexity of algorithms, followed by general algorithmic concepts like greedy algorithms and dynamic programming. This is followed by a comprehensive discussion on graph algorithms including network flow techniques. This is followed by discussions on NP completeness and computational geometry. The chapter ends with the description of the technique of simulated annealing.

5.1 BASIC COMPLEXITY ANALYSIS

An algorithm is essentially a sequence of simple steps used to solve a complex problem. An algorithm is considered good if its overall runtime is small and the rate at which this runtime increases with the problem size is small. Typically, this runtime complexity is analytically measured/modeled as a function of the total number of elements in the input problem. To make this analysis simpler, several notations and conventions have been developed.

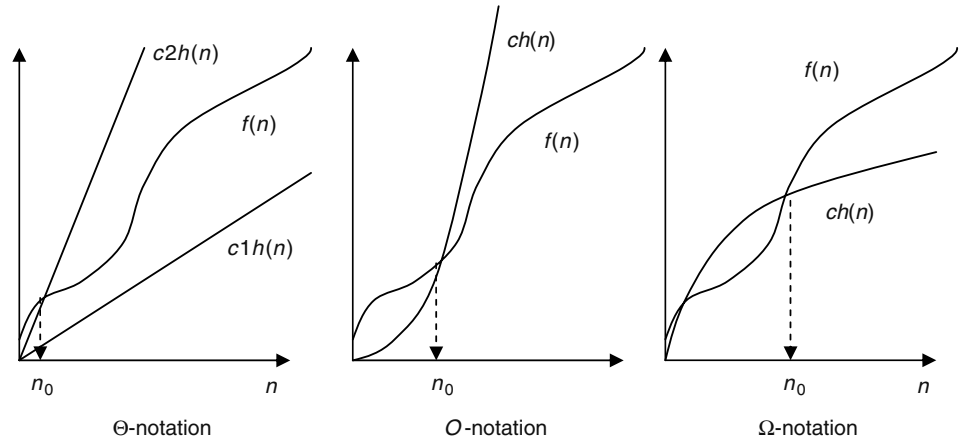


FIGURE 5.1 Complexity analysis.

Θ-Notation

For a function $h(n)$, $\Theta[h(n)]$ represents the set of all functions that satisfy the following:

$$\Theta[h(n)] = \{f(n): \text{there exist positive constants } c1 \text{ and } c2 \text{ and an } n_0 \text{ such that}$$
$$0 \leq c1h(n) \leq f(n) \leq c2h(n) \quad \forall\, n \geq n_0\}$$

Conceptually, the set of functions $f(n)$ are sandwiched between $c1h(n)$ and $c2h(n)$. In such scenarios, $h(n)$ is said to be the asymptotically tight bound (see Figure 5.1) for $f(n)$. Therefore, if an algorithm has a complexity of $f(n)$ (takes $f(n)$ steps to execute), then its complexity could be classified as $\Theta[h(n)]$.

O-Notation

For a function $h(n)$, $O[h(n)]$ represents a set of functions that satisfy the following:

$$O[h(n)] = \{f(n): \text{there exist positive constants } c \text{ and an } n_0 \text{ such that}$$
$$0 \leq f(n) \leq ch(n) \quad \forall\, n \geq n_0\}$$

The O-notation represents an upper bound (see Figure 5.1) for the set of functions $f(n)$. Therefore, an algorithm with complexity $f(n)$ could be classified as an algorithm with complexity $O[h(n)]$.

Ω-Notation

For a function $h(n)$, $\Omega[h(n)]$ represents a set of functions that satisfy the following:

$$\Omega[h(n)] = \{f(n): \text{there exist positive constants } c \text{ and an } n_0 \text{ such that}$$
$$0 \leq ch(n) \leq f(n) \quad \forall\, n \geq n_0\}$$

The Ω-notation represents a lower bound (see Figure 5.1) for the set of functions $f(n)$.

EXAMPLE

Analysis of the Complexity of Sort

```
Sort (Array:A, size:N):
 last = N
 While last >= 1
```

```
            max = A[1]
            max-location = 1
            For i = 1 to last
                    If (A[i] > max)
                              max = A[i]
                              max-location = i
            temp = A[max-location]
            A[max-location] = A[last]
            A[last] = temp
            last = last − 1
Return A
```

The outer while loop runs N times. For the first time the inner loop runs N times, followed by $N-1$ and then $N-2$, etc. So the total number of iterations in this algorithm become $N+N-1+N-2+\cdots+1 = N(N+1)/2$.

