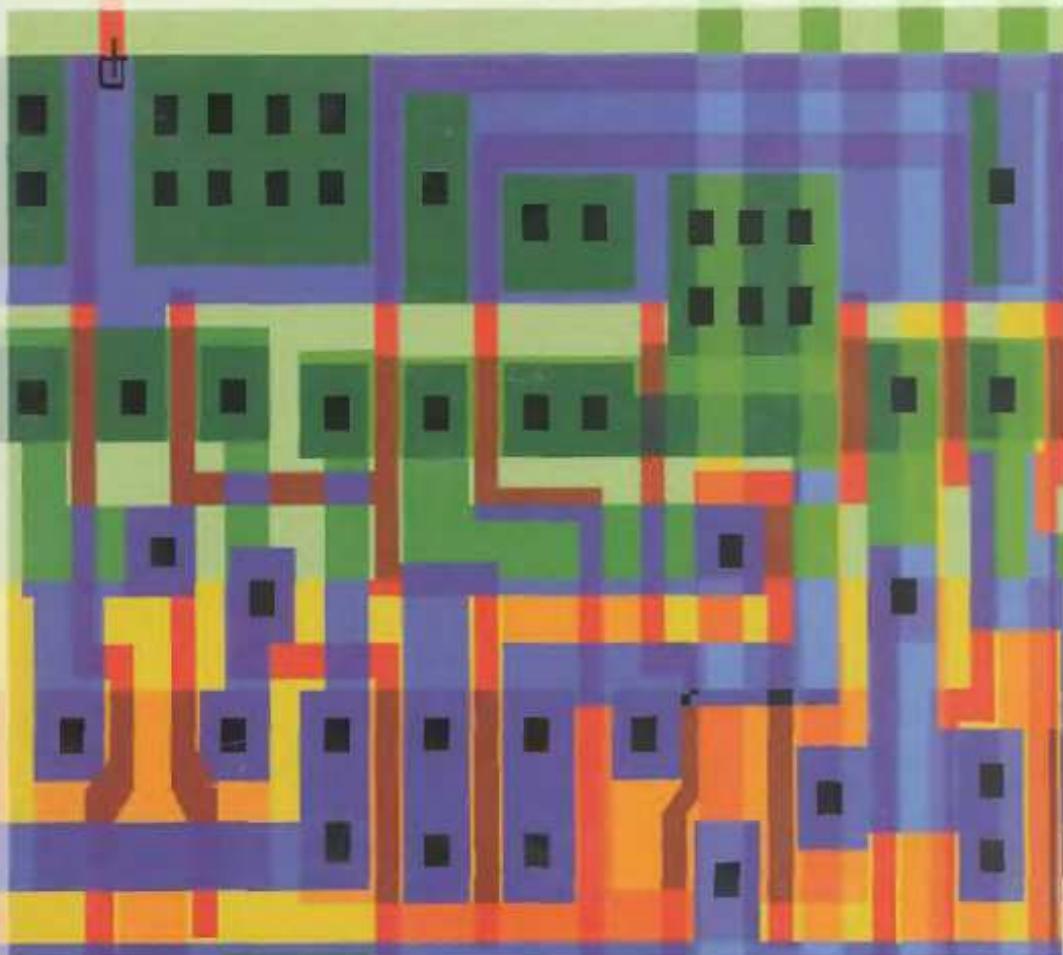


Algorithms for VLSI Design Automation



Sabih H. Gerez



WILEY

Algorithms for VLSI Design Automation

Algorithms for VLSI Design Automation

Sabih H. Gerez

*University of Twente, Department of Electrical Engineering,
The Netherlands*

JOHN WILEY & SONS

Chichester • New York • Weinheim • Brisbane • Singapore • Toronto

Copyright © 1999 by John Wiley & Sons Ltd.
Baffins Lane, Chichester,
West Sussex PO19 1UD, England

National 01243 779777
International (+44) 1243 779777

e-mail (for orders and customer service enquiries): cs-books@wiley.co.uk

Visit our Home Page On <http://www.wiley.co.uk>
or
<http://www.wiley.com>

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except under the terms of the Copyright, Designs and Patents Act 1988 or under the terms of a licence issued by the Copyright Licensing Agency, 90 Tottenham Court Road, London, W1P 9HE, UK, without the permission in writing of the publisher.

Other Wiley Editorial Offices

John Wiley & Sons, Inc., 605 Third Avenue,
New York, NY 10158-0012, USA

Wiley-VCH Verlag GmbH, Pappelallee 3,
D-69469 Weinheim, Germany

Jacaranda Wiley Ltd, 33 Park Road, Milton,
Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01,
Jin Xing Distripark, Singapore 129 809

John Wiley & Sons (Canada) Ltd, 22 Worcester Road,
Rexdale, Ontario M9W 1L1, Canada

Library of Congress Cataloging-in-Publication Data

Gerez, Sabih H.

Algorithms for VLSI design Automation / Sabih H. Gerez – Draft
ed.

p. cm.

Includes bibliographical references and index.

ISBN 0-471-98489-2

1. Integrated Circuits – Very large scale integration – Design and
construction – Data processing. 2. Computer-aided design.

3. Algorithms. I. Title.

TK7874.75.G47 1998

621.39'5 — dc21

98-39574

CIP

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library.

ISBN 0 471 98489 2

Produced from PostScript files supplied by the author.

Printed and bound in Great Britain by Bookcraft (Bath) Ltd.

This book is printed on acid-free paper responsibly manufactured from sustainable
forestry, in which at least two trees are planted for each one used in paper production.

Contents

Preface	x1
I Preliminaries	1
1 Introduction to Design Methodologies	3
1.1 The VLSI Design Problem	3
1.2 The Design Domains	5
1.3 Design Actions	7
1.4 Design Methods and Technologies	8
1.5 Bibliographic Notes	9
2 A Quick Tour of VLSI Design Automation Tools	11
2.1 Algorithmic and System Design	11
2.2 Structural and Logic Design	13
2.3 Transistor-level Design	15
2.4 Layout Design	15
2.5 Verification Methods	17
2.6 Design Management Tools	18
2.7 Bibliographic Notes	19
3 Algorithmic Graph Theory and Computational Complexity	21
3.1 Terminology	22
3.2 Data Structures for the Representation of Graphs	24
3.3 Computational Complexity	26
3.4 Examples of Graph Algorithms	29
3.4.1 Depth-first Search	30
3.4.2 Breadth-first Search	32
3.4.3 Dijkstra's Shortest-path Algorithm	34
3.4.4 Prim's Algorithm for Minimum Spanning Trees	37

3.5	Bibliographic Notes	39
3.6	Exercises	40
4	Tractable and Intractable Problems	41
4.1	Combinatorial Optimization Problems	41
4.2	Decision problems	43
4.3	Complexity Classes	45
4.4	NP-completeness and NP-hardness	46
4.5	Consequences	49
4.6	Bibliographic Notes	50
4.7	Exercises	51
5	General-purpose Methods for Combinatorial Optimization	53
5.1	The Unit-size Placement Problem	54
5.2	Backtracking and Branch-and-bound	55
5.2.1	Backtracking	56
5.2.2	Branch-and-bound	59
5.3	Dynamic Programming	62
5.4	Integer Linear Programming	65
5.4.1	Linear Programming	65
5.4.2	Integer Linear Programming	67
5.5	Local Search	69
5.6	Simulated Annealing	71
5.7	Tabu Search	73
5.8	Genetic Algorithms	75
5.9	A Few Final Remarks on General-purpose Methods	78
5.10	Bibliographic Notes	79
II	Selected Design Problems and Algorithms	81
6	Layout Compaction	83
6.1	Design Rules	83
6.2	Symbolic Layout	85
6.3	Problem Formulation	86
6.3.1	Applications of Compaction	86
6.3.2	Informal Problem Formulation	86
6.3.3	Graph-theoretical Formulation	87
6.3.4	Maximum-distance Constraints	90
6.4	Algorithms for Constraint-graph Compaction	91
6.4.1	A Longest-path Algorithm for DAGs	91
6.4.2	The Longest Path in Graphs with Cycles	92

6.4.3	The Liao-Wong Algorithm	93
6.4.4	The Bellman-Ford Algorithm	95
6.4.5	Discussion: Shortest Paths, Longest Paths and Time Complexity	96
6.5	Other Issues	97
6.6	Bibliographic Notes	99
6.7	Exercises	99
7	Placement and Partitioning	101
7.1	Circuit Representation	102
7.2	Wire Length Estimation	105
7.3	Types of Placement Problem	106
7.4	Placement Algorithms	108
7.4.1	Constructive Placement	108
7.4.2	Iterative Improvement	109
7.5	Partitioning	112
7.5.1	The Kernighan-Lin Partitioning Algorithm	112
7.6	Bibliographic Notes	117
7.7	Exercises	118
8	Floorplanning	119
8.1	Floorplanning Concepts	121
8.1.1	Terminology and Floorplan Representation	121
8.1.2	Optimization Problems in Floorplanning	124
8.2	Shape Functions and Floorplan Sizing	125
8.3	Bibliographic Notes	130
8.4	Exercises	131
9	Routing	133
9.1	Types of Local Routing Problems	133
9.2	Area Routing	134
9.3	Channel Routing	138
9.3.1	Channel Routing Models	139
9.3.2	The Vertical Constraint Graph	140
9.3.3	Horizontal Constraints and the Left-edge Algorithm	143
9.3.4	Channel Routing Algorithms	146
9.4	Introduction to Global Routing	150
9.4.1	Standard-cell layout	151
9.4.2	Building-block Layout and Channel Ordering	153
9.5	Algorithms for Global Routing	154
9.5.1	Problem Definition and Discussion	155
9.5.2	Efficient Rectilinear Steiner-tree Construction	157

9.5.3 Local Transformations for Global Routing	163
9.6 Bibliographic Notes	163
9.7 Exercises	166
10 Simulation	167
10.1 General Remarks on VLSI Simulation	167
10.2 Gate-level Modeling and Simulation	169
10.2.1 Signal Modeling	170
10.2.2 Gate Modeling	171
10.2.3 Delay Modeling	171
10.2.4 Connectivity Modeling	172
10.2.5 Compiler-driven Simulation	173
10.2.6 Event-driven Simulation	176
10.3 Switch-level Modeling and Simulation	180
10.3.1 Connectivity and Signal Modeling	181
10.3.2 Simulation Mechanisms	183
10.4 Bibliographic Notes	191
10.5 Exercises	192
11 Logic Synthesis and Verification	195
11.1 Introduction to Combinational Logic Synthesis	195
11.1.1 Basic Issues and Terminology	195
11.1.2 A Practical Example	199
11.2 Binary-decision Diagrams	201
11.2.1 ROBDD Principles	201
11.2.2 ROBDD Implementation and Construction	206
11.2.3 ROBDD Manipulation	208
11.2.4 Variable Ordering	215
11.2.5 Applications to Verification	217
11.2.6 Applications to Combinatorial Optimization	219
11.3 Two-level Logic Synthesis	222
11.3.1 Problem Definition and Analysis	222
11.3.2 A Heuristic Based on ROBDDs	225
11.4 Bibliographic Notes	230
11.5 Exercises	232
12 High-level Synthesis	235
12.1 Hardware Models for High-level Synthesis	235
12.1.1 Hardware for Computations, Data Storage, and Interconnection	236
12.1.2 Data, Control, and Clocks	238
12.2 Internal representation of the Input Algorithm	239

12.2.1 Simple Data Flow	239
12.2.2 Conditional Data Flow	241
12.2.3 Iterative Data Flow	243
12.2.4 Data-flow Graph Representation	245
12.3 Allocation, Assignment and Scheduling	247
12.3.1 Goals and Terminology	247
12.3.2 A detailed Example	248
12.3.3 Optimization Issues	251
12.4 Some Scheduling Algorithms	253
12.4.1 ASAP Scheduling	253
12.4.2 Mobility-based Scheduling	254
12.4.3 Force-directed Scheduling	256
12.4.4 List Scheduling	259
12.5 Some Aspects of the Assignment Problem	261
12.5.1 Optimization Issues	261
12.5.2 Graph Theoretical problem Formulation	262
12.5.3 Assignment by Interval and Circular-arc Graph Coloring	264
12.5.4 Assignment by Clique Partitioning	265
12.6 High-level Transformations	266
12.7 Bibliographic Notes	271
12.8 Exercises	273
III Appendices	275
Appendix A CMOS Technology	277
A.1 The MOS Transistor and CMOS Logic Design	278
A.2 Transistor Layout in CMOS and Related Issues	282
A.3 Bibliographic Notes	285
Appendix B About the Pseudo-code Notation	287
B.1 Data Structures and Declarations	288
B.2 C-language Constructs	289
B.3 Pseudo-code Constructs	292
B.4 Bibliographic Notes	294
Appendix C List of Acronyms	295
References	297
Index	315

Preface

Algorithms for VLSI Design Automation intends to show current and future users of VLSI CAD tools what is going on inside these tools. This should lead to insight into what tasks can or cannot typically be performed by such tools, and why some problems can only be approximately solved after long computation times while others are exactly solved in a short time. A secondary goal is to provide a first introduction to those students that want to specialize in the development of the tools themselves.

The book is targeted firstly at students of electrical engineering. It assumes only an elementary knowledge of programming and some familiarity with digital IC design. However, the necessary knowledge of IC design is quite minimal and students of computer science or applied mathematics should be perfectly able to follow the text after reading Appendix A that explains the very basics of CMOS technology. The book is also interesting for computer scientists and applied mathematicians as it shows the many applications of combinatorial optimization within the field of VLSI design automation.

After studying this book, the students should be sufficiently familiar with the notions and terminology of the field and be able to understand more specialized books and articles on their own. It is recommended that the study of the book is supplemented with programming exercises such that the student will not only understand typical CAD algorithms but be able to implement them efficiently as well.

The book consists of two groups of chapters and a group of appendices. The first group of chapters consists of introductions to "VLSI design" and "CAD tools", followed by introductions to the mathematical topics of "algorithmic graph theory", "computational complexity", "intractability" and "general methods for combinatorial optimization". The mathematical introductions have been included because many students of electrical engineering may not be familiar with them.

The second group of chapters presents a selection of CAD problems and algorithms to solve them. Although attention is paid to simulation, logic synthesis, high level synthesis, and several aspects of layout design, the wide range of VLSI design automation tools is only partially covered by these chapters. The reason for this is that I consider it more important to achieve some depth in a limited number of topics rather than to have a shallow coverage of many topics. Besides, a more complete but superficial presentation of all tools is given in Chapter 2 entitled "A Quick Tour of VLSI Design Automation Tools". Another reason for not attempting to cover a wide range of tools and algorithms is that the focus of research in the field continuously moves to new topics. The reader will have to consult the most recent

literature in order to learn about them. I believe that the material presented in this book sufficiently prepares the reader for the study of other texts dealing with VLSI design automation, whether they are provided by the teacher as additional material or collected by the reader. Many pointers for further reading can be found in the "Bibliographic Notes" sections at the end of each chapter. These sections list review texts, texts that present example implementations, and texts that have served as a source of information for me.

Apart from the first appendix on the basics of CMOS technology mentioned earlier, there is an appendix that presents the language that is used in the book for the specification of algorithms in pseudo-code. The language is based on C with some extensions to write compact pseudo-code. The last appendix lists all acronyms used in this book.

Algorithms are the central theme of this book and the book presents many of them. Most of the algorithms are illustrated by means of small examples to make it easier to understand them. The choice to include some of them was a natural one because their relevance to the field is undisputed. In other cases, I have made a choice among the many alternative algorithms that solve the same problem. This was a subjective choice based on issues like complexity, elegance, relation with other algorithms in the text, etc. In all cases, the reader should realize that there is still a large gap between the algorithms as presented in this book and actual implementations inside tools due to many practical details that have been left out of consideration. The algorithms in this book should not be considered recipes that can directly be applied. The goal of the book is rather to train the student in thinking about algorithms related to the field.

A WWW page is available containing additional information related to the book (additional exercises, useful links, supplementary material for teachers, etc). It can be found at:

<http://utelnt.e1.utwente.nl/links/gerez/cadvlsi/book.html>

This page (and future versions possibly located elsewhere) can also be reached from the publisher's site:

<http://www.wiley.com/college/wave/>

Acknowledgements

This book has evolved from the lecture notes of the course that I teach at Twente University. Throughout the years, colleagues and students have detected mistakes and suggested modifications to the text. Although some of their remarks may have referred to parts of the text that have not made it into this last version, I would like to thank all those that were willing to give me feedback. Three persons have carefully read large parts of the almost final manuscript. I am very grateful to Jaap Hofstede (University of Twente, Department of Computer Science), Marc Heiljilgers (Philips Research) and Nick van der Meijs (Delft University of Technology) for their valuable comments.

The following figures were published earlier in [Ger98] and are reprinted here with kind permission of Kluwer Academic Publishers: 9.12, 12.13, 12.15, 12.16, 12.30, and 12.35. Figures 10.11 and 10.15 originate from [Bry87a] and are reprinted here with kind permission of the IEEE.

Of course, the publication of this book would not have been possible without the support of the staff of *John Wiley and Sons*. I would like to thank Simon Plumtree, Robert Hambrook, and their colleagues for their readiness to answer all my questions and their flexible interpretation of deadlines.

It is common that authors apologize to their families for not having been available often during the writing of their book. Given the fact that my partner Sonia Heemstra is as heavily loaded as myself with a full-time job, asking for sacrifices from her side was hardly possible. It is important to me that Sonia and our two children continually provide me with inspiration for enjoying my private and professional life and that they have motivated me in this way to undertake the task of writing a book. I would finally like to thank my parents for encouraging me to study and work hard throughout my life.

Sabih H.Gerez (s.h.gerez@el.utwente.nl)
Enschede, September 9, 1998

Part I

Preliminaries

1

Introduction to Design Methodologies

This text deals with the algorithms that are used inside VLSI design automation tools, also called computer-aided design (CAD) tools. It does not make sense to discuss internals of VLSI design automation tools without having a clear idea of how these tools are used. For this reason, a brief review of VLSI *design methodologies* is given in this first chapter. The term "design methodology" refers to the approach followed to solve the VLSI design problem. The discussion in this chapter is rather abstract. A review of VLSI design automation with concrete references to the subproblems to be solved and the tools used follows in Chapter 2. A minimal knowledge of CMOS technology is necessary to understand this chapter and most of the other chapters in this text. Appendix A supplies this knowledge for those that are unfamiliar with the topic.

This chapter is organized as follows. First, the different entities to be optimized during VLSI design are discussed. Then some attention is paid to the three VLSI design domains and the design actions. The chapter concludes with a short discussion of design methods and technologies.

1.1 The VLSI Design Problem

As is probably known to the reader, the abbreviation VLSI stands for *Very Large Scale Integration*, which refers to those integrated circuits that contain more than 10^5 transistors (in current-day technologies, circuits of 10^7 transistors can already be produced). The circuits designed may be *general-purpose integrated circuits* such as microprocessors, digital signal processors, and memories. They are characterized by a wide range of applications in which they can be used. They may also be *application-specific integrated circuits* (ASICs) which are designed for a narrow range of applications (or even a single one).

Designing such a circuit is a difficult task. A first requirement is, of course, that a given specification is realized. Besides this, there are different entities that one would like to optimize. These entities can often not be optimized simultaneously (one can

only improve one entity at the expense of one or more others). The most important entities are:

- **Area.** Minimization of the chip area is not only important because less silicon is used but also because the *yield* is in general increased. Not all circuits that are manufactured function properly: the *yield* is the percentage of correct circuits. Causes of failure, like crystal defects, defects in the masks, defects due to contact with dust particles, etc. are less likely to affect a chip when its area is smaller.
- **Speed.** The faster a circuit performs its intended computation, the more attractive it may be to make use of it. Increasing the operation speed will normally require a larger area (one may e.g. duplicate the hardware in order to parallelize the computation). The design process should, therefore, always carefully consider the trade-off between speed and area. Often, the operation speed is part of the specification and the area should be minimized without violating this specification. Speed is then a *design constraint* rather than an entity to optimize.
- **Power dissipation.** When a chip dissipates too much power, it will either become too hot and cease working or will need extra (expensive) cooling. Besides, there is a special category of applications, viz. portable equipment powered by batteries, for which a low power consumption is of primary importance. Here again there are trade-offs: designing for low power may e.g. lead to an increase in the chip area.
- **Design time.** The design of an integrated circuit is almost never a goal on its own; it is an economical activity. So, a chip satisfying the specifications should be available as soon as possible. The design costs are an important factor, especially when only a small number of chips need to be manufactured. Of course, good CAD tools help to shorten the design time considerably as does the use of semicustom design (see Section 1.4).
- **Testability.** As a significant percentage of the chips fabricated is expected to be defective, all of them have to be tested before being used in a product. It is important that a chip is easily testable as testing equipment is expensive. This asks for the minimization of the time spent to test a single chip. Often, increasing the testability of a chip implies an increase in its area.

One can combine all these entities into a single *cost function*, the *VLSI cost function*. It is impossible to try to design a VLSI circuit at one go while at the same time optimizing the cost function. The complexity is simply too high. Two main concepts that are helpful to deal with this complexity are *hierarchy* and *abstraction*. Hierarchy shows the structure of a design at different levels of description. Abstraction hides the lower level details. The use of abstraction makes it possible to reason about a limited number of interacting parts at each level in the hierarchy. Each part is itself composed of interacting subparts at a lower level of abstraction. This decomposition continues until the basic building blocks (e.g. transistors) of a VLSI circuit are reached. Figure 1.1 illustrates the concepts of hierarchy and abstraction. Figure 1.1(a)

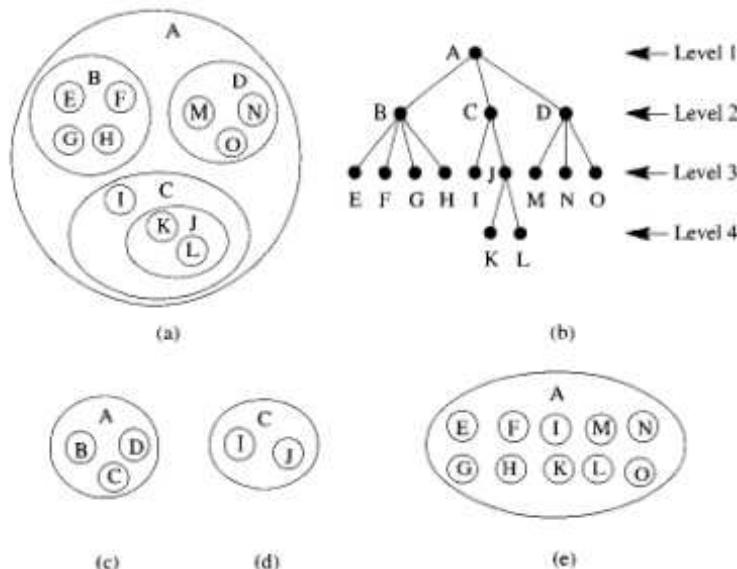


Figure 1.1 A stylized view of a design (a), its decomposition tree (b), a view of the design decomposition of entity *A* at Abstraction Levels 1 and 2 (c), of entity *C* at Levels 2 and 3 (d) and the whole design without a hierarchical organization (e).

shows a design and the way its parts have been partitioned hierarchically. A hierarchical design can be visualized by a *decomposition tree* as is shown in Figure 1.1(b). Figure 1.1(c) shows the entire design as it is seen when leaving out the details below Abstraction Level 2. Similarly, Figure 1.1(d) shows part *C* of the design as seen at Levels 2 and 3. The situation without a hierarchical organization is pictured in Figure 1.1(e) where the ten primitive entities are seen as direct parts of the top-level entity. If there are in the order of a million primitives instead of ten, as is the case for a VLSI circuit, the situation without hierarchy clearly becomes unmanageable.

1.2 The Design Domains

A single hierarchy is not sufficient to properly describe the VLSI design process. There is a general consensus to distinguish *three* design domains, each with its own hierarchy. These domains are:

- *The behavioral domain.* In this domain, a part of the design (or the whole) is seen as a black box; the relations between outputs and inputs are given without a reference to the implementation of these relations. A behavioral description at the transistor level is e.g. an equation giving the channel current as a function of the voltages at source, drain and gate or the description of a transistor as ideal switch. At a higher level, a design unit with the complexity of several transistors can easily be described by means of expressions in Boolean algebra or by means

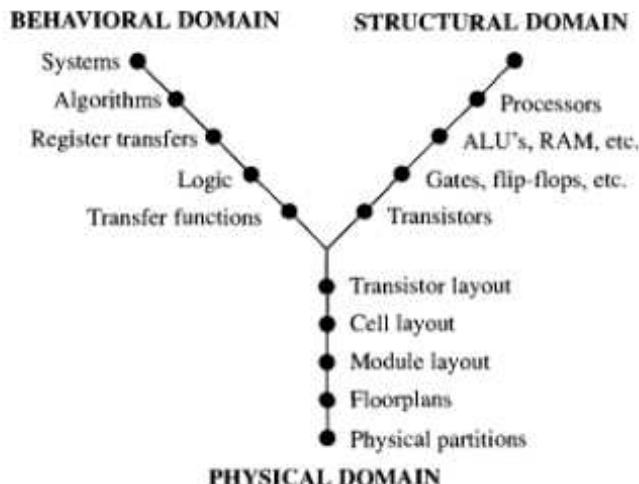


Figure 1.2 The visualization of the three design domains in Gajski's Y-chart.

of truth tables. Going up in abstraction level, one reaches the *register-transfer level*, where a circuit is seen as sequential logic consisting of memory elements (registers) and functions that compute the next state given the current memory state. The highest behavioral descriptions are algorithms that may not even refer to the hardware that will realize the computation described.

- *The structural domain.* Here, a circuit is seen as the composition of subcircuits. A description in this domain gives information on the subcircuits used and the way they are interconnected. Each of the subcircuits has a description in the behavioral domain or a description in the structural domain itself (or both). A schematic showing how transistors should be interconnected to form a NAND gate is an example of a structural description, as is the schematic showing how this NAND gate can be combined with other logic gates to form some arithmetic circuit.
- *The physical (or layout) domain.* A VLSI circuit always has to be realized on a chip which is essentially two-dimensional. The physical domain gives information on how the subparts that can be seen in the structural domain, are located on the two-dimensional plane. For example, a *cell* that may represent the layout of a logic gate will consist of mask patterns that form the transistors of this gate and the interconnections within the gate.

The three domains and their hierarchies can be visualized on a so-called *Y-chart* as depicted in Figure 1.2. Each axis represents a design domain and the level of abstraction decreases from the outside to the center. It was introduced by Gajski in 1983 and has been widely used since then.

Apart from showing the three design domains in one picture, the Y-chart is a powerful tool to illustrate different design methodologies. An example is shown in

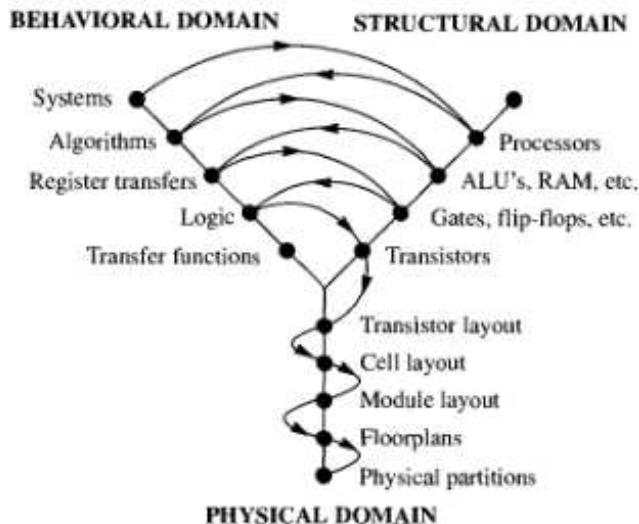


Figure 1.3 A design methodology based on top-down structural decomposition and bottom-up layout reconstruction illustrated by means of the Y-chart.

Figure 1.3. The path drawn in the Y-chart illustrates a top-down design methodology where parts with known behavior are decomposed into smaller blocks with simpler behavior and an interconnection structure. This corresponds with a transition from the behavioral to the structural domain. Each subpart can again be thought to be located on the behavioral axis and is decomposed in its own turn. This process continues until a sufficiently low level of abstraction is reached. The layout has not been considered in this process. Once the circuit has been fully specified down to the lowest structural level, the layout is specified in a bottom-up fashion: transistors are grouped to form cells, cells are grouped to form modules, etc.

Of course, many other design methodologies exist. It is, for example, possible to design in a fully top-down manner, where the layout is determined immediately after structural decomposition. This corresponds with an inward spiral on the Y-chart. The method will be discussed in more detail in Chapter 8 when dealing with *floorplanning*.

1.3 Design Actions

The actions involved to design a VLSI circuit can be grouped in different categories according to their types. These categories will be shortly reviewed in this section as different types of activities require different types of CAD tools.

Some of the design actions can clearly be visualized as a transition in the Y-chart either within a single domain or from one domain to another. These are *synthesis* steps; they add detail to the current state of the design. Synthesis steps may be

performed fully automatically by some synthesis tool or manually by the designer (there may exist interactive tools that support the designer in taking some design decisions and capturing their results).

Ideally, a synthesis step is *correct by construction*, which means that the transformation required to obtain the new description from the old one has the property that it preserves the behavior of the subcircuit involved. When synthesis steps do not lead to a result that is correct by construction, there is a necessity to use *verification* tools. Steps that were performed fully automatically can often be trusted to be correct. However, as human skills and creativity cannot always simply be replaced by computer programs, human intervention is necessary in several stages of the design process. On the other hand, a weak point of humans is that they make mistakes from time to time. Most of these mistakes can be detected by computer programs. In some cases, even a fully-automatic synthesis step should be verified, especially when the synthesis tool itself is very complex and is likely to be imperfect.

Another category of tools consists of *analysis* tools. They provide data on the quality of the design (e.g. how fast is the circuit? how large is its area?) and hints on how to optimize the design (e.g. which part determines the maximal speed and should receive more attention by the designer?).

Yet another important category of tools is the group of *optimization* tools. These improve the quality of a design without necessarily making a transition to another level of abstraction or design domain. For example, a logic optimization tool can replace a set of Boolean expressions by an equivalent set which is cheaper to realize. It should be noted that most synthesis tools need to perform some type of optimization in order to be interesting and useful.

Apart from these tools, there are tools that do not directly contribute to the design itself, but support the other tools: the so-called *design management* tools. They take care of design data storage, tool communication, the invocation of tools in the right order, etc.

1.4 Design Methods and Technologies

As has already been mentioned, the design of VLSI circuits is a complex process. The more degrees of freedom there are, the bigger the search space for the optimal design. In *full-custom* design there is maximal freedom: the designer has the ability to determine the shape of every mask layer for the production of the chip.

Although it may make inaccessible the parts of the search space which contain optimal designs, limiting the freedom of the designer has the advantage of a smaller search space and, therefore, a shorter design time. Design methods with limited freedom are referred to by the term *semicustom*. As explained below, semicustom design implies the use of *gate arrays*, *standard cells*, *parameterizable modules* or a combination of the three.

Many manufacturers of integrated circuits make available chips that have all their transistors preplaced in regular patterns. The designer only needs to specify the wiring patterns (in one or more layers of metal) to interconnect these transistors.

Circuits of this type are called *gate arrays*. The older types have groups of transistors separated by wiring channels. More modern versions do not have wiring channels: wiring is done on top of the transistors, sometimes at the expense of a lower transistor utilization. This type of gate array is called a *sea of gates*. Gate arrays allow the manufacturer to prefabricate chips without metallization in large quantities. As only the metal patterns are customer specific, both the price and the turn-around time can be less than in full-custom fabrication.

One can say that gate arrays as described above are *mask programmable*: by choosing the right mask patterns for the realization of the connections, one fixes the functionality of the circuit. There also exist so-called *field-programmable gate arrays* (FPGAs). These are chips that have been fully fabricated, but whose interconnections can temporarily or permanently be configured by applying electrical signals on some inputs. So, as opposed to mask-programmable gate arrays, they can be programmable in the field, after they have left the factory.

Besides replacing transistors, a faster design time can be achieved by combining elementary circuits, so-called *standard cells*, (e.g. simple logic gates, flip-flops, etc.) that have been predesigned and have been made available to the designer in a *library*. In such a case, one can build a complete design from cells, which saves the cumbersome low-level design steps. Working with standard cells is also considered semicustom design. It can be done both when the fabrication is to be full custom or when the realization will be on a gate array.

Another way of increasing the design productivity is the use of *module generators*. These generators exist for those subcircuits in a design that have a regular structure, such as adders, multipliers, and memories. Such a subcircuit is called a *module*. Due to the regularity of the structure, the module can be described by one or two parameters like the number of bits of an adder or the word length and number of words in a memory. These parameters are sufficient for the module generator to build the desired module from a set of elementary circuits. These elementary circuits could have been designed in a full-custom fashion, or they could be composed from standard cells (it may even be likely that the elementary circuit may be available as a single cell in a standard cell library).

The term *technology* (sometimes *fabrication technology*) refers to the semiconductor process used to produce a circuit. It can refer to the type of semiconductor (Si or GaAs), to the type of transistors (e.g. CMOS, nMOS, bipolar) or to the details of a certain transistor technology (e.g. 1 micron CMOS of manufacturer X).

Clearly the design method and the technology that are chosen, can have consequences for the CAD tools to be used. They even create the necessity for specific tools (e.g. a cell generator for sea-of-gates). An overview of the most common CAD tools is given in the next chapter.

1.5 Bibliographic Notes

VLSI design is the topic of many books including [Wes93], [Wol94b] and [Rab96]. A comprehensive book on both VLSI design and modern CAD tools is [Smi97].

Recently designing for *low power* has received quite some attention; the interested reader can consult the books [Cha95a], [Mac96], [Rab95], [Neb97] and [Ben98] or the review paper [Ped96] for more information on this topic.

Two interesting publications on the notions of hierarchy and abstraction in the context of VLSI design are [Tri81] and [Séq83]. The original paper in which Gajski proposed to visualize the three design domains by means of the Y-chart is [Gaj83]. The paper also illustrates how different design methodologies can be expressed as paths in the Y-chart. Similar ideas can be found in later publications by Gajski like [Gaj88b]) as well as publications by other authors (see e.g. [Wal85]).

2

A Quick Tour of VLSI Design Automation Tools

As mentioned in Chapter 1, designing an integrated circuit is a sequence of many actions most of which can be done by computer tools. It is the goal of this chapter to briefly mention the most relevant tools. Only a few of these tools will receive detailed attention later on in this text. For those tools, a pointer to the appropriate chapter will be provided. For most of the other tools, references to relevant sources are given in the Bibliographic Notes at the end of the chapter.

In order to keep their discussion in this chapter somewhat structured, the tools have been grouped according to the keywords "algorithmic and system design", "structural and logic design", "transistor-level design", "layout design", "verification" and "design management". The first four groups more or less completely cover the Y-chart (see Section 1.2) as is shown in Figure 2.1. "Verification" is action that occurs almost anywhere in the Y-chart, whereas "design management" does not deal directly with a specific design and cannot be shown on the Y-chart. The tools are not necessarily discussed in the order that they should be invoked during a design.

2.1 Algorithmic and System Design

At the earliest stage of the design, there is a necessity to experiment with specifications, to try to formalize them, etc. The designer is mainly concerned with the initial algorithm to be implemented in hardware and works with a purely behavioral description of it. Some designers use general-purpose programming languages like *C* or *Pascal* at this stage. However, it becomes more and more popular to use so-called *hardware description languages* (HDLs). Being specially created for this goal, they allow for a more natural description of hardware. For example, the statements of a program written in a general-purpose programming language are supposed to be executed sequentially whereas the semantics of HDLs imply parallel execution. Many HDLs have been designed in the past, both as part of commercial tools and for research purposes. Currently, the languages *VHDL* and *Verilog* are the most widely used.

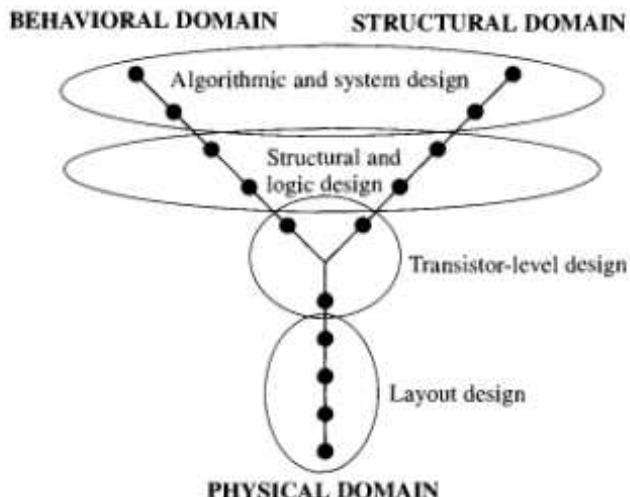


Figure 2.1 The location in the Y-chart of the tool groups as discussed in this chapter.