Now it can be seen that the algorithmic complexity of sort, $f(N) = N(N+1)/2$ is $O(N^2)$ and also $\Theta(N^2)$.

5.2 GREEDY ALGORITHMS

An algorithm is defined as a sequence of simple steps that solves a more complicated problem. At each step, the algorithm makes a decision from a set of choices. Greedy algorithms [1] have the property of making a choice that looks the best at that time. This may or may not guarantee the optimality of the final solution. The key advantage of greedy algorithms is simplicity. In this section, we will discuss the basic properties that a problem must have for greedy strategies to yield the optimal solution. If we can demonstrate the following properties in a problem, then greedy methods will yield the optimal solution:

1. Problem can be modeled as a combination of a greedy choice and a smaller subproblem.
2. There exists an optimal solution to the problem in which the greedy choice has been made.
3. Combination of the optimal solution to the subproblem and the greedy choice results in the optimal solution to the overall problem.

EXAMPLE

Fractional Knapsack Problem

Given a knapsack of a certain size W *and* n *items, with the* ith *item having a value of* v_i *and a quantity of* w_i. *We would like to fill the knapsack with the maximum valued goods.*

The algorithm is as follows:

1. Sort the items in decreasing order of v_i/w_i.
2. Start from the first item in the list and pick as much as you can.
3. If space still left, then go to the next item and repeat.

Note that we select as much as possible of the most valuable item (largest v_i/w_i). This is a greedy step. The remaining space in the knapsack is filled by the remaining items. This constitutes the subproblem. It can be shown that the above three properties hold for the fractional knapsack problem and therefore it is solvable optimally using greedy strategies.

There are many problems (including the 0–1 generalization of the knapsack problem where we are forced to choose the entire item or none at all) where a greedy scheme cannot guarantee optimality. In

such scenarios, greedy schemes are usually employed as heuristics resulting in quick but good solutions to the problem, although not provably optimal.

5.3 DYNAMIC PROGRAMMING

The technique of dynamic programming (DP) [1] essentially is a way of utilizing the availability of cheap memory to improve the runtime of algorithms. This technique was invented by Richard Bellman in 1953. Before we go into the details of this technique, let us discuss the following sequence of steps for solving a problem:

1. Break the problem into smaller subproblems.
2. Solve the smaller subproblems optimally.
3. Combine the optimal solutions to the smaller subproblems to get a solution to the original problem.

Now the term optimal substructure means that the optimal solution to the subproblems can be used to generate the optimal solution to the overall problem. If indeed this is true then the above-mentioned sequence of steps for solving a problem must generate the optimal solution to the overall problem. DP also generates the optimal solution using the same principle. Let us illustrate the DP philosophy using an example.

EXAMPLE

Generation of the Nth Fibonacci Number

Solution

A simple way of generating the Nth Fibonacci number could be as follows:

```
FIBONACCI(N)
     If N = 0 or 1
              then return N
     Else
              return FIBONACCI(N−1) + FIBONACCI(N−2)
```

Note that this problem demonstrates optimal substructure because the optimal solution to the problem of size N can be generated by the optimal solution for subproblem of size $N-1$ and $N-2$. The complexity of this algorithm could be analyzed as follows. Let $T(N)$ represent the complexity of optimally solving a problem of size N. So

$$T(N) = T(N-1) + T(N-2) \quad \text{for } N > 1$$

It could be shown that $T(N)$ is an exponential function of N, which clearly is impractical for large problems. Nonetheless, from close inspection, we find that to solve the subproblem of size $N-1$, we will inevitably solve a subproblem of size $N-2$. This property is called overlapping subproblems. Existence of overlapping subproblems could be utilized to improve the complexity of the above algorithm. Basically, every time a subproblem of a certain size is encountered for the first time, its optimal solution could be stored. Next time, if the optimal solution to this subproblem is needed, it could simply be accessed from memory. Using such techniques, a modified algorithm for Fibonacci numbers is as follows:

```
MODIFIED FIBONACCI(N)
For i = 1 to N
         M[i] = −1
Function Fib(N)
```

```
If M[N] ! = −1
        M[N] = Fib(N − 1) + Fib(N − 2)
return M[N]
```

In this algorithm, the array M stores the optimal solution (Fibonacci values). Whenever the solution of a subproblem is needed, it could be simply read from this array without having to perform the whole computation again from scratch. This technique is called memoization. It could be seen that the complexity of this algorithm is no longer exponential.

Although the Fibonacci example is not an optimization problem, it illustrates the concept behind DP quite well. DP is essentially a divide-and-conquer approach in which larger complex problems are subdivided in simpler subproblems. The existence of the optimal substructure property ensures that optimality of the overall problem will be maintained. Furthermore, overlapping subproblems could be stored in memory (memoization) for improving the runtime complexity of the algorithm. DP-based approaches for a given problem could be developed as follows:

1. Express the overall problem in the form of subproblems.
2. Investigate if the optimal substructure property holds.
3. Investigate the existence of overlapping subproblems.
4. Develop a memoization-based approach in which the solutions to overlapping subproblems are stored in memory, hence improving the computational complexity.

Several physical design/synthesis problems including buffer insertion for wiring trees and technology mapping could be solved optimally using DP [5].

5.4 INTRODUCTION TO GRAPH THEORY

Graph theory [1,2] is believed to have begun in the year 1736 with the publication of the solution to the Konigsberg bridge problem, developed by Euler. A graph is characterized by $G = (V, E)$, where V is the set of vertices and E is the set of edges between them (see Figure 5.2). These edges could either be directed (leading to a directed graph) or undirected (undirected graph). Graphs provide an excellent way to abstract various problems in physical synthesis and design. Combinational circuits are typically modeled as directed acyclic graphs and placement netlists are also modeled as graphs.

Definition 1 *Path: A sequence of vertices and edges in which no vertex is repeated.*

Definition 2 *Cycle: A sequence of vertices $v_0, v_1, v_2, \ldots, v_n$ where $v_n = v_0$ and all other vertices are different.*

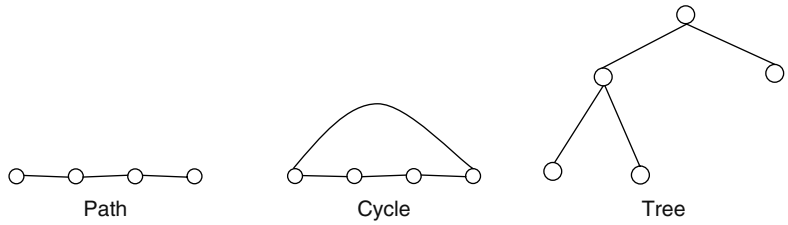


FIGURE 5.2 Examples of graphs.

5.4.1 Graph Traversal/Search

Searching a graph is the process of hopping from one vertex to the other in search for the appropriate vertex or edge. Graph search is used extensively in physical synthesis and design problems when a gate of a specific characteristic is being searched. It also finds widespread application in timing analysis. Two schemes for searching on a graph have been developed

5.4.1.1 Breadth First Search

Given a graph $G = (V, E)$ and a source vertex s, breadth first search (BFS) systematically investigates all the vertices that can be reached from s. The algorithm is outlined below:

```
BFS(G(V,E),s)
For each vertex u ∈ V − {s}
    Status[u] = untouched
    Distance[u] = ∞
Distance[s] = 0
QUEUE = {s}
While QUEUE! = NULL
    u = FRONT(QUEUE) /* The function FRONT returns the front
         of a queue */
    For each vertex v that can be directly reached from u
           If Status[v] = untouched
                    Status[v] = touched
                    Distance[v] = Distance[u] + 1
                    ENQUEUE(QUEUE, v)
    DEQUEUE(QUEUE)  /* Remove the Front Vertex from the Queue */
    Status[u] = Finished
```

In this algorithm, the frontier between the discovered and undiscovered vertices proceeds like a wavefront. Starting from the source, all vertices immediately adjacent to it are investigated. This is followed by investigation of all vertices adjacent to these and so on. This algorithm finds the minimum number of edges between the source s and the vertices that are reachable from s (this information gets stored in the array Distance). If a vertex cannot be reached then its distance from the source is infinity.

5.4.1.2 Depth First Search

Unlike BFS that proceeds as a wavefront, depth first search (DFS) investigates deeper in the graph till it cannot go any further. At this point, it backtracks to the nearest vertex and investigates its neighbors once again in a depth first manner. This process continues till no further vertices can be explored. The algorithm is outlined below:

```
DFS(G(V,E))                    Touch-DFS(u)
For each vertex u              Status[u] = touched
       Status[u] = untouched   Time = Time + 1
Time = 0                       Starting-Time[u] = Time
For each vertex u              For each v that can be reached from u
    If Status[u] = untouched          If Status[v] = untouched
```

```
Touch-DFS(u)                    Touch-DFS(v)
                Status[u] = finished
                Time = Time + 1
                Finishing-Time[u] = Time
```

As indicated in the algorithm above, we start with a vertex and investigate deeper into the neighborhood till we cannot go any further. At this point, we go one level above to the previous vertex and investigate deep into the graph once again. A vertex is deemed finished if all the vertices adjacent to it have been touched in a depth first manner. Note that Starting-Time and Finishing-Time, respectively, indicate the time stamp at which we begin investigating a vertex and at which we have investigated its entire neighborhood.