A formal description of hardware by means of an HDL already helps to make an unambiguous specification as opposed to a specification in a natural language like English. However, an HDL becomes more interesting when a *simulator* is available for it. Simulation helps in the detection of errors in the specification and allows the comparison of the highest-level description with more detailed versions of the designs that are created during the design process. A second application of formal description is the possibility of automatic *synthesis*: a "synthesizer" reads the description and generates an equivalent description of the design at a much lower level. Such a low-level description may e.g. consist of a set of interconnected standard cells. The degree of abstraction at which the input to the synthesis tool is given determines the power of the tool: the higher this level, the less the number of design steps to be performed by the human designer. The synthesis from the algorithmic behavioral level to structural descriptions consisting of arithmetic hardware elements, memories and wiring is called *high-level synthesis* and is discussed in Chapter 12. Yet another application of formal descriptions is in *formal verification*, a topic that is shortly discussed in Section 2.5.

The term *silicon compiler* came already into existence in the early days of VLSI design when CAD tools were relatively primitive compared to what they are now. The goal was to construct a tool similar to a compiler for software by reasoning that mapping a computation to the instruction set of a general-purpose computer was not very different from mapping the computation to hardware. A similar program to the one that served as the input for a software compiler could be used as the input for the "silicon compiler" which would produce the mask patterns for a chip. This ideal is being approximated more and more by the synthesis tools mentioned above.

A formal specification does not always need to be in a textual form by means

of a hardware description language. Tools exist to capture part of the specification in a graphical way, e.g. in the case that structural information is available. Another situation in which graphical entry may be preferable above text is for the specification of finite state machines (FSMs). Especially *hierarchical FSMs* in which some states may be hierarchical FSMs themselves, are useful for the specification of so-called *control-dominated applications*. The tools generally have the possibility to convert the graphical information into a textual equivalent expressed in a language like VHDL that can be accepted as input by a synthesis tool.

Design starting from a system specification will normally not result in a single ASIC. It is much more realistic that the final design for a complex system will consist of several chips, some of which are programmable. In such a case, the design process involves decisions on which part of the specification will be realized in hardware and which in software. Such a design process is called *hardware-software co-design*. One of the main tasks in this process is the partitioning of the initial specification in software and hardware parts. This task is very difficult to automate, but tools exist that support the designer, e.g. by providing information on the frequency at which each part of the specification is executed. Clearly, the parts with the highest frequencies are the most likely to be realized in hardware. The result of co-design is a pair of descriptions: one of the hardware (e.g. in VHDL) that will contain programmable parts, and the other of the software (e.g. in C). Mapping the high-level descriptions of the software to the low-level instructions of the programmable hardware is a CAD problem of its own and is called *code generation*. One possibility for the verification of the correctness of the result of co-design is simulation. Because the simulator should be able to cope simultaneously with descriptions of hardware and software, this process is called *hardware-software co-simulation*.

2.2 Structural and Logic Design

In many situations, it is not possible to provide a high-level description of a circuit and leave the rest of the design to synthesis tools: the tools might not be able to cope with the desired behavior or may produce results whose quality is unacceptable. In such a case, the designer can use a *schematic editor* program. This CAD tool allows the interactive specification of the blocks composing a circuit and their interconnections by means of a graphics computer screen, mouse, menus etc. Often, the schematics constructed in this way are hierarchical: a block at one level is an interconnection of blocks one level lower. The blocks at the lowest level are normally elementary logic gates (e.g. a 3-input NAND or a D-flipflop), although more abstract (e.g. an adder) or more detailed (e.g. a transistor) blocks could form the lowest level as well.

Once the circuit schematics have been captured by an editor, it is a common practice to verify the circuit by means of *simulation*. Simulation may even be the only reason why the circuit schematics were drawn on the computer (documentation may be another reason). Simulation is performed to detect errors in the design and increase the confidence in the correct functioning of the circuit. Chapter 10 discusses

the most important techniques that are used in a simulator. A topic closely related to simulation is *fault simulation*: one checks whether a set of *test vectors* or *test patterns* (input signals used for testing) will be able to detect faults caused by imperfections of the fabrication process. Going one step further, one could let the computer search for the best set of test vectors by using a tool for *automatic test-pattern generation* (ATPG).

Logic synthesis is concerned with the generation and optimization of a circuit at the level of Boolean gates. In this field, three different types of problems can roughly be distinguished:

1. *Synthesis of two-level combinational logic.* As the reader should know, any Boolean function can be written as a two-level expression, e.g. as a sum of products or a product of sums. These forms can e.g. directly be implemented as *programmable logic arrays* (PLAs), a specific regular arrangement of transistors. It is, therefore, important to minimize two-level expressions.
2. *Synthesis of multilevel combinational logic.* Some parts of integrated circuits consist of so-called *random logic*. This is the circuitry that does not have the regular structure that can be found in e.g. adders, multipliers, etc. Random logic is often built of standard cells, which means that the implementation does not restrict the depth of the logic (the maximal number of gates between an input and an output). Using more than two levels normally decreases the area but increases the propagation delay. The goal is often to minimize the area while satisfying delay constraints.
3. *Synthesis of sequential logic.* Opposite to combinational logic, sequential logic has a *state* which is normally stored in memory elements such as flip-flops. One of the main problem here is to find a state encoding such that the logic necessary to compute the state transitions is minimized.

In general a large library of distinct logic gates are available for the realization of digital circuitry. Logic synthesis algorithms normally do not deal with the library directly in order to be as independent as possible of the available technologies and libraries. An abstract circuit representation is, therefore, used instead during the initial stages of the synthesis process. Once that the circuit is estimated to satisfy the optimization constraints, it is converted into a circuit composed of actually available library cells by a *technology mapping* tool. Chapter 11 deals with the synthesis of combinational logic, concentrating mainly on two-level synthesis.

In Section 1.1 it was mentioned that the speed of the circuit to be designed is either an entity to be optimized or a constraint that should be satisfied. In both cases, the designer should be informed about the maximum delay paths in the circuit: the shorter these delays, the faster the operation of the circuit. One possibility of finding out about these delays is by means of simulation, using methods that model the delays with sufficient accuracy. Another, more efficient, possibility is the use of *timing analysis* tools. These are tools that are able to compute delays through the circuit without performing any simulation.

2.3 Transistor-level Design

Logic gates are composed of transistors. Designing at the transistor level requires its own design tools, most of which are simulation tools. Depending on the accuracy required, transistors can be simulated at different levels. At the *switch level*, transistors are modeled as ideal bidirectional switches and the signals are essentially digital, although the model is often augmented to deal with different signal strengths, capacitances of nodes, etc. (see Chapter 10). At the *timing level*, analog signals are considered, but the transistors have simple models (e.g. piecewise linear functions). At the *circuit level*, more accurate models of the transistors are used which often involve nonlinear differential equations for the currents and voltages. The equations are then solved by numerical integration. The more accurate the model, the more computer time is necessary for simulation and, therefore, the lower the maximum size of the circuit that can be simulated in reasonable time.

The fact that an integrated circuit will be realized in a mainly two-dimensional physical medium has implications for design decisions at many levels. This is certainly true for the transistor level. The exact values of (parasitic) capacitances and resistors depend on the shapes of the patterns in the different layers of material. Therefore, it is the custom to *extract* the circuit from the layout data, i.e. to construct the network of transistors, resistors and capacitances taking the mask patterns as inputs (see also Section 2.4). The extracted circuit can then be simulated at the circuit or switch level especially to find out how parasitic capacitances and resistors affect the circuit behavior. Circuit extraction is especially important when performing full-custom design. In the case of standard cells (semicustom design), the so-called *characterization* of the cells, i.e. the determination of their timing behavior is done once by the library developer rather than by the designer who makes use of the library.

2.4 Layout Design

Design actions related to layout are very diverse and there are, therefore, many different layout tools. The most important ones will be discussed in this section.

Suppose that one has the layout of the subblocks of a design available, together with the list of interconnections to be made. From an abstract point of view, these subblocks are rectangles (or polygons) with terminals at their periphery to which wires can be connected. The problem is to compose the layout of the entire integrated circuit. It is often solved in two stages. First, a position in the plane is assigned to each subblock, trying to minimize the area to be occupied by interconnections. This is called the *placement* problem and is discussed in Chapter 7. The next step is to generate the wiring patterns that realize the correct interconnections between these blocks. This is called the *routing* problem and is discussed in Chapter 9. The goal of placement and routing is to generate the minimal chip area, while possibly satisfying some constraints. Constraints may e.g. be derived from timing requirements. As the length of a wire affects the propagation time of a signal along the wire, it may be

important to keep specific wires short in order to guarantee an overall execution speed of the circuit. Layout design with timing constraints is called *timing-driven layout*.

The *partitioning* problem concerns the grouping of the subblocks in a structural description such that those subblocks that are tightly connected are put in the same group while the number of connections from one group to the other is kept low. This problem is not strictly a layout problem. It is e.g. encountered when some design is too large to fit in a single integrated circuit and has to be distributed among several chips. Partitioning can also help to solve the placement problem as is discussed in Chapter 7.

The simultaneous development of structure and layout is called *floorplanning*. In a top-down design methodology, when making a transition of a behavioral description to a structure, one also fixes the relative positions of the subblocks. Through floorplanning, layout information becomes available at early stages of the design. It gives early feedback on e.g. long wires in the layout and may lead to a reconsideration of the decisions on structural decomposition. The floorplanning problem is closely related to the placement problem with the difference that detailed layout information is available in placement whereas floorplanning has mainly to deal with estimations. Floorplanning is discussed in Chapter 8.

A *cell compiler* generates the layout for a network of transistors (consisting of, say, at most 100 transistors). One could follow a placement and routing approach in this case. However, such a two-stage method does not work very well here. As a transistor occupies the same area as a short wire segment, it is very difficult to estimate the area necessary for wiring at the placement stage. Most cell compilers therefore do not exploit the full freedom of being able to place and interconnect transistors arbitrarily. They target regular arrangements of transistors, such as linear or matrix orderings, and use specific optimization techniques suitable for the particular regular arrangement.

A problem somewhat related to cell compilation is *module generation*. A module is normally understood to be a hardware block, the layout of which can be composed by an arrangement of cells from a small subset. These elementary cells are sometimes called *microcells*. They have a complexity of around 10 transistors. Examples of microcells are a full adder or single-bit memory cell. Given some parameters (e.g. the number of bits for an adder or the word length and number of words in a memory), the module generator puts the right cells at the right place and composes a module in this way.

When designing full-custom chips, the designer should have the possibility to modify the layout at the level of mask patterns. The computer tool that supports this action is called a *layout editor*. This tool is essential and any set of tools meant for full-custom design includes it. Its basic function is to allow the insertion, deletion and modification of patterns in specific layers. Most layout editors have additional functions, like the possibility of designing hierarchically and duplicating subcells, which speed up the tedious work of operating at the mask level.

Working at the mask level gives the freedom of manipulating the layout at the lowest level, but the increased freedom is as well a source of errors. In a correct design, the mask patterns should obey some rules, e.g. on minimal distances and the minimal widths, called *design rules* (see Chapter 6). Tools that analyze a layout to detect violations of these rules are called *design-rule checkers*. A somewhat related tool that also takes the mask patterns as its input is the *circuit extractor*. It constructs a circuit of transistors, resistors and capacitances that can be simulated by the methods mentioned in Section 2.3. Both design-rule checking and circuit extraction lean on knowledge from the field called "computational geometry".

One serious disadvantage of full-custom design is that the layout has to be redesigned when the technology changes. Even when the change only involves the minimum feature size (e.g. from a 1 micron process to a 0.5 micron process), often the designs cannot be ported to the new technology by simply scaling dimensions. As a remedy to this problem and to speed up the design time in general, *symbolic layout* has been proposed. In symbolic layout widths and distances of mask patterns are irrelevant. What matters is the positions of the patterns relative to each other, the so-called *topology* of the design. Symbolic layout can only be used in combination with a *compactor*. This is a tool that takes the symbolic description, assigns widths to all patterns and spaces the patterns such that all design rules are satisfied. Compaction is the topic of Chapter 6.

2.5 Verification Methods

There are three ways of checking the correctness of an integrated circuit without actually fabricating it:

1. *Prototyping*, i.e. building the system to be designed from discrete components rather than one or a few integrated circuits. A form of prototyping called *breadboarding* used to be a common practice in the early days of integrated circuit design, but is out of use nowadays, both because of the huge number of components that would be needed and the fact that the behavior of devices on a chip is totally different from that of discrete components when it comes to delays, parasitics, etc. However, prototyping using programmable devices such as *field-programmable gate arrays* (see Section 1.4) is quite popular as a means to investigate the algorithms that a system should realize. This type of prototyping is called *rapid system prototyping* and is especially used in audio and video processing. The prototype is supposed to show the effects of algorithms in *real time*, meaning that the computations should be as fast as in the final design. The advantage of prototyping over simulation (see below) is that simulation will in general not operate in real time. A prerequisite for rapid system prototyping is the availability of a compiler that can "rapidly" map some algorithm on the programmable prototype.
2. *Simulation*, i.e. making a computer model of all relevant aspects of the circuit, executing the model for a set of input signals, and observing the output signals.

Simulation has the disadvantage that it is impossible to have an exhaustive test of a circuit of reasonable size, as the set of all possible input signals and internal states grows too large. One has to be satisfied with a subset that gives sufficient confidence in the correctness of the circuit. So, simulation that does not check all possible input patterns and internal states always includes the risk of overlooking some errors.

3. *Formal verification*, i.e. the use of mathematical methods to *prove* that a circuit is correct. In mathematics it is possible to reason about sets with a large number of elements or even infinite sets without enumerating all elements of the set. So, a mathematical proof, as opposed to simulation, gives certainty on the correctness of the circuit. The problem is that performing these proofs by hand is too time consuming. Therefore, the attention is focused on those techniques that can be performed by computers.

Formal verification methods consider different aspects of VLSI design. The most common problem is to check the equivalence of two descriptions of a circuit, especially a behavioral description and its structural decomposition (see Figure 1.3). In this context the behavioral description is called the *specification* and the structural one its *implementation*. The formal descriptions of specification and implementation are presented to the verifier that manipulates these descriptions and tries to prove that they are equivalent. An introduction to a technique that can be used for this purpose is presented in Chapter 11.

Another problem is to prove the internal consistency of some formal specification, e.g. to check that all possible conditions are covered in the case of a conditional computation. Yet another issue related to formal verification is *correctness by construction*. One can see the design process as the subsequent application of small transformations on an initial design. Each transformation adds some detail to an intermediate design until the final design has been obtained. If one can prove that all transformations that one intends to use preserve the behavior of the system, one can conclude that final design is "correct by construction" (assuming that the initial specifications were correct).

2.6 Design Management Tools

As was mentioned in Chapter 1, there are tools that are not directly related to the progress of the design itself, but are indispensable in a CAD system. First of all, CAD tools consume and produce design data in different design domains and at different levels of abstraction. These data have to be stored in databases. The quantity of data for a VLSI chip can be enormous and appropriate *data management* techniques have to be used to store and retrieve them efficiently. Besides, design is an iterative activity: a designer might modify a design description in several steps and sometimes discard some modifications if they do not satisfy. *Version management* allows for the possibility of undoing some design decisions and proceeding with the design from

some earlier situation without having to copy the entire design description for each design decision taken. Yet another aspect of design management is to maintain a consistent description of the design while multiple designers work on different parts of the design.

One of the earliest *tool integration* problems originated from the different data formats used by distinct tools e.g. due to the fact that the tools were supplied by different vendors. Using the tools in combination asked for data conversion tools such that the tools could use each other's data. In order to keep the number of conversion problems low, efforts have been spent in defining single interchange formats that all tools could read and write. A famous standard format is *EDIF* (Electronic Design Interchange Format).

Data conversion only solves the tool integration problem when the tools that make use of the same data, are invoked sequentially. It is, however, desirable that design data can be shared by tools that are running simultaneously. One would e.g. like to edit a circuit with a schematic editor, simulate it with a tool of another vendor, go back to the schematics, simulate again, without having to bother about data conversion. This is possible if a *framework* is used. One of its features is a uniform interface to access design data from a database. Then all tools can exchange data by making use of standardized procedure calls to read from and write to the common database provided by the framework. A framework can also provide for uniform *user interfaces* using a similar mechanism of standardized procedure calls. The ideal is that a user can buy any tool from any vendor and plug it into the framework to create a personal tool set rather than having to make a choice for one vendor and accept all its tools.

Yet another feature of a framework is *methodology management*. This refers to the possibility to guide the user through the design flow by indicating the order in which some tools should be invoked, e.g. by making sure that some verification program is called after the invocation of a synthesis tool.

2.7 Bibliographic Notes

The reader who is interested in alternative presentations of the suite of CAD tools, the way they can be used, as well as the evolution of the tools throughout the years, can consult the review papers [New81], [Car86], [New86], [Row91] and [Keu97]. Extensive information on modern CAD tools can also be found in [Smi97].

Many books on VHDL exist including [Lip89] and [Nav93]. Synthesis from VHDL is covered in [Bha96], [Cha97], and [Nay97]. Verilog is explained in [Tho91] and [Gol96]. Examples of graphics tools for high-level specifications are presented in [Har90a] and [Ama94]. Silicon compilation is the topic of [Gaj88b] and [Dut90]. An interesting paper that considers the evolution of the ideas for building a commercial silicon compiler over a period of ten years is [Joh89].

Hardware-software co-design has received quite some attention in recent years leading to several books on the topic including [Gup95], [DM96], [Bal97], [Ber97] and [Sta98]. A review paper on the topic is [Wol94c]. Different issues related to code

generation are discussed in [Mar95]. Two monographs on this topic are [Leu97] and [Lie98].

The reader interested in knowing more on fault simulation and ATPG is referred to [Abr90] and [Zob93] among the many sources available on these topics.

A nice book on specification and verification is [Mil94]. More on formal verification can be found in the review sources [Yoe90] and [Gup92]. A discussion on formal methods from a practical point of view is given in [Keu96].

A review of cell compilers is given in [Gaj88a]. Examples of module generators are described in [Chu84], [May86] and [Smi88]. A famous example of a "layout editor" is Magic [Ous85].

General review papers on design-rule checking and circuit extraction are [Yos86] and [Szy88]. A review that emphasizes computational geometry is [Asa86]. Magic's circuit extractor is described in [Sco86]. A complete layout system including editing, design rule checking, extraction, compaction and cell compilation is the topic of [Hil89].

More information on frameworks can be found in the review paper [Har90b] and the books [Bar92] and [Wol94a]. A paper on EDIF is [Kah92a].

3

Algorithmic Graph Theory and Computational Complexity

A *graph* is a mathematical structure that describes a set of objects and the connections between them. The use of graphs may facilitate the mathematical formulation, analysis, and solution of a problem. A road map can, for example, be represented by a graph: points in a plane represent cities and line segments between these points show those cities connected by a direct road. Graphs are often encountered in the field of design automation for integrated circuits, both when dealing with entities that naturally look like a network (e.g. a circuit of transistors) as well as in more abstract cases (e.g. precedence relations in the computations of some algorithm, see Chapter 12).

Algorithmic graph theory, as opposed to *pure* graph theory, emphasizes the design of algorithms that operate on graphs, instead of concentrating on mathematical properties of graphs and theorems expressing those properties. The distinction between the two is not very sharp, however, and algorithmic graph theory certainly benefits from results in pure graph theory.

Computational complexity refers to the time and memory required by a certain algorithm as function of the size of the algorithm's input. The concept applies to algorithms in general and is not restricted to graph algorithms.

The theory that is presented in this chapter consists of the minimum knowledge that will be of help to understand the applications of graph theory to specific problems in the field of design automation for VLSI. After studying this chapter, the reader is supposed to be able to understand the description of graph algorithms, analyze their computational complexity, slightly modify them, and write simple programs that implement graph algorithms (if already sufficiently experienced in programming). The chapter starts with the terminology of graph theory, followed by a discussion of elementary data structures for the representation of graphs. An introduction to the theory of computational complexity follows next. The chapter concludes with a small selection of graph algorithms.

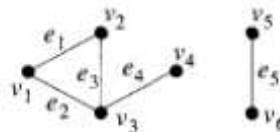


Figure 3.1 An example of a graph.

3.1 Terminology

A graph is characterized by two sets: a *vertex set* V and an *edge set* E . It is customary to denote a graph by $G(V, E)$, where G refers to the graph itself. Not everybody uses the same terms when discussing graphs: a vertex is also called a *node* and an edge is called an *arc* or a *branch*.

The sets V and E fully characterize a graph. Figure 3.1 shows a graph where $V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ and $E = \{e_1, e_2, e_3, e_4, e_5\}$. The two vertices that are joined by an edge are called the edge's *endpoints*. An edge can be identified by its two endpoints $u, v \in V$, in which case the notation (u, v) is used. In the example of Figure 3.1, $e_1 = (v_1, v_2)$, $e_2 = (v_1, v_3)$, etc. When $e = (u, v)$, it is said that e is *incident with* u (and also with v). The vertices u and v such that $(u, v) \in E$, are called *adjacent*.

When one removes vertices and/or edges from a given graph G , one gets a *subgraph* of G . In this process, the following rule should be obeyed: removing a vertex implies the removal of all edges connected to it. More formally: given a graph $G(V, E)$, a subgraph induced by a set $V_s \subset V$ is the graph $G_s(V_s, E_s)$, with $E_s = \{(v_i, v_j) | v_i \in V_s \wedge v_j \in V_s \wedge (v_i, v_j) \in E\}$.

A graph $K(V_k, E_k)$ for which the edge $(v_i, v_j) \in E_k$ for every pair of vertices $v_i, v_j \in V_k$ is called a *complete* graph. A subgraph that is complete, and that is not contained in a larger complete subgraph, is called a *clique*. The graph of Figure 3.1 has three cliques identified by the vertex sets $\{v_1, v_2, v_3\}$, $\{v_3, v_4\}$ and $\{v_5, v_6\}$. Some authors call *any* complete subgraph a clique and one that is not contained in a larger one a *maximal clique*.

The *degree* of a vertex is equal to the number of edges incident with it. An edge (u, u) , i.e. one starting and finishing at the same vertex, is called a *selfloop*. Two edges of the form $e_1 = (v_1, v_2)$ and $e_2 = (v_1, v_2)$, i.e. having the same endpoints, are called *parallel edges*. A graph without selfloops or parallel edges is called a *simple* graph. In most contexts, only simple graphs are discussed and therefore the term *graph* is used instead of simple graph. A graph without selfloops but with parallel edges is called a *multigraph*.

If the vertex set V of a graph is the union of two disjoint sets V_1 and V_2 and all edges of this graph exclusively connect a vertex from V_1 with a vertex V_2 , the graph is called *bipartite*. Such a graph is often denoted by $G(V_1, V_2, E)$. An example is given in Figure 3.2, where the two types of vertices have been colored black and white respectively.

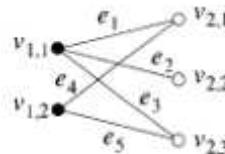


Figure 3.2 An example of a bipartite graph.

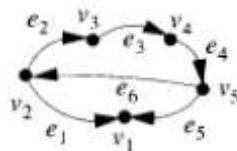


Figure 3.3 An example of a directed graph.

A graph that can be drawn on a two-dimensional plane without any of its edges intersecting is called *planar*. Otherwise, the graph is called *nonplanar*.

A sequence of alternating vertices and edges, starting and finishing with a vertex, such that an edge $e = (u, v)$ is preceded by u and followed by v in the sequence (or vice versa), is called a *path*. In Figure 3.1 v_1, e_1, v_2, e_3, v_3 is an example of a path. The *length* of a path equals the number of edges it contains. Note that a path can have a length of zero. A path, of which the first and last vertices are the same and the length is larger than zero, is called a *cycle* (sometimes also: *loop* or *circuit*). A path or a cycle not containing two or more occurrences of the same vertex is a *simple* path or cycle (except, of course, for the first and last vertices of a cycle). The adjective ‘simple’ is often omitted when all paths and cycles in some context are known to be simple.

Two vertices u and v are called *connected* if there is a path starting at u and finishing at v . In Figure 3.1 v_1 and v_4 are connected but v_1 and v_5 are not. If all pairs of vertices in a graph are connected, the graph is called a *connected graph*. Otherwise, the graph is *disconnected*. The graph of Figure 3.1 is disconnected. It consists of two *connected components*: a connected component is a subgraph induced by a maximal subset of the vertex set, such that all pairs in the set are connected (a “maximal subset” is the largest possible subset obeying some property).

Sometimes a *direction* is associated with the edges in a graph. Such a graph is called a *directed graph* or *digraph*. An example of a directed graph is shown in Figure 3.3. A graph whose edges do not have directions is consequently called an *undirected graph*. The terminology and notation presented above have to be extended for directed graphs. In the notation for an edge $e = (u, v)$, u and v cannot be freely interchanged anymore. The edge e is directed from u to v . It is said that e is *incident from* u and *incident to* v . The *in-degree* of a vertex is equal to the number of edges incident to it; the *out-degree* of an edge is equal to the number of edges incident from

it.

A *path* for a directed graph is defined in the same way as for an undirected graph: in the sequence of alternating vertices and edges, an edge does not need to be directed from the vertex preceding it to the vertex following it. However, if all the edges are directed in this way, the path is called a *directed path*. In the directed graph of Figure 3.3, v_2, e_1, v_1, e_5, v_5 is an example of a path and v_2, e_2, v_3, e_3, v_4 is an example of a directed path. In an analogous way, the terms *cycle* and *directed cycle* can be defined. In the graph of Figure 3.3, $v_2, e_1, v_1, e_5, v_5, e_5, v_2$ is an example of a cycle and $v_2, e_2, v_3, e_3, v_4, e_4, v_5, e_6, v_2$ is an example of a directed cycle. In many discussions related to directed graphs, often only directed paths and directed cycles are relevant and the adjective 'directed' is often omitted when no confusion is possible.

Two vertices u and v in a directed graph are called *strongly connected* if there is both a directed path from u to v and a directed path from v to u . The property of being 'strongly connected' partitions the directed graph into *strongly connected components*. In each such component, all possible pairs of vertices are strongly connected and no other vertex can be added to it that is strongly connected to all those already in the component. In the example of Figure 3.3, there are two strongly connected components: $\{v_1\}$ and $\{v_2, v_3, v_4, v_5\}$. If there is a path between two vertices u and v , the vertices are called *weakly connected*. The property of being 'weakly connected' analogously partitions the graph into *weakly connected components*. In Figure 3.3, there is only one weakly connected component consisting of the entire graph.

Depending on the problem that one wants to solve by means of graphs, a graph can be extended to carry more information. One of the most common extensions is to assign *weights* to the edges of a graph. For example, if a graph is used to represent a road map with the vertices representing cities and edges representing the roads between cities, an edge weight might represent the distance between the two cities connected by a road. A graph of this type is called an *edge-weighted graph*. In some applications, it might be necessary to associate a weight with a vertex; such a graph is called a *vertex-weighted graph*.

The terminology mentioned in this section is just a basic set of terms. More graph theoretical terms will be introduced in other chapters, in the context of specific problems from the field of VLSI design automation.

3.2 Data Structures for the Representation of Graphs

If one wants to implement graph algorithms, one of the first issues to be settled is how to represent the graph in a computer. In other words, one should choose a suitable *data structure* for graphs. There is no optimal data structure that should be used in all algorithms. Different algorithms have different requirements.

One of the most straightforward ways to represent graphs is by means of an *adjacency matrix*. If the graph $G(V, E)$ has n vertices, an $n \times n$ matrix A is used. $A_{ij} = 1$ if $(v_i, v_j) \in E$, and $A_{ij} = 0$ if $(v_i, v_j) \notin E$. As the edges do not have

$$(a) \quad \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad (b) \quad \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Figure 3.4 The adjacency matrices for the graphs of Figure 3.1 (a) and Figure 3.3 (b).

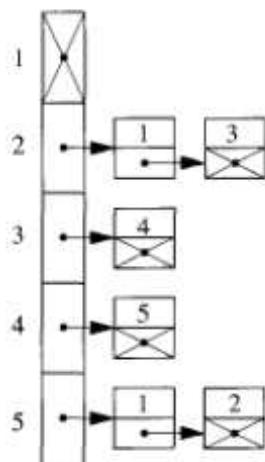


Figure 3.5 The the adjacency list representation of the graph given in Figure 3.3.

directions in undirected graphs, the adjacency matrices of such graphs are symmetric. This data structure is illustrated in Figure 3.4, where the adjacency matrices for the graphs of Figures 3.1 and 3.3 are given.

When using the adjacency matrix, testing whether two given vertices are connected or not can be performed in constant time, i.e. the time to perform the test does not depend on the size of the graph. However, finding all vertices connected to a given vertex requires inspection of a complete row and a complete column of the matrix. This is not very efficient when most of the entries in the matrix are equal to zero.

The *adjacency list* representation is a better choice in such a case. It consists of an array that has as many elements as the number of vertices in the graph. So, an array element identified by an index i corresponds with the vertex v_i . Each array element points to a *linked list* that contains the indices of all vertices to which the vertex corresponding to the element is connected. Figure 3.5 illustrates the adjacency list representation for the graph of Figure 3.3.

```

struct vertex {
    int vertex_index;
    struct edge *outgoing.edges;
};

struct edge {
    int edge_index;
    struct vertex *from, *to;
    struct edge *next;
};

```

Figure 3.6 The data structure for a graph representation with explicit vertex and edge structures.

If an algorithm operates on edges rather than vertices and needs to efficiently identify the vertices that an edge connects, a data structure that is built from explicit structures (records in the *Pascal* jargon), can be used for both vertices and edges. The description in C of this data structure is given in Figure 3.6. It is assumed that edges as well as vertices are identified by an integer number. This number can also be the index in an array pointing to all vertices or all edges. It is supposed that the graph is directed and that it is only required to traverse the graph according to the edge directions. Each vertex points to the list of edges incident from it by means of the member (field in *Pascal*) *outgoing.edges* in the *vertex* structure and the member *next* in the *edge* structure. Each edge points to the two vertices that it connects by means of the members *to* and *from*. This data structure is illustrated in Figure 3.7 for a small directed graph.

The choice of a suitable data structure in the implementation of a graph algorithm can be very important and may directly affect the computational effort necessary to solve some problem. This section has presented the most common data structures for graphs. Given the characteristics of the algorithm to be implemented, one should carefully consider which of the data structures is most appropriate and how these basic data structures should be modified and extended to satisfy the requirements of the algorithm.

3.3 Computational Complexity

A main concern for the design of an algorithm is how the algorithm will perform. How fast will it run when implemented on a computer? How much of the computer's memory will it claim? The theory of computational complexity tries to answer these questions without having to deal with specific hardware and also without providing absolute values in seconds and bytes. Instead, the behavior of an algorithm is characterized by mathematical functions of the algorithm's "input size".

The *input size* or *problem size* of an algorithm is related to the number of symbols necessary to describe the input. Suppose that a sorting algorithm has to sort a list of n words, each consisting of at most 10 letters, then the input size is bounded by $10n$.

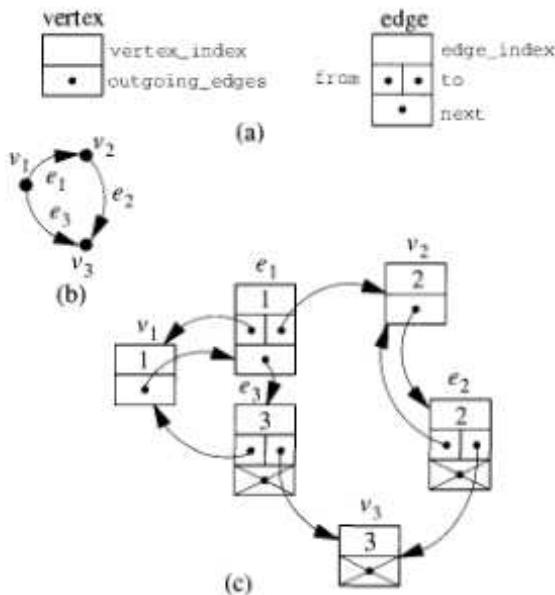


Figure 3.7 The visualization of the `vertex` and `edge` structures (a), a small directed graph (b) and its representation with a data structure built from those structures (c).

Constant factors are not relevant in this context (one wants to avoid technical details such as e.g. the number of bits to code a letter; using the ASCII code that uses 8 bits for each letter, one might as well claim that the input size of the problem is bounded by $80n$). Therefore, constants are eliminated: in the sorting problem the input size is then defined to be n .

A few more examples: an algorithm that takes as input a single natural number n of arbitrary size, has an input size of $\log n$, as the number of symbols in a *reasonable* encoding grows logarithmically with n (a reasonable encoding is e.g. binary or decimal notation; an unreasonable encoding is e.g. unary notation, that uses n bits to encode number n). The input size of a graph algorithm operating on a graph $G(V, E)$ is characterized by two parameters: the size of the vertex set $|V|$ and the size of the edge set $|E|$.

Two types of computational complexity are distinguished: *time complexity*, which is a measure for the time necessary to accomplish a computation, and *space complexity* which is a measure for the amount of memory required for a computation. Space complexity is often given less importance than time complexity (although an algorithm that is using more memory than what is available will not run at all). Because most of the statements made on time complexity in the rest of this text hold as well for space complexity, space complexity will not receive any special attention. In the rest of this text, sometimes "time complexity" will simply be called "complexity".

Before defining the time complexity of an algorithm, it is necessary to introduce

the notion of the *order* of a function. Given two functions f and g that map a natural number n to some positive value, it is said that $f(n) = \mathcal{O}(g(n))$ (pronounced as "f(n) is big O of g(n)" or as "f(n) is of order at most g(n)") if two constants n_0 and K can be found, such that the following proposition is true:

$$\forall n \geq n_0 : f(n) \leq K \cdot g(n)$$

Note that the equals sign in $f(n) = \mathcal{O}(g(n))$ actually indicates set membership, viz. of the set all functions that are $\mathcal{O}(g(n))$. The notation is maintained for historical reasons. Examples:

$$\begin{aligned} \frac{1}{3}n^2 &= \mathcal{O}(n^2) \\ 0.02n^2 + 127n + 1923 &= \mathcal{O}(n^2) \\ 3n \log n + n &= \mathcal{O}(n \log n) \\ 5n^3 &= \mathcal{O}(\frac{1}{3}n^3) \\ 5n^3 &= \mathcal{O}(n^4) \\ 5n^3 &= \mathcal{O}(n^3 + n^2) \end{aligned}$$

Although the last three equalities are correct, they are not used to describe computational complexity. One chooses the order expression to be as simple and as sharp as possible.

The big-O notation describes the upper bound of a function. If one wants to describe a lower bound, the *big-Omega* notation is used: $f(n) = \Omega(g(n))$ (pronunciation: "f(n) is big Omega g(n)" or "f(n) is of order at least g(n)") if $g(n) = \mathcal{O}(f(n))$. The fact that two functions have the same *rate of growth* is indicated by the *big-Theta* notation: $f(n) = \Theta(g(n))$ (pronunciation: "f(n) is big Theta g(n)" or "f(n) is of order exactly g(n)") if $f(n) = \mathcal{O}(g(n))$ and $g(n) = \mathcal{O}(f(n))$. For most algorithms, and especially those that are discussed in this text, the big-O notation is sufficient for the description of their computational complexities.

The duration of a computation is expressed in *elementary computational steps*. This is a simple computation that does not depend on the input size, e.g. the addition, multiplication or comparison of two 32 bit integers, or the access of a record when a pointer to it is available.

The time complexity of a computation is a function that gives the number of elementary computational steps executed for inputs of a specific size. Normally, it is not only the size of the input that determines the number of computational steps: conditional constructs in the algorithm are the reason that the time required by the algorithm is different for different inputs of the same size. Therefore one works with the *worst-case* time complexity, assuming that the condition that requires the largest number of computational steps will be true for an input of a given size.

Apart from the worst-case time complexity, there are other time complexity measures. For example, the *average-case* time complexity is the expected value of the computation time for a given distribution of the distinct inputs of the algorithm. Actually, average-case time complexity has a higher practical value than worst-case time complexity. Its analysis is, however, more complex. For this reason, discussions

on time complexity in the rest of this book will be limited to worst-case time complexity.