The runtime complexity of both BFS and DFS is $O(|V| + |E|)$.

5.4.1.3 Topological Ordering

Definition 3 *Directed Acyclic Graph (DAG): A directed graph $G = (V, E)$ in which there are no directed cycles.*

Directed acyclic graphs can be used to model most combinational circuits and therefore are particularly important for VLSI computer-aided design (CAD). Topological ordering in DAGs is an ordering $v_0, \ldots, v_n$ of all vertices in V such that for a given vertex v_i, all the vertices in V that have a path either directly or indirectly to v_i must come before v_i in this ordering.

Topological ordering can be generated using DFS by sorting the nodes in decreasing order of their finishing times.

5.4.2 Minimum Spanning Tree

Let us suppose we have an undirected graph $G = (V, E)$ where each edge (u, v) has a weight $w(u, v)$. A spanning tree on such a graph is defined as follows:

Definition 4 *Spanning tree: A spanning tree of a graph $G = (V, E)$ is a subgraph $G' = (V, E')$, which has the same vertices as G and the edges $E' \subseteq E$ such that G' forms a tree.*

A minimum spanning tree (MST) of a graph G is a spanning tree with the minimum total weight (of all edges) among all possible spanning trees of G (see Figure 5.3). There are two popular algorithms for finding the MST of a graph: Kruskal's algorithm and Prim's algorithm.

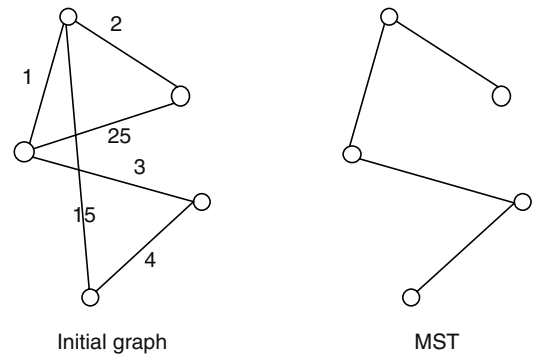


FIGURE 5.3 Minimum spanning tree.

5.4.2.1 Kruskal's Algorithm

Kruskal's algorithm proceeds by starting with a set of disconnected trees (a forest) of vertices in G and merges them in such a way that we eventually get the MST of G. The algorithm is as follows:

```
Kruskal(G=(V,E))
Each node in G represents a trivial Tree.
Sort all edges in E in non-decreasing order of weights
For each edge (u,v) ∈ E in the non-decreasing order
    If u and v are in separate trees
              Merge the two trees into one by connecting them
                  through the edge (u,v)
```

The algorithm starts by assigning all nodes to separate trees. Then it traverses the edges in nondecreasing order of their weights. If an edge merges two separate trees then it is used to create a larger tree otherwise it is discarded. The algorithm terminates after generating the MST.

5.4.2.2 Prim's Algorithm

Unlike Kruskal's algorithm, that maintains multiple trees and merges them iteratively, Prim's algorithm has only one tree and merges more vertices in this tree till the MST is created. The algorithm is outlined as follows:

```
Prim(G=(V,E))
Start with any vertex in V and assign it to a Tree T
While there exist vertices in G not in T
       Find a vertex in G-T which is closest to T
       Expand T by including this vertex
```

MSTs are used extensively in physical design to predict the wirelengh of interconnects when the placement information is available and routing is not known.

5.4.3 Shortest Paths in Graphs

The problem of shortest paths in graphs has several important practical applications. Given a graph $G = (V, E)$ (directed or undirected) and edge weights, try to find the shortest weighted path from a given source s to all other vertices (single-source shortest path problem) or between all pair of vertices. The overall weight of a path is simply the sum of all the edge weights on it.

Let us start the discussion with the single-source shortest path problem. Given a source s, we would like to find the shortest path to all other vertices in the graph. Definition of a shortest path between two vertices becomes ambiguous when there exists a negative weight cycle between the source and the destination. We can simply find a shorter route by indefinitely going around this negative cycle (and therefore reducing the overall path weight). We describe two algorithms for finding the shortest paths: Dijkstra's algorithm and Bellman Ford algorithm. Dijkstra's algorithm assumes all the edge weights are positive and therefore there are no negative weighted cycles either. On the other hand, Bellman Ford algorithm can handle negative weighted edges and also detect the existence of negative weighted cycles (a case where shortest path is not defined).