Only the order of an algorithm's time complexity is normally relevant. One e.g. says that an algorithm operates in $\mathcal{O}(n^2)$ time. Depending on the magnitude of the input size, a number of different criteria can be used for qualifying an algorithm:

1. *Polynomial vs. exponential order.* As an exponential function grows faster than any polynomial and the exponents of a polynomial tend to be small, an algorithm with a polynomial time complexity is to be preferred over an exponential algorithm. Actually, this distinction corresponds to the one between *tractable* and *intractable* problems that is the topic of Chapter 4.
2. *Linear vs. quadratic order.* Suppose that the input size of an algorithm is determined by the number of transistors in a circuit and that the algorithm has to be applied to VLSI circuit containing some 10^6 transistors. Then, running an algorithm with a linear time complexity is feasible on a computer with a realistic speed, but an algorithm with quadratic time complexity is not. Between these categories are the functions with complexities $\mathcal{O}(n \log n)$, $\mathcal{O}(n \log \log n)$, $\mathcal{O}(n \log^2 n)$, etc. As logarithmic functions grow very slowly, algorithms with this type of time complexities are also acceptable in the above context.
3. *Sublinear order.* When the input of an algorithm is structured in some way, an algorithm might find the solution to some problem without processing all input elements separately. For example, finding an element when the input is already sorted and is available in an array can be done by "binary search" and requires $\mathcal{O}(\log n)$ time. An extreme case is when an algorithm's computation is completely independent of the input size: one says that the algorithm operates in constant time and its complexity is written as $\mathcal{O}(1)$.

A final issue to be mentioned is how to compute an algorithm's time complexity. This can be done in different ways. Direct analysis can lead to an upper bound to the number of computational steps, or the time complexity of the whole can be derived from the complexities of its parts. For example, the time complexity of a loop construct is the time complexity of the loop's body multiplied by the number of times the loop is executed (or an upper bound for this number) and the time complexity of an "if statement" is the sum of the time complexities of its "then" and "else" parts. The rule for multiplication of $f_1 = \mathcal{O}(g_1)$ and $f_2 = \mathcal{O}(g_2)$ is: $f_1 \times f_2 = \mathcal{O}(g_1 \times g_2)$. And for addition of $f_1 = \mathcal{O}(g_1)$ and $f_2 = \mathcal{O}(g_2)$: $f_1 + f_2 = \mathcal{O}(g_1 + g_2)$, followed by a simplification if either of g_1 or g_2 is of a lower order than the other.

3.4 Examples of Graph Algorithms

In this section a number of relatively simple graph algorithms are discussed. The goal of their presentation is twofold: to become acquainted with graph algorithms as such and to apply the theory of computational complexity given above.

```

/* Given is the graph  $G(V, E)$  */

struct vertex {
    ...
    int mark;
};

dfs(struct vertex v)
{
    v.mark ← 0;
    "process  $v$ ";
    for each  $(v, u) \in E$  {
        "process  $(v, u)$ ";
        if ( $u$ .mark)
            dfs( $u$ );
    }
}

main ()
{
    for each  $v \in V$ 
        v.mark ← 1;
    for each  $v \in V$ 
        if ( $v$ .mark)
            dfs( $v$ );
}

```

Figure 3.8 A pseudo-code description of the depth-first search algorithm `dfs`.

3.4.1 Depth-first Search

In many graph algorithms, one needs to traverse the graph in one way or the other and “do something” with the nodes and/or edges that one encounters during the traversal. One systematic way of doing this is by means of *depth-first search*. Another systematic way is by means of *breadth-first search* as is discussed in Section 3.4.2. Depending on the application, both search methods may be equivalent or only one of them may be appropriate. What is actually being done at each vertex and/or edge visited is not specified here. In this respect, the descriptions contain “generic actions” to be filled by the applications. The description in pseudo-code of a version of depth-first search for *directed* graphs is given in Figure 3.8, where the function that performs the depth-first search is called `dfs`.

The goal is to visit all vertices only once. This is achieved by introducing a member `mark` in the `vertex` structure. This member is initialized with the value 1 and given the value 0 when the vertex is visited. As the value is never restored to 1 and only the vertices whose `mark` members have value 1 are visited, it is guaranteed that each vertex is visited at most once. Note that an actual implementation does not necessarily need to use a `struct vertex` data structure. What the presentation

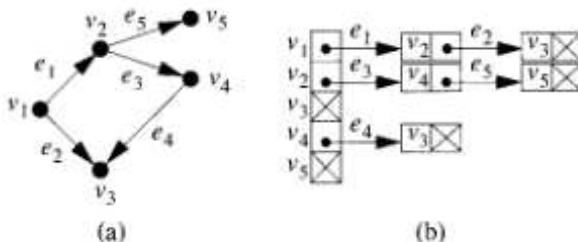


Figure 3.9 A directed graph (a) and its adjacency-list representation (b).

$\text{dfs}(v_1)$	$\text{dfs}(v_2)$	$\text{dfs}(v_4)$	
$\rightarrow e_1 = (v_1, v_2)$	$\rightarrow e_3 = (v_2, v_4)$	$\rightarrow e_4 = (v_4, v_3)$	$\text{dfs}(v_3)$
$\rightarrow e_2 = (v_1, v_3)$	$\rightarrow e_5 = (v_2, v_5)$	$\text{dfs}(v_5)$	

Figure 3.10 The different steps of the depth-first search algorithm applied to the graph of Figure 3.9(a).

in Figure 3.8 tries to convey is that somehow a “mark” *attribute* for a vertex should be provided in the data structures used (a convenient alternative would be to use an array to hold the mark information).

The function `dfs` is a recursive function that takes a vertex as its argument. It “processes” the vertex (the generic vertex action) and then inspects all its outgoing edges one by one. After “processing the edge” (the generic edge action), the vertex to which this edge is incident is used for a recursive call of `dfs`, unless the vertex had already been visited. In the main program, the function is applied to each vertex of the graph to account for the fact that not all vertices may be reachable from a single vertex.

An adjacency-list representation is appropriate in this situation as it gives direct access to the outgoing edges of each vertex. Using this data structure, the analysis of the time complexity of depth-first search is rather simple. It was already mentioned that each vertex is visited exactly once. Since the outgoing edges of a vertex are only visited when the vertex itself is visited, all edges are also visited exactly once (an edge is an outgoing edge of one vertex only). Assuming that the generic vertex and edge actions have a constant time complexity, this leads to a time complexity of $\mathcal{O}(n + |E|)$ for depth-first search, where $n = |V|$.

The idea of depth-first search is illustrated using the graph of Figure 3.9(a) represented by means of its adjacency list shown in Figure 3.9(b). Figure 3.10 shows the evolution of the algorithm applied to the graph. In the figure, the recursion depth increases from left to right. The edges processed as a result of a call to `dfs` at a specific recursion depth are shown in the same column as the function call (preceded

by an arrow). Time increases from left to right and from top to bottom. From the figure, it can be concluded that the vertices are visited in the order v_1, v_2, v_4, v_3 and v_5 while the edges are visited in the order e_1, e_3, e_4, e_5 and e_2 . Because all vertices are reachable from v_1 , the first vertex for which `dfs` is called, there is only a single call to `dfs` at the lowest level of recursion.

The last remark directly indicates a straightforward application of depth-first search: the identification of vertices reachable from a specific vertex $v_s \in V$ for a graph $G(V, E)$. If the main program of Figure 3.8 is modified such that it consists of a single call of `dfs` with argument v_s after setting all `mark` attributes to 1, exactly those vertices that are reachable from v_s will be marked 0 after termination of the function call.

3.4.2 Breadth-first Search

As was mentioned in Section 3.4.1, *breadth-first search* is an alternative to depth-first search for systematically visiting all vertices of a graph. In this section a breadth-first search algorithm will be presented for a situation similar to the one in Section 3.4.1, viz. for directed graphs represented by an adjacency list. A pseudo-code description is provided in Figure 3.11, where the function that performs the actual breadth-first search is called `bfs`.

The central element in the description is the *FIFO queue* (first-in first-out queue) Q . Objects can be added and removed from such a queue in such a way that the order in which objects are removed is identical to the order in which the objects were originally added. The exact implementation is not very relevant here (see Section 3.5 for pointers to the literature). It is sufficient to know that the call `shift.in(q, o)` adds an object o to the queue q , that `shift.out(q)` removes the oldest object from the queue q and that the empty queue is denoted by `()` in pseudo-code. Besides, one should know that adding and removing objects from a FIFO queue can be done in constant time.

The main difference with depth-first search is that calling the `dfs` function recursively with a new vertex adjacent to the current vertex has been replaced by the addition of the new vertex to the FIFO queue. This results in the fact that the two methods visit the vertices and edges in a different order, although all vertices and edges are eventually visited by both methods. Using a similar reasoning as in Section 3.4.1 based on the `mark` attribute of a vertex and the same assumptions for the complexity of the generic actions, it is easy to prove that the algorithm visits all vertices and all edges of a graph exactly once leading to a time complexity of $\mathcal{O}(n + |E|)$. Actually, changing the behavior of the queue from FIFO to LIFO (last in first out) changes the breadth-first algorithm into the depth-first algorithm (see Exercise 3.2).

The application of `bfs` to the graph of Figure 3.9(a) is illustrated in Figure 3.12. Each line of the figure corresponds to one iteration of the `do` loop in the pseudo-code description. For each iteration, the contents of the FIFO queue (the leftmost object is the oldest), the vertex processed and the edges processed are given. Note that the

```
/* Given is the graph  $G(V, E)$  */
```

```
struct vertex {
    ...
    int mark;
};

bfs(struct vertex v)
{
    struct fifo *Q;
    struct vertex u, w;
    Q ← 0;
    shift.in(Q, v);
    do { w ← shift.out(Q);
        "process w";
        for each  $(w, u) \in E$  {
            "process  $(w, u)$ ";
            if ( $u.\text{mark}$ ) {
                u.mark ← 0;
                shift.in(Q, u);
            }
        }
    } while ( $Q \neq 0$ )
}

main ()
{
    for each  $v \in V$ 
        v.mark ← 1;
    for each  $v \in V$ 
        if ( $v.\text{mark}$ ) {
            v.mark ← 0;
            bfs(v);
        }
}
```

Figure 3.11 A pseudo-code description of the breadth-first search algorithm `bfs`.

Q	w	edges processed
(v_1)	v_1	$e_1 = (v_1, v_2), e_2 = (v_1, v_3)$
(v_2, v_3)	v_2	$e_3 = (v_2, v_4), e_5 = (v_2, v_5)$
(v_3, v_4, v_5)	v_3	-
(v_4, v_5)	v_4	$e_4 = (v_4, v_3)$
(v_5)	v_5	-

Figure 3.12 The different steps of the breadth-first search algorithm applied to the graph of Figure 3.9(a).

edges processed determine the vertices added to the queue. From the figure it can be concluded that the vertices are visited in the order v_1, v_2, v_3, v_4, v_5 while the edges are visited in the order e_1, e_2, e_3, e_5 and e_4 .

In Section 3.4.1 it was mentioned that depth-first search could be used to find all vertices connected to a specific vertex $v_s \in V$ of a graph $G(V, E)$. Breadth-first search could be used for the same purpose. However, breadth-first search has an additional property. First all vertices adjacent to v_s are visited. These are the vertices connected to v_s with a path of length 1. The vertices visited next are reachable from v_s through a path of length 2, etc. It is therefore not difficult to see that the vertices are visited in the order of their *shortest path* from v_s . Breadth-first search can indeed be turned into a shortest-path algorithm by adding some bookkeeping statements and substituting appropriate code for the generic vertex and edge actions (see Exercise 3.3).

The shortest-path problem becomes more complex, however, if the length of the path between two vertices is not simply the number of edges in the path. This is the case in edge-weighted graphs, where the length of a path is defined as the sum of the edge weights of the edges in the path. Such a graph could e.g. model a group of cities and the distances between these cities (the graph can be constructed by taking a vertex for each city, an edge for each pair of cities that have a direct connection, and edge weights corresponding to the distance between adjacent cities). This version of the shortest-path problem is the topic of the next section.

3.4.3 Dijkstra's Shortest-path Algorithm

Suppose that a weighted directed graph $G(V, E)$ is given with the edge weights $w(e), w(e) > 0$, for each edge $e \in E$. The problem addressed in this section is finding the *shortest path* from a source vertex $v_s \in V$ to a target vertex $v_t \in V$. The solution discussed is an algorithm proposed by Dijkstra in 1959 and nowadays known as "Dijkstra's shortest-path algorithm".

How should one solve the shortest-path problem for an edge-weighted directed graph? It looks a good idea to start at vertex v_s and visit its adjacent vertices as both depth-first and breadth-first search do. The problem is, of course, that a single edge is not always the shortest connection between two adjacent vertices. There may exist a path (or multiple paths) between two vertices going through several edges of which the total weight is lower than the weight of the edge connecting the two vertices directly. A straightforward way is to enumerate all possible paths between v_s and v_t and then select the shortest one. This is not very efficient. It is for example possible to construct graphs whose sizes grow linearly with some parameter k , while the number of paths grows exponentially. An example of a family of graphs with this property is shown in Figure 3.13: it has $2k + 2$ vertices (k pairs of vertices together with v_s and v_t) and $4k$ edges, while the number of possible paths from v_s to v_t is 2^k (at each of the first k vertical positions one can choose between two edges to proceed to the right).

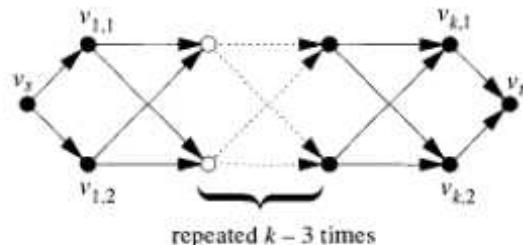


Figure 3.13 A family of graphs with an exponentially growing number of paths from v_s to v_t .

It turns out that there are more clever ways of computing the shortest path than merely enumerating all paths. This is e.g. the case for Dijkstra's shortest-path algorithm. As it will be shown later, its time complexity is a polynomial function of n (the number of vertices) and $|E|$ in spite of the fact that the number of paths may grow exponentially with n and $|E|$. This is due to the fact that many nonoptimal paths are directly eliminated while the algorithm proceeds.

A description of the algorithm in pseudo-code is given in Figure 3.14. This algorithm actually only computes the *length* of the shortest path (the computation of the shortest path itself is left as an exercise to the reader, see Exercise 3.4). In its main loop, the vertices of the set V are transferred one by one to a set T . The vertex that is selected for transfer, has the property that the shortest-path length to it is known at that moment (i.e. it will not change in the next iterations). Therefore, if the vertex to be transferred corresponds to v_t , the algorithm can stop.

The crucial element of the algorithm is to select the vertices in the correct order. This ordering results from the minimal value for the vertex attribute *distance* (similar to the descriptions of depth-first and breadth-first search, the use of struct *vertex* with member *distance* in the pseudo-code does not necessarily imply an implementation with *vertex* structures). In the first iteration, the *distance* attribute of a vertex $v \in V$ is equal to the edge weight $w((v_s, v))$ for all edges incident from v_s . Clearly, if the vertex with minimal *distance* value is selected, say vertex u , there cannot be a shorter path from v_s to u than the one through the single edge (v_s, u) (all edge weights are positive, so any path from v_s to u via an intermediate vertex is longer). Directly after the transfer of u from V to T , the *distance* attributes of the vertices at the endpoints of the edges incident from u are updated in the case that a path via u is shorter than the direct connection from v_s . Continuing this reasoning leads to the conclusion that the algorithm indeed computes shortest paths (see also Exercise 3.5). In general, it can be remarked that the value of the *distance* attribute of some vertex in V is equal to the length of the shortest path starting in v_s and passing only through the vertices in T .

To illustrate its effects, Dijkstra's shortest-path algorithm has been applied on a small graph shown in Figure 3.15. The growth of the set T and the subsequent values of the *distance* attributes of the vertices in the graph have been listed in

```

struct vertex {
    ...
    int distance;
};

dijkstra(set of struct vertex  $V$ , struct vertex  $v_s$ , struct vertex  $v_t$ )
{
    set of struct vertex  $T$ ;
    struct vertex  $u, v$ ;
     $V \leftarrow V \setminus \{v_s\}$ ;
     $T \leftarrow \{v_s\}$ ;
     $v_s$ .distance  $\leftarrow 0$ ;
    for each  $u \in V$ 
        if  $((v_s, u) \in E)$ 
             $u$ .distance  $\leftarrow w((v_s, u))$ 
        else  $u$ .distance  $\leftarrow +\infty$ ;
    while ( $v_t \notin T$ ) {
         $u \leftarrow "u \in V, \text{ such that } \forall v \in V : u \text{.distance} \leq v \text{.distance}"$ ;
         $T \leftarrow T \cup \{u\}$ ;
         $V \leftarrow V \setminus \{u\}$ ;
        for each  $v$  "such that  $(u, v) \in E"$ 
            if  $(v \text{.distance} > w((u, v)) + u \text{.distance})$ 
                 $v \text{.distance} \leftarrow w((u, v)) + u \text{.distance}$ ;
    }
}

```

Figure 3.14 A pseudo-code description of Dijkstra's shortest-path algorithm.

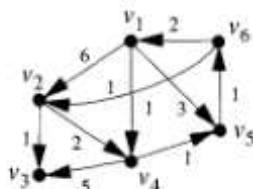


Figure 3.15 A small edge-weighted directed graph.

Figure 3.16. An underlined distance value indicates that the associated vertex is transferred from V to T at that moment. From the figure it can be seen that the $v_t = v_2$ is reached after 5 iterations and that continuing for one iteration more computes the lengths of the shortest paths for all vertices in the graph.

The time complexity of Dijkstra's algorithm depends on the exact type of data structures used. In a straightforward implementation, selecting the next element to be included in set T (the first statement in the `while` loop of Figure 3.14) requires $\mathcal{O}(n)$ time, where $n = |V|$, because all vertices in V have to be inspected in order to

iteration	T	v_i .distance, for $i =$					
		1	2	3	4	5	6
1	{ v_1 }	6	∞	1	3	∞	
2	{ v_1, v_4 }	6	6		2	∞	
3	{ v_1, v_4, v_5 }	6	6			3	
4	{ v_1, v_4, v_5, v_6 }	4	6				
5	{ v_1, v_4, v_5, v_6, v_2 }		5				
6	{ $v_1, v_4, v_5, v_6, v_2, v_3$ }						

Figure 3.16 The evolution of the `distance` attributes in Dijkstra's algorithm when applied to the graph of Figure 3.15; $v_1 = v_1$ and $v_7 = v_2$.

find the one with minimal `distance` attribute. As the loop is executed $\mathcal{O}(n)$ times, this results in a contribution to the overall time complexity of $\mathcal{O}(n^2)$. The body of the “for each” loop inside the while loop is executed $|E|$ times in total, as all edges are visited exactly once, viz. after the vertex from which they are incident is added to set T . This gives a contribution of $\mathcal{O}(|E|)$ to the overall time complexity, assuming that the edges incident from a node are directly accessible (e.g. because of an adjacency list representation). In a simple graph there are at most n^2 edges and therefore the worst-case time complexity of $\mathcal{O}(n^2 + |E|)$ can be simplified to $\mathcal{O}(n^2)$.

3.4.4 Prim's Algorithm for Minimum Spanning Trees

A *tree* is a connected graph without cycles. A *spanning tree* of a connected graph $G(V, E)$ is a subgraph of G that is a tree and contains all vertices of V . So, one gets a spanning tree by removing edges from E until all cycles in the graph have disappeared while all vertices remain connected. Unless the graph is already a tree, a graph has several spanning trees, all of which have the same number of edges (equal to the number of vertices minus one). In the case of edge-weighted undirected graphs, it is often interesting to find the spanning tree with the least total edge weight, also called the *tree length*. This problem is the *minimum spanning tree* problem. The problem is e.g. encountered when one has a set of cities and wants to build a railroad system of minimal total length that connects all cities by means of pairwise connections, or in layout design for integrated circuits, when it is necessary to connect a set of terminals with the minimal total wire length¹. In this section an algorithm originally published by Prim in 1957 will be presented.

The description of the algorithm in pseudo-code is given in Figure 3.17. The algorithm starts with an arbitrary vertex which is considered the initial tree (all the vertices are part of the spanning tree!). In the main loop, edges are added to the tree one by one (adding an edge automatically adds a new vertex as well) until the tree has become a spanning tree and contains all the vertices. Both the partial tree that is

¹ Actually, one often needs to find the minimal *Steiner tree*; this type of tree and a way to find an approximate solution are discussed in Section 4.5, where also its relation to the spanning tree is explained.

```

struct vertex {
    ...
    int distance;
    struct edge *via_edge;
};

struct edge {
    ...
};

prim(set of struct vertex V)
{
    set of struct edge F;
    set of struct vertex W;
    struct vertex u;
    u  $\leftarrow$  "any vertex from V";
    V  $\leftarrow$  V \ {u};
    W  $\leftarrow$  {u};
    F  $\leftarrow$   $\emptyset$ ;
    for each v  $\in$  V
        if ((u, v)  $\in$  E) {
            v.distance  $\leftarrow$  w((u, v));
            v.via_edge  $\leftarrow$  (u, v);
        }
        else v.distance  $\leftarrow$   $+\infty$ ;
    while (V  $\neq$   $\emptyset$ ) {
        u  $\leftarrow$  "u  $\in$  V, such that  $\forall v \in V : u.distance \leq v.distance$ " ;
        W  $\leftarrow$  W  $\cup$  {u};
        V  $\leftarrow$  V \ {u};
        F  $\leftarrow$  F  $\cup$  {u.via_edge};
        for each v "such that (u, v)  $\in$  E"
            if (v.distance  $>$  w((u, v))) {
                v.distance  $\leftarrow$  w((u, v));
                v.via_edge  $\leftarrow$  (u, v);
            }
        }
    }
}

```

Figure 3.17 The pseudo-code description of Prim's minimum spanning tree algorithm.

being constructed as well as the final spanning tree are graphs. Their vertex and edge sets are represented by the variables *W* and *F* respectively.

Here again, the data structures used are left vague intentionally. Even the use of separate records for vertices and edges is not necessarily required. They are only mentioned in order to be able to deal with abstract sets of vertices and edges. Besides, it is assumed that a vertex has the attributes *distance* and *via_edge*. The first one is used to register the shortest distance from a vertex not yet in the tree to a vertex already selected. The second one represents the edge through which this shortest

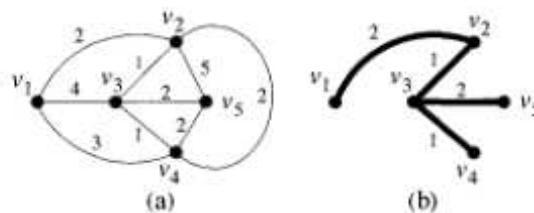


Figure 3.18 A small edge-weighted undirected graph (a) and its minimum spanning tree (b).

iteration	u	$v_i.\text{distance}$, $v_i.\text{via_edge}$, for $i =$				
		1	2	3	4	5
0	v_1		<u>2, (v_1, v_2)</u>	4, (v_1, v_3)	3, (v_1, v_4)	$+\infty?$
1	v_2			<u>1, (v_2, v_3)</u>	2, (v_2, v_4)	5, (v_2, v_5)
2	v_3				<u>1, (v_3, v_4)</u>	2, (v_3, v_5)
3	v_4					<u>2, (v_3, v_5)</u>
4	v_5					

Figure 3.19 The evolution of Prim's algorithm when applied to the graph of Figure 3.18.

connection is made. In each iteration, the vertex with the minimal distance value is added to the tree together with the edge of its `via_edge` attribute. Then the values of the `distance` and `via_edge` attributes are updated for the vertices not yet in the tree.

Figure 3.18(a) shows an example of a graph the minimal spanning tree of which should be determined. The result found by Prim's algorithm is depicted in Figure 3.18(b). Figure 3.19 shows the steps leading to this result. For each iteration, the figure shows the vertex u selected for inclusion in the set W and the values of the `distance` and `via_edge` attributes for the vertices not yet in W .

In a straightforward implementation, Prim's algorithm will require $\mathcal{O}(n^2)$ time, if n is the number of vertices. The main `while` loop is executed n times and within the loop finding the vertex with lowest `distance` value requires $\mathcal{O}(n)$ time. This worst-case time complexity can be improved by using more sophisticated data structures (see Section 3.5).

3.5 Bibliographic Notes

Several books entirely dedicated to algorithmic graph theory are available. Examples are: [Gib85] and [McH90]. Many books that deal with algorithms in general pay attention to graph theory in one or more chapters. Examples are: [Sed88] (this book uses Pascal as the language to describe algorithms; a version of the same book that uses C also exists [Sed90]), [Cor90] and [Mor91]. In these last books one can also find more details about the implementation of a FIFO queue necessary

for the implementation of breadth-first search. A discussion on the order notation, its standardization and its history can be found in [Knu76]. Dijkstra's shortest-path algorithm can be found in almost any book that is entirely or partially dedicated to algorithmic graph theory. The original publication is [Dij59]. It is possible to improve the worst-case time complexity of Dijkstra's algorithm from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n + |E|)$ by using more sophisticated data structures [Cor90]. Prim's algorithm for the minimum spanning tree is described in most of the books mentioned above. The original publication is [Pri57]. A discussion on how to improve the worst-case time complexity with respect to the description in Figure 3.17 can e.g. be found in [Mor91].

3.6 Exercises

- 3.1 Consider the graph data structure with "explicit edge and vertex representation" as shown in Figure 3.7. This data structure gives direct access to the outgoing edges of each vertex. Suppose that an algorithm also needs direct access to the incoming edges of a vertex. How should the data structure be modified to provide this direct access? Draw the representation of the graph of Figure 3.7(a) using this new data structure. How should the data structure be modified for the representation of undirected graphs? Draw the representation of the graph of Figure 3.1 using the modified data structure.
- 3.2 Suppose that the FIFO queue in the pseudo-code of Figure 3.11 is replaced by a queue with LIFO (last in first out) behavior. Show that the algorithm now more or less behaves as depth-first search by applying the modified algorithm to the graph of Figure 3.9.
- 3.3 Consider the shortest-path problem for directed graphs in which the length of a path is the number of edges in the path. It can be solved by the breadth-first search method described in Figure 3.11. What actions should be performed by the functions that "process" a vertex and an edge in order to actually compute the shortest path from any start vertex v_s to all other vertices in the graph?
- 3.4 Indicate how the description of Dijkstra's algorithm in Figure 3.14 should be modified when one wants to find the shortest path itself instead of merely the length of the shortest path. Pay special attention to the data structure for the representation of the path.
- 3.5 Prove that the algorithm given in Figure 3.14 correctly computes the length of the shortest path between v_s and v_t . Hint: use induction on the size of set T .
- 3.6 Prove the correctness of the algorithm given in Figure 3.17.

4

Tractable and Intractable Problems

In Chapter 3 the notion of computational complexity has been introduced. It was mentioned there that the distinction between problems that can be solved in polynomial time and those that need exponential time is a crucial one. As the exponent k in the time complexity $\mathcal{O}(n^k)$ of most algorithms normally is rather low (say, 1, 2, or 3), it is often feasible to apply the algorithm to problems of nontrivial size. A problem that can be solved in polynomial time is, therefore, called *tractable*. It is called *intractable* otherwise. A key notion in this context is the class of *NP-complete* problems, which contains those problems which are “likely to be intractable”. This chapter provides some background information on this topic, especially because many NP-complete problems occur in the field of CAD for VLSI. Often, for tractable problems, it is feasible to use exact algorithms that find the optimal solution, whereas for intractable problems, one should be satisfied with algorithms that do not guarantee an optimal solution.

In this chapter, the definitions of a *combinatorial optimization problem* and its *decision version* are given first. These definitions are then used for the introduction of the notion of NP-completeness, which is the main topic of this chapter. At the end of the chapter the consequences of the theory for the design of CAD tools are discussed.

The presentation in this chapter is kept rather informal. The goal is that the reader becomes aware that some problems in VLSI design automation are inherently difficult. When this is understood, one can better appreciate the performance of an algorithm meant to solve such a problem (both its execution speed and solution quality).

4.1 Combinatorial Optimization Problems

It is important to distinguish between a *problem*, and its *instances*. The term *problem* refers to a general class, e.g. the “shortest-path problem”. The term *instance* refers to a specific case of a problem, e.g. “the shortest-path problem between vertex v_s and vertex v_t in a given graph G ”.

Instances of *optimization* problems can be characterized by a finite set of variables; the correct choice for the values of these variables specifies the optimal solution. If the variables range over real numbers, the problem is called a *continuous* optimization problem. If the variables are discrete, i.e. they only can assume a finite number of distinct values, the problem is called a *combinatorial* optimization problem (also: *discrete* optimization problem). Most of the problems encountered in CAD for VLSI are combinatorial.

An example of a simple combinatorial optimization problem is the *satisfiability* problem. The input for this problem is a Boolean expression in the "product of sums" form as in the following example: $(x_1 + x_2 + x_3 + x_4)(\bar{x}_2 + \bar{x}_5)(\bar{x}_3 + \bar{x}_4 + \bar{x}_5)$. The problem is to assign Boolean values to the variables x_i in such a way that the whole expression becomes true. If this is possible, one says that the expression can be *satisfied*. Clearly, each of the variables x_i can only assume two values, making the problem a combinatorial optimization problem.

Another example is the shortest-path problem of the type solved by Dijkstra's algorithm (see Section 3.4.3). A graph $G(V, E)$, with given source and target vertices $v_s, v_t \in V$, defines an instance of the problem. One could associate Boolean variables b_i with each edge $e_i \in E$, such that $b_i = 1$ means that the edge is "selected" and $b_i = 0$ means that it is not. Then solving the shortest-path problem for this graph can be seen as assigning Boolean values to the variables b_i , such that the selected set of edges forms the shortest path.

Generally, a *combinatorial optimization problem* is defined as the set of all the *instances* of the problem, each instance I being defined as a pair (F, c) . F is called the set of *feasible solutions* (or the *search space*), while c is a function assigning a *cost* to each element of F . Thus $c : F \rightarrow R$, where R is the set of real numbers (for some type of problem, the cost only has integer values). One could say that each distinct set of values for the variables mentioned above defines one feasible solution. Solving a particular instance of a problem consists of finding a feasible solution $f \in F$ with minimal cost, i.e. an f such that $\forall y \in F : c(f) \leq c(y)$. Note: it is assumed in this section that the solution with minimal cost has to be found. If one is looking for the solution with maximal cost, the definitions and reasonings should be modified accordingly.

In the case of the satisfiability problem, for each instance, any assignment of Boolean values to the variables defines a feasible solution. So, all 2^N assignments for a problem with N variables define the set F of feasible solutions. Because the expression is either satisfied or not satisfied, the cost function c could be defined as having value 0 for those assignments that satisfy the expression and value 1 for the other ones.

For an instance of the shortest-path problem, each set of values for the Boolean variables b_i defines a subset of the edge set E . So, the set F consists of all subsets of E . A suitable definition of the cost function c is to assign the value $+\infty$ to all subsets that do not form a path from v_s to v_t and to assign the path length to those subsets that represent such a path.

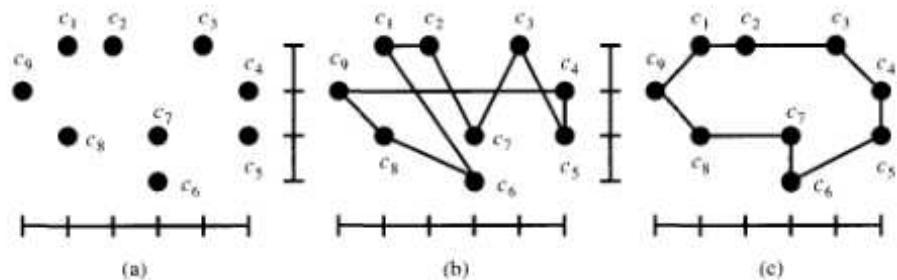


Figure 4.1 An instance of the Euclidean TSP (a), a nonoptimal (b) and an optimal solution (c).

Another example of a famous combinatorial optimization problem is the *traveling salesman* problem (TSP). As the name says, the problem involves finding the shortest tour for a salesman who needs to visit a number of cities and then to return to the first city. In its *Euclidean* version, the cities are located in Euclidean space (most of the time this space has two dimensions) and a distance matrix gives the distances between all pairs of cities. Obviously, any permutation of the cities defines a feasible solution and the cost of the feasible solution is the length of the cycle represented by the solution. Figure 4.1(a) shows an example of the Euclidean TSP problem with cities \$c_1, c_2, \dots, c_9\$. If the coordinates of a city \$c_i\$ are given by \$(x_i, y_i)\$, the distance between two cities \$c_i\$ and \$c_j\$ is simply given by \$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}\$. The Figures 4.1(b) and (c) show a nonoptimal and an optimal solution for the problem of Figure 4.1(a).

Another, more general, version of TSP is based on graphs: the cities can be thought as vertices of a graph and the distances are the weights of the edges. The goal is to find a simple cycle through all vertices, the tour length of which is minimal (the sum of the edge weights for the edges in the tour). Note that a graph version can be constructed for any instance of Euclidean TSP, but that the reverse is not true in general. It is also not true that any permutation of the vertices defines a tour due to the possible absence of an edge between two neighboring vertices in the permutation. A cost function should assign the value \$+\infty\$ to these illegal tours while it should assign the sum of the edge weights to the legal tours. Figure 4.2(a) shows an edge-weighted graph for which the TSP should be solved and Figure 4.2(b) an optimal tour in this graph.

4.2 Decision Problems

As was stated above, given an instance of a combinatorial optimization problem, the goal is to find the feasible solution with minimal cost. One could also have a more modest goal, viz. to only find the cost of the optimal solution without necessarily knowing the solution itself. As opposed to the *optimization version* of the problem, the problem that one gets then is called the *evaluation version* of the problem.

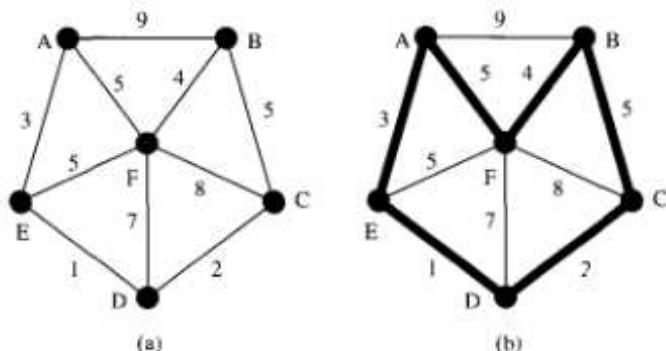


Figure 4.2 An instance of the graph version of TSP (a) and an optimal solution (b).

Example: the optimization version of the shortest-path problem in graphs requires that the edges forming the shortest path are identified, whereas the evaluation version merely asks for the length of the shortest path.

For mathematical reasons, the theory of NP-completeness only deals with so-called *decision problems*. These are problems that only have two possible answers: "yes" or "no". This is not a serious restriction, as far as combinatorial optimization problems are concerned. Each optimization problem has a *decision version* as well. In the case of a minimization problem, instead of asking the question "What is the minimal solution?", one could ask "Is there a solution with cost less than or equal to k ?".

It should be remarked that the decision version of a problem is not harder than the optimization version. Suppose that one could solve the optimization version in polynomial time, then the decision version can also be solved in polynomial time: once the minimal solution is known, it is known as well whether its cost is less than or equal to k . The opposite is not true: if there is an algorithm that is able to decide in polynomial time whether there is a solution with cost less than or equal to k , it is not always obvious how to get the solution itself in polynomial time. Therefore, the computational complexity of the decision version of a problem gives a *lower bound* for the computational complexity of the optimization version. Note: the time complexity of the evaluation version of a combinatorial optimization problem is located in between the time complexities of the decision and optimization versions.

More formally, the decision version of a combinatorial problem Π can be defined as the set D_Π of its instances (F, c, k) . Note that each instance is now characterized by an extra parameter k ; k is the parameter in the question "Is there a solution with cost less than or equal to k ?" An interesting subset of instances is formed by those instances for which the answer to the question is "yes". This set is called Y_Π ($Y_\Pi \subseteq D_\Pi$).

A task associated with a decision problem is *solution checking*. It is the problem of verifying whether $c(f) \leq k$, given an $f \in F$. Solution checking actually only

involves the computation of the cost function, given a feasible solution. For most problems it can be done in polynomial time.

4.3 Complexity Classes

For the purpose of reasoning about the degree of difficulty of decision problems, it is useful to group problems with the same degree in one *complexity class*. The class of decision problems for which an algorithm is known that operates in polynomial time is called P (which is an abbreviation of "polynomial").

Before introducing another class called NP , it is necessary to give an informal definition of a *nondeterministic computer*. This is not a computer that can be built from physical components, but only a conceptual tool used in the theory of computational complexity. For a common (deterministic) computer it always is clear how a computation continues at a certain point in the computation. This is also reflected in the programming languages used for them. In languages like Pascal or C, there is e.g. a sequence of statements separated by semicolons that are executed one after the other. A nondeterministic computer allows for the specification of multiple computations at a certain point in a program: the computer will make a *nondeterministic choice* on which of them to perform. This is not just a *random choice*, but a choice that will lead to the desired answer. Another way of looking at such a machine is to say that the machine splits itself into as many copies as there are choices, evaluates all choices in parallel, and then merges back to one machine. In the context of decision versions of computational optimization problems, one can say that a nondeterministic computer is able to guess the right feasible solution in constant time and then verify that the decision for this solution is "yes" by the deterministic application of the solution checking computation.

The complexity class NP (an abbreviation of "nondeterministic polynomial") consists of those problems that can be solved in polynomial time on a nondeterministic computer. Any decision problem for which solution checking can be done in polynomial time is in NP .

Because all algorithms with a polynomial time complexity on a deterministic computer will certainly execute in polynomial time on a nondeterministic one, the class P is a subset of the class NP . In the field of complexity theory many more classes of decision problems are distinguished, apart from P and NP . They are outside the scope of this text. As far as this text is concerned, the universe of decision problems can be pictured as in Figure 4.3. The class NPC , of *NP-complete* problems is very important and will be discussed in detail in the next section. In this section it will also become clear that it is not completely certain that the universe has the structure shown.

Note: the reader should realize that there are decision problems for which no algorithm at all exists, regardless of its complexity. A well-known example is the *halting problem*, where the problem is to find an algorithm that accepts a computer program as its input and has to decide whether or not this program will stop after a finite computation time. Such a problem is called *undecidable*.

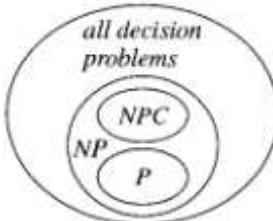


Figure 4.3 A simple classification of all decision problems as it is likely to be.

4.4 NP-completeness and NP-hardness

Within NP one can distinguish a special class NPC formed by the *NP-complete* problems. It is very likely that there are no problems that are both in P and NPC , although nobody has been able to prove this fact so far. If some problem with this property would be found, by definition $NP = P$ would hold, instead of the strong belief $NP \supset P$. This is a consequence of the special properties required for the membership of the class NPC , as will be explained below.

The class NPC is characterized by the fact that all decision problems contained in it are *polynomially reducible* to each other. Informally stated, this means that an instance of any NP-complete problem can be expressed as an instance of any other NP-complete problem using transformations that have a polynomial time complexity. Then, if an algorithm can be found that solves some problem in NPC in polynomial time, all problems in NPC would be solved in polynomial time. The consequence would be that all NP-complete problems would become elements of complexity class P . Stated more formally, a problem Π_1 is polynomially reducible to a problem Π_2 , if a polynomial-time transformation f exists that transforms any instance $I \in D_{\Pi_1}$ into an instance $f(I) \in D_{\Pi_2}$ in a polynomial number of steps. Besides, f should have the following property:

$$I \in Y_{\Pi_1} \Leftrightarrow f(I) \in Y_{\Pi_2}$$

If such a transformation can be found, it means that Π_1 can be solved in polynomial time if a polynomial-time algorithm exists for Π_2 . This idea has been illustrated in Figure 4.4.

Proving that a certain problem Π is NP-complete consists of two steps:

1. Showing that Π is in NP , i.e. that it can be solved in polynomial time on a nondeterministic machine. It amounts to showing that solution checking can be done in polynomial time on a deterministic machine, which is trivial in most cases.
2. Showing that some problem that is already known to be NP-complete can be polynomially reduced to Π .

To illustrate the procedure followed in proving NP-completeness, it will be proven that the *graph version* of TSP (the traveling salesman problem, see Section 4.1) is

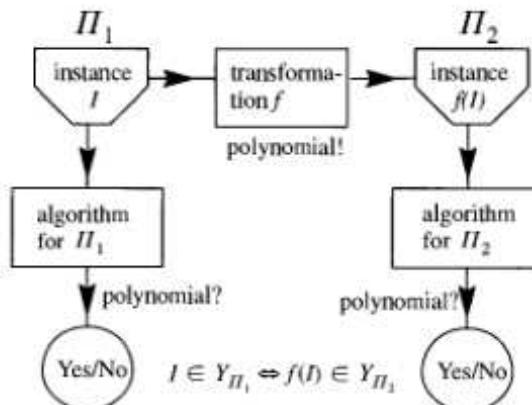


Figure 4.4 The reduction of a problem Π_1 to a problem Π_2 .

NP-complete, given the fact that the problem HAMILTONIAN CYCLE¹ is known to be NP-complete. This last problem asks whether a given undirected graph $G(V, E)$ contains a so-called *Hamiltonian cycle*, i.e. a simple cycle that goes through all vertices of V . TRAVELING SALESMAN, the decision version of TSP amounts to answering the question of whether there is a tour (simple cycle) through all vertices, the length of which is less than or equal to k . Showing that the problem is in NP is trivial: for any sequence of vertices, it can be checked in polynomial time whether the sequence corresponds to a cycle in the graph going through all vertices and that the sum of all edge weights of the edges in the cycle is less than k . The second part of the proof consists of a reduction from HAMILTONIAN CYCLE. Given an instance of HAMILTONIAN CYCLE with graph $G(V, E)$, use the same graph and assign a weight of 1 to all edges. So, in this case, the transformation f just adds unity edge weights to the graph. It is clearly a polynomial-time transformation. If the number of vertices is n , the question to be answered is now whether there is a simple cycle of length n going through all vertices. Obviously, there is a tour of length n in the instance of TRAVELING SALESMAN if and only if there is a Hamiltonian cycle in the graph $G(V, E)$. This completes the NP-completeness proof of TRAVELING SALESMAN. The consequence is that if a polynomial-time algorithm could be found to solve the graph version of TSP, there would also exist a polynomial-time algorithm for HAMILTONIAN CYCLE and indeed for any problem member of NPC.

The proof above was almost trivial to find as the two problems were similar in nature. Often, the reduction of a problem to another is far more complex.

Proving NP-completeness using reduction only works when there is a first problem that can be proved to be NP-complete in another way. Such a problem is the

¹ Following the convention used in [Gar79], the names of decision problems are given in capitals.

SATISFIABILITY problem, already mentioned in Section 4.1. *Cook's theorem* proves that SATISFIABILITY is NP-complete. This amounts to showing that *any* problem in *NP* is polynomially reducible to SATISFIABILITY. Such a general statement clearly requires a sophisticated proof using mathematical notions that are beyond the scope of this text. Therefore, only its "spirit" presented here.

It starts with a mathematical model of a nondeterministic computer and the programs that it can run (this model includes e.g. the instructions that the computer can execute). This model is based on a *Turing machine*, a computer with a sequentially accessible memory (a "tape") and a very simple instruction set. The set only includes instructions for writing a symbol (from a finite set) to the memory location pointed at by the memory pointer and move the pointer one position up or down. Besides, a finite number of "internal states" should be provided for a specific Turing machine. A "program" then consists of a set of conditional statements that map a combination of a symbol at the current memory location and an internal state to a new symbol to be written, a new internal state and a pointer movement. The input to the algorithm to be executed on a Turing machine is the initial state of the memory. The machine stops when it enters one of the special internal states labeled by "yes" and "no" (corresponding to the answers to a decision problem). In spite of its simplicity, the Turing machine is able to perform any computation that any other computer can perform if one can tolerate an increase in the time complexity by a factor that is a polynomial of the input size (this can indeed be tolerated in the theory of NP-completeness). Many versions of the Turing machine exist, including the nondeterministic one that is used in the proof of Cook's theorem.

The sequence of state transitions that a Turing machine performs can be expressed as a product-of-sums type Boolean expression. Stated more strongly, any computation that can be performed by a nondeterministic Turing machine has an equivalent product-of-sums type Boolean expression that is satisfiable if and only if the computation finishes in the special internal state "yes". This means that any problem in *NP* can be polynomially reduced to SATISFIABILITY. So, Cook's theorem establishes that any problem in *NP* could be solved in polynomial time if SATISFIABILITY could be solved in polynomial time, or equivalently, that SATISFIABILITY is *NP*-complete.

An open question in computer science is whether or not the complexity classes *P* and *NPC* are disjoint. Nobody has been able to find a polynomial-time algorithm that solves any of the *NP*-complete problems. If such an algorithm could be found, it would provide a polynomial solution to all *NP*-complete problems including SATISFIABILITY. Because all problems in *NP* can be polynomially reduced to SATISFIABILITY by Cook's theorem, all problems in *NP* would be solvable in polynomial time, proving that *P* = *NP*. On the other hand, nobody has been able to prove that an *NP*-complete problem is not an element of *P*. The theory of *NP*-completeness only supports the statement that it is very unlikely that *P* = *NP* and that *NP*-complete problems are probably inherently more difficult than problems in *P*.

The qualification *NP-hard* is often encountered when discussing the computational complexity of a problem. It refers to problems that are at least as hard as NP-complete problems. It might be the case that it cannot be proved that a problem is in *NP*, but that an NP-complete problem can be reduced to it. The term NP-hard is also used for the optimization versions of combinatorial problems for which the decision version is known to be NP-complete. As mentioned earlier, the optimization version of a combinatorial problem is at least as difficult as the decision version.

4.5 Consequences

An important lesson from the previous theory is that one should not have the naive attitude of wanting to solve a problem optimally without checking the complexity class to which that problem belongs.

Even when two problems seem to be similar, one of them can have a polynomial-time algorithm that solves it, while the other is NP-complete or NP-hard. For example the shortest-path problem in graphs can be solved in polynomial time (with e.g. Dijkstra's algorithm mentioned in Section 3.4.3), whereas the *longest-path* problem that has the goal of finding the longest simple path between two points in a graph, is NP-hard (see Exercise 4.1; see as well Section 6.4.5 for a more detailed discussion).

The existence of NP-complete problems justifies the use and design of algorithms that do not guarantee an optimal solution. Such algorithms can be classified into two groups: *approximation algorithms* and *heuristics*. On the other hand, one should not forget that the differences between complexity classes become more relevant for large input sizes. If the inputs for an NP-complete problem to be solved are known to be relatively small, one should certainly consider to solve the problem exactly (using an algorithm with an exponential worst-case time complexity).

Approximation algorithms guarantee a solution with a cost that is within some margin of the optimum. Consider as an example two related problems MINIMUM RECTILINEAR SPANNING TREE and MINIMUM RECTILINEAR STEINER TREE. In both cases a set of points in a plane is given. The points have to be interconnected by a tree of shortest length, with the restriction that the interconnections should consist of horizontal and vertical line segments only (these are called *rectilinear* segments). A tree in this context can best be understood by thinking of the points as vertices of a *complete planar graph*. If a point p_i has coordinates (x_i, y_i) , the distance between two points p_i and p_j is given by $|x_i - x_j| + |y_i - y_j|$, as only rectilinear segments are allowed for the interconnection. Distances defined in this way are the edge weights of the graph. The tree length is then the sum of the edge weights for the edges included in the tree. The two problems arise e.g. in VLSI routing. In the case of the spanning tree, no new points can be added to the original set; in the case of a Steiner tree, additional points, so-called *Steiner points*, can be added to reduce the length of the tree (see Figure 4.5).

MINIMUM RECTILINEAR SPANNING TREE is a special case of the general spanning tree problem for graphs and can be solved in polynomial time by Prim's algorithm (see Section 3.4.4). On the other hand, MINIMUM RECTILINEAR

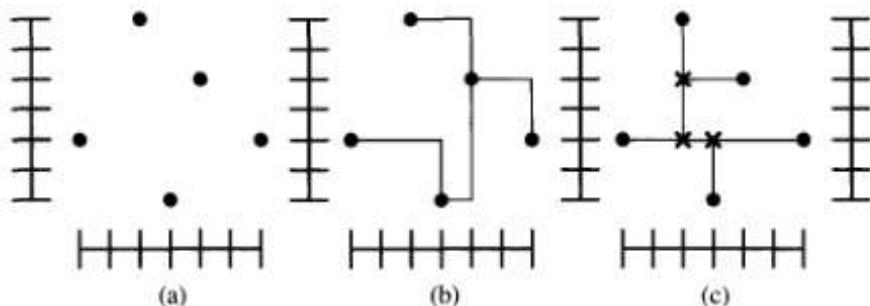


Figure 4.5 A set of points on a grid (a), their minimal rectilinear *spanning tree* (b), and one of their minimal rectilinear *Steiner trees* (c) (crosses indicate Steiner points).

STEINER TREE is NP-complete. However, if one takes the minimal rectilinear spanning tree as an approximation for the minimal rectilinear Steiner tree, the resulting tree is at most 1.5 times longer than the optimal one² (check that this is true in Figure 4.5).

Heuristics are algorithms that are constructed based on “rules-of-thumb”, ideas that seem to be helpful for some typical instances, although nothing can be said in advance on the general quality of the result. Even when it might be unsatisfactory, for most NP-complete or NP-hard problems in CAD for VLSI, heuristics seem to be the only way to solve problems. They are acceptable if they perform well for most typical problem instances.

4.6 Bibliographic Notes

Readers that want to study the topic of NP-completeness and related notions in all detail are referred to [Gar79], which is the standard text on the topic. A similar level of detail and depth can be found in the more recent text [Pap94]. These two sources each include a detailed discussion of Turing machines and a proof of Cook’s theorem. The topic also receives a more or less formal treatment in many books with a (much) wider scope; examples are: [Baa78], [Hor78], [Pap82], [Wil86], and [Cor90]. The chapters on computational complexity (including NP-completeness) in [Har87] are informal and easy to read.

The formulation of a combinatorial optimization problem as the set of its instances, each instance being defined as a pair of feasible solutions and a cost function, is based on the formulation in [Pap82]. The *evaluation version* of a combinatorial optimization problem is not so often mentioned in textbooks; a book that mentions it is [Pap82].

Many texts on theoretical computer science discuss the existence of undecidable problems of which the *halting problem* is an important one. The topic is e.g.

² Using other algorithms, one can obtain even tighter approximations.

mentioned in [Pap94]. An informal and entertaining treatment can be found in [Har87].

A comprehensive text that is completely dedicated to Steiner trees is [Hwa92]; it includes several chapters on the rectilinear Steiner tree. The proof that MINIMUM RECTILINEAR STEINER TREE is NP-complete is given in [Gar77]. The result that the length of the minimum rectilinear spanning tree is at most 1.5 times the length of the minimum rectilinear Steiner tree is from [Hwa76]. More references to literature on Steiner trees are given in the Bibliographic Notes section at the end of Chapter 9.

A book that considers VLSI design from the point of view of computational complexity is [Ull84].

4.7 Exercises

- 4.1** Show the NP-completeness of the LONGEST PATH problem using a reduction from HAMILTONIAN CYCLE. The LONGEST PATH problem asks whether there exists a path with a length larger than k from a source to a target vertex in a graph with positive edge weights (the same type of graph for which Dijkstra's algorithm discussed in Section 3.4.3 computes the shortest path). Hint: consider the longest path from a vertex to itself. Explain now why Dijkstra's algorithm cannot be adopted to compute longest paths by systematically replacing " $<$ " by " $>$ ", etc. in the algorithm.

5

General-purpose Methods for Combinatorial Optimization

In VLSI design automation one encounters many combinatorial optimization problems. Most of them are intractable in the sense of Chapter 4. For others, polynomial solutions are known. In Chapter 4, it was mentioned that the algorithm designer has three possibilities when confronted with an intractable problem.

The first option is to try to solve the problem exactly if the problem size is sufficiently small using an algorithm that has an exponential (or even a higher order) time complexity in the worst case. This can be done in many different ways, all of which have the same theoretical worst-case time complexity, although some are clearly preferable in practice. The simplest way to look for an exact solution is *exhaustive search*: it simply visits all points in the search space in some order and retains the best solution visited. Other methods only visit part of the search space, albeit the number of points visited may grow exponentially (or worse) with the problem size. The first sections of this chapter are dedicated to methods in this category, viz. *backtracking with branch-and-bound*, *dynamic programming* and *integer-linear programming*. (Note that the term *programming* does not refer here to the act of writing code in order to implement a computation on a computer, but to the mathematical formulation of the problem.) One can call these methods “general purpose” in the sense that they are applicable to almost any combinatorial optimization problem as opposed to the “special purpose” methods that only work for specific problems.

Approximation algorithms and heuristics are the other two options to tackle intractable problems. General-purpose approximation algorithms do not exist: guarantees in finding a solution, the cost of which is within a certain margin of the optimal cost, can only be given by involving problem-specific issues in the analysis of the algorithm. The second part of this chapter consists of sections dealing with general-purpose heuristics that do not guarantee an optimal solution: *local search*, *tabu search*, *simulated annealing* (under theoretical conditions that are impossible to satisfy in practice, this method finds an exact solution), and *genetic algorithms*.

It is remarkable that some general-purpose methods for combinatorial optimization find their inspiration in other domains than mathematics and computer science.

Simulated annealing was inspired by physics, whereas evolution theory was a source of inspiration for *genetic algorithms*.

The goal of this chapter is to provide some additional background for the discussion of the combinatorial optimization problems that are found in different places in VLSI design automation as explained in various chapters of this text. Special-purpose algorithms are described in the appropriate chapters, but algorithms that are applicable in more than one place can better be presented centrally. No attempt has been made to achieve completeness for general-purpose methods and the selection is mainly based on the occurrence of the methods in CAD for VLSI. The methods are presented rather informally, sometimes at the expense of mathematical precision.

The very first issue discussed in this chapter is, however, not an optimization method but a simplified CAD problem that has been called the *unit-size placement problem*. It can be tackled by most of the general-purpose methods to be introduced. By defining the problem at this stage, it will be possible to use it for the illustration of the different methods besides more mathematical problems such as TSP.

5.1 The Unit-size Placement Problem

An instance of the placement problem is defined by a set of *cells* and a description of how these cells should be interconnected. Cells are small subcircuits the internal layout of which is known. The interconnections to be made are specified by *nets*. A net can be seen as a set of cells that share the same electrical signal. The goal of placement is to assign a location to each cell such that the total chip area occupied is minimized. As the number of cells is not modified by placement, minimizing the area amounts to avoiding empty space and keeping the wires that will realize the interconnections as short as possible.

In this section, only a "toy version" of the placement problem is defined. More realistic versions of the problem are presented in Chapter 7. The toy problem will be called *unit-size placement* for the simple reason that all cells in the circuit are supposed to have a layout with dimensions 1×1 (measured in some abstract length unit). Furthermore it can be assumed that the only positions on which a cell can be put on the chip are the grid points of a grid created by horizontal and vertical lines with unit-length separation. A nice property of unit-size placement is that the assignment of distinct coordinate pairs to each cell guarantees that the layouts of the cells will not overlap. If the range of coordinates available in two dimensions is fixed in advance, the only contribution to the cost function will come from the area occupied by wiring.

Figure 5.1(a) shows the specification of a circuit description consisting of 7 cell instances *A* to *F*, and 8 nets n_1 to n_8 . For each net, the specification gives the cells to which the net is connected. Such a description is called a *netlist*. A possible placement of this circuit on a unit-size grid is given in Figure 5.1(b).

The best way to evaluate the quality of a solution for unit-size placement is to route all nets and measure the extra area necessary for wiring. In the unit-size placement model, as much area as required can be created by pulling apart adjacent rows or

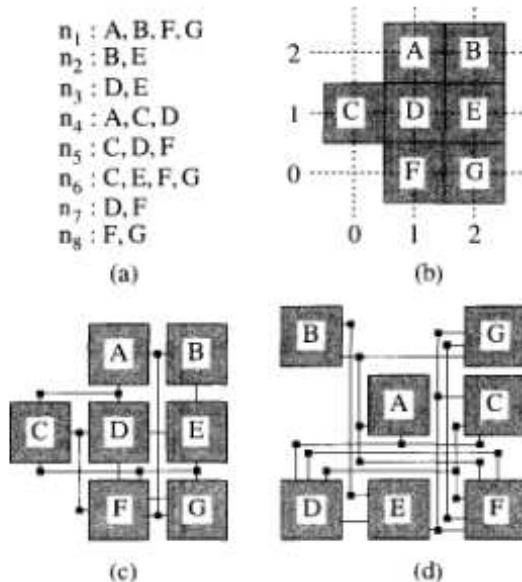


Figure 5.1 An example netlist for the unit-size placement problem (a), a possible placement solution (b), the routing for this placement (c), and the routing for a placement of worse quality (d).

columns. This is shown in Figure 5.1(c) for the placement of Figure 5.1(b). The solution presented is just a sketch. For a real solution, one needs to know to which specific port of a cell's layout a net should connect. A bad placement will have longer connections which normally will lead to more routing *tracks* between the cells and therefore to a larger chip area. This is illustrated in Figure 5.1(d).

Most versions of the routing problem are intractable and even heuristics require a considerable amount of computation time. Therefore, solving the routing problem is an expensive way to evaluate the quality of a placement. This is especially the case if many tentative placements have to be evaluated in an algorithm that tries to find a good one. An alternative used in most placement algorithms is to only *estimate* the wiring area. The first approximation made is to estimate the total wire length instead of wiring area. At this stage, it is sufficient to know that techniques for wire length estimation exist that require the position of all cells and the netlist as their only inputs. They will be discussed in detail in Section 7.2.

5.2 Backtracking and Branch-and-bound

This section presents two techniques for finding the optimal solution of a combinatorial optimization problem that are quite similar: backtracking and branch-and-bound. Actually, branch-and-bound is a refinement of backtracking. Before explaining these techniques some general remarks have to be made.

In Chapter 4, an instance I of a combinatorial optimization problem was defined by a pair (F, c) , with F the “set of feasible solutions” (also called the “search space” or “solution space”) and c a cost function assigning a real number to each element in F . Suppose that each feasible solution can be characterized by an n -dimensional vector $\mathbf{f} = [f_1 f_2 \dots f_n]^T$ and each f_i ($1 \leq i \leq n$) can assume a finite number of values, called the *explicit constraints* of f_i . Besides, the values assigned to the different components of \mathbf{f} may sometimes not be chosen independently. In such a case one speaks of *implicit constraints*. Consider e.g. a combinatorial optimization problem related to some graph $G(V, E)$ in which a path with some properties is looked for. One can then associate a variable f_i with each edge $e_i \in E$, whose value is either 1 to indicate that the corresponding edge is part of the path or 0 to indicate the opposite. The explicit constraints then state that $f_i \in \{0, 1\}$ for all i . The implicit constraints say that the edges selected by the variables should form a path.

Note: the specification of a solution by means of a fixed-length vector is not always the most appropriate choice for a combinatorial optimization problem. In the example above, a more natural representation of a path is given by a sequence of vertices and edges with a variable length (cf. the definition of a path in Section 3.1). However, the notions presented here do not change essentially if a variable-length encoding of the solution is chosen instead of a fixed-length one.

Assuming the fixed-length vector representation of a solution, one defines a *partial solution* by leaving one or more elements of the vector unspecified. One might as well say that a partial solution corresponds to a subspace of the solution space. A partial solution will be denoted by $\tilde{\mathbf{f}}$.

5.2.1 Backtracking

The principle of using *backtracking* for an *exhaustive search* of the solution space is to start with an initial partial solution in which as many variables as possible are left unspecified, and then to systematically assign values to the unspecified variables until either a single point in the search space is identified or an implicit constraint makes it impossible to process more unspecified variables. In the first case, the cost of the feasible solution found can be computed. In both cases, the algorithm continues by going back to a partial solution generated earlier and then assigning a next value to an unspecified variable (hence the name “backtracking”). The pseudo-code of such a backtracking algorithm is given in Figure 5.2. It is assumed that all variables f_i have type `solution.element`. The partial solutions are generated in such a way that the variables f_i are specified for $1 \leq i \leq k$ and are unspecified for $i > k$. Partial solutions having this structure will be denoted by $\tilde{\mathbf{f}}^{(k)}$. Note that $\tilde{\mathbf{f}}^{(n)}$ corresponds to a fully-specified solution (a member of the set of feasible solutions). The global array `val` corresponds to the vector $\tilde{\mathbf{f}}^{(k)}$. The value of f_k is stored in `val[k - 1]`. So, the values of array elements with index greater than or equal to k are meaningless and should not be inspected. The procedure `cost(val)` is supposed to compute the cost of a feasible solution using the cost function c . It is obviously only called when $k = n$, i.e. when a solution is fully specified. The

```

float best_cost;
solution.element val[n], best.solution[n];

backtrack(int k)
{
    float new_cost;
    if (k == n) {
        new_cost := cost(val);
        if (new_cost < best_cost) {
            best_cost := new_cost;
            best.solution := copy(val);
        }
    }
    else
        for each (el ∈ allowed(val, k)) {
            val[k] := el;
            backtrack(k + 1);
        }
}
}

main ()
{
    best_cost := ∞;
    backtrace(0);
    report(best.solution);
}

```

Figure 5.2 The pseudo-code of an algorithm for an exhaustive search by means of backtracking.

procedure `allowed(val, k)` returns a set of values allowed by the explicit and implicit constraints for the variable f_{k+1} given $\bar{f}^{(k)}$. The best solution found is stored in the global array `best.solution` and it is reported at the end of the search process.

Consider the graph version of the *traveling salesman problem* (TSP) introduced in Chapter 4, in which the number of vertices in a graph $G(V, E)$ equals $n - 1$. A solution can be specified by a sequence of n vertices such that an edge between subsequent vertices exists, the first vertex in the sequence equals the last one and the first $n - 1$ vertices are distinct. The sequence can be characterized by n variables f_i ($1 \leq i \leq n$), one for each position in the sequence. The value of f_i is the vertex at position i in the sequence. Because any vertex $v \in V$ in the cycle can be taken as the first and last element of the sequence, one can select any of the vertices in the graph and state that f_1 and f_n can only have value v . These are the explicit constraints for f_1 and f_n . The explicit constraints for the other variables are the remaining vertices of V ($V \setminus \{v\}$). The implicit constraints are that any sequence specified by

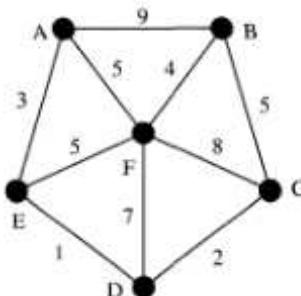


Figure 5.3 An example traveling salesman problem instance.

the variables should form a path¹. Figure 5.3 shows an example problem instance. The six vertices have been labeled A to F . The cost of traveling between two adjacent vertices is given by a weight along the edge connecting the two vertices. It will be assumed that $f_1 = f_7 = A$ and that the other variables take any of the values in the set $\{B, C, D, E, F\}$.

Consider as another example the *unit-size placement* problem defined in Section 5.1. If there are n cells to be placed, a solution can be described by $2n$ variables f_1 to f_{2n} , where f_{2i-1} corresponds to the x -coordinate of cell i and f_{2i} to its y -coordinate ($1 \leq i \leq n$). The explicit constraints say that values should be chosen in the available range of coordinates (x or y). The implicit constraints disallow that two cells are assigned the same coordinate pair.

The subsequent generations of (partial) solutions by the backtracking algorithm can be visualized by nodes in a tree, called the *search tree*². The search tree for the example of Figure 5.3 is shown in Figure 5.4. A *tree* is a directed graph here. It is normally drawn in such a way that all edges have an implicit direction from top to bottom. The single node at the highest level, Level 1, is called the *root* of the tree. The nodes *incident from* the node (connected to it at the next lower level) are called its *children*. Every level of the tree corresponds to a variable. The children at level $k+1$ of a node at level k correspond to the partial solutions obtained by specifying a value for f_{k+1} . So, a node at level k represents a partial solution $\tilde{\Gamma}^{(k)}$. Nodes without children are called *leaf nodes*. Note that each node except for the root has exactly one incoming edge. The other endpoint of this edge is called the node's *parent*. Removing this incoming edge for a node $\tilde{\Gamma}^{(k)}$ gives a *subtree* having $\tilde{\Gamma}^{(k)}$ as its root. Each path in the tree starting at the root and ending in a leaf node corresponds to either a fully

¹ One could say that a variable that can only assume a single value is not a real variable and claim that a feasible solution could be described with $n - 2$ instead of n variables. However, the description chosen here makes it e.g. easier to exclude by means of implicit constraints a path that contains all vertices but does not form a cycle.

² In order not to get confused with the vertices of e.g. the graph of the TSP example, the vertices of the search tree will be called *nodes* in this section.

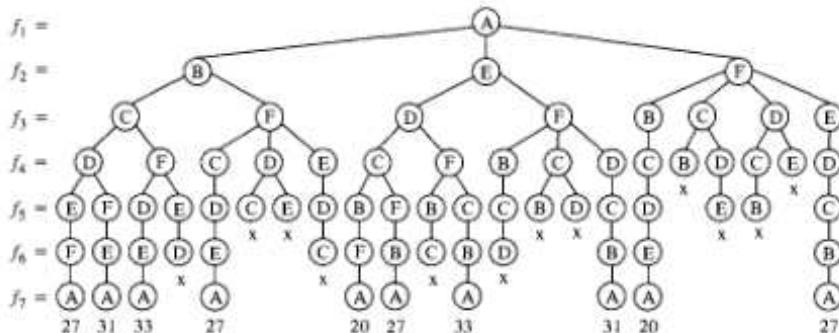


Figure 5.4 The search tree obtained by the exhaustive search backtracking algorithm for the example of Figure 5.3.

specified solution of which the cost can be computed (indicated with the cost value in the figure) or to a state in the backtracking procedure from which no new (partial) solution can be generated due to the fact that no legal assignment to the next variable is possible (indicated with an 'x' in the figure). Note that each tour is found twice, the second of each pair of equivalent solutions being the reverse of the first (this can be avoided by an extra implicit constraint, e.g. putting that City B should always precede City C in each solution; such a constraint will reduce the search space and, therefore, speed up the search). Obviously, the optimal tour visits the vertices in the order A, E, D, C, B, F, and A (or in the reverse order) and has a total length of 20.

5.2.2 Branch-and-bound

Normally, it is not necessary to visit all (partial) solutions that the backtracking procedure generates. Let $D(\bar{f}^{(k)})$ denote the set of fully-specified solutions in the subtree with root $\bar{f}^{(k)}$. Information about a certain partial solution $\bar{f}^{(k)}$, $1 \leq k < n$, at a certain level can indicate that any fully-specified solution $\bar{f}^{(n)} \in D(\bar{f}^{(k)})$ derived from it can never be the optimal solution. This conclusion is based on the estimation of the lower bound of the cost of all solutions in $D(\bar{f}^{(k)})$. The function that estimates this cost lower bound will be denoted by \bar{c} . If inspection of $\bar{f}^{(k)}$ can guarantee that all of the solutions in $D(\bar{f}^{(k)})$ have a higher cost than some solution already found earlier during the backtracking, none of the children of $\bar{f}^{(k)}$ need any further investigation. One says that the node in the tree corresponding to $\bar{f}^{(k)}$ can be *killed*. The modification of the backtracking algorithm that provides in killing partial solutions is called *branch-and-bound*. One also says that one can *prune* the search tree by removing the subtree having $\bar{f}^{(k)}$ as its root.

The pseudo-code describing the branch-and-bound version of backtracking is presented in Figure 5.5. The main recursive procedure is called `b_and_b`. It is quite similar to the procedure `backtrack` of Figure 5.2 with the difference that the procedure `lower_bound_cost` is called to get a lower bound of the partial solution

```

float best_cost;
solution_element val[n], best_solution[n];

b.and.b(int k)
{
    float new_cost;
    if (k == n) {
        new_cost := cost(val);
        if (new_cost < best_cost) {
            best_cost := new_cost;
            best_solution := copy(val);
        }
    }
    else if (lower_bound_cost(val, k) ≥ best_cost)
        /* No action, node is killed. */
    else
        for each (el ∈ allowed(val, k)) {
            val[k] := el;
            backtrack(k + 1);
        }
}
}

main ()
{
    best_cost := ∞;
    b.and.b(0);
    report(best_solution);
}

```

Figure 5.5 The pseudo-code of the branch-and-bound algorithm.

based on the function \tilde{c} . A next level of recursion is entered only if the node in the search tree cannot be killed.

An important issue is how to define \tilde{c} such that it is useful. It is quite natural to compose it out of two terms:

$$\tilde{c}(\tilde{\mathbf{f}}^{(k)}) = \tilde{g}(\tilde{\mathbf{f}}^{(k)}) + \tilde{h}(\tilde{\mathbf{f}}^{(k)})$$

where $\tilde{g}(\tilde{\mathbf{f}}^{(k)})$ is a function that is computed given the specified variables of $\tilde{\mathbf{f}}^{(k)}$ and $\tilde{h}(\tilde{\mathbf{f}}^{(k)})$ is a term that is based on the unspecified variables of $\tilde{\mathbf{f}}^{(k)}$. Taking the traveling salesman problem as an example, $\tilde{g}(\tilde{\mathbf{f}}^{(k)})$ can give the length of the path fixed by the specified variables and $\tilde{h}(\tilde{\mathbf{f}}^{(k)})$ can give a lower bound on the remaining tour length. One possibility is to define $\tilde{h}(\tilde{\mathbf{f}}^{(k)}) = 0$ for all partial solutions. This certainly satisfies the requirement that all final solutions in the set $D(\tilde{\mathbf{f}}^{(k)})$ have a cost higher than the estimated one. However, this definition does not help much to reduce the search space. A more effective possibility is to use the length of the *minimum spanning tree* (see Section 3.4.4) as a lower bound. Given a sequence of vertices that represents

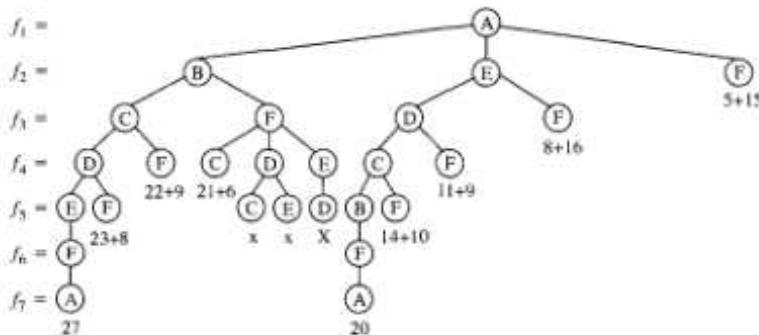


Figure 5.6 The search tree for the TSP example of Figure 5.3 when branch-and-bound is used.

a partial solution, one knows that the fully-specified solution derived from it will somehow connect the vertices not yet in the sequence with the first and last vertex. Call the set consisting of the vertices not in the sequence augmented by the first and last one of the sequence the *remaining* vertices. A path that connects all remaining vertices and that starts at the last one already in the sequence and finishes at the first vertex already in the sequence, will turn the partial solution into a fully-specified one. Note that such a path is a spanning tree in the subgraph induced by the remaining vertices. So, the length of the minimum spanning tree in this subgraph is a lower bound on the length of the required path.

The branch-and-bound search tree for the TSP example of Figure 5.3 that is obtained using a function $\hat{h}(\bar{\mathbf{f}}^{(k)})$ based on the minimal spanning tree, is shown in Figure 5.6. In the figure, a label ' $a + b$ ' means that the node is killed due to the fact that the sum exceeds the cost of the best known solution up to that moment. The term a is the value of $\tilde{g}(\bar{\mathbf{f}}^{(k)})$ and b is the value of $\hat{h}(\bar{\mathbf{f}}^{(k)})$. A label 'X' means that the subgraph induced by the remaining nodes is not connected and that a spanning tree does not exist. Clearly, as a consequence, a solution to TSP cannot be found in the subtree of that node. Note that the number of nodes in the search tree is 72 for the exhaustive search version and 27 for the branch-and-bound version.

The application of branch-and-bound search to the unit-size placement problem is more complex. As was mentioned before and will be discussed in more detail in Section 7.2, the cost function of this problem computes an estimate of the wire length on the basis of the positions of the cells. In a partial solution, only a subset of the cells has a fixed position. This makes it necessary to use more advanced estimation techniques that give lower bounds for the lengths of the wires connected to cells without a fixed position.

An essential point is that the function $\tilde{c}(\bar{\mathbf{f}}^{(k)})$ that computes the lower bound of the solutions in $D(\bar{\mathbf{f}}^{(k)})$ should, in general, be easier to compute than the mere traversal of the subtree at $\bar{\mathbf{f}}^{(k)}$ in order to have some computational gain. There is some preservation of the required effort here: more sophisticated functions can estimate

tighter lower bounds allowing the killing of nodes in an earlier stage, but they will normally require a higher computational effort themselves.

In this section only a simple version of backtracking (with and without branch-and-bound) has been presented. It traverses the search tree in a *depth-first* fashion (see also Section 3.4.1). In depth-first search, one focuses on a single partial solution of which a child at the next level is generated. Only when a solution has been killed or has been fully specified, does the algorithm go back to a higher level. The algorithm only keeps track of a single partial solution at a time.