5.4.3.1 Dijkstra's Algorithm

This algorithm takes a weighted graph G with positive edge weights, a source vertex, and generates the shortest weighted path solution. It initializes two sets S and S'. The set S consists of all vertices in G whose shortest path from s has been calculated and the set S' consists of all the remaining vertices. Initially, $S = \{s\}$ and $S' = V - \{s\}$. We also initialize a label array L, which stores the labeling for the vertices. The moment a vertex u is included in the set S, its labeling $L[u]$ is exactly the weight of the shortest path between s and u. Initially, $L[s] = 0$ and $L[u] = \infty \; \forall \; u \in V - \{s\}$. In the next step, the labels of all the vertices v in S', which are adjacent to a vertex u in S are updated as follows. If $L[u] + \text{weight}(u, v) \leq L[v]$ then $L[v] = L[u] + \text{weight}(u,v)$. After updating all the labels, the vertex in S' that has the smallest label is chosen and moved to the set S. At this point, the label of this node corresponds to the weight of the shortest path from s. These sequence of steps are continued till S' is null. The algorithm is formally outlined below:

```
Dijkstra(G=(V,E))
S = {s}, S′ = V − {s}
L[s] = 0, L[u] =∞ ∀ u ∈ V − {s}
L[u] = weight[su] ∀ u adjacent to s
While S′ ! = NULL
        Find Minimum L[u] ∀ u in S′
        S = SU {u}
        S′ = S′ − {u}
        For each v in S′ that is adjacent to u
             If L[v] ≥ L[u] + weight(uv)
                  L[v] = L[u] + weight(uv)
```

It could be seen that this is a greedy algorithm because at each step a greedy choice is executed (the vertex with the smallest labeling is chosen). This greedy algorithm indeed results in the optimal solution.

5.4.3.2 Bellman Ford Algorithm

Dijkstra's algorithm cannot handle edge weights that are negative. Bellman Ford algorithm not only handles negative edge weights but also detects the existence of negative weighted cycles (that are reachable from the source s). The algorithm is iterative in nature. Once again it has a label array L. $L[s]$ is initializes to 0 and infinity for all other vertices. The algorithm is outlined below:

```
Bellman Ford (G=(V,E))
L[s] = 0, L[u] = ∞ ∀ u ∈ V − {s}
For i = 1 to Number of Vertices
        For each edge (u,v) ∈ E
             If L[v] ≥ L[u] + weight(uv)
                  L[v] = L[u] + weight(uv)
```

The algorithm is quite self-explanatory. It could be proved that if there are no negative weighted cycles reachable from s then the array L has the shortest path to each vertex in the graph. Detection of negative weighted cycles (reachable from s) can be done by the following simple procedure:

```
Negative Cycle Detection
Let L be the labeling of all nodes after application
     of Bellman Ford
```

```
For each edge (u,v) ∈ E
     If L[v] > L[u] + weight(uv)
          Return: Negative Weighted Cycle Exists
```

The all pair shortest path problem tries to find the shortest paths between all vertices. Of course, one approach is to execute the single-source shortest path algorithm for all the nodes. Much faster algorithms like Floyd Warshall algorithm, etc. have also been developed.

5.5 NETWORK FLOW METHODS

Definition 5 *A* network *is a directed graph* $G = (V, E)$ *where each edge* $(u, v) \in E$ *has a capacity* $c(u, v) \geq 0$. *There exists a node/vertex called the source,* s *and a destination/sink node,* t. *If an edge does not exist in the network then its capacity is set to zero.*

Definition 6 *A* flow *in the network* G *is a real value function* f: $VXV \rightarrow R$. *This has the following properties:*

1. *Capacity constraint*: *Flow* $f(u, v) \leq c(u, v) \quad \forall\, u, v \in V$
2. *Flow conservation*: $\forall\, u \in V - \{s, t\}, \Sigma_{v \in V} f(u, v) = 0$
3. *Skew symmetry*: $\forall\, u, v \in V, f(u, v) = -f(v, u)$

The value of a flow is typically defined as the amount of flow coming out of the source to all the other nodes in the network. It can equivalently be defined as the amount of flow coming into the sink from all the other nodes in the network. Figure 5.4 illustrates an example of network flow.

Definition 7 Maximum flow problem *is defined as the problem of finding a flow assignment to the network such that it has the maximum value (note that a flow assignment must conform to the flow properties as outlined above).*

Network flow [4] formulations have large applicability in various practical problems including supply chain management, airline industry, and many others. Several VLSI CAD applications like low power resource binding, etc. can be modeled as instances of network flow problems. Network flow has also been applied in physical synthesis and design problems like buffer insertion.