Alternative ways to explore the search tree become possible if one is prepared to use *queues* to store intermediate nodes of the search tree. The principle is to evaluate *all* children of a node (the first node to be evaluated is, of course, the root) and store them in a queue. Then the elements already stored are removed one by one from the queue while their children are added to it. The partial solutions in the queue can be processed in different orders: first-in-first-out (FIFO), last-in-first-out (LIFO) and least-cost (LC). Using a FIFO queue will result in *breadth-first* search, while using a LIFO queue will amount to *depth-first* search. LC uses the function $\tilde{c}(\tilde{P}^k)$ to have an estimated cost of all elements in the queue and then first generates the children of the partial solution with minimal cost. Depending on the problem and its typical instances, using any of the search strategies just mentioned may lead to significant improvements in the time necessary to obtain an optimal solution.

More variations are possible. Instead of having the *static* search tree discussed above, in which partial solutions are generated by assigning values to the f_i in order of increasing index i , one can have *dynamic* search trees, both for depth-first and breadth-first search. In a dynamic search tree, those parameters are assigned a value first that will either lead to a fast detection of a solution or an early killing of a large subtree.

5.3 Dynamic Programming

Dynamic programming is a technique that systematically constructs the optimal solution of some problem instance by defining the optimal solution in terms of optimal solutions of smaller size instances. Suppose that some problem instance can be characterized by some "complexity parameter" p (this parameter is not necessarily the input size). Dynamic programming can be applied to such a problem if there is a rule to construct the optimal solution for $p = k$ from the optimal solutions of instances for which $p < k$. Given such a rule and the fact that the lowest complexity problem instances (those instances for which, say, $p = 0$ or $p = 1$) are easy to obtain, the optimal solutions for instances with any complexity can be constructed. The fact that an optimal solution for a specific complexity can be constructed from the optimal lower complexity problems only, is essential for dynamic programming. In this way, many parts of the search space can be discarded without visiting them. This idea is called the *principle of optimality*.

The explanation just given may be a little abstract. It will, therefore, be illustrated by applying dynamic programming to two example problems. The first problem is the shortest-path problem as solved by Dijkstra's algorithm and the second is TSP.

Following its presentation in Section 3.4.3, the goal in the shortest-path problem is to find the shortest path from a source vertex $v_s \in V$ to a destination vertex $v_t \in V$ in a directed graph $G(V, E)$ where the distance between two vertices $u, v \in V$ is given by the edge weight $w((u, v))$. Actually, no new shortest-path algorithm will be presented here, but Dijkstra's algorithm will be "reinterpreted" as an application of dynamic programming. The variable names used refer to those in the pseudo-code of Figure 3.14.

An instance of the shortest-path problem is extended with a parameter p for the purpose of dynamic programming. If $p = k$, the optimization goal becomes: find the shortest path from v_s to all other vertices in the graph considering paths that only pass through the first k closest vertices to v_s . Once the problem has been formulated in this way, Dijkstra's algorithm becomes an almost obvious solution to the shortest-path problem. The optimal solution for the instance with $p = 0$ is found in a trivial way by assigning the edge weight $w((v_s, u))$ to the `distance` attribute of all vertices $u \in V$. Suppose that the optimal solution for $p = k$ is known and that the k closest vertices to v_s have been identified and transferred from V to T . Then, solving the problem for $p = k + 1$ is simple: transfer the vertex u in V having the lowest value for its `distance` attribute from V to T and update the value of the `distance` attributes for those vertices remaining in V . A crucial observation is, of course, that the shortest path length from v_s to u has now been found. By the assumption, it is known that the `distance` attribute of u gives the length of the shortest path that passes through any of the vertices in T . Suppose now that the actual shortest path to u also passes through some vertex $w \in V$; then the shortest path length of u would at least equal $w.\text{distance} + w((w, u))$. This leads to a contradiction, however, because all edge weights are positive and $w.\text{distance} > u.\text{distance}$ (otherwise w would have been selected instead of u). Note that the reasoning just presented is a correctness proof of the algorithm (see also Exercise 3.5).

There is a single optimal solution for each value of p in the dynamic programming interpretation of Dijkstra's algorithm. One only needs to keep the optimal solution for $p = k$ for the computation of the optimum for $p = k + 1$. This solution is kept in the `distance` attributes of the vertices. In general, many optimal solutions for $p = k$ have to be stored such that they can be consulted when constructing solutions for the instances with $p = k + 1$. Said more precisely, additional parameters may be necessary to distinguish multiple instances of the problem for the same value of p . This is the case in the dynamic programming approach to the graph version of the traveling salesman problem discussed below.

As was the case in Section 5.2, the graph considered is an undirected graph $G(V, E)$, with edge weights $w((u, v))$ for an edge $(u, v) \in E$. Some vertex $v_s \in V$ should be chosen as the starting point of the tour (as any optimal tour visits all vertices, the actual choice for v_s is irrelevant). The introduction of a parameter p in TSP creates a set of problem instances derived from the original instance. If $p = k$,

the goal becomes to find the shortest paths to any $v \in V$ ($v \neq v_s$) starting from v_s that go through exactly k intermediate vertices.

In the following, $C(S, v)$ will denote the shortest path length from v_s to v exactly passing through the intermediate vertices given by the set S . The shortest tour length for the original instance of TSP is then given by $C(V \setminus \{v_s\}, v_s)$. The rule for constructing solutions with $p = k + 1 = |S|$ from solutions with $p = k$ is:

$$C(S, v) = \min_{m \in S} [C(S \setminus \{m\}, m) + w((m, v))] \quad (5.1)$$

Clearly, for each value of k , $C(S, v)$ has to be computed for all possible S and all possible v . The construction of the solution starts with the values $C(\emptyset, v)$ which equals the weight $w((v_s, v))$ when $(v_s, v) \in E$, and ∞ otherwise.

In the process of arriving at the final solution almost all subsets of V have to be considered (several times). This leads to a time complexity that is at least an exponential function of the number of vertices (a set with n elements has 2^n subsets). This is not surprising given the NP-completeness of TSP. However, the dynamic programming approach may still be interesting because it does not visit all points in the search space due to the elimination of all suboptimal solutions of size k when computing the solutions for size $k + 1$. It is comparable to the branch-and-bound approach in this sense, which also had an exponential worst-case time complexity in spite of the fact that it eliminates many parts of the search space.

This dynamic programming approach for TSP will now be illustrated for the example graph of Figure 5.3 where the choice $v_s = A$ has been made. First the values for $|S| = 0$ have to be computed:

$$\begin{aligned} C(\emptyset, B) &= 9 \\ C(\emptyset, C) &= \infty \\ C(\emptyset, D) &= \infty \\ C(\emptyset, E) &= 3 \\ C(\emptyset, F) &= 5 \end{aligned}$$

The next step is to compute all possible values for $|S| = 1$ applying the construction rule given in Equation (5.1). Below a selection of the 20 values to be computed is given (5 values for S and 4 for v not contained in S , when computing $C(S, v)$):

$$\begin{aligned} C(\{B\}, C) &= C(\emptyset, B) + w((B, C)) = 9 + 5 = 14 \\ C(\{B\}, F) &= C(\emptyset, B) + w((B, F)) = 9 + 4 = 13 \\ C(\{F\}, B) &= C(\emptyset, F) + w((F, B)) = 5 + 4 = 9 \end{aligned}$$

There are 30 values to be computed for $|S| = 2$. One of them is computed below:

$$\begin{aligned} C(\{B, F\}, C) &= \min [C(\{B\}, F) + w((F, C)), C(\{F\}, B) + w((B, C))] \\ &= \min [13 + 8, 9 + 5] = 14 \end{aligned}$$

This result indicates that the shortest path from A to C that passes through B and F has a length of 14 (by keeping track of the choices that led to this minimum, one could also report that the shortest path goes first to F and then to B). By continuing this process for ever larger sizes of $|S|$, one eventually computes $C(\{B, C, D, E, F\}, A)$ and obtains the solution of the example TSP instance.

The application of dynamic programming for the other example problem considered in this chapter, viz. unit-size placement, is not so obvious. It is not clear how the optimal placement of $k + 1$ cells should be defined in terms of the optimal solutions of all subproblems with k cells in such a way that parts of the search space are eliminated. An application of dynamic programming to a problem that occurs in a channel routing algorithm is presented in Section 9.3.4.

5.4 Integer Linear Programming

Integer linear programming (ILP) is a specific way of casting a combinatorial optimization problem in a mathematical format. More precisely, many combinatorial optimization problems can relatively easily be *reduced* to ILP. This does not help from the point of view of computational complexity as ILP is NP-complete itself. However, ILP formulations for problems from the field of VLSI design automation are often encountered due to the existence of "ILP solvers". ILP solvers are software packages that accept any instance of ILP as their input and generate an exact solution for the instance. As a consequence of the structure of an ILP formulation, the solver does not need any knowledge of the problem. The only effort to be made is to find the right reduction to correctly translate all features of the problem into the ILP format. This section actually only deals with the format and the translation process. The techniques used inside an ILP solver are not explained. It will seldom be necessary to implement an ILP solver because powerful solvers are available both commercially and in the public domain.

There are several reasons why ILP is useful in CAD for VLSI. First, the input sizes of the problems involved may be small enough for an ILP solver to find a solution in reasonable time. One then has an easy way of obtaining exact solutions, compared to techniques such as branch-and-bound (by the way, an ILP solver may use branch-and-bound itself). It may also be that one needs the exact solutions of some benchmark circuits in order to judge the merits of some problem-specific heuristics. Excessive run times may be acceptable in such a case. Finally, an ILP formulation may be a source of inspiration for the development of problem-specific heuristics.

5.4.1 Linear Programming

Integer linear programming is a special variant of *linear programming* (LP) and a basic understanding of LP is necessary in order to understand ILP. LP will be introduced by means of an example. Suppose that a factory produces two different food products P_1 and P_2 that are composed of the same two ingredients I_1 and I_2 in

different proportions. P_1 is made of a_{11} units of I_1 and a_{21} units of I_2 . Similarly, P_2 is made of a_{12} units of I_1 and a_{22} units of I_2 . The factory sells one unit of P_1 for price c_1 and one unit of P_2 for c_2 . The goal is to maximize the daily sales, given that the company can receive at most b_1 units of I_1 and b_2 units of I_2 per day. The demand is such that all production will be sold. The unknowns in this problem are x_1 and x_2 , the quantities of P_1 and respectively P_2 to be produced daily. Stated mathematically, the goal is to maximize:

$$c_1x_1 + c_2x_2$$

while satisfying the constraints:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 &\leq b_1 \\ a_{21}x_1 + a_{22}x_2 &\leq b_2 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \end{aligned}$$

Such a formulation is called a *linear program*. In general a linear program can be formulated in terms of a vector \mathbf{x} of variables, a cost vector \mathbf{c} and a matrix \mathbf{A} and a vector \mathbf{b} that determine the constraints (\mathbf{A} is not necessarily square!). The goal is either to maximize or minimize:

$$\mathbf{c}^T \mathbf{x}$$

while satisfying:

$$\begin{aligned} \mathbf{Ax} &\leq \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

Note that the expressions for the cost and constraints are *linear* in \mathbf{x} . The problem can be changed from a minimization problem to a maximization problem by simply multiplying \mathbf{c} by -1 . Similarly, one can convert a constraint containing a "less than or equal" sign to one containing containing a "greater than or equal" sign by multiplying by -1 the appropriate elements in \mathbf{A} and \mathbf{b} . The formulation just given is called the *canonical form* of LP. Although all variables are apparently restricted to be positive, this restriction can easily be overcome by replacing a variable x_i , that may assume negative values, by the difference $x_j - x_k$ of two new variables x_j and x_k that are both restricted to be positive.

Apart from the canonical form, LP also has a *standard form* in which the constraints look like:

$$\begin{aligned} \mathbf{Ax} &= \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

Here, the constraints to be satisfied are expressed as equations rather than inequalities. Inequalities can be converted to equations by adding to or subtracting from each

equation a new variable, a so-called *slack* variable. For example, $a_{11}x_1 + a_{12}x_2 \leq b_1$ becomes $a_{11}x_1 + a_{12}x_2 + x_3 = b_1$. Conversely, an equation can be replaced by a pair of inequalities when going from the standard form to the canonical form. One can for example replace $a_{11}x_1 + a_{12}x_2 = b_1$ by the two inequalities $a_{11}x_1 + a_{12}x_2 \leq b_1$ and $a_{11}x_1 + a_{12}x_2 \geq b_1$.

So-called "LP solvers" are available to solve LP problems. It is outside the scope of this text to present the algorithms used in LP solvers, but some interesting remarks can be made about them. It is possible to solve LP problems by a polynomial-time algorithm called the *ellipsoid algorithm*. This algorithm is, however, outperformed in practice by the *simplex algorithm* which has an exponential worst-case time complexity. LP is therefore one of the few problems for which an exponential algorithm is often preferable above a polynomial one. This illustrates the importance of average-case time complexity in practice.

5.4.2 Integer Linear Programming

ILP is a variant of LP in which the variables are restricted to be integers. Although it seems to be a minor modification, this restriction makes ILP NP-complete. The techniques used for finding solutions of LP are in general not suitable for ILP. It does, for example, not help to treat an instance of ILP as an instance of LP and then to round the results: the solution obtained may not be the optimal one or even a feasible one. Therefore, other techniques that more explicitly deal with the integer values of the variables should be used. As was mentioned before, however, solution techniques will not be discussed here.

In many situations, the integer variables are restricted further to assume either of the values zero or one. This variant of ILP is called *zero-one integer linear programming*. A way to obtain a zero-one ILP formulation for the TSP in graphs is presented below.

Consider a graph $G(V, E)$ where the edge set E contains k edges: $E = \{e_1, e_2, \dots, e_k\}$. The ILP formulation requires a variable x_i for each edge $e_i \in E$. The variable x_i can either have the value 1, which means that the corresponding edge e_i has been selected as part of the solution, or the value 0 meaning that e_i is not part of the solution. Given the fact that the weight of an edge e_i is denoted by $w(e_i)$, the cost function can be written as:

$$\sum_{i=1}^k w(e_i)x_i \quad (5.2)$$

In addition, the following restriction applies:

$$x_i \in \{0, 1\}, \quad i = 1, 2, \dots, k \quad (5.3)$$

In the optimal solution, only those x_i that correspond to edges in the optimal tour have a value 1. The constraints should enforce that the edges selected actually form

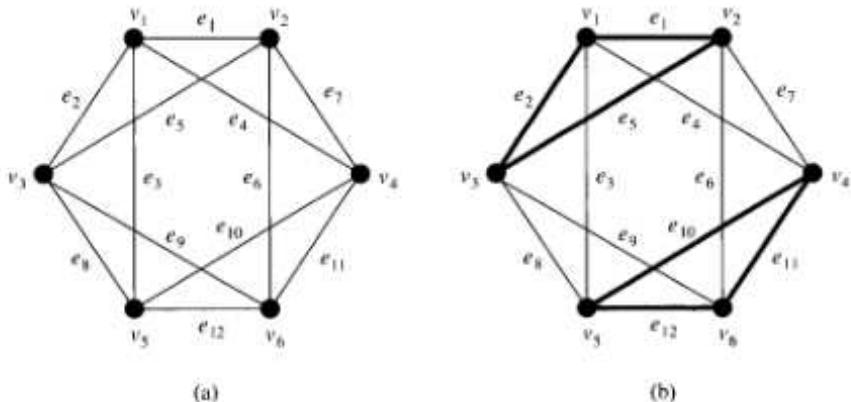


Figure 5.7 A graph to illustrate the ILP formulation of TSP (a) and an illegal solution consisting of two tours (b).

a tour that visits all the vertices. A necessary (but not sufficient) condition to achieve this is that exactly two edges of those incident to a vertex are selected. In the example of Figure 5.7(a), this means that the following constraints should be satisfied:

$$\begin{aligned}
 v_1 : & x_1 + x_2 + x_3 + x_4 = 2 \\
 v_2 : & x_1 + x_5 + x_6 + x_7 = 2 \\
 v_3 : & x_2 + x_5 + x_8 + x_9 = 2 \\
 v_4 : & x_4 + x_7 + x_{10} + x_{11} = 2 \\
 v_5 : & x_3 + x_8 + x_{10} + x_{12} = 2 \\
 v_6 : & x_6 + x_9 + x_{11} + x_{12} = 2
 \end{aligned} \tag{5.4}$$

These constraints are, however, not enough to enforce a tour as they do not exclude that ILP finds a solution that consists of multiple disjoint tours. An example of such an illegal solution that satisfies the constraints is shown in Figure 5.7(b). More constraints are necessary to avoid these illegal solutions. These additional constraints can be obtained by looking at bipartitions of the vertex set: some subset $V_1 \subset V$ and its complement $V_2 = V \setminus V_1$. A tour that visits all vertices in the graph should pass through at least two of the edges that connect a vertex in V_1 with a vertex V_2 . If e.g. the subset $\{v_1, v_2, v_3\}$ is considered in the graph of Figure 5.7, it can be stated that a tour should pass through at least two of the vertices e_3, e_4, e_6, e_7, e_8 and e_9 . Such a condition prevents the occurrence of the illegal solution shown in Figure 5.7(b).

So, an additional set of constraints should be generated systematically to prevent solutions containing multiple tours based on the subsets of the vertex set. A number of observations can be made in this respect:

- All relevant subsets V_1 should be taken into consideration in order to prevent illegal solutions.

- Both V_1 and V_2 should contain at least three vertices (no multiple tours are otherwise possible). Also, all vertices in any subgraph induced by either V_1 or V_2 should have at least a degree two. Otherwise, no multiple subtours covering all vertices are possible and it is therefore not necessary to have extra constraints to avoid them.

Taking these observations into account leads to the following relevant partitions and corresponding constraints for the graph of Figure 5.7:

$$\begin{aligned} \{v_1, v_2, v_3\} + \{v_4, v_5, v_6\} : x_3 + x_4 + x_6 + x_7 + x_8 + x_9 \geq 2 \\ \{v_1, v_3, v_5\} + \{v_2, v_4, v_6\} : x_1 + x_4 + x_5 + x_9 + x_{10} + x_{12} \geq 2 \\ \{v_1, v_2, v_4\} + \{v_3, v_5, v_6\} : x_2 + x_3 + x_5 + x_6 + x_{10} + x_{11} \geq 2 \end{aligned} \quad (5.5)$$

In summary, Equations (5.2) through (5.5) define the ILP formulation of TSP for the graph of Figure 5.7. The formulation given is neither in the canonical form nor in the standard form, but it can easily be converted to any of the two forms using the "tricks" mentioned in Section 5.4.1.

An interesting issue is the size of the ILP formulation with respect to the size of the problem instance. In the example above, the number of variables is equal to the number of edges. The number of constraints of the type presented in Equation (5.4) is equal to the number of vertices. The number of constraints of the type given in Equation (5.5), however, can grow exponentially as the number of subsets of V equals $2^{|V|}$. If additional *real-valued* variables are allowed in the formulation, an alternative set of constraints can be constructed whose size is a polynomial function of the number of vertices. Such a formulation will not be presented here (see the Bibliographic Notes at the end of the chapter for pointers to the literature). Linear programming that contains both integer as well as real variables is called *mixed integer linear programming*.

ILP can be used for a large range of combinatorial optimization problems. An important reason why ILP *cannot* directly be used is a nonlinear cost function. The unit-size placement problem is an example of such a problem. Its cost function is an estimation of the wire length and there is no obvious way to relate e.g. cell locations to wire lengths using linear operators (the estimation methods mentioned in Section 7.2 use operations like squaring, minimum, maximum and absolute value).

5.5 Local Search

Local search is a general-purpose optimization method that works with fully-specified solutions $f \in F$ of a problem instance (F, c) . It makes use of the notion of a *neighborhood* $N(f)$ of a feasible solution f , which consists of a subset of F that is "close" to f in some sense. More formally, a neighborhood is a function that assigns a number of feasible solutions to each feasible solution: $N : F \rightarrow 2^F$. Here, 2^F denotes the *power set* of F , the set of all its subsets. Any $g \in N(f)$ is called a *neighbor* of f (with respect to the neighborhood N).

```

local_search()
|
struct feasible_solution f;
set of struct feasible_solution G;

f  $\leftarrow$  initial_solution();
do [
    G  $\leftarrow$  {g | g  $\in$  N(f), c(g)  $<$  c(f)};
    if (G  $\neq$   $\emptyset$ )
        f  $\leftarrow$  "any element of G";
    while (G  $\neq$   $\emptyset$ );
    "report f";
]

```

Figure 5.8 The pseudo-code description of local search.

Consider for example the unit-size placement problem in which each cell has to be assigned two coordinates with the goal of minimizing the estimated wire length. The neighborhood of a feasible solution *f* could consist of those solutions in *F* that can be obtained from *f* by exchanging the coordinates of two cells. Supposing that the instance has *n* cells, each feasible solution has a neighborhood of $\frac{n(n-1)}{2}$ elements, while the size of the complete search space is in the order of *n!*. More complex neighborhoods could be obtained by selecting three or more cells and interchanging their coordinates. In an extreme case, a neighborhood involving all *n* cells could be defined, which means that the neighborhood of each feasible solution *f* \in *F* consists of the complete search space *F*. In such a case, as will become clear later, local search becomes equivalent to exhaustive search.

The principle of local search is to subsequently visit a number of feasible solutions in the search space, each solution being in the neighborhood of the previous one. Such a transition from one solution to the next is called a *move* or a *local transformation*. One starts with a feasible solution *f* and then moves to some *g* \in *N(f)*, such that *c(g)* $<$ *c(f)*. This is repeated until some feasible solution is found that is cheaper than all its neighbors. The pseudo-code presenting this process is given in Figure 5.8. It is assumed that feasible solutions can be represented by the data structure **feasible_solution**. The function **initial_solution** has the task to generate a feasible solution. This could be done randomly or by some problem-specific heuristic. The moves that transform the initial solution into the final one are executed in the body of the **do** loop. In a practical implementation, it is not always necessary to generate the complete set *G* of all cheaper neighbors of *f*, unless one is specifically interested in selecting the cheapest one as the next value of *f*. A strategy that moves to the first cheaper neighbor encountered is called *first improvement* and the one moving to the cheapest one in *G* is called *steepest descent*. The latter strategy clearly requires a larger computational effort than the former one. It may still be interesting because of the belief that the final solution will be reached after fewer iteration steps or that a better quality solution will be found. Neither of these

is guaranteed, though.

The effectiveness of local search depends on the "shape" of the cost function c . If the function has a single minimum, it will be found sooner or later. However, most of the interesting combinatorial optimization problems have cost functions with many minima, most of which are local, while only one (or a few) are global. So, local search has the property that it can get stuck in a local minimum. This disadvantage cannot be fully avoided unless the extreme neighborhoods that coincide with the whole search space are considered. In general, the larger the neighborhoods considered, the larger is the part of the search space explored and the higher is the chance of finding a solution with good quality. However, the investigation of larger neighborhoods also requires a larger computational effort.

One can generalize local search in several ways. One possibility is to repeat the search a number of times with different initial solutions. In this way, it becomes more likely that larger parts of the search space will be explored and multiple minima will be encountered. Another possibility is to adapt the size of the neighborhood during local search based on properties of the feasible solution that one is visiting.

In order to circumvent the main disadvantage of local search, viz. the fact that it will get stuck in a local minimum, one should be able to move to a solution with a higher cost, by means of so-called *uphill moves*. However, this should be done in a way that still guarantees convergence to some solution. The methods *simulated annealing* and *tabu search*, to be discussed in the next sections, are examples of methods operating in this way.

5.6 Simulated Annealing

Simulated annealing (sometimes also called *statistical cooling*) performs a computation that is analogous to a physical process. In the process concerned, a material is first heated up to a temperature that allows all its molecules to move freely around (the material becomes liquid), and is then cooled down very slowly. The freedom of movement for the molecules decreases gradually until all the molecules take a fixed position. At the end of the process, the total energy of the material is minimal provided that the cooling is very slow. The idea of applying this analogy to combinatorial optimization was first published by Kirkpatrick *et al.* in an article that included applications from VLSI design automation (placement and global routing) as examples.

The analogy with the physical model has the following points of correspondence with a combinatorial optimization problem:

- The energy corresponds to the cost function.
- The movement of the molecules corresponds to a sequence of moves in the set of feasible solutions.
- The temperature corresponds to a control parameter T which controls the acceptance probability for a move from $f \in F$ to $g \in N(f)$. A good move, i.e. a

```

int accept(struct feasible_solution f, g)
{
    float  $\Delta c$ ;  

     $\Delta c \leftarrow c(g) - c(f)$ ;
    if ( $\Delta c \leq 0$ )
        return 1;
    else return ( $e^{-\frac{\Delta c}{T}} > \text{random}(1)$ );
}

simulated_annealing()
{
    struct feasible_solution f, g;
    float T;  

    f  $\leftarrow$  initial_solution();
    do {
        do {
            g  $\leftarrow$  "some element of  $N(f)$ ";  

            if (accept(f, g))
                f  $\leftarrow g  

            while (!thermal_equilibrium());
            T  $\leftarrow$  new_temperature(T);
            while (!stop());
            "report f";
        }
    }
}$ 
```

Figure 5.9 A pseudo-code description of simulated annealing.

move for which $c(g) \leq c(f)$, is always accepted irrespective of the value of *T*. A bad move, for which $c(g) > c(f)$, is accepted with a probability $e^{-\frac{\Delta c}{T}}$, where $\Delta c = c(g) - c(f)$. So, for high values of *T* nearly all bad moves are accepted, while hardly any bad move is accepted when *T* is low. (The Boltzmann distribution in statistical mechanics states that the number of molecules N_1 with energy level ϵ_1 divided by the number of molecules N_0 with energy level ϵ_0 equals $e^{\frac{\Delta \epsilon}{kT}}$, where $\Delta \epsilon = \epsilon_1 - \epsilon_0$, k is the Boltzmann constant and *T* is the absolute temperature.)

The algorithm itself consists of an outer loop in which the temperature is gradually lowered and an inner loop in which the configuration is randomly perturbed by moves that are either accepted or rejected. The inner loop should be executed a number of times large enough to reach "thermal equilibrium" before going back to the outer loop. This is shown in the pseudo-code description given in Figure 5.9. The strategy for accepting or rejecting a move is represented by the function *accept*, in which the function *random(k)* generates a real-valued random number between 0 and *k* with a uniform distribution. (Check that the code of *accept* correctly implements the acceptance strategy described above.) The function *thermal_equilibrium*

should only return a value unequal to zero if the inner loop has been executed a “sufficient” number of times. This number is normally a function of the problem instance size. The function `new.temperature` computes a new, lower, value for the temperature to be used for the next execution of the inner loop. In practice, this is often implemented by a multiplication of T by a constant between 0 and 1. The function `stop`, finally, decides about the termination of the search. One possibility is to stop when none of the moves in the inner loop has been accepted. Note that simulated annealing may visit an optimal solution and then move away from it. It is therefore wise to record the best solution in a separate variable and report its value at the end of the search instead of the final value of f .

The combination of the functions `thermal.equilibrium`, `new.temperature` and `stop` realizes a strategy for simulated annealing, which is called the *cooling schedule*. Theoretical analysis shows that the cooling schedule can be chosen in such a way that the probability of finding the global minimum becomes equal to one. However, these schedules imply an infinite number of moves before stopping. In practice, one is interested in finding a solution as fast as possible and uses cooling schedules that cannot guarantee an optimal solution.

Simulated annealing has been applied with varying success to almost any combinatorial optimization problem in VLSI design automation. Especially in the case of the placement problem (see Chapter 7), it has proved itself as one of the best methods, if not the overall best one. On the other hand, when good tailor-made heuristics are available, they are often preferable to simulated annealing: they obtain better quality results in a fraction of the time needed for simulated annealing. An adapted version of the Kernighan-Lin algorithm for graph partitioning (see Section 7.5.1), for example, clearly outperforms simulated annealing when applied to circuit partitioning (a problem that is somewhat more complex than graph partitioning).

5.7 Tabu Search

Simulated annealing allows many uphill moves at the beginning of the search and gradually decreases their frequency. In this way, a convergence mechanism is imposed to the search. The *tabu search* method, on the other hand, does not directly restrict uphill moves throughout the search process. Given a *neighborhood subset* $G \subseteq N(f)$ of a feasible solution f , the principle of tabu search is to move to the cheapest element $g \in G$ even when $c(g) > c(f)$. In order to avoid a circular search pattern, a so-called *tabu list* containing the k last visited feasible solutions is maintained. Transitions to these solutions are prohibited (they are *taboo*, hence the name of the method). This only helps, of course, to avoid cycles of length $\leq k$.

A pseudo-code description of tabu search is given in Figure 5.10. In the description, the tabu list is represented by the FIFO (first-in first-out) queue Q of length k . The variable b is used to store the best solution encountered in the search process. The function `stop` decides when to terminate the search, e.g. when no improvement on the best solution is found in the last n iterations.

```

tabu_search()
{
    struct feasible_solution f, g, b;
    set of struct feasible_solution G;
    "k-element FIFO queue of" feasible_solution Q;

    Q ← "empty";
    b ← initial_solution();
    f ← initial_solution();
    do {
        G ← "some subset of N(f) such that ∀s ∈ Q, s ∉ G";
        if (G ≠ ∅) {
            g ← "cheapest element of G";
            "shift g into Q";
            f ← g;
            if (c(f) < c(b))
                b ← f;
        }
    }
    while (G ≠ ∅ or stop());
    "report b";
}

```

Figure 5.10 A pseudo-code description of tabu search.

Consider as an example an instance of the unit-size placement problem, where 10 000 cells have to be placed on grid points identified by 100 x-coordinates and 100 y-coordinates. Suppose that a move selects two cells and exchanges their places. For such a problem, the quantity of data to be stored for each solution kept in the tabu list is quite high: the 10 000 coordinate pairs that define a feasible solution. The data to be stored can be drastically reduced by e.g. only storing the two cells that have been moved to arrive at the solution, which implies that these two cells will not be moved during the next k iterations. The price for the storage reduction is a reduction of the search space: many feasible solutions that would not give rise to a circular search pattern will also remain out of consideration as long as the cells are kept in the taboo list.

The tabu search method described above is the most elementary form of the many variations possible. Actually, tabu search should be seen as a large family of optimization methods offering (and sometimes requiring) extensive adaptation to the combinatorial optimization problem at hand. An unsatisfactory aspect of tabu search is that no theoretical analysis exists to guide the choice for the parameter k (the tabu list size) and the stopping criterion: they should be determined empirically.

5.8 Genetic Algorithms

In this section a simple version of search by *genetic algorithms* is presented. Many variations on this version are possible and some of them will be mentioned at the end of this section.

As in local search and its related methods, a genetic algorithm also works with fully specified solutions f included in the set of feasible solutions F . However, instead of repetitively transforming a single current solution into a next one by the application of a move, the algorithm simultaneously keeps track of a set P of feasible solutions, called the *population*. In an iterative search process, the current population $P^{(k)}$ is replaced by the next one $P^{(k+1)}$ using a procedure that is characteristic for genetic algorithms.

In order to generate a feasible solution $f^{(k+1)} \in P^{(k+1)}$, two feasible solutions $f^{(k)}$ and $g^{(k)}$, called the *parents* of the *child* $f^{(k+1)}$, are first selected from $P^{(k)}$. $f^{(k+1)}$ is generated in such a way that it *inherits* parts of its "properties" from one parent and the other part from the second parent by the application of an operation called *crossover*. First of all, this operation assumes that all feasible solutions $f \in F$ can be encoded by a fixed length vector $\mathbf{f} = [f_1 \ f_2 \ \dots \ f_n]^T \equiv f$ as was the case for the backtracking algorithm discussed in Section 5.2. In their basic form, genetic algorithms use bit strings to represent feasible solutions. This not only implies that the number of vector elements n is fixed, but that the number of bits to represent the value of each element f_i ($1 \leq i \leq n$) is fixed as well. The string of bits that specifies a feasible solution in this way, is called a *chromosome*. Consider an instance of the unit-size placement problem with 100 cells and a 10×10 grid. As 4 bits are necessary to represent one coordinate value (each value is an integer between 1 and 10) and 200 coordinates (100 coordinate pairs) specify a feasible solution, the chromosomes of this problem instance have a length of 800 bits.

It is important to be aware of the distinction between a feasible solution (in biological terms, the *phenotype*) and its encoding as a chromosome (its *genotype*). In the rest of this section, however, an $f \in F$ that is formally the phenotype, is sometimes used to denote the genotype as well in order to avoid the introduction of a more complex notation.

Given two chromosomes, a crossover operator will use some of the bits of the first parents and some of the second parent to create a new bit string representing the child. A simple crossover operator works as follows:

- Generate a random number r between 1 and the length l of the bit strings for the problem instance.
- Copy the bits 1 through $r - 1$ from the first parent and the bits r through l from the second parent into the bit string for the child. Sometimes, it is customary to generate a second child using the same r , now reversing the roles of both parents when copying the bits.

The generation of a pair of children $f^{(k+1)}$ and $g^{(k+1)}$ from a pair of parents $f^{(k)}$ and $g^{(k)}$ is illustrated in Figure 5.11. The example also illustrates a complication

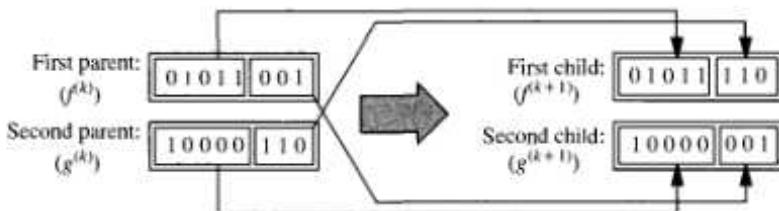


Figure 5.11 The generation of a pair of children by crossover.

that arises in genetic algorithms due to the encoding of the solution as a bit string. Suppose that the bit strings of the example represent the coordinates of the placement problem on a 10×10 grid mentioned earlier, now with only a single cell to place (an artificial problem). The bit string for a feasible solution is then obtained by concatenating the two 4-bit values of the coordinates of the cell. So, $f^{(k)}$ is a placement on position (5, 9) and $g^{(k)}$ one on position (8, 6). The children generated by crossover represent placements at respectively (5, 14) and (8, 1). Clearly, a placement at (5, 14) is illegal: it does not represent a feasible solution as coordinate values cannot exceed 10.

One solution to this complication is to use strings that consist of more sophisticated data structures than single bits. A chromosome could then be represented as a vector of e.g. integers or characters. In this way, the complication mentioned in the previous paragraph cannot occur. The combination of the chromosome representation and the crossover operator for generating new feasible solutions, leads, however, to more complications. Consider e.g. the traveling salesman problem for which each of the feasible solutions can be represented by a permutation of the cities (at least in those versions of the problem where there is a direct connection between each pair of cities, such as in the Euclidean version). Two example chromosomes for a six-city problem instance with cities c_1 through c_6 could then look like " $c_1c_3c_6c_5c_2c_4$ " and " $c_4c_2c_1c_5c_3c_6$ ". In such a situation, the application of the crossover operator as described for binary strings is very likely to produce solutions that are not feasible. Performing crossover on the two chromosomes just mentioned by cutting them after the second city would lead to the illegal solution " $c_1c_3c_1c_5c_3c_6$ " (or " $c_4c_2c_6c_5c_2c_4$ ") that is not a permutation of all cities in the problem instance.

This behavior can be avoided by using special-purpose crossover operators tailored for chromosomes that represent permutations. One such operator is called *order crossover*. This operator copies the elements of the first parent chromosome until the point of the cut into the child chromosome. The remaining part of the child is composed of the elements missing in the permutation in the *order* in which they appear in the second parent chromosome. Consider again the chromosomes " $c_1c_3c_6c_5c_2c_4$ " and " $c_4c_2c_1c_5c_3c_6$ " cut after the second city. Then the application of order crossover would lead to the child " $c_1c_3c_4c_2c_5c_6$ " (reversing the roles of the first and second parents would lead to: " $c_4c_2c_1c_3c_6c_5$ ").

The aspects of genetic algorithms that have been introduced until now do not

```

genetic()
{
    int pop_size;
    set of struct chromosome pop, newpop;
    struct chromosome parent1, parent2, child;

    pop ← ∅;
    for (i ← 1; i ≤ pop_size; i ← i + 1)
        pop ← pop ∪ {"chromosome of random feasible solution"};
    do {
        newpop ← ∅;
        for (i ← 1; i ≤ pop_size; i ← i + 1) {
            parent1 ← select(pop);
            parent2 ← select(pop);
            child ← crossover(parent1, parent2);
            newpop ← newpop ∪ {child};
        }
        pop ← newpop;
    } while (!stop());
    "report best solution";
}

```

Figure 5.12 The pseudo-code description of a genetic algorithm.

lead to any optimization. One needs to favor good solutions above bad solutions in some way. This is done by giving a stronger preference to parents with a lower cost when selecting pairs of parents to be submitted to the crossover operator. So, the better the cost of some feasible solution, the higher the chances that it will be selected for reproduction. Children generated in this way combine “good” features of their parents and lead to improved solutions. The reverse, children that mainly inherit “bad” features of their parents, can also happen. Those children are, however, unlikely to survive the transition to the next generation.

The pseudo-code description of search by genetic algorithms presented in Figure 5.12 summarizes the concepts introduced above and gives some more details as well. Note that the description in the figure deals with chromosomes that are manipulated, not the feasible solutions themselves. In the main loop of the code, two parents at a time are selected and used to generate one new child in the new population. The function `select` is responsible for the selection of feasible solutions from the current population favoring those that have a better cost. The function `crossover` actually generates a new child from two parent chromosomes. The function `stop` decides when to terminate the search, e.g. when there has been no improvement on the best solution in the population during the last m iterations, where m is a parameter of the algorithm.

A final main feature of genetic algorithms is *mutation*. This is a phenomenon also encountered in nature, viz. that errors can be made during the copying of a chromosome from a parent to the child (this happens inside the procedure `crossover` in

the version described here). Mutation helps to avoid getting stuck in a local minimum.

Many variations are possible to the scheme presented above. Examples are:

- One can work with more sophisticated crossover operators, e.g. operators that make multiple cuts in a chromosome.
- One can copy some members of the population entirely to the new generation instead of generating new children from them.
- Instead of distinguishing between the populations $P^{(k)}$ and $P^{(k+1)}$, one could directly add a new child to the population and simultaneously remove some "weak" member of the population. This allows for the coexistence of different generations as is often the case in nature.

5.9 A Few Final Remarks on General-purpose Methods

Quite a variety of general-purpose methods for combinatorial optimization has been presented in this chapter. Even more methods exist, such as *neural networks* and *simulated evolution*, to name just two methods that have sometimes successfully been applied to problems in VLSI design automation. Yet another method, based on the *satisfiability* problem, is discussed in Chapter 11 in the context of Boolean function manipulation.

The advantage of using a general-purpose method for some problem is clearly in the reduction in algorithm design time as compared to development time for a special-purpose method. Sometimes, a general-purpose method directly leads to a powerful solution method that cannot be outperformed by any special-purpose method. This is e.g. the case for placement by simulated annealing. It may, however, be that the general-purpose method gives unsatisfactory results. This is often due to the loss of problem-specific knowledge that occurs when trying to reformulate a specific problem in terms of the general-purpose method. Genetic algorithms, for example, require a feasible solution to be represented by a chromosome which is a linear data structure, whereas a more complex data structure may be a more natural representation of the solution. Or, it may not always be easy to find "moves" for local search that do not create illegal solutions, due to the complexity of the representation on which the moves have to operate.

Sometimes good results can be obtained by a *hybrid* algorithm that has both general-purpose and problem-specific parts. One may e.g. have a problem-specific heuristic that can be controlled by means of some parameters (different solutions for a problem instance will be generated for different parameter values) and that has a low time complexity. Such a heuristic should have the property that any interesting part of the search space, and especially the part that contains the optimal solution, can be reached by the correct choice of the parameter values. Besides, the parameter values themselves should be simple data structures, such as integers or the permutations of a finite set. A general-purpose method can then be used to find those

parameter values that will lead to a good solution by repetitively calling the heuristic with different values. What has been achieved is that the heuristic has replaced the complex data structures required to adequately represent a feasible solution, by a simple data structure suitable for a successful application of the general-purpose method. One can also say that the heuristic has become a sophisticated cost function that calculates the cost of a feasible solution represented by the parameters.

5.10 Bibliographic Notes

A book that presents general-purpose heuristics for combinatorial optimization, including many of those presented in this chapter is [Ree95].

Backtracking and branch-and-bound are clearly explained in [Hor78]. Some of the terminology used here in the explanation of this topic also originates from this book. The idea of using the minimum spanning tree length as a lower bound for the unspecified part of the partial solution of a traveling salesman problem is mentioned in [Sed88].

Dynamic programming is a common technique that is discussed in many books on combinatorial optimization. A short but clear presentation is e.g. given in [Pap82]. The application of dynamic programming to TSP as presented in this chapter is based on this source.

Many books and review papers can be found on linear and integer linear programming. A readable introduction to these topics as well as some other issues in combinatorial optimization is given in [Fou84]. Other examples are [Hu69], [Pap82], [Nem88], and [Nem89]. These sources pay attention to ILP formulations of TSP including the one that uses mixed integer linear programming. All formulations are based on directed graphs as opposed to the version for undirected graphs discussed in Section 5.4.2. The NP-completeness of ILP is mentioned in [Gar79].

The discussion on local search given in this chapter is based on [Pap82]. The seminal paper on simulated annealing including its application to VLSI design automation is [Kir83]. A theoretical study for the optimal cooling schedule is provided in [Haj88]. Readers interested in an extended analysis of the convergence of simulated annealing are referred to books dedicated to the topic, like [Laa87], [Aar89] and [Ott89]. One of the best results of simulated annealing in VLSI design automation has been obtained by the Timberwolf placement program as reported in [Sec85] and [Sun95]. The comparison between the Kernighan-Lin algorithm and simulated annealing for the placement problem, as reported in this chapter, is made in [Nah89]. The book [Won88] is entirely dedicated to applications of simulated annealing to VLSI layout design problems. The book [Ban94] that deals with parallel algorithms for VLSI design automation, also covers a number of layout applications of simulated annealing.

A review of tabu search and possible variations is given in [Glo93]. A successful application of tabu search in VLSI design automation has been reported for the high-level synthesis scheduling problem in [Ame94].

The presentation of genetic algorithms in this chapter roughly follows the description given in [Gol89]. Other books on the topic are [Dav91] and [Mic92a]. Genetic algorithms have been applied to various combinatorial optimization problems arising in VLSI design automation. Interesting results have e.g. been reported for the standard-cell placement problem [Coh87, Sha90]. The results obtained are comparable in quality to simulated annealing [Sec85], but at the expense of longer run times. It is remarkable, though, that the number of feasible solutions considered by the genetic algorithm reported in [Sha90] is 10 to 50 times less than in simulated annealing (so the computational effort per feasible solution is relatively high).

More information on the use of neural networks for combinatorial optimization can be found in [Hop86], [Ram88], [Bou90], and [Phi95]. Examples of the application of simulated evolution to VLSI design automation are presented in [Kli89] and [Lin89]. The application of hybrid genetic algorithms to high-level synthesis has been reported in [Hei95] and [Hei96]. They have later been extended for *multiprocessor scheduling* in [Bon97].

Part II

Selected Design Problems and Algorithms

6

Layout Compaction

As mentioned in Chapter 1, one of the goals in VLSI design is to minimize the area of the final circuit. Design decisions at all levels of abstraction have a consequence for the area of the final circuit. At the lowest level, the level of the mask patterns for the fabrication of the circuit, a final optimization can be applied to remove redundant space. This optimization is called *layout compaction*.

The main topic of this chapter is the so-called *constraint-graph compaction* method. Before introducing it, some attention is paid to the notions of *design rules* and *symbolic layout*. Then, the compaction problem is formulated in graph-theoretical terms for the purpose of constraint-graph compaction and algorithms to solve the problem are presented. The chapter concludes with a short presentation of related topics that cannot be covered in depth.

6.1 Design Rules

The mask patterns that are used for the fabrication of an integrated circuit have to obey certain restrictions on their shapes and sizes. These restrictions are called the *design rules*. Sticking to the design rules decreases the probability that the fabricated circuit will not work due to shortcircuits, disconnections in wires, parasitics, etc.

The shape of the patterns is often restricted to *rectilinear polygons*, i.e. polygons that are made of horizontal and vertical segments only (see Figure 6.1(a)). Some technologies also allow 45-degree segments in polygons, segments that are parallel

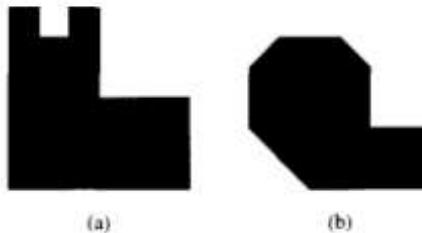


Figure 6.1 A rectilinear polygon (a) and one with 45-degree segments (b).

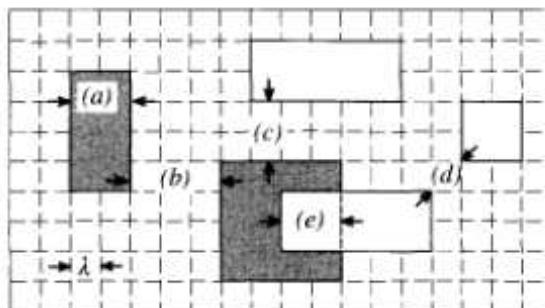


Figure 6.2 Minimum-distance design rules on a lambda grid: minimum width (a), minimum separation (b, c and d) and minimum overlap (e).

to the lines $y = x$ or $y = -x$ on an x-y plane (see Figure 6.1(b)). In the rest of this text, for reasons of simplicity, the mask patterns will be assumed to consist of rectangles only (a rectilinear polygon can be decomposed into a set of rectangles). There are design rules for layout elements located in the same fabrication layer and rules for elements in different layers. If patterns in two specific layers are constrained by one or more design rules, the layers are said to *interact*. For example, polysilicon and diffusion are interacting layers as their overlapping creates a transistor, whereas polysilicon and metal form noninteracting layers (if one ignores parasitic capacitances). Design rules can be quite complex. However, most of them can be expressed as *minimum-distance* rules.

As the minimum feature size that can be realized on a chip is subject to continual change (from several microns a few years ago to a few tenths of microns nowadays), distances are often expressed in integer multiples (or small fractions) of a relative length unit, the λ , rather than absolute length units. In this way, designers can deal with simple expressions independent of actual length values. This means that all mask patterns are drawn along the lines of a so-called *lambda grid* as shown in Figure 6.2.

The most common types of minimum-distance rules are:

- *Minimum width*: a pattern in a certain layer cannot be narrower than a certain distance (see Figure 6.2(a)).
- *Minimum separation*: two patterns belonging to the same (see Figure 6.2(b)) layer or to different but interacting layers (see Figure 6.2(c)) cannot be positioned closer to each other than a certain distance; this is also true when the rectangles are diagonally separated (see Figure 6.2(d)).
- *Minimum overlap*: a pattern in one layer located on top of a pattern in another interacting layer, should have a minimal overlap (see Figure 6.2(e)).

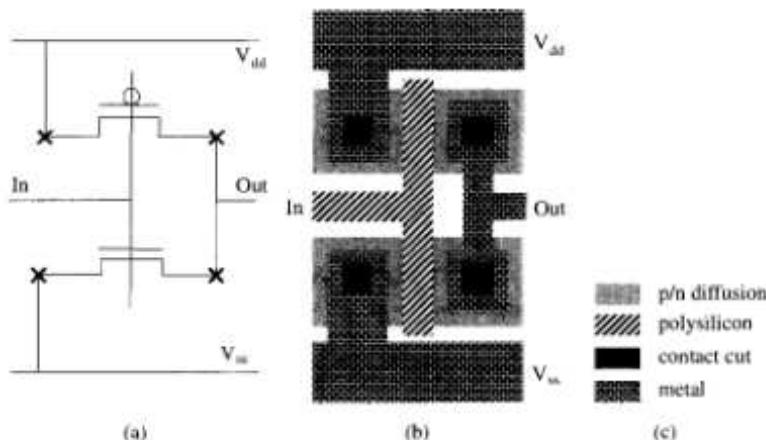


Figure 6.3 The symbolic layout of a CMOS inverter (a), its geometric layout (b) and a legend explaining the layers in the geometric layout (c).

6.2 Symbolic Layout

The existence of design rules makes the design of layout very cumbersome; there are many design rules and a human designer can overlook one of them very easily when fixing the position of a certain rectangle. Although special tools, so-called *design-rule checkers*, exist for detecting this type of mistakes, the mistake might not be easy to correct, requiring one to reposition many other rectangles in the neighborhood of the wrong one. As a remedy to this problem, it has been proposed not to design the *geometry* of the layout directly, but only to fix its *topology* instead. In this context, geometry means that the coordinates of the rectangles are absolute (or in multiples of λ), and topology means that only relations between layout elements, such as "to the left of" or "below", are known. In the rest of this chapter, a distinction will be made between *geometric* or *mask* layout on one hand, and *symbolic* or *topological* layout on the other hand.

The entities of which a symbolic layout consists are, not surprisingly, symbols: there is a symbol for a transistor, a wire in a specific layer, a contact cut, etc. In a symbolic layout, the interconnections of these symbols fix the topology of the circuit: the relative positions of transistors, the shapes of wires, etc. are known. The symbolic layout of a simple CMOS inverter is given in Figure 6.3(a).

Symbolic layout can normally be created interactively on a graphics computer screen, by means of a *symbolic layout editor* or it can be specified in textual form, by means of a formal *layout language*.

The advantages of symbolic layout can only be exploited if tools exist for automatically converting symbolic layout into geometric layout. Such a tool should e.g. accept some description of Figure 6.3(a) together with a *technology file* that contains all design rule information for the target technology, and produce the layout of Figure 6.3(b) (the shading patterns used to identify the different layers are given in

Figure 6.3(c)). The conversion from symbolic to geometric layout is one of the applications of layout compaction. Some other applications are mentioned in the next section.

6.3 Problem Formulation

6.3.1 Applications of Compaction

Layout compaction can be applied in four situations, of which the first two have already received some attention.

- Converting symbolic layout to geometric layout.
- Removing redundant area from geometric layout.
- Adapting geometric layout to a new technology. A new technology means that the design rules have changed; as long as the new and old technologies are compatible (e.g. both are CMOS technologies with the same mask layers), this adaptation can be done automatically, e.g. by means of so-called *mask-to-symbolic extraction*. In such a case geometric layout in the old technology is converted to a symbolic layout and then the design rules of the new technology are used for the generation of the new geometric layout.
- Correcting small design rule errors. If there are methods to put layout elements closer to each other to remove redundant space, it is plausible to assume that pulling layout elements apart when they are too close to each other can be done similarly. This is true as long as the layout with design-rule errors is topologically correct: the relative ordering of the rectangle edges in interacting layers should be the same as in the correct design.

6.3.2 Informal Problem Formulation

As mentioned before, a layout is considered to consist of rectangles. However, not all rectangles are the same. Basically, the rectangles can be classified into two groups: *rigid* rectangles and *stretchable* rectangles. Rigid rectangles correspond to transistors and contact cuts whose length and width are fixed (as a consequence of the technology or the wishes of the designer). When they are moved during a compaction process, their lengths and widths do not change. Stretchable rectangles correspond to *wires*. In principle the width of a wire cannot be modified (a designer may have reasons to make a wire wider than strictly necessary in order to decrease its resistance). The length of a wire, however, can be changed by compaction (the wire connecting two contact cuts, for example, should become shorter if the contact cuts are moved closer to each other).

Layout is essentially two-dimensional and layout elements can in principle be moved both horizontally and vertically for the purpose of compaction. When *one-dimensional compaction* tools are used, the layout elements are only moved along

one direction (either vertically or horizontally). This means that the tool has to be applied at least twice: once for horizontal and once for vertical compaction. *Two-dimensional compaction* tools move layout elements in both directions simultaneously. Theoretically, only two-dimensional compaction can achieve an optimal result. However, this type of compaction is NP-complete. On the other hand, one-dimensional compaction can be solved optimally in polynomial time as is shown later on in this chapter. Actually, repeated one-dimensional compaction can be considered a straightforward but valuable heuristic for two-dimensional compaction.

This idea is illustrated in Figure 6.4. The layout in the figure consists of nine squares labeled *A* to *I*, each with a size of $2\lambda \times 2\lambda$. These squares are rigid and are interconnected by wires of the same material. The design rules are such that the minimum separation between rectangles of the same material is 1λ . Figure 6.4(a) shows the starting situation with redundant spacing that covers an area of $11\lambda \times 11\lambda$. If the layout is first compacted horizontally from right to left and then vertically from top to bottom, the $8\lambda \times 11\lambda$ solution of Figure 6.4(c) is obtained. If the layout is first compacted vertically from top to bottom and then horizontally from right to left, the $11\lambda \times 8\lambda$ solution of Figure 6.4(e) is obtained. The optimal solution that has an area of $8\lambda \times 8\lambda$ is shown in Figure 6.4(f). In order to obtain this solution from the starting position in one pass, the layout elements must be moved in two dimensions. Note that the optimal result could be obtained if the solution of Figure 6.4(c) is compacted from left to right and then once more from top to bottom. More complex examples can be constructed, however, in which iterated one-dimensional compaction will never obtain the optimal result of two-dimensional compaction.

As mentioned above, two-dimensional compaction is NP-complete and exact as heuristic algorithms to solve the problem can be quite complex. Most practical compaction tools are based on repeated one-dimensional compaction and therefore algorithms for this case will be presented in the rest of this chapter. For this purpose, the compaction problem is first formulated in graph-theoretical terms in the next two sections.

6.3.3 Graph-theoretical Formulation

In one-dimensional, say horizontal, compaction a rigid rectangle can be represented by one x -coordinate (of its center, for example) and a stretchable one by two (one for each of the endpoints). Figure 6.5 shows some rectangles that are horizontally stretchable.

For the purpose of the algorithms to be explained, it is assumed that there are n distinct x -coordinates. They will be indicated as x_1, x_2, \dots, x_n . A minimum-distance design rule between two rectangle edges can now be expressed as an inequality:

$$x_j - x_i \geq d_{ij} \quad (6.1)$$

In the example of Figure 6.5, assuming that the minimum width for the layer concerned is a and the minimum separation is b , the following (and many other)

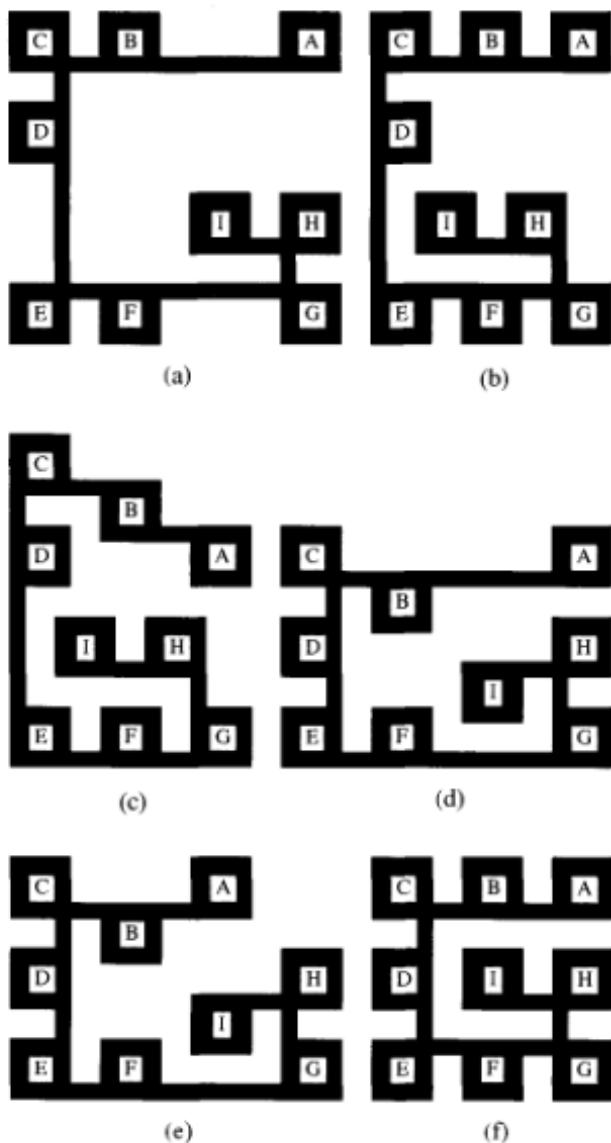


Figure 6.4 A layout with redundant space (a); the compaction result after horizontal compaction (b) followed by vertical compaction (c); the result after vertical (d) followed by horizontal compaction (e); the optimally compact result (f).

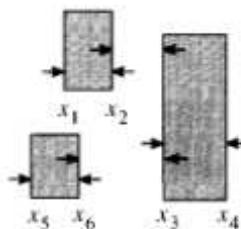


Figure 6.5 The pairs of variables associated with horizontally stretchable rectangles.

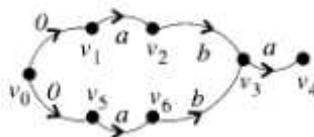


Figure 6.6 The constraint graph for the example of Figure 6.5.

inequalities are valid:

$$\begin{aligned}x_2 - x_1 &\geq a \\x_3 - x_2 &\geq b \\x_3 - x_6 &\geq b\end{aligned}$$

It is now possible to represent all these inequalities in a so-called *constraint graph* $G(V, E)$, constructed in the following way:

- The vertex set V is composed by associating a vertex v_i with each variable x_i that occurs in an inequality.
- The edge set E is composed of edges (v_i, v_j) with weight $w((v_i, v_j)) = d_{ij}$ for each inequality $x_j - x_i \geq d_{ij}$.
- There is a source vertex v_0 , located at $x = 0$. So, there are $n + 1$ vertices in total: v_0, v_1, \dots, v_n . All layout elements are assumed to have a positive x -coordinate. This is incorporated in the graph by edges from the source vertex to those vertices that do not have any other vertices constraining them at the left.

The constraint graph for the example of Figure 6.5 has been given in Figure 6.6.

A constraint graph derived from only minimum-distance constraints has no cycles (why?). It is called a *directed acyclic graph*, often denoted by the abbreviation *DAG*. The following observation can be made: the length of the longest path from the source vertex v_0 to a specific vertex v_i in the constraint graph $G(V, E)$ gives the minimal x -coordinate x_i associated to that vertex. By taking the longest path to v_i , one makes sure that all inequalities in which x_i participates are satisfied. So, computing the lengths of the longest paths to *all* vertices in the constraint graph

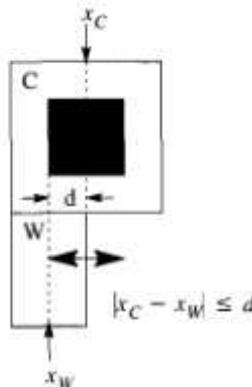


Figure 6.7 Maintaining a wire connected to a contact cut gives rise to a maximum distance constraint.

results in a solution for the one-dimensional compaction problem. An algorithm to compute these longest paths efficiently is given in Section 6.4.1. Prior to the presentation of that algorithm, a more general case of the compaction problem is formulated in the next section.

6.3.4 Maximum-distance Constraints

Some situations necessitate the limitation of the maximum distance between the coordinates of layout elements. An example of such a situation is a connectivity constraint: moving an element too far away with respect to another can break an electrical connection. An illustration for the case of a wire segment connected to a contact cut is given in Figure 6.7. The situation in the figure gives rise to two constraints: $x_C - x_W \leq d$ and $x_W - x_C \leq d$.

Maximum-distance constraints can in general be written as:

$$x_j - x_i \leq c_{ij}$$

where $c_{ij} \geq 0$. This can also be written as:

$$x_i - x_j \geq -c_{ij}$$

The last inequality has the same form as Inequality (6.1) and can be represented in the constraint graph by an edge (v_j, v_i) with weight $d_{ij} = -c_{ij}$. The addition of this type of edges can create cycles in the constraint graph. In the presence of cycles, the solution of the compaction problem still amounts to computing the lengths of the longest paths. However, the problem is more difficult than finding the longest path in a DAG as will become clear below.

```

longest-path( $G$ )
|
  for ( $i \leftarrow 1; i \leq n; i \leftarrow i + 1$ )
     $p_i \leftarrow \text{"in-degree of } v_i\text{"};$ 
     $Q \leftarrow \{v_0\};$ 
    while ( $Q \neq \emptyset$ ) {
       $v_i \leftarrow \text{"any element from } Q\text{"};$ 
       $Q \leftarrow Q \setminus \{v_i\};$ 
      for each  $v_j$  "such that"  $(v_i, v_j) \in E$  {
         $x_j \leftarrow \max(x_j, x_i + d_{ij});$ 
         $p_j \leftarrow p_j - 1;$ 
        if ( $p_j \leq 0$ )
           $Q \leftarrow Q \cup \{v_j\};$ 
      }
    }
  }

main ()
{
  for ( $i \leftarrow 0; i \leq n; i \leftarrow i + 1$ )
     $x_i \leftarrow 0;$ 
  longest-path( $G$ );
}

```

Figure 6.8 A longest-path algorithm for DAGs.

6.4 Algorithms for Constraint-graph Compaction

In this section, algorithms for constraint-graph compaction are presented. This type of compaction amounts to finding the longest path in a directed graph as was discussed above. First an algorithm for the simpler case of the longest path in a DAG is given. Then two different algorithms for graphs with cycles are explained.

6.4.1 A Longest-path Algorithm for DAGs

The longest-path problem for DAGs can be solved efficiently by an algorithm that is quite similar to breadth-first search (see Section 3.4.2). A description in pseudo-code of an algorithm that computes the longest paths from the source vertex v_0 of a graph $G(V, E)$ to any other vertex is shown in Figure 6.8. A variable p_i is associated with each vertex v_i to keep count of the edges incident to v_i that have already been processed. Because the graph is acyclic, once all incoming edges have been processed, the longest path to v_i is known. Then v_i is included in a set Q . It will be taken out later on to traverse the edges incident from it in order to propagate the longest-path values to the vertices at their endpoints. The data structure used to implement Q is left open here. Any data structure that is able to implement the semantics of a "set" can be used. The two actions to be supported are: to add

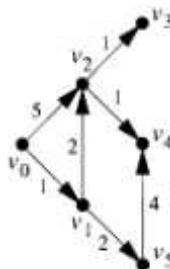


Figure 6.9 A directed acyclic graph.

Q	p_1	p_2	p_3	p_4	p_5	x_1	x_2	x_3	x_4	x_5
"not initialized"	1	2	1	2	1	0	0	0	0	0
$\{v_0\}$	0	1	1	2	1	1	5	0	0	0
$\{v_1\}$	0	0	1	2	0	1	5	0	0	3
$\{v_2, v_5\}$	0	0	0	1	0	1	5	6	6	3
$\{v_3, v_5\}$	0	0	0	1	0	1	5	6	6	3
$\{v_5\}$	0	0	0	0	0	1	5	6	7	3
$\{v_4\}$	0	0	0	0	0	1	5	6	7	3

Figure 6.10 The evolution of the variables in the longest-path algorithm for DAGs when applied to the graph of Figure 6.9.

a new element and to remove an arbitrary element. Note: the initialization of the variables x_i has been put outside the procedure body as this makes it possible to use `longest-path` as part of another algorithm that will be presented later.

The longest-path algorithm presented has a time complexity $\mathcal{O}(|E|)$. This is easy to see: all edges in the graph are visited exactly once during the execution of the inner for-each loop.

Consider the DAG given in Figure 6.9. The evolution of the variable values throughout the different iterations of the while loop is given in Figure 6.10. The value given for the set Q is the value at the beginning of the iteration. The first element is removed. The remaining elements in the same row are the updated values after processing this first element.

6.4.2 The Longest Path in Graphs with Cycles

Before presenting algorithms that compute the longest path in graphs with cycles, some general remarks on this case will be made in this section. Two cases can be distinguished:

1. The graph only contains *negative* cycles, i.e. the sum of the edge weights along any cycle is negative.

```

count ← 0;
for ( $i \leftarrow 1; i \leq n; i \leftarrow i + 1$ )
     $x_i \leftarrow -\infty;$ 
 $x_0 \leftarrow 0;$ 

do [ flag ← 0;
    longest-path( $G_f$ );
    for each  $(v_i, v_j) \in E_b$ 
        if ( $x_j < x_i + d_{ij}$ ) {
             $x_j \leftarrow x_i + d_{ij};$ 
            flag ← 1;
        }
    count ← count + 1;
    if (count >  $|E_b|$  && flag)
        error("positive cycle")
}
while (flag);

```

Figure 6.11 The Liao-Wong compaction algorithm.

2. The graph contains *positive cycles*.

The problem for graphs with positive cycles is NP-hard. However, a constraint graph with positive cycles corresponds to a layout with conflicting constraints (in which e.g. the minimum distance for two coordinates exceeds the maximum one). Such a layout is called *overconstrained* and is impossible to realize. So, the best to be done in such a case is to detect the existence of positive cycles. They can be detected in polynomial time. In fact, the two algorithms to be described for the longest-path problem for graphs with negative cycles can detect the existence of positive cycles (but not localize them). The two algorithms presented next, the Liao-Wong algorithm and the Bellman-Ford algorithm, are not the only algorithms known for the computation of longest paths in directed graphs. They have been selected because they have successfully been applied to layout compaction.

6.4.3 The Liao-Wong Algorithm

Liao and Wong have proposed an algorithm that partitions the edge set E of the constraint graph $G(V, E)$ into two sets E_f and E_b . The edges in E_f have been obtained from the minimum-distance inequalities and are called *forward* edges. The edges in E_b correspond to maximum-distance inequalities and are called *backward* edges (they create cycles by going backward).

The pseudo-code for the algorithm is given in Figure 6.11. The main idea is to start with $G_f(V, E_f)$ which is acyclic and to which the DAG longest-path algorithm of Figure 6.8 can be applied. Then the backward edges are considered and modifications to the minimal x-coordinates are made, followed by a call to the DAG longest-path algorithm to propagate the effects of the modifications through the forward

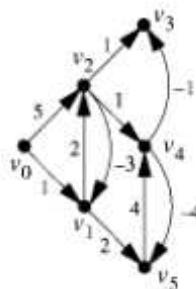


Figure 6.12 A directed graph with “backward” edges.

Step	x_1	x_2	x_3	x_4	x_5
Initialize	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
Forward 1	1	5	6	7	3
Backward 1	2	5	6	7	3
Forward 2	2	5	6	8	4
Backward 2	2	5	7	8	4
Forward 3	2	5	7	8	4
Backward 3	2	5	7	8	4

Figure 6.13 The evolution of the distances for the graph of Figure 6.12 as computed by the Liao-Wong algorithm.

edges. This process (modifications due to backward edges followed by propagation) is repeated until the values of the minimal x -coordinates stabilize or a maximum number of iterations equal to $|E_b|$ have been performed. In the last case, a positive cycle exists. The execution of the algorithm is then terminated by calling the function `error` that prints the error message provided as an argument and then performs a “long jump” to an appropriate environment in the software.

In order to understand that the algorithm is correct, one should realize that at the k th iteration of the do loop, the values of the x_i represent the longest-paths going through all forward edges and possibly k backward edges. As a longest-path contains each edge at most once, the algorithm should terminate after at most $|E_b|$ iterations in the absence of positive cycles.

As the DAG longest-path algorithm has a time complexity of $\mathcal{O}(|E_f|)$ and is called at most E_b times, the Liao-Wong algorithm has a time complexity of $\mathcal{O}(|E_b| \times |E_f|)$. This makes the algorithm interesting in cases when the number of backward edges is relatively small.

An example graph that will be used for the illustration of the algorithm is given in Figure 6.12. The graph has been obtained from the graph of Figure 6.9 by adding to it three backward edges (recognizable by their negative weights). The evolution of the Liao-Wong algorithm applied to this graph is shown in Figure 6.13. The figure shows

```

for ( $i \leftarrow 1; i \leq n; i \leftarrow i + 1$ )
     $x_i \leftarrow -\infty;$ 
     $S_1 \leftarrow [v_0];$ 
     $S_2 \leftarrow \emptyset;$ 
    while (count  $\leq n \&& S_1 \neq \emptyset$ ) {
        for each  $v_i \in S_1$ 
            for each  $v_j$  "such that"  $(v_i, v_j) \in E$ 
                if ( $x_j < x_i + d_{ij}$ ) {
                     $x_j \leftarrow x_i + d_{ij};$ 
                     $S_2 \leftarrow S_2 \cup \{v_j\}$ 
                }
         $S_1 \leftarrow S_2;$ 
         $S_2 \leftarrow \emptyset;$ 
        count  $\leftarrow$  count + 1;
    }
    if (count  $> n)$ 
        error("positive cycle");
}

```

Figure 6.14 The Bellman-Ford algorithm.

the values of the found distances after the “forward” and “backward” steps in each iteration (respectively, the results obtained by the DAG longest-path algorithm and the update steps). No more changes are found in the third iteration and the algorithm stops. Note that the result obtained after the first “forward” step is equal to the final result of Figure 6.9.

6.4.4 The Bellman-Ford Algorithm

An alternative to the Liao-Wong algorithm is the *Bellman-Ford* algorithm. The algorithm does not discriminate between forward and backward edges. It is comparable to the longest-path algorithm for DAGs with the difference that several iterations through the graph are necessary before the lengths of the longest paths have been computed.

The pseudo-code for the algorithm is given in Figure 6.14. One way of looking at the algorithm is to see it as repeated wave front propagation: S_1 contains the current wave front and S_2 is the one for the next iteration. As in the Liao-Wong algorithm, if there are more than n iterations, where n is the number of vertices in the graph $G(V, E)$, it can be concluded that the graph has positive cycles. Informally, this can be seen as follows: after k iterations, the algorithm has computed the longest-path values for paths going through $k - 1$ intermediate vertices. If there are no cycles, the algorithm should terminate after at most n iterations, as the longest path to a vertex can go through at most $n - 1$ vertices.

The time complexity of the Bellman-Ford algorithm is $\mathcal{O}(n \times |E|)$ as each iteration

S_1	x_1	x_2	x_3	x_4	x_5
"not initialized"	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$\{v_0\}$	1	5	$-\infty$	$-\infty$	$-\infty$
$\{v_1, v_2\}$	2	5	6	6	3
$\{v_1, v_3, v_4, v_5\}$	2	5	6	7	4
$\{v_4, v_5\}$	2	5	6	8	4
$\{v_4\}$	2	5	7	8	4
$\{v_3\}$	2	5	7	8	4

Figure 6.15 The evolution of the distances for the graph of Figure 6.12 as computed by the Bellman-Ford algorithm.

visits all edges at most once and there are at most n iterations. If the graph is dense, i.e. the number of edges is $\mathcal{O}(n^2)$, this would mean a worst-case time complexity of $\mathcal{O}(n^3)$. However, under assumptions that are realistic for compaction, the *average* time complexity turns out to be $\mathcal{O}(n^{1.5})$.

The evolution of the Bellman-Ford algorithm when applied to the graph of Figure 6.12, that was also used for the illustration of the Liao-Wong algorithm, is shown in Figure 6.15. In each row, the first column gives the contents of the set S_1 at the beginning of the algorithm and the remaining entries of the row show the distance value after having processed all elements of S_1 . Not surprisingly, the final result is the same as the one obtained by the Liao-Wong algorithm.

6.4.5 Discussion: Shortest Paths, Longest Paths and Time Complexity

Until now different versions of shortest and longest-path problems have been presented in this text. In order to eliminate possible confusion, a short discussion follows here that shows the relationships between the different algorithms presented as well as the type of graphs on which they operate.

Consider first a DAG. The longest-path problem for a DAG can be solved with the algorithm given in Figure 6.8. If one wants to solve the shortest-path problem in a DAG, a slightly modified version of the algorithm can be applied by initializing all path lengths to $+\infty$ and taking the minimum instead of the maximum in the inner loop. One can also apply the longest-path algorithm without modification if it operates on a "transformed" graph where all weights have been multiplied with -1 . The multiplication by -1 of the longest path lengths found gives the shortest path lengths.

The situation is more complicated when the graph is cyclic. If it only contains negative cycles, the longest-path problem can be solved in polynomial time. Analogously, if the graph only contains positive cycles, the shortest-path problem can be solved in polynomial time (actually, the original version of the Bellman-Ford algorithm is for shortest paths).

The longest-path problem in graphs with positive cycles and analogously the shortest-path problem in graphs with negative cycles is NP-hard (see Exercise 4.1).

Note that Dijkstra's algorithm for shortest paths (see Section 3.4.3) only works when all edge weights are positive. Therefore, it does not operate correctly if the edge weights are first multiplied by -1 in an attempt to adapt the algorithm for a longest-path computation.

6.5 Other Issues

There follows a short discussion of a number of additional issues that are important for compaction but that cannot be dealt with in detail in this text:

- *Noncritical layout elements*

Consider horizontal, right-to-left, one-dimensional compaction using a constraint graph as in the text above. The width of the layout after compaction is determined by the vertex v that has the largest longest-path length. The longest path starting at the source vertex v_0 and finishing at v is called the *critical path* of the layout (there may be more than a single critical path). In the case of the graph of Figure 6.9 for example, v_4 receives the largest coordinate after compaction and the critical path consists of the vertices v_0, v_2, v_1, v_5 and v_4 . The layout elements corresponding to these vertices cannot be moved further to the right without making the total width of the layout larger than strictly necessary. This is, however, not the case with the layout element corresponding to the remaining vertex v_3 : it can be moved one length unit to the right (to position 8) without affecting the minimal width of the layout. In general, all layout elements not on the critical path have some *mobility* or *freedom*. The interval within which a layout element can be moved can easily be found by performing both a right-to-left and a left-to-right compaction and taking the longest path lengths found as the interval boundaries. Simply assigning each layout element its leftmost or rightmost possible value does not exploit the mobility of the noncritical elements. It may be that a subsequent compaction step in a perpendicular dimension will achieve better results if the layout elements are assigned a coordinate that is not on the boundary of this interval. Finding the best position for these noncritical layout elements is a difficult task (an optimal algorithm would actually solve the two-dimensional compaction problem that is known to be NP-complete) but some heuristics could always be applied to exploit mobility.