Next, an algorithm is presented that solves the maximum flow problem optimally. This algorithm was developed by Ford and Fulkerson. This is an iterative approach and starts with $f(u, v) = 0$ for

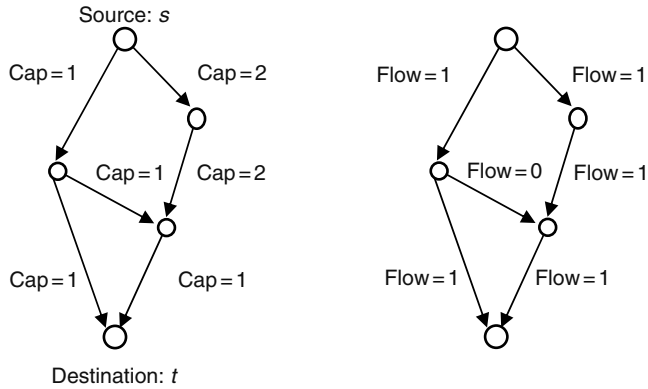


FIGURE 5.4 Network flow.

all vertex pairs (u, v). At each step/iteration, the algorithm finds a path from s to t that still has available capacity (this is a very simple explanation of a more complicated concept) and augments more flow along it. Such a path is therefore called the augmenting path. This process is repeated till no augmenting paths are found. The basic structure of the algorithm is as follows:

```
Ford-Fulkerson (G,s,t)
     For each vertex pair (u,v)
          f(u,v) = 0
     While there is an augmenting path p from s to t
          Send more flow along this path without violating
               the capacity of any edge
     Return f
```

At any iteration, augmenting path is not simply a path of finite positive capacity in the graph. Note that capacity of a path is defined by the capacity of the minimum capacity edge in the path. To find an augmenting path, at a given iteration, a new graph called the residual graph is initialized. Let us suppose that at a given iteration all vertex pairs uv have a flow of $f(u, v)$. The residual capacity is defined as follows:

$$c_{\mathrm{f}}(u, v) = c(uv) - f(u, v)$$

Note that the flow must never violate the capacity constraint. Conceptually, residual capacity is the amount of extra flow we can send from u to v. A residual graph G_{f} is defined as follows:

$$G_{\mathrm{f}} = (V, E_f) \quad \text{where } E_{\mathrm{f}} = \{(u, v), \in V\,X\,V : c_{\mathrm{f}}(u, v) > 0\}$$

The Ford Fulkerson method finds a path from s to t in this residual graph and sends more flow along it as long as the capacity constraint is not violated. The run-time complexity of Ford Fulkerson method is $O(E^* f_{\max})$ where E is the number of edges and $f_{\max}$ is the value of the maximum flow.

Theorem 1 *Maximum Flow Minimum Cut: If f is a flow in the network then the following conditions are equivalent*

1. *f is the maximum flow*
2. *Residual network contains no augmenting paths*
3. *There exists a cut in the network with capacity equal to the flow f*

A cut in a network is a partitioning of the nodes into two: with the source s on one side and the sink t on another. The capacity of a cut is the sum of the capacity of all edges that start in the s partition and end in the t partition.

There are several generalizations/extensions to the concept of maximum flow presented above.

Multiple sources and sinks: Handling multiple sources and sinks can be done easily. A super source and a super sink node can be initialized. Infinite capacity edges can then be added from super source to all the sources. Infinite capacity edges can also be added from all the sinks to the super sink. Solving the maximum flow problem on this modified network is similar to solving it on the original network.

Mincost flow: Mincost flow problems are of the following type. Assuming we need to pay a price for sending each unit of flow on an edge in the network. Given the cost per unit flow for all edges in the network, we would like to send the maximum flow in such a way that it incurs the minimum total

cost. Modifications to the Ford Fulkerson method can be used to solve the mincost flow problem optimally.

Multicommodity flow: So far the discussion has focused on just one type of flow. Several times many commodities need to be transported on a network of finite edge capacities. The sharing of the same network binds these commodities together.

These different commodities represent different types of flow. A version of the multicommodity problem could be described as follows. Given a network with nodes and edge capacities/costs, multiple sinks and sources of different types of flow, satisfy the demands at all the sinks while meeting the capacity constraint and with the minimum total cost. The multicommodity flow problem is NP-complete and has been an active topic of research in the last few decades.

5.6 THEORY OF NP-COMPLETENESS

For algorithms to be computationally practical, it is typically desired that their order of complexity be polynomial in the size of the problem. Problems like sorting, shortest path, etc. are examples for which there exist algorithms of polynomial complexity. A natural question to ask is "Does there exist a polynomial complexity algorithm for all problems?". Certainly, the answer to this question is no because there exist problems like halting problem that has been proven to not have an algorithm (much less a polynomial time algorithm). NP-complete [3] problems are the ones for which we do not know, as yet, if there exists a polynomial complexity algorithm. Typically, the set of all problems that are solvable in polynomial time is called P. Before moving further, we would like to state that the concept of P or NP-complete is typically developed around problems for which the solution is either yes or no, a.k.a., decision problems. For example, the decision version for the maximum flow problem could be "Given a network with finite edge capacities, a source, and a sink, can we send at least K units of flow in the network?" One way to answer this question could be to simply solve the maximum flow problem and check if it is greater than K or not.

Polynomial time verifiability: Let us suppose an oracle gives us the solution to a decision problem. If there exists a polynomial time algorithm to validate if the answer to the decision problem is yes or no for that solution, then the problem is polynomially verifiable. For example, in the decision version of the maximum flow problem, if an oracle gives a flow solution, we can easily (in polynomial time) check if the flow is more than K (yes) or less than K (no). Therefore, the decision version of the maximum flow problem is polynomially verifiable.

NP class of problems: The problems in the set NP are verifiable in polynomial time. It is trivial to show that all problems in the set P (all decision problems that are solvable in polynomial time) are verifiable in polynomial time. Therefore, $P \subseteq \text{NP}$. But as of now it is unknown whether $P = \text{NP}$.

NP-complete problems: They have two characteristics:

1. Problems can be verified in polynomial time.
2. These problems can be transformed into one another using a polynomial number of steps.

Therefore, if there exists a polynomial time algorithm to solve any of the problems in this set, each and every problem in the set becomes polynomially solvable. It just so happens that to date nobody has been able to solve any of the problems in this set in polynomial time. Following is an procedure for proving that a given decision problem is NP-complete:

1. Check whether the problem is in NP (polynomially verifiable).
2. Select a known NP-complete problem.
3. Transform this problem in polynomial steps to an instance of the pertinent problem.
4. Illustrate that given a solution to the known NP-complete problem, we can find a solution to the pertinent problem and vice versa.

If these conditions are satisfied by a given problem then it belongs to the set of NP-complete problems.

The first problem to be proved NP-complete was Satisfiability or SAT by Stephen Cook in his famous 1971 paper "The complexity of theorem proving procedures," in *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*. Shortly after the classic paper by Cook, Richard Karp proved several other problems to be NP-complete. Since then the set of NP-complete problems has been expanding. Several problems in VLSI CAD including technology mapping on DAGs, gate duplication on DAGs, etc. are NP-complete.

EXAMPLE

Illustrative Example: NP-Completeness of 3SAT

3SAT: Given a set of m clauses anded together $F = C_1 \cdot C_2 \cdot C_3 \cdot \cdots \cdot C_m$. Each clause C_i is a logical OR of at most three boolean literals $C_i = (a + b + \bar{o})$ where $\bar{o}$ is the negative phase of boolean variable o. Does there exist an assignment of 0/1 to each variable such that F evaluates to 1? (Note that this is a decision problem.)

Proof of NP-Completeness: Given an assignment of 0/1 to the variables, we can see if each clause evaluates to 1. If all clauses evaluate to 1 then F evaluates to 1 else it is 0. This is a simple polynomial time algorithm for verifying the decision given a specific solution or assignment of 0/1 to the variables. Therefore, 3SAT is in NP. Now let us transform the well-known NP-complete problem SAT to an instance of 3SAT.

SAT: Given a set of m clauses anded together $G = C_1 \cdot C_2 \cdot C_3 \cdot \cdots \cdot C_m$. Each clause C_i is a logical OR of boolean literals $C_i = (a + b + \bar{o} + e + f + \cdots)$. Does there exist an assignment of 0/1 to each variable such that G evaluates to 1. (Note that this is a decision problem.)