- *Automatic Jog Insertion*

In Section 6.3.2 layout elements were partitioned in rigid and stretchable rectangles. Considering a wire as a rectangle that is merely stretchable in one dimension, does not exploit all the possible freedom that one has with wires. One such possibility is the insertion of *jogs*. This is the splitting of a wire in segments such that these segments can be moved with respect to each other. This situation can lead to a reduction of the layout width as illustrated in Figure 6.16. Compaction algorithms capable of inserting jogs are quite common. They especially introduce jogs to reduce the critical-path length.

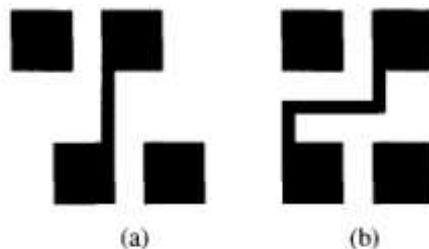


Figure 6.16 A simple compacted layout before (a) and after (b) the introduction of jogs.

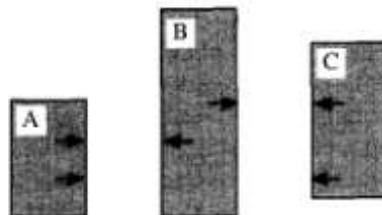


Figure 6.17 The redundancy of the minimal-distance constraint between rectangles A and C.

- *Constraint Generation*

The algorithms discussed above operate on a constraint graph. However, compaction has to be applied on a layout. Therefore, an efficient algorithm is necessary to convert a layout into a constraint graph. In a straightforward algorithm, assuming horizontal compaction, one could inspect all pairs of layout elements and generate a weighted edge between them, if they overlap when projected on the vertical axis. When the original layout consists of n rectangles, this will result in an algorithm with a time complexity of $\mathcal{O}(n^2)$. There are two problems with this approach. First, too many rectangle pairs are inspected in the generation process and second, the resulting graph is likely to have many redundant edges (remember that the time complexity of compaction algorithms depends on the number of edges in the constraint graph). An edge (v_i, v_j) in a graph $G(V, E)$ is called *redundant* if there also exist edges (v_i, v_k) and (v_k, v_j) , while $w((v_i, v_j)) \leq w((v_i, v_k)) + w((v_k, v_j))$. This has been illustrated in Figure 6.17, that shows three rectangles A, B, and C located in the same layer. The minimum-distance constraints for the rectangle pairs (A,B) and (B,C) make the minimum-distance constraint for pair (A,C) redundant.

- *Hierarchy*

It is important to note that compaction will seldom be applied to the layout of a complete integrated circuit. As was mentioned in Chapter 1, the physical domain can be described hierarchically, such that small groups of transistors form a cell, groups of cells form modules, etc. If special attention is paid to cell boundaries, it

is normally sufficient to apply compaction to cells only.

6.6 Bibliographic Notes

The reader that is interested in knowing more about compaction or in an alternative presentation can consult textbooks dedicated to physical VLSI design automation such as [Len90], [She93], [Sai95] or [Sar96]. Reviews on the compaction problem include [Cho85], [Mly86], [Boy88], [Wol88], [Hil89] presents the results of a layout generation project that included symbolic layout, a formal *layout language* to describe it and compaction methods.

More on design rules can be found in textbooks on CMOS VLSI design such as [Wes93] but also in [Uye95], which is a more practice-oriented book on CMOS circuit and layout design. One of the early sources to introduce the symbol λ for the minimal feature size was [Mea80].

An example of a symbolic layout editor is described in [Cro88]. More on *mask-to-symbolic extraction* can e.g. be found in [Beu90].

A proof that two-dimensional compaction is NP-complete is given in [Sch83b]. The fact that it is an NP-complete problem justifies the use of simulated annealing as reported in [Mos87]. An interesting compaction method that allows some lateral movement during one-dimensional compaction is reported in [Séq87].

A one-dimensional compaction method that has some similarities with constraint-graph compaction but has not been discussed here is *virtual-grid compaction*. It was introduced by Weste [Wes81] and has the property that layout elements placed by a designer on the same horizontal or vertical "virtual grid" line remain on the same line during compaction.

The Liao-Wong algorithm is explained in [Lia83]. The application of the Bellman-Ford algorithm, which is well-known in graph theory (see e.g. [Law76], [Cor90] or [McH90]), to the compaction problem has been pointed out by Schiele [Sch83a]. The pseudo-code in the text is based on Schiele's version of the algorithm. The average time complexity result of $\mathcal{O}(n^{1.5})$ has also been reported by Schiele.

More on constraint generation can be found in the textbooks and review papers mentioned above. The best algorithm for the constraint-graph generation problem has been proposed by Doenhardt and Lengauer [Doe87, Len90]. It has a time complexity of $\mathcal{O}(n \log n)$, while it only generates irredundant constraints.

6.7 Exercises

- 6.1 Write a recursive procedure for computing the longest-path lengths based on the algorithm of Figure 6.8 that does not store vertices to be processed explicitly in a data structure: so, the new procedure should not contain the variable Q .

- 6.2** The three algorithms for longest-path computations presented in this chapter (in Figures 6.8, 6.11 and 6.14) all compute longest-path lengths rather than longest paths. How should these algorithms be modified to compute the longest paths themselves? Do the modifications that you propose affect the time complexity of the algorithms?

Placement and Partitioning

Placement is a very common problem in VLSI design. The input to the problem is the structural description of a circuit. Such a description consists of a list of design subentities (hardware subparts) and their interconnection patterns that together specify the complete circuit. At the moment that the placement problem has to be solved, the layouts of these subentities should be available. The goal of placement is to determine the location of these layouts on the chip such that the total resulting chip area is minimal, while the layouts of the subentities do not overlap and sufficient space is left for wiring. The wiring should realize exactly the interconnections specified in the structural description. The determination of the wiring patterns on chip forms the *routing* problem that is discussed in Chapter 9. During placement, it is sufficient to estimate the area occupied by wiring.

A "toy" version of the *placement* problem, viz. the *unit-size placement* problem, has already been introduced in Chapter 5. There, some indications were given on how to solve this problem using general-purpose optimization methods. In this chapter, two versions of the placement problem that are more realistic than unit-size placement, viz. *standard-cell placement* and *building-block placement*, are first defined. Most of the general-purpose optimization methods can also be used for these versions of the placement problem. The emphasis in this chapter is on problem-specific solution methods for placement. They can be subdivided in the categories *constructive* and *iterative placement* and are discussed separately.

Before dealing with the placement problem and possible solutions, however, attention is paid to the representation of an electric circuit such that a placement program (and many other design automation tools) can access the circuit easily. Some possibilities are then explained for the estimation of the wiring area.

The *partitioning* problem deals with splitting a network into two or more parts by cutting connections. Although this problem has major importance as a problem on its own in the field of VLSI design automation, it is treated here together with placement because solution methods for the partitioning problem can be used as a subroutine for some type of placement algorithms. Only one partitioning algorithm is presented here in detail, viz. the one proposed by Kernighan and Lin.

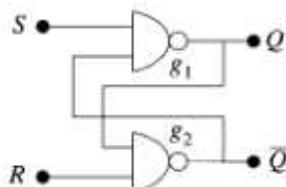


Figure 7.1 The schematics of an RS-latch.

7.1 Circuit Representation

The structural description of an electric circuit is a central issue in design automation. It is the input of tools like placement, simulation, etc., while it is the output of e.g. tools for logic and high-level synthesis. Consider the schematics shown in Figure 7.1 that indicate how an *RS-latch* can be constructed from two NAND gates. In the schematics, one can distinguish the two NAND gates g_1 and g_2 , two *input terminals* S and R , two *output terminals* Q and \bar{Q} and the wires connecting the gates and the terminals. Besides, a complete description of the circuit should indicate to which specific input or output a wire is connected. Obviously, a *data model* of an electric circuit (the organization of the data structures that represent it) should correctly deal with all issues mentioned for the example.

The data model proposed here consists of the three structures *cell*, *port* and *net*. A *cell* is the basic building block of a circuit. A NAND gate is an example of a cell. The point at which a connection between a wire and a cell is established is called a *port*. So, a cell has one or more ports. The wire that electrically connects two or more ports is a *net*. So, a set of ports is associated with each net and a port can only be part of a single net. These notions can be expressed as data structure definitions in pseudo-code, as shown in Figure 7.2.

A cell in a circuit is an *instance* of a *master* cell. The master contains all information that all cells of a specific type, e.g. all NAND gates, have in common. The term instance refers to each occurrence of the cell in the circuit. One property stored in the master is, of course, the name of the cell: "NAND". Another property is a list of its inputs and outputs. Still other properties could be related to electrical properties, such as the switching delay or layout properties, such as width and height. Note that the information stored in masters originates from a *library*, either a library of predesigned cells such as standard cells, or a library of cells designed by the designer.

For each occurrence of a cell in the circuit there is a separate instance. It might be useful to uniquely identify each cell instance with a name (e.g. for easy communication in a design team). For this reason each *cell* structure has an attribute *i.d.*. Finally, each cell instance needs to have access to its ports, identified here by the sets *in_ports* and *out_ports*. The ports have been partitioned into inputs and outputs. This may not always be relevant, but it surely is the case in e.g. the context of simulation where the cells have to be processed in the order of signal flow.

```

struct cell {
    struct cell_master *cell.type; /* Access to cell type, e.g. NAND
                                   gate and other generic properties */
    char id[ ]; /* A string that uniquely identifies the cell, e.g. g1 */
    set of struct port in_ports, out_ports;
};

struct port {
    struct port_master *port.type; /* Access to generic port information */
    char id[ ]; /* Unique identification */
    struct cell *parent.cell; /* To which cell does this port belong? */
    struct net *connected.net; /* To which net is this port connected? */
};

struct net {
    char id[ ]; /* Unique identification */
    set of struct port joined_ports; /* Ports connected by the net */
};

```

Figure 7.2 Data structure definitions for circuit representation.

What has been said about instances and masters for cells also applies to ports. The information to be stored in a port master could be the name of the port (e.g. "carry-in"), whether the port is an input or an output, etc. The attributes `parent.cell` and `connected.net` in the `port` structure identify respectively the cell instance to which the port instance belongs and the net to which the instance is connected.

The structure `net`, finally, is quite simple. For each net, it should give access to the ports to which the net is connected.

Any electric circuit communicates with the external world in one way or the other through its terminals (see *S*, *R*, etc. in Figure 7.1). These terminals cannot be directly incorporated in the data model just presented. For a consistent modeling, pseudo-cells called *input cells* and *output cells* are introduced. An input cell has a single port through which it sends a signal to the circuit and an output cell has a single port through which it receives a signal from the circuit. In Figure 7.3(a) the RS-latch of Figure 7.1 is shown once more, now explicitly showing the cells, pseudo-cells and ports. Input cells are shown using an "arrow" symbol pointing to its port and output cells using the same symbol, but now pointing away from its port. Dashed lines show the cell boundaries. Ports are indicated by small squares located on the cell boundaries.

It is quite straightforward to derive a *graph model* of an electric circuit from the data model just presented. The graph will have three distinct sets of vertices: a cell set, a port set and a net set. There will be two edge sets: one for edges connecting cells with ports and one for edges connecting nets with ports. Note that edges never connect vertices of the same type and neither do they connect nets with cells. One could call such a graph a *tripartite graph*. This graph model of the RS-latch circuit

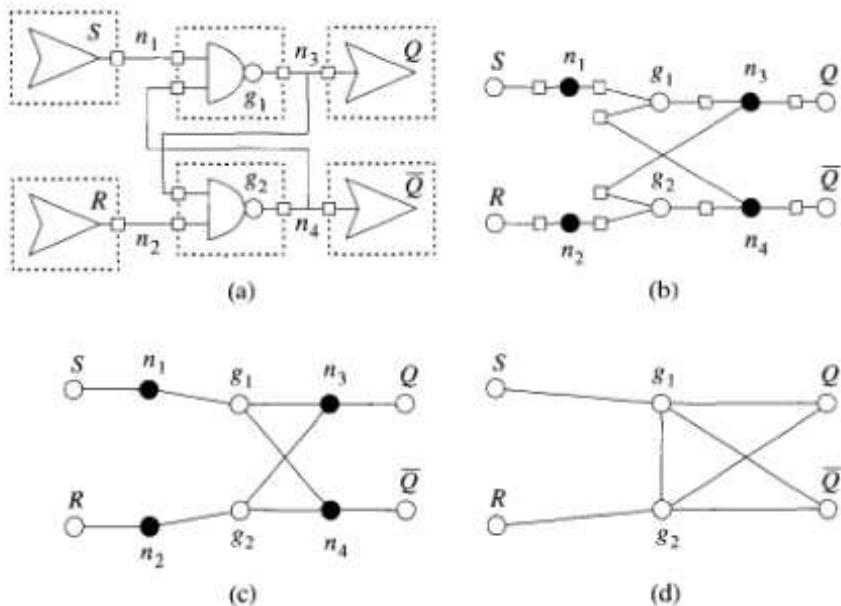


Figure 7.3 The representation of the RS-latch using the cell-port-net data model (a), the tripartite the graph model (b), the bipartite graph model (c) and the clique model (d).

is shown in Figure 7.3 (b). In the figure, white circles represent cells, black circles represent nets and squares represent ports.

In some applications, such as the placement problem to be discussed in this chapter, it is not always important to distinguish the ports of a cell. In such a case all ports can be merged with their associated cell vertices. The result is a *bipartite graph* consisting of cell and net vertices and a single edge set that links nets with cells. The representation of the RS-latch example using this graph model is shown in Figure 7.3(c). This model is equivalent to the *hypergraph* model of a circuit. A hypergraph consists of vertices and *hyperedges*, where hyperedges connect two or more vertices instead of exactly two as is the case in a common graph. Clearly, in a hypergraph model, the vertices represent the cells and the hyperedges the nets.

The model can be simplified even further by omitting the explicit representation of nets. Only the set of cell vertices remains in the graph. For each set of cells that are connected to the same net, edges are created between each pair of cells in this set. So, there is a *clique* in the graph (see Section 3.1) for each net in the circuit. The representation of the RS-latch using this clique model is shown in Figure 7.3(d). Clearly, the graph models discussed here differ in the accuracy with which they model an electric circuit. It is up to the designer of CAD software to chose the most suitable one for the problem at hand.

As was discussed in Chapter 1, the notions of *hierarchy* and *abstraction* are very important in VLSI design. The complexity of VLSI circuits can only be mastered

through the use of hierarchy and abstraction. The data model presented here does not directly support hierarchy, but can easily be modified to do so. A complete circuit can e.g. be converted into a single cell by assigning the circuit's contents to a new *master*. In this process, the input and output pseudo-cells become the ports of the new cell (see Exercise 7.1). Applying these ideas to the circuit of the RS-latch would e.g. result in a new master cell "RS-latch" that can be used as a component in a larger circuit.

The data model presented in this section is a general one that can be used by several applications. Each application will require additional information to be stored in the model. A simulator e.g. will extend the `net` data structure to store the value of the signal carried by the net. Layout information will be added to a `cell` data structure by a placement program, etc. The use of *object-oriented* programming techniques could be of help here as extending the basic data structures and the functions operating on them can be done without modifying the basic code itself.

7.2 Wire-length Estimation

As mentioned at the end of the previous section, the estimation of the total wire length is used to evaluate the quality of placement. A wire-length *metric* is applied to each net, resulting in a length estimate per net. The total wire length estimation is then obtained by summing the individual estimates. The total wiring area (the cost function for the unit-size placement problem) can then be derived from this length by assuming a certain wire width and a wire separation distance.

All metrics refer to a cell's coordinates. In the unit-size placement problem, these are the coordinates of the unit-size grid as shown e.g. in Figure 5.1(b). In more general placement problems, one will need a finer grid to accurately indicate the location of a cell. A cell's coordinates can then be the coordinates of its center. Some common metrics are:

- *Half perimeter*: This metric computes the smallest rectangle that encloses all terminals of a net and takes the sum of the width and height of the rectangle as an estimation of the wire length. The estimation is exact for two- and three-terminal nets and gives a lower bound for the wire length of nets with four or more terminals.
- *Minimum rectilinear spanning/Steiner tree*: The two types of tree were introduced in Section 4.5. In both cases, a wiring pattern is computed that interconnects all terminals of the net. Remember that the spanning tree is relatively easy to compute, whereas the minimum Steiner tree problem is NP-complete. The minimum Steiner tree always has a length shorter than or equal to the spanning tree length which means that the latter is an upper bound for the former. For both types of tree, when computing the length between two points (x_1, y_1) and (x_2, y_2) , one can either use the *rectilinear* or *Manhattan* distance defined as $|x_1 - x_2| + |y_1 - y_2|$ or the *Euclidean* distance defined as $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

- *Squared Euclidean distance:* This method is meant for the clique-model representation of an electric circuit (see Section 7.1). As nets are not explicitly present in this model, the total cost is obtained by summing over the cells rather than over the nets. The cost of a placement is then defined as:

$$\frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \gamma_{ij} [(x_i - x_j)^2 + (y_i - y_j)^2]$$

Here, n denotes the number of vertices (cells), γ_{ij} is the edge weight of the edge between vertex v_i and v_j , and x_i , x_j , y_i , and y_j are the x-coordinates and y-coordinates of the corresponding cells. The factor $\frac{1}{2}$ just expresses that all vertex pairs are counted twice, but could obviously be omitted without affecting the optimization result obtained using this metric. Also the fact that the squared distance instead of the distance itself is used here, does not affect the results of optimization. γ_{ij} is zero if there is no edge between the vertices v_i and v_j . For nets that are connected to many cells, its value can be made lower to express that not all pairwise connections in the clique representing the net will be realized during routing. Also, the fact that the two cells v_i and v_j are both connected to more than one net can be translated in a higher value of γ_{ij} .

7.3 Types of Placement Problem

The most common placement problems are *standard-cell placement* and *building-block placement*. They are introduced in this section.

As was mentioned in Section 1.4, *standard cells* are predesigned small circuits (e.g. simple logic gates, flip-flops, etc.) for a specific fabrication process that are stored in a library. For each cell, the library contains behavioral data (e.g. Boolean equations, delay parameters) that can be used by a simulator, and also layout data (e.g. size, positions of terminals, etc.) to be used by placement and routing programs. The designer combines the standard cells into the desired circuit by interconnecting them in an appropriate way.

Standard cells obey some restrictions for their layout. Connections that are shared by all or most cells, like e.g. power and clock connections, cross the cells from left to right at fixed locations. These connections are sometimes called the *logistic signals* as opposed to *logic signals*. Signals related to the specific I/O of the cell have to leave the cell either at the top or the bottom. This is illustrated in Figure 7.4. In a standard-cell design, the cells are collected into *rows* separated by wiring or routing *channels*. Within one row, the logistic signals automatically realize continuing connections due to their fixed positions. It is said that the cells *abut* horizontally or that they are connected by horizontal *abutment*. Figure 7.5 shows a small arrangement of cells obeying the standard-cell layout style.

In full-custom design where designers have the freedom to give arbitrary (but often rectangular) shapes to their cells or where *macro cells* are used that consist of regular parameterized layout structures, the cells need wiring space all around. The layout

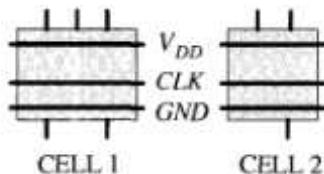


Figure 7.4 Standard cells: logistic signals like power and clock cross the cell at fixed positions while I/O signals specific to a cell have terminals at the top and bottom.

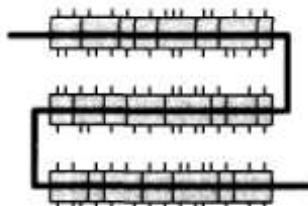


Figure 7.5 An example of standard cell placement: there are three rows separated by wiring channels, while the logistic signals connecting to all cells have symbolically been represented by a thick line.

style that supports designs with these type of cells is called *building-block layout* (or sometimes also *general-cell layout*). Figure 7.6 shows a stylized representation of a placement in building-block layout.

Apart from the standard-cell and building-block layout styles, a combination of both is also common. One e.g. uses standard cells for the implementation of the logic functions of a design together with a large RAM module for the memory. Such a module occupies far less area than an equivalent memory built of standard cells only (in a standard-cell implementation of the memory, much more space is occupied by wiring).

Clearly, the placement problem for standard cells or building blocks is more complex than the *unit-size* placement problem discussed in Section 5.1. One obvious difference is that *moves* that exchange two cells as encountered in many general-purpose algorithms (see Chapter 5) are not always possible due to the size difference

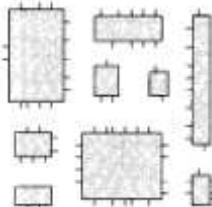


Figure 7.6 An example of the placement of cells in the building-block layout style.

of cells. This is one of the issues discussed in the next section.

7.4 Placement Algorithms

Placement algorithms can be grouped into two categories:

- *constructive placement*: the algorithm is such that once the coordinates of a cell have been fixed they are not modified anymore;
- *iterative placement*: all cells have already some coordinates and cells are moved around, their positions are interchanged, etc. in order to get a new (hopefully better) configuration.

Most placement algorithms contain both approaches: an *initial placement* is obtained in a constructive way and attempts are made to increase the quality of the placement by *iterative improvement*. Note: this searching strategy does not apply to placement only, but is applicable to many combinatorial optimization problems.

Below, separate attention is given to the constructive and iterative placement approaches.

7.4.1 Constructive Placement

There are many ways to perform constructive placement. Two important methods are based on *min-cut partitioning* and *clustering*. Both methods are essentially *partitioning* methods which divide the circuit in two or more subcircuits of a given size while minimizing the number of connections between the subcircuits. More information on partitioning will be given in Section 7.5.

The basic idea of min-cut placement is to split the circuit into two subcircuits of more or less equal size while minimizing the number of nets that are connected to both subcircuits (a net that is connected to both subcircuits is "cut", hence the name "min-cut"). The two subcircuits obtained will each be placed in separate halves of the layout (upper and lower halves or left and right halves). Because the number of nets crossing from one half to another has been minimized, one can assume that the number of long wires crossing from one half of the chip to the other has been minimized as well. This type of *bipartitioning* is recursively applied to the subcircuits until some criterion is satisfied (e.g. until all subcircuits only contain one cell). Also, after each bipartitioning stage, the area available for the subcircuit concerned is bisected and a decision is made on the direction of the bisection (horizontal or vertical) and on which of the two smaller subcircuits obtained will be assigned to which half. The idea is illustrated in Figure 7.7, that shows three stages of min-cut placement. For each stage, the netlist status is shown at the top and the placement at the bottom. Note that a clique model is used for the netlist where parallel edges indicate multiple connections between the cells.

There are two important tasks in min-cut placement: the partitioning of the graph and the assignment of the partitions to relative layout positions. An algorithm for

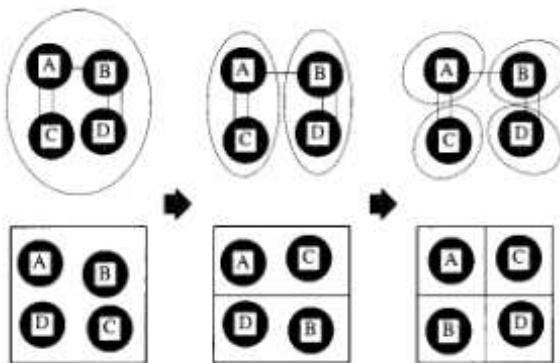


Figure 7.7 The recursive bipartitioning of a circuit leading to a min-cut placement.

partitioning will be presented in Section 7.5.1. The second task can be based on different heuristics. One such heuristic is to look at the parts of the circuit that already have a fixed position (either because the placement of these parts is already fixed or because they are connected to the inputs or outputs of the chip that are located at the chip's periphery). The decision can then be based on keeping the connections to these fixed parts as short as possible. Consider the example of Figure 7.7: after the first bipartitioning, there are no fixed parts and the cells *A* and *C* are assigned to the upper half of the layout, while the cells *B* and *D* are assigned to the lower half. In the next stage, due to the lack of fixed cells, *A* is assigned to the upper right and *C* to the upper left (the decision to assign them both to the upper half had already been taken before). Now, when taking decisions for the cells *B* and *D*, one can see that it is preferable to put *B* at the lower left rather than the lower right because of its connections with *A*.

Min-cut placement is a *top-down* method: one starts with the whole circuit and ends with small subcircuits. A *bottom-up* method for initial placement is *clustering*. Here one starts with a single cell and finds one or more cells that share many nets with it. These cells are all taken together to form a cluster. More cells are added to the cluster in this way until the cluster contains the whole circuit. As was the case in the bipartitioning-based approach, each time after a cell has been selected into a cluster, a decision should be taken on its location. Again, the connectivity of a cell with those already in the cluster can guide the decision. The idea is illustrated in Figure 7.8.

7.4.2 Iterative Improvement

Iterative improvement is a method that perturbs a given placement by changing the positions of one or more cells and evaluates the result. If the new cost is less than the old one, the new placement replaces the old one and the process continues. If the new situation is worse than the old one, the perturbed situation may or may not be

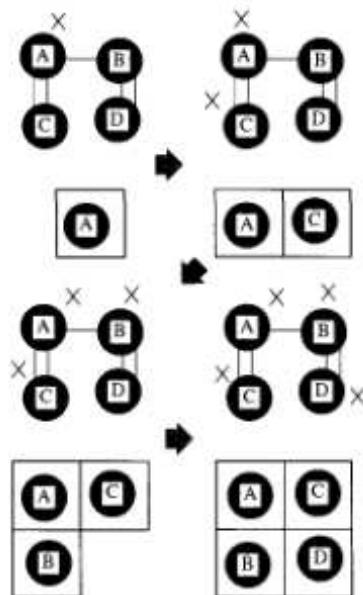


Figure 7.8 Placement by clustering (the cells that are already part of the cluster are marked by a cross).

```

iterative_improvement()
{
    s ← initial_configuration();
    c ← cost(s);
    while (!stop()) {
        s' ← perturb(s);
        c' ← cost(s');
        if (accept(c, c'))
            s ← s';
    }
}

```

Figure 7.9 A generic algorithm for iterative improvement.

accepted depending on the exact search method used. A general framework for an iterative improvement algorithm is given in Figure 7.9.

The behavior of the algorithm depends on the definitions of the functions `initial.configuration`, `cost`, `stop`, `perturb` and `accept`. `initial.configuration` generates an initial placement and `cost` computes the costs of a given placement. The function `stop` decides when the main loop in `iterative_improvement` should be terminated. This function can be very simple, counting a fixed number of iterations, or arbitrarily more complex. The Boolean function

accept (c, c') normally simply implements the test $c' < c$, which means that perturbations that decrease the cost function (downhill moves), are always accepted. An appropriate choice for the functions mentioned can make the algorithm operate as some of the general-purpose optimization methods discussed in Chapter 5, such as *local search*, *simulated annealing* or *tabu search*.

The function `perturb` computes a new feasible solution s' from a given feasible solution s . This function is easy to implement for the case of *unit-size* placement (see Section 5.1), as was indicated in several places in Chapter 5: any exchange of cells or the displacement of a cell to an empty slot will result in a feasible solution. Perturbation of a feasible solution for standard cell or building-block placement is more complex due to the inequality of the cell sizes. Different approaches are possible:

- One can allow that cells in a feasible solution overlap and make the overlap part of the cost function to be minimized. So, the cost function becomes some weighted sum of the estimated wire length and the overlap cost. This will direct a placement algorithm towards solutions with little or no overlap. Any overlap that remains can be eliminated by pulling apart the cells in the final layout (at the expense of a larger overall chip area).
- One can eliminate overlaps directly after each move by shifting an appropriate part of the cells in the layout. In general, this is a computation-intensive operation as the coordinates of many cells in the layout have to be recomputed as well as the estimated wire lengths of all nets that connect to the displaced cells. It is, however, feasible for standard cell placement, because overlaps can occur in one dimension only. This means that at most half of the cells in a row need to be shifted (when inserting a cell at a position without sufficient space, shift to the side that contains the smallest number of cells).
- One can resort to more sophisticated encoding methods for feasible solutions, instead of the mere assignment of coordinates to cells. These methods store the "relative position" of the cells with respect to each other. One could think e.g. of representations similar to the *slicing tree* to be discussed in Chapter 8. Applying transformations on these encodings to find neighboring solutions will always result in placements without overlap. However, the price paid for feasible solutions that are free of overlap is again a computational one, because the absolute coordinates of each cell and the net length estimates have to be recomputed after each move.

Apart from random perturbations that are characteristic for optimization methods such as simulated annealing, one could use more domain-specific methods for placement by means of iterative improvement. *Force-directed placement* is an example of such a method. It assumes that cells that share nets, feel an attractive "force" from each other. The goal is to reduce the total force in the network. To achieve this goal one can compute the "center of gravity" of a cell, the position where the cell feels a force zero, given the positions of all the cells to which it is connected.

Consider the “clique representation” of a netlist by means of an edge-weighted graph $G(V, E)$ as discussed in Section 7.1. The weight of an edge $(i, j) \in E$ is given by γ_{ij} ($\gamma_{ij} = 0$ when $(i, j) \notin E$). Then the center of gravity (x_i^g, y_i^g) of a cell i is defined as:

$$x_i^g = \frac{\sum_j \gamma_{ij} x_j}{\sum_j \gamma_{ij}}$$

$$y_i^g = \frac{\sum_j \gamma_{ij} y_j}{\sum_j \gamma_{ij}}$$

So, the center of gravity is found by computing the weighted average of the coordinates of those cells connected to i . An example of a perturbation is then to move a cell to a legal position close to its center of gravity and if there is another cell at that position to move that cell to some empty location or to its own center of gravity (which creates a chain of moves until the last cell in the chain is put in an empty location).

7.5 Partitioning

The direct motivation for paying attention here to the partitioning problem originates from min-cut placement. However, partitioning is itself an important problem in the field of VLSI design automation. It shows up e.g. when a large circuit has to be implemented with multiple chips and the number of pins on the IC packages necessary for interchip communication should be minimized.

Many versions of the partitioning problem exist and many algorithms for each version. A famous algorithm was published by Kernighan and Lin in 1970. It will be discussed in the next section. The section on Bibliographic Notes at the end of this chapter provides pointers to other algorithms.

7.5.1 The Kernighan-Lin Partitioning Algorithm

The model assumed by the algorithm is as follows: there is an edge-weighted undirected graph $G(V, E)$; the graph has $2n$ vertices ($|V| = 2n$); an edge $(a, b) \in E$ has a weight γ_{ab} ; if $(a, b) \notin E$, $\gamma_{ab} = 0$. The problem is to find two sets A and B , subject to $A \cup B = V$, $A \cap B = \emptyset$, and $|A| = |B| = n$, which minimizes the *cut cost* defined as follows:

$$\sum_{(a, b) \in A \times B} \gamma_{ab}$$

In other words, the goal is to minimize the total weight of the edges cut by the partitioning of V into the sets A and B . Note that the algorithm assumes that the clique model has been used for the representation of nets (see Section 7.1).

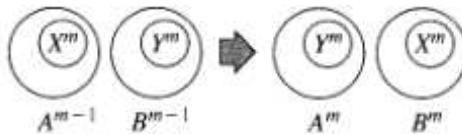


Figure 7.10 The interchange of subsets in the Kernighan-Lin algorithm.

The first thing to remark is that the problem is NP-complete and that the algorithm proposed by Kernighan and Lin is a heuristic that turns out to be rather successful. The principle of the algorithm is to start with an initial partition consisting of the sets A^0 and B^0 which, in general, will not have a minimal cut cost. In an iterative process, subsets of both sets are isolated and interchanged. In iteration number m , the set isolated from A^{m-1} will be denoted by X^m and the set isolated from B^{m-1} will be denoted by Y^m . The new sets, A^m and B^m are then obtained as follows:

$$\begin{aligned}A^m &= (A^{m-1} \setminus X^m) \cup Y^m \\B^m &= (B^{m-1} \setminus Y^m) \cup X^m\end{aligned}$$

This idea is illustrated in Figure 7.10. The iteration goes on until no improvement in the cut cost is possible.

An important issue in the algorithm is the construction of the sets X^m and Y^m . Note that for any nonoptimal partition, there are subsets X and Y that will lead to an optimal partition in one interchange step. The difficulty of the problem arises, of course, from the fact that these subsets cannot be identified easily. Therefore, the algorithm makes an attempt to find suitable subsets, interchanges them and then tries to make a new attempt, until the attempt does not lead to an improvement of the cut cost. In this context, each exchange of subsets is called a *pass*. The total number of passes needed turns out not to be dependent on the problem size n : most examples reported in the literature do not need more than 4.

The construction of the sets X^m and Y^m is based on *external* and *internal* costs for vertices in the sets A^{m-1} and B^{m-1} . The external cost E_a of $a \in A^{m-1}$ is defined as follows:

$$E_a = \sum_{y \in B^{m-1}} \gamma_{ay}, \quad a \in A^{m-1}$$

So, the external cost for vertex $a \in A^{m-1}$ is a measure for the *pull* that the vertex experiences from the vertices in B^{m-1} . In a similar way, the external cost E_b for a vertex $b \in B^{m-1}$ and the internal costs I_a and I_b can be defined:

$$E_b = \sum_{x \in A^{m-1}} \gamma_{bx}, \quad b \in B^{m-1}$$

$$I_a = \sum_{x \in A^{m-1}} \gamma_{ax}, \quad a \in A^{m-1}$$

$$I_b = \sum_{y \in B^{m-1}} \gamma_{by}, \quad b \in B^{m-1}$$

The difference between internal and external costs gives an indication about the desirability to move the vertex: a positive value shows that the vertex should be better moved to the opposite set, a negative value shows a preference to keep the vertex in its current set. The differences for the vertices in both sets are given by the variables D_a and D_b :

$$\begin{aligned} D_a &= E_a - I_a, \quad a \in A^{m-1} \\ D_b &= E_b - I_b, \quad b \in B^{m-1} \end{aligned}$$

Now the gain in the cut cost, Δ , resulting from the interchange of two vertices can be expressed as:

$$\Delta = D_a + D_b - 2\gamma_{ab}, \quad a \in A^{m-1}, \quad b \in B^{m-1}$$

The last term is a correction for a possible edge between a and b , which will continue to cross the cut after swapping the vertices.

The pseudo-code code of the Kernighan-Lin algorithm is shown in Figure 7.11. As mentioned earlier, each execution of the outer loop is a new pass. In the main inner loop, the `for` loop with iteration variable i , the subsets to be interchanged are constructed element by element. In each iteration of this loop, the pair $(a_i, b_i) \in A^{m-1} \times B^{m-1}$ giving the best improvement for the cut cost is selected. The vertices are then "locked", meaning that they cannot be selected once more in the inner loop. They are candidates to be included in the subsets. Actually, it is pretended that they have already been interchanged, and the differences between external and internal costs of the unlocked vertices are therefore updated (check the correctness of the expressions that update the values of D_x and D_y).

It is important to realize that the best cut cost improvement leading to the selection of a pair (a_i, b_i) may be negative. Once all vertices have been locked, the pairs are investigated in the order of selection: the actual subsets to be interchanged correspond to the sequence of pairs (starting with $i = 1$) giving the best improvement. So, pairs in the sequence may have negative cost improvements as long as the pairs following them compensate for it. Such a situation would e.g. occur when the exchange of two clusters of tightly connected vertices results in an improvement, while the exchange of individual vertices from each cluster does not improve the cut cost.