To perform this tranformation, we look at each clause C_i in the SAT problem with more than three literals. Let $C_i = (x_1 + x_2 + x_3 + \cdots + x_k)$. This clause is replaced by $k-2$ new clauses each with length 3. For this to happen, we introduce $k-3$ new variables $u_i, \ldots, u_{k-3}$. These clauses are constructed as follows.

$$P_i = (x_1 + x_2 + u_1)(x_3 + \bar{u}_1 + u_2)(x_4 + \bar{u}_2 + u_3)(x_5 + \bar{u}_3 + u_4) \cdots (x_{k-1} + x_k + \bar{u}_{k-3})$$

Note that if there exists an assignment of 0/1 to $x_1, \ldots, x_k$ for which C_i is 1, then there exists an assignment of 0/1 to $u_1, \ldots, u_{k-3}$ such that P_i is 1. If C_i is 0 for an assignment to $x_1, \ldots, x_k$, then there cannot exist an assignment to $u_1, \ldots, u_{k-3}$ such that P_i is 1. Hence, we can safely replace C_i by P_i for all the clauses in the original SAT problem with more than three literals. An assignment that makes C_i 1 will make P_i 1. An assignment that makes C_i as 0 will make P_i as 0 as well. Therefore, replacing all the clauses in SAT by the above-mentioned transformation does not change the problem. Nonetheless, the transformed problem is an instance of the 3SAT problem (because all clauses have less than or equal to three literals). Also, this transformation is polynomial in nature. Hence, 3 SAT is NP-complete.

5.7 COMPUTATIONAL GEOMETRY

Computational geometry deals with the study of algorithms for problems pertaining to geometry. This theory finds application in many engineering problems including VLSI CAD, robotics, graphics, etc.

5.7.1 CONVEX HULL

Given a set of n points on a plane, each characterized by its x and y coordinates. Convex hull is the smallest convex polygon P for which these points are either in the interior or on the boundary of the polygon (see Figure 5.5). We now present an algorithm called Graham's scan for generating a convex hull of n points on a plane.

```
Graham Scan(n points on a plane)
Let p0 be the point with minimum y coordinate
Sort the rest of the points p1, ..., pn-1 by the polar
     angle in counterclockwise order w.r.t. p0
```

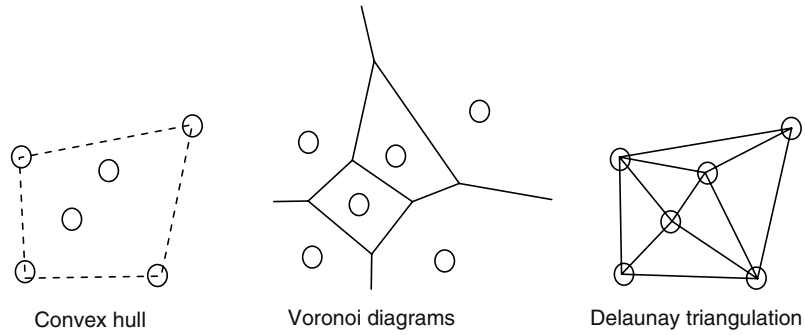


FIGURE 5.5 Computational geometry.

```
Initialize Stack S
Push(p0,S)
Push(p1,S)
Push(p2,S)
For i = 3 to n−1
        While (the angle made by the next to top point on S,
               top point on S and pi makes a non left turn)
                  Pop(S)
        Push(pi, S)
Return S
```

The algorithm returns the stack S that contains the vertices of the convex hull. Basically, the algorithm starts with the bottom-most point p_0. Then it sorts all the other points in increasing order of the polar angle made in counterclockwise direction w.r.t p_0. It then pushes p_0, p_1, and p_2 in the stack. Starting from p_3, it checks if top two elements and the current point p_i forms a left turn or not. If it does then p_i is pushed into the stack (implying that it is part of the hull). If not then that means the current stack has some points not on the convex hull and therefore needs to be popped. Convex hulls, just like MSTs, are also used in predicting the wirelength when the placement is fixed and routing is not known.

5.7.2 Voronoi Diagrams and Delaunay Triangulation

A Voronoi diagram is a partitioning of a plane with n points (let us call them central points) into convex polygons (see Figure 5.5). Each convex polygon has exactly one central point. Also, any point within the convex polygon is closest to the central point associated with the polygon.

Delaunay triangulation is simply the dual of Voronoi diagrams. This is a triangulation of central points such that none of the central points are inside the circumcircle of a triangle (see Figure 5.5).

Although we have defined these concepts for a plane, they are easily extendible to multiple dimensions as well.

5.8 SIMULATED ANNEALING

Simulated annealing is a general global optimization scheme. This technique is primarily inspired from the process of annealing (slow cooling) in metallurgy where the material is cooled slowly to form high quality crystals. The simulated annealing algorithm basically follows a similar principle. Conceptually, it has a starting temperature parameter that is usually set to a very high quantity. This