The algorithm will be illustrated using the example graph given in Figure 7.12. The graph consists of the vertices v_1 to v_8 . The initial partition consists of the two sets $\{v_2, v_3, v_6, v_7\}$ and $\{v_1, v_4, v_5, v_8\}$ with cut cost 14 as shown in Figure 7.12(a). The evolution of the algorithm in the first pass is shown in Figure 7.13. Each row in the figure corresponds to a value of the loop variable i as is shown in the first column. The values under a column headed by a vertex name v show the subsequent values of the variable D_v (the difference between the external cost E_v and the internal cost

```

initialize( $A^0, B^0$ );
 $m \leftarrow 1$ ;
do { for each  $a \in A^{m-1}$ 
    “compute  $D_a$ ”;
    for each  $b \in B^{m-1}$ 
        “compute  $D_b$ ”;
    for ( $i \leftarrow 1; i \leq n; i \leftarrow i + 1$ ) {
        “find unlocked vertices  $a_i \in A^{m-1}, b_i \in B^{m-1}$  such that
         $\Delta_i = D_{a_i} + D_{b_i} - 2\gamma_{a_i b_i}$  is maximal”;
        “lock  $a_i$  and  $b_i$ ”;
        for each “unlocked”  $x \in A^{m-1}$ 
             $D_x \leftarrow D_x + 2\gamma_{x a_i} - 2\gamma_{x b_i}$ ;
        for each “unlocked”  $y \in B^{m-1}$ 
             $D_y \leftarrow D_y - 2\gamma_{y a_i} + 2\gamma_{y b_i}$ ;
    }
    “find a  $k$  such that  $\sum_{i=1}^k \Delta_i$  is maximal”;
     $G \leftarrow \sum_{i=1}^k \Delta_i$ ;
    if ( $G > 0$ ) {
         $X^m \leftarrow \{a_1, \dots, a_k\}$ ;
         $Y^m \leftarrow \{b_1, \dots, b_k\}$ ;
         $A^m \leftarrow (A^{m-1} \setminus X^m) \cup Y^m$ ;
         $B^m \leftarrow (B^{m-1} \setminus Y^m) \cup X^m$ ;
        “unlock all vertices in  $A^m$  and  $B^m$ ”;
         $m \leftarrow m + 1$ 
    }
}
while ( $G \leq 0$ );

```

Figure 7.11 The pseudo-code description of the Kernighan-Lin algorithm.

I_v). The variables that are locked at a specific stage of the inner loop are underlined and the value Δ_i corresponding to the selected pair is given in the last column.

Once the inner loop has been traversed, the value k has to be determined such that the sum $\sum_1^k \Delta_i$ is maximal. In this case, the value of k is 1 and the subsets of vertices to be exchanged are $\{v_2\}$ and $\{v_4\}$ (note that $k = 3$ is optimal as well). This leads to the new sets $A^1 = \{v_3, v_4, v_6, v_7\}$ and $B^1 = \{v_1, v_2, v_5, v_8\}$ as is shown in Figure 7.12(b). The cut cost has now decreased from 14 to 8.

The evolution of the variables in the second pass is shown in Figure 7.14. The conclusion of the second iteration is that the maximal gain corresponds to the value of $k = 2$, which means that the vertex subsets to be exchanged are $\{v_3, v_4\}$ and $\{v_5, v_8\}$. The exchange of these subsets leads to $A^2 = \{v_1, v_2, v_3, v_4\}$ and $B^2 = \{v_5, v_6, v_7, v_8\}$ which is the optimal solution for the example. The cut cost now equals 2. Clearly, a third pass will not lead to an improvement and the algorithm will stop. The final partition is illustrated in Figure 7.12(c). Note that the Kernighan-Lin algorithm will not always find the optimal solution: it is just a powerful heuristic.

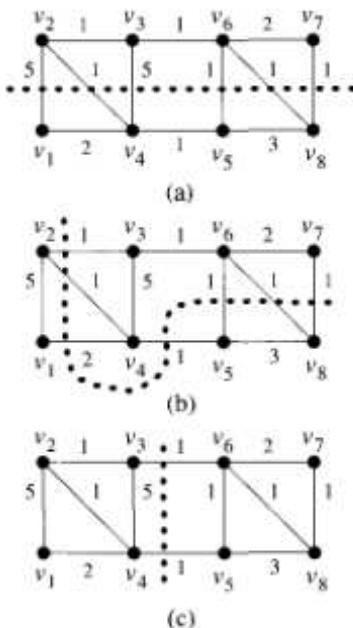


Figure 7.12 Different steps in the application of the Kernighan-Lin algorithm: the initial partition (a), the situation after the first iteration (b) and the final solution (c).

i	A^0				B^0				Δ_i
	v_2	v_3	v_6	v_7	v_1	v_4	v_5	v_8	
1	<u>5</u>	3	-1	-1	3	<u>3</u>	-1	-3	6
2		-5	-1	<u>-1</u>	-3		<u>-1</u>	-1	-2
3		-5	<u>1</u>		-3			<u>3</u>	2
4		<u>-3</u>			<u>-3</u>				-6

Figure 7.13 The first pass of the Kernighan-Lin algorithm applied to the graph of Figure 7.12.

i	A^1				B^1				Δ_i
	v_3	v_4	v_6	v_7	v_1	v_2	v_5	v_8	
1	-5	<u>-1</u>	-1	-1	-3	-3	-1	<u>-1</u>	-2
2	<u>5</u>		-3	-3	-7	-5	<u>3</u>		8
3			<u>-3</u>	-3	<u>-7</u>	-7			-10
4				<u>1</u>		<u>3</u>			4

Figure 7.14 The second pass of the Kernighan-Lin algorithm applied to the graph of Figure 7.12.

The time complexity of the algorithm is determined by the inner loop that is executed n times (remember that the number of executions of the outer loop was not problem-dependent). Finding the best pair of vertices to be locked next requires the pairwise comparison of elements each with at most n elements. Therefore, the number of comparisons is $\mathcal{O}(n^2)$ and the total time complexity becomes $\mathcal{O}(n^3)$. In practical situations (but not in the worst case) sorting the elements of A^{m-1} and B^{m-1} according to their difference values will limit the number of comparisons to the first few elements of the sorted lists. As sorting can be done in $\mathcal{O}(n \log n)$ time, this would lead to an overall time complexity of $\mathcal{O}(n^2 \log n)$. A better time complexity can be achieved by the use of more sophisticated data structures and a modified search strategy (see also the Bibliographic Notes).

Note that the Kernighan-Lin algorithm can be seen as a local search with variable neighborhood (see Section 5.5): the number of elements exchanged between the sets A^{m-1} and B^{m-1} depends on the feasible solution that one is visiting.

7.6 Bibliographic Notes

Many of the textbooks that deal with physical design automation, such as [Len90], [She93], [Sai95] and [Sar96], pay quite some attention to the placement problem. In addition, general information on the placement problem can be obtained through the review papers [Got86], [Bra87] and [Pre88].

Electric circuit representation as presented in this chapter is not so often discussed in the literature, as it is quite straightforward and because different applications have different requirements. Circuit representations that are e.g. appropriate for placement or partitioning are paid attention to in [Len90] and [Alp95]. What has been presented here, is an *internal* data structure, meant to reside in the computer's memory while an algorithm is processing it. A related issue is the *external* representation meant to be stored on file, usually in a human-readable format, that can be shared by many applications. Such a format is EDIF (Electronic Design Interchange Format), that has been standardized by ANSI [Kah92a]. It deals with the notions of cells, ports and nets and many, many more issues that are relevant for real-world CAD.

The wire-length metrics mentioned in the text and some others are listed in [Shi88], [Len90] and [Sai95].

The term *logistic signal* (for power and clock wires) was coined by Spaanenburg [Spa85]. An example of a min-cut algorithm for placement is given in [Bre77]. Examples of clustering algorithms can be found in [Ake82].

As reported in [Sun95] simulated annealing is probably the best algorithm currently known for placement, especially for standard-cell placement. Already the first publication on simulated annealing as a general-purpose optimization method [Kir83] presented results applied to the placement problem. Also, the introductory paper on simulated annealing [Rut89] uses the placement problem as the main illustration of the method. The Timberwolf system [Sec85] is famous for its good results in standard-cell placement by simulated annealing. The version discussed in [Sun95] directly eliminates overlaps after each move. A "rectangle packing" method based

on simulated annealing that is applicable to the building-block placement problem is reported in [Mur95]; this method uses an "encoding" for feasible solutions that excludes overlaps.

Apart from simulated annealing, the application of other general-purpose optimization methods to placement has been reported, including genetic algorithms [Coh87, Sha90] and simulated evolution [Kli89].

More information on the partitioning problem can be found in textbooks dealing with physical design automation, such as [Len90], [She93], [Sai95] and [Sar96]. Also, a number of review papers are available: [Don88] is rather mathematics oriented, [Alp95] is a very extended and detailed review, while [Joh96] mainly concentrates on the new developments in the years 1994-1996.

The original description of the Kernighan-Lin algorithm is given in [Ker70]. The NP-completeness of the version of partitioning problem as solved by the Kernighan-Lin algorithm is mentioned in [Gar79]. A modified version of the algorithm that deals with an explicit net representation (the bipartite-graph model) is given in [Sch72]. Algorithms that have the same objectives as the Kernighan-Lin algorithm but a better time complexity have been reported in [Fid82] and [Kri84]. However, the quality of the results obtained is often worse than the result of the Kernighan-Lin algorithm due to the fact that fewer possibilities for the exchange of subsets are investigated. The use of the method presented in [Dut93], on the other hand, does not affect the search space of the original Kernighan-Lin algorithm, but significantly improves the time complexity by using more sophisticated data structures and a better search strategy.

7.7 Exercises

- 7.1 Indicate how the data model for the representation of electric circuits, as presented in Section 7.1, can be extended for the description of hierarchically specified circuits. Which modifications are necessary for the data structure definitions given in Figure 7.2?
- 7.2 Apply the metrics mentioned in Section 7.2 to the two placements given in Figure 5.1. Do they consistently predict that the solution of Figure 5.1(c) is better than the one of Figure 5.1(d)?
- 7.3 The Kernighan-Lin algorithm as described in the text partitions the vertex set of a graph with $2n$ vertices in two sets of exactly n elements. How can one use the algorithm for a vertex set with an odd number of elements $2n + 1$ that is to be partitioned into sets of n and $n + 1$ elements? What about a situation in which the two sets do not need to have the same size but may differ by at most m elements?

Floorplanning

Often in VLSI design layout aspects are dealt with in a bottom-up fashion. The designer either uses cells from a library or designs his/her cells and subsequently composes the overall layout of the chip by means of placement and routing. This often results in poor utilization of the chip area due to large portions of the chip that are occupied by wiring.

Of course, only a well-conceived design methodology can result in final designs of high quality. When looking for such a methodology, it was obvious that the structured design methods developed for software systems (e.g. the hierarchical partitioning of the software in small procedures) could also be used for systems to be implemented in hardware. However, hardware design is harder, as a hardware system has to be embedded in two or three dimensions (a VLSI circuit can be said to be realized in two dimensions only as a relatively few number of layers constitute the third dimension). Software has an arbitrary number of dimensions: there are no limitations in the number of procedures that one procedure can call. The hardware analog of a procedure call is the exchange of signals between modules. When the number of interconnections between modules increases, it becomes harder to design a satisfactory circuit.

These insights have led to the so-called *floorplan-based design methodology*. This top-down design methodology advocates that layout aspects should be taken into account in all design stages. When the three design domains of VLSI are considered (see Section 1.2), a synthesis step from the behavioral domain to the structural domain should be followed by synthesis step from the structural domain to the physical domain. Note that at higher levels of abstraction, due to the lack of detailed information, only the relative positions of the subblocks in the structural description can be fixed. The illustration of the floorplan-based design methodology on the Y-chart is given in Figure 8.1. Taking layout into account in all design stages also gives early feedback: structural synthesis decisions can immediately be evaluated for their layout consequences and corrected if necessary. The presence of (approximate) layout information allows for an estimation of wire lengths. From these lengths, one can derive performance properties of the design such as timing and power consumption. They both increase when the wire lengths grow, as a consequence of the parasitic capacitances between the wires and the substrate.

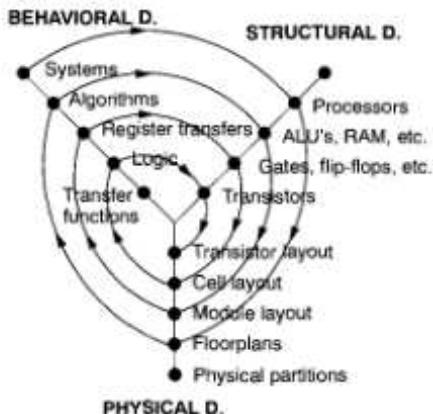


Figure 8.1 The visualization of the floorplan-based design methodology on Gajski's Y-chart.

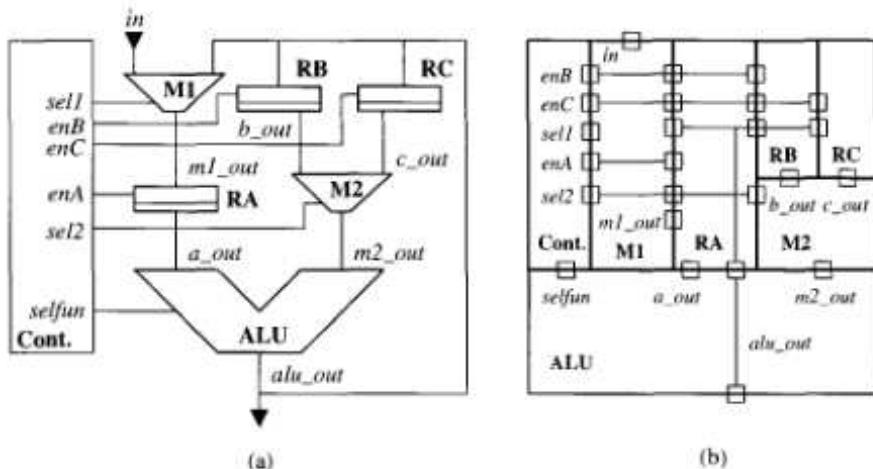


Figure 8.2 A structural description of some circuit (a) and a possible floorplan (b).

It is easy to deal with layout when structural detail at the lowest abstraction is available: one knows the exact number of transistors in the circuit and the way they are interconnected. At the moment that this type of structural information is not fully available, one can estimate the area to be occupied by the various subblocks and, together with a precise or estimated interconnection pattern, try to allocate distinct regions of the integrated circuit to the specific subblocks. This process is called *floorplanning*. The idea is illustrated in Figure 8.2, where a hardware structure has tentatively been mapped on a floorplan. Figure 8.2(a) shows the schematic of some circuit at the register-transfer level that consists of an ALU (arithmetic logic unit),

three registers, two multiplexers and a controller (see Section 12.1 for an explanation of the symbols used in the figure). Figure 8.2(b) shows a possible floorplan annotated with the names of the cells and nets of the schematic. Terminal positions (ports) are indicated by small squares on a cell boundary. *Feedthrough wires*, wires that cross a cell without the wire necessarily carrying a signal relevant to the cell, are indicated by lines connecting the terminals. A real-life floorplan will contain more information than shown in the figure such as the bit width of the wires (control signals only require a single wire, but data signals may e.g. be 16-bit wide). Note that functionally equivalent subblocks (registers, multiplexers) have different shapes and terminal positions. This is one of the main characteristics of floorplan-based design: one chooses the shape and terminal positions such that they fit best with the original structure, and assumes that there is a way to design the module satisfying the chosen shape and terminal positions.

The rest of this chapter is organized in two main parts. First some concepts related to floorplanning are explained followed by a discussion of "shape functions" and "floorplan sizing".

8.1 Floorplanning Concepts

8.1.1 Terminology and Floorplan Representation

The layout of an integrated circuit or, more precisely, its floorplan can be represented hierarchically: cells are built from other cells, except for those cells that are at the lowest level of the hierarchy. These lowest-level cells are called *leaf cells*. Cells that are made from leaf cells are called *composite cells*. Composite cells can contain other composite cells as well, which makes it possible to have an arbitrary number of levels in the hierarchy. The direct subcells of a composite cell are called its *children*. Conversely, every cell, except for the one representing the complete circuit, has a *parent cell*.

For the sake of simplicity, both leaf cells and composite cells are assumed to have a rectangular shape. If the children of all composite cells can be obtained by bisecting the cell horizontally or vertically, the floorplan is called a *slicing floorplan*. So, in a slicing floorplan a composite cell is made by combining its children horizontally or vertically (putting them next to each other from left to right or stacking them on top of each other). A natural way to represent a slicing floorplan is by means of a *slicing tree*. The leaves of this tree correspond to the leaf cells. Other nodes correspond with horizontal and vertical composition of the children nodes. The floorplan presented in Figure 8.2 is slicing; its slicing tree is given in Figure 8.3. In the figure a node labeled with an 'H' was obtained by horizontal composition and a node labeled with a 'V' by vertical composition. The ordering of a composite node's children is relevant: 'from left to right' in horizontal composition and 'from bottom to top' in vertical composition.

Not all floorplans are slicing. For example, in the so-called *wheel* or *spiral* floorplan presented in Figure 8.4, the children of the given cell cannot be obtained

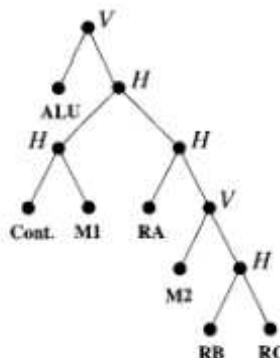


Figure 8.3 The slicing tree for the floorplan of Figure 8.2.

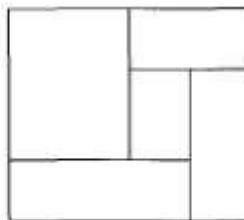


Figure 8.4 The "wheel" floorplan.

by bisections. Note that a composite cell needs to be composed of at least five cells in order not to be slicing. One can derive new composition operators from the wheel floorplan and its mirror image and use them in combination with the horizontal and vertical composition operators in a *floorplan tree*. A floorplan that can be described in this way is called a *floorplan of order 5* because the operators in the floorplan tree have at most 5 operands. Figure 8.5(a) gives an example of a floorplan of order 5 while the corresponding floorplan tree is given in Figure 8.5(c). Figure 8.5(b) shows the locations of the children cells by means of numeric labels that are also used in the tree of Figure 8.5(c). Clearly, a slicing floorplan can also be called a floorplan of *order 2*, because its floorplan tree only consists of the horizontal and vertical composition operators which have two operands.

Higher order floorplans that require operators with more than 5 operands also exist. In an extreme situation, a floorplan with n cells will require operators of order n to describe it and the floorplan tree will have a single parent cell having all leaf cells as its children. However, *hierarchical* descriptions using floorplan trees are seldom used for floorplans of order higher than 5. A representation mechanism that can deal with any floorplan is the *polar graph* which actually consists of two directed graphs: the horizontal polar graph and the vertical one. These graphs can be constructed by identifying the longest possible line segments that separate the cells in the floorplans.

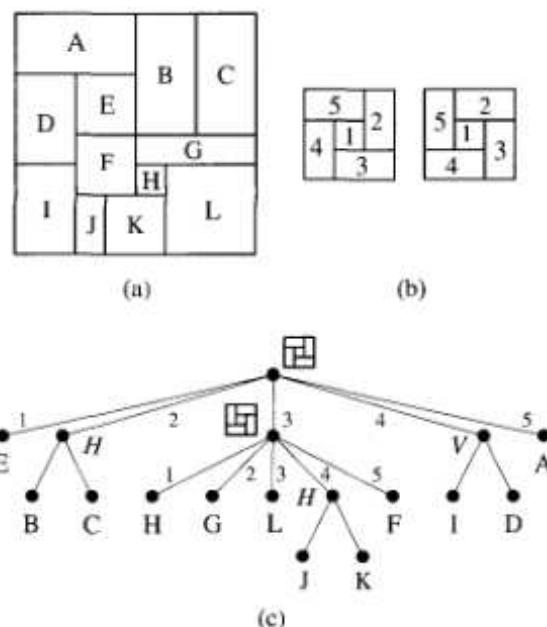


Figure 8.5 A floorplan of order 5 (a), the characterization of the operators used (b) and the floorplan tree (c).

The horizontal segments are used as vertices in the horizontal polar graph and the vertical segments as the vertices in the vertical polar graph. Each cell is represented by an edge in the polar graph. In the horizontal one, there will be an edge directed from the line segment that is the cell's top boundary to the line segment that is its bottom boundary. In the vertical one, a similar idea is used where the edge direction is from the left boundary to the right one. Figure 8.6 gives an example of the polar graph representation.

When two cells that need to be electrically connected have their terminals in the right order and separated correctly, the cells can simply be put against each other without the necessity for a routing channel in between them. Such cells are said to *abut*. The creation of a composite cell by means of the abutment of two cells is shown in Figure 8.7.

Ideally, all composite cells are created by abutment and no routing channels are used in a floorplan-based design methodology. This requires the existence of *flexible* cells, leaf cells that are generated by a cell generation program that is able to generate a certain function in a given region with predefined terminal positions. Besides, flexible cells should be able to accommodate *feedthrough* wires. Feedthrough wires are a consequence of the absence of routing channels (see Figure 8.2 for an illustration). On the other hand, floorplan-based design does not exclude the existence of routing channels. The channels can be taken care of by

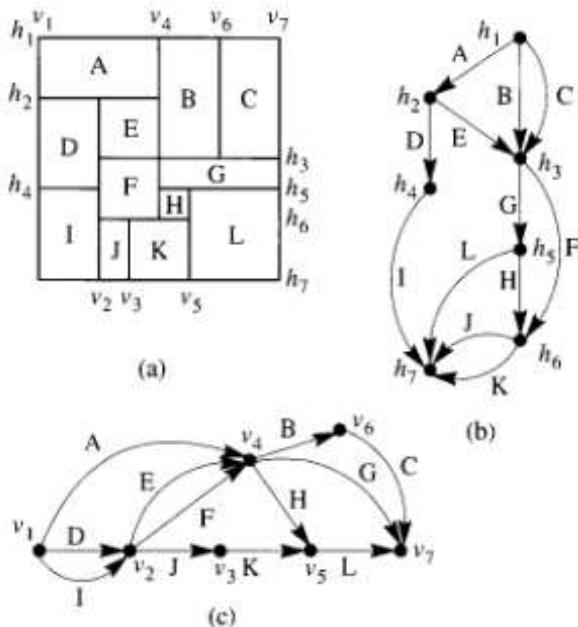


Figure 8.6 A floorplan with labeled horizontal and vertical line segments (a), its polar horizontal graph (b) and polar vertical graph (c).

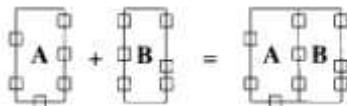


Figure 8.7 The creation of a composite cell by abutment.

incorporating them in the area estimations for the cells.

8.1.2 Optimization Problems in Floorplanning

There are a number of optimization problems related to floorplanning:

1. *Mapping of a structural description to a floorplan* (e.g. a slicing tree). In a true top-down design methodology, floorplanning will probably be performed manually or interactively as the number of children cells in which a parent cell is subdivided is relatively small and good decisions can be made based on designer experience. Another application of floorplanning is, however, the construction of a floorplan representation that is independent of the hierarchy in the structural domain. The circuit is then seen as the interconnection of many cells which have to be assigned a position. Floorplan construction is then just meant for

taking advantage of the flexible shapes of the leaf cells using floorplan sizing. The floorplan construction problem has similarities with the placement problem described in Chapter 7. The difference is that precise layout data that are known in placement are unknown in floorplanning. Anyhow, techniques known from placement like min-cut partitioning can also be used in floorplanning. At the same time, the terminal positions of abutting cells and the feedthrough routing patterns have to be determined. This problem is related to global routing and is sometimes called *abstract routing* in this context. These issues will not be discussed any further in this text.

2. *Floorplan sizing.* The availability of flexible cells implies the possibility of having different shapes for the same hardware unit. It is therefore possible to choose a suitable shape for each leaf cell such that the resulting floorplan is optimal in some sense (e.g. minimal area). This topic is discussed in the next section.
3. *Generation of flexible cells.* This task takes as input a cell shape, data on desired positions of terminals and a netlist of the circuit to be synthesized at some abstraction level (e.g. a netlist of transistors or standard cells) and uses a *cell compiler* to generate the layout that complies with the input. The problem is especially complex when the layout has to be composed of individual transistors because of the many degrees of freedom and the huge search space that is associated with it. Also, as this style of design amounts to full-custom design, quite some extra effort has to be spent in the *characterization* of the generated cells. Characterization is the process of determining all kind of electrical properties of a cell, such as parasitic capacitances and propagation delay, that will be necessary for an accurate simulation of the circuit containing the generated cell. Flexibility in a cell's shape can also be achieved using primitives belonging to a level higher than the transistor level. An example is a register file of 64 registers that can be laid out in many different ways, such as 8×8 , 16×4 , 4×16 or 1×64 . Another possibility for cell compilation with higher-level primitives is to generate a macro cell out of standard cells and take information on cell shape and terminal positions into account during the layout generation process. The topic of cell compilation will not be discussed further in this text.

8.2 Shape Functions and Floorplan Sizing

A cell implements a certain function. When the cell is flexible, one could say that the realization needs an area A . Whichever shape the cell will have, its height h and its width w have to obey the constraint $hw \geq A$. The minimal height given as a function of the width is called the *shape function* of the cell. Figure 8.8(a) gives the shape function of a cell with constant area: $h(w) = \frac{A}{w}$. The shaded area, where $h \geq \frac{A}{w}$, contains all points (h, w) corresponding to feasible shapes of the cell. It is not realistic to have cells that are very elongated. Due to design rules neither the height nor the width will asymptotically approach zero. The shape function can be

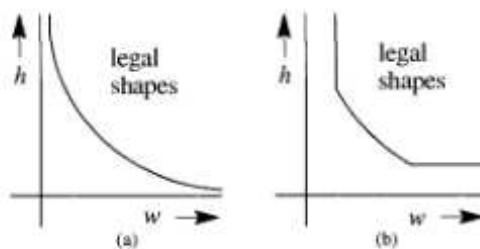


Figure 8.8 Shape functions for a cell without (a) and with (b) minimal width/height restrictions.

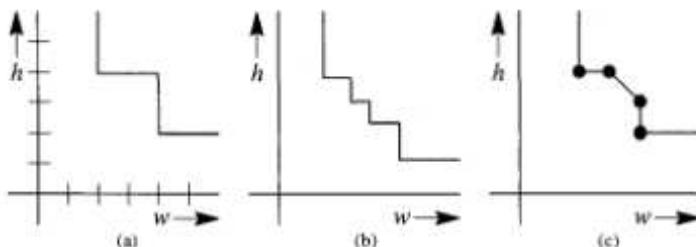


Figure 8.9 The shape function of an inset cell (a); the shape function of a cell with a discrete set of (h, w) pairs (b); a piecewise linear shape function (c).

modified such that the legal shapes have a minimal height and width as is shown in Figure 8.8(b).

Note: the concept of a function is used here in an informal sense. In order to define the shape function as a function in the mathematical sense, there should be at most one h value for each w value. It is left to the reader to refine the figures in this text and to solve the problems arising when computing the inverse of a shape function.

The shape functions mentioned so far refer to cells where height and width have continuous values. In reality, due to design rules, the height and width can only assume discrete values. Let us first consider an extreme case of a discrete-valued shape function: the *Inset* or *Rigid* cell. An inset cell, a predesigned cell residing in a library, has the possibility of rotations (only in multiples of 90°) and mirrorings as the only flexibility to be fit in a floorplan (note that such a cell will in general need routing channels around it). The shape function of a 2×4 inset cell is given in Figure 8.9(a). The figure has two horizontal segments, one corresponding to the horizontal orientation of the cell and the other to the vertical one. The shape function of a cell that can have a discrete number of (h, w) pairs is given in Figure 8.9(b). Each horizontal segment corresponds to a different realization of the cell.

The method for floorplan sizing, to be described below, can deal with any piecewise linear shape function. An example of such a function is given in Figure 8.9(c). A piecewise linear function is characterized by a partitioning of its input domain

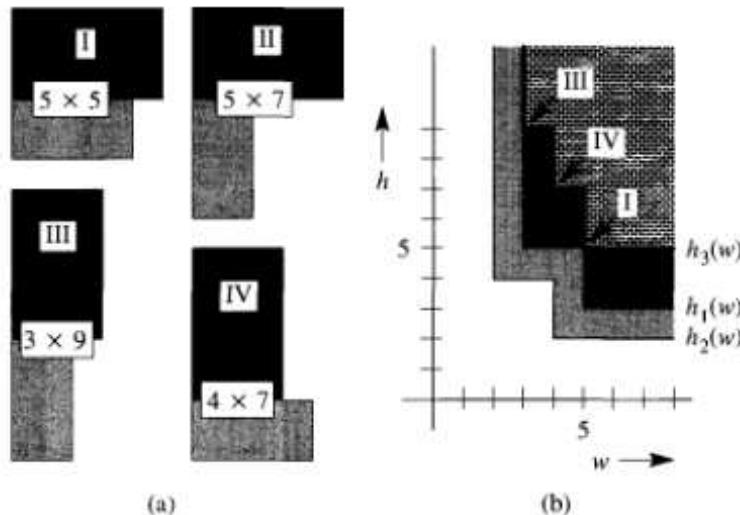


Figure 8.10 The four possible ways of vertical composition for two inset cells (a) and the shape functions related to this situation (b).

into intervals and a distinct linear behavior in each of these intervals. The points delimiting the intervals are called the function's *break points*.

The shape function of a composite cell in a slicing floorplan can be computed from the shape function of its children cells. First *vertical* composition is considered, i.e. the case where a cell c_1 is stacked on top of a cell c_2 . If the shape function of c_1 is indicated by $h_1(w)$ and the one of c_2 by $h_2(w)$, then the shape function $h_3(w)$ of the composite cell can be expressed as: $h_3(w) = h_1(w) + h_2(w)$. As only piecewise linear functions are considered, h_3 can efficiently be computed by evaluating h_1 and h_2 at their break points.

This idea is illustrated in Figure 8.10 for a small example where both c_1 and c_2 are inset cells with respective sizes of 4×2 and 5×3 . Clearly, there are four ways to stack the two cells vertically as shown in Figure 8.10(a); of these, the case labeled by 'I' has the smallest area. The shape functions $h_1(w)$ of c_1 and $h_2(w)$ of c_2 are given in Figure 8.10(b). $h_1(w)$ and $h_2(w)$ each have two break points: those of $h_1(w)$ are located at respectively $w = 2$ and $w = 4$, and those of $h_2(w)$ at $w = 3$ and $w = 5$. The shape function $h_3(w)$ of the composite cell can easily be constructed by evaluating $h_1(w)$ and $h_2(w)$ at these break points. It turns out that only three of the four break points are retained in $h_3(w)$ as the composite cell cannot be narrower than 3. The function values at the three break points of $h_3(w)$ in Figure 8.10(b) correspond to three of the four possible ways to realize a vertical composition shown in Figure 8.10(a), viz. the cases labeled with 'I', 'III' and 'IV'. The case labeled 'II' is a so-called *redundant* floorplan: one does not need to consider this case when searching for an optimal floorplan size.

In the case of *horizontal* composition, the shape function of a composite cell has

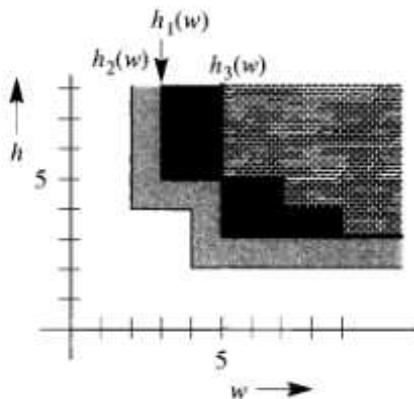


Figure 8.11 The shape functions of two inset cells and of the cell obtained by their horizontal composition.

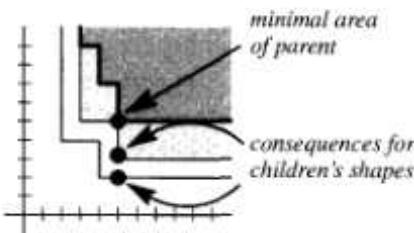


Figure 8.12 The consequence for the shape of children cells of fixing the shape of the parent cell.

to be computed using a detour via the inverses of the children's shape functions. The inverse of the composite cell's shape function is the sum of the inverses of its children cell's shape functions: $h_3^{-1}(w) = h_1^{-1}(w) + h_2^{-1}(w)$. Figure 8.11 shows the shape function resulting from the horizontal composition of the two inset cells used in the example of Figure 8.10.

Once the shape function of a composite cell is known, it is possible to choose a suitable shape for it from its legal shapes. Although one could consider other optimization goals, the most obvious entity to optimize is the area of the composite cell. A choice for the shape of the parent cell constrains the shapes of the children cells. If one chooses for a point on the boundary of the parent's shape function, the corresponding shapes of the children cells are uniquely determined. It is not so difficult to prove that the shapes of minimal area are always located on the boundary of the shape function and they will always be one of the break points if the shape function is piecewise linear.

Figure 8.12 illustrates how the children's shapes can be obtained from the optimal parent shape for the vertical composition example given in Figure 8.10. The eval-

ation of the area at the break points of the shape function $h_3(w)$ gives $w = 5$ (and $h = 5$) as the one with minimal area. Tracing back how $h_3(w)$ was obtained from $h_1(w)$ and $h_2(w)$, one finds that the minimal area is obtained for the orientation of the inset cells shown as 'I' in Figure 8.10(a).

The procedure just described for vertical composition can easily be adapted for horizontal composition, which means that children shapes can be easily derived from the chosen parent shape for both types of composition. It is now possible to formulate the *sizing* algorithm for slicing floorplans:

1. Construct the shape function of the top-level composite cell in a bottom-up fashion starting with the lowest level and combining shape functions while moving upwards.
2. Choose the optimal shape of the top-level cell.
3. Propagate the consequences of the choice for the optimal shape down the slicing tree until the shapes of all leaf cells are fixed.

The sizing problem for slicing floorplans can be solved in polynomial time by the algorithm just mentioned. Assume that there are n cells with a piecewise linear shape function and that the total number of break points in all shape functions is q ($q \geq n$). Assume further that the depth of the slicing tree is d . Then, at each level of the tree the total number of computations for the computation of the new shape functions is $\mathcal{O}(q)$ which gives $\mathcal{O}(dq)$ for the time complexity of the complete algorithm, as Step 1 is the dominant part of the sizing algorithm. If the tree is balanced, i.e. $d = \mathcal{O}(\log n)$, then the time complexity becomes $\mathcal{O}(q \log n)$. At the other extreme, viz. a totally unbalanced tree, the time complexity becomes $\mathcal{O}(qn)$. Note that the slicing tree of the example in Figure 8.3 is rather unbalanced.

An interesting question to consider is the sizing of floorplans that are not slicing. The sizing problem is then NP-complete. Suppose that the floorplan has n leaf cells and that each leaf cell has at most k different shapes. The last assumption is quite realistic, but more restrictive than the piecewise linear shape functions that may allow an infinite number of shapes. One possible approach that could be used for any floorplan, is to the use of a general purpose optimization method such as genetic algorithms (see Section 5.8). A chromosome can consist of a string with n positions where each position indicates which of the possible shapes for each leaf cell has been included in the floorplan. The corresponding area can then be obtained by computing the height and width of the floorplan, which amounts to the computation of the longest paths in the vertical and horizontal polar graphs respectively. This can be done in $\mathcal{O}(n)$ time using the algorithm of Section 6.4.1, because the polar graphs have exactly n edges each.

The sizing of a hierarchical floorplan of order 5 can in principle be done in a similar way as the sizing of slicing floorplans. The only problem is how to find the shape function of a composite cell using an operator of order 5, given the shape function of the children cells. A brute force enumeration will lead to k^5 possibilities, if each child has k possible shapes. It turns out that one can obtain all nonredundant

floorplans using more clever algorithms having a time complexity of $\mathcal{O}(k^2 \log k)$. The use of such an algorithm in the bottom-up traversal of the floorplan tree with depth d will result in an algorithm of exponential complexity $\Omega(k^{2d})$ (disregarding the logarithmic factor for the sake of simplicity). This is not surprising given the NP-completeness of the problem.

8.3 Bibliographic Notes

Most of the textbooks on VLSI physical design automation, such as [Len90], [She93], [Sai95] and [Sar96], pay some attention to the floorplanning problem. The discussion in these books includes algorithms on how to obtain a slicing or nonslicing floorplan from a structural description, a topic that has been left out of this chapter. Floorplanning receives considerable attention in the review paper by Kuh and Ohtsuki [Kuh90] that covers VLSI layout in general.

An interesting article on structured design for VLSI is [Seq83]. The floorplan-based design methodology has been advocated in many publications including [Gin84], [Zim86] and [Pig91].

Possible graph representations of floorplans that can be used both for the slicing as the nonslicing case, are discussed in detail in [Ott88].

An example of a system that uses a min-cut partitioning technique for floorplanning is [Hea85]. An abstract router to be used for floorplanning is described in [Hea87]. A system that includes floorplan construction, sizing and routing is presented in [Len93]. A review paper on cell compilation, including the generation of flexible cells, is [Gaj88a]. The example of the multiple shapes for a register file containing 64 registers originates from [Wim89].

Otten and Stockmeyer were the first to publish on floorplan sizing. The paper of Otten [Ott83] considers the sizing of slicing floorplans for cells with piecewise linear shape functions and shows that the evaluation of the shape function at its breakpoints is sufficient in order to get the minimal area. The article by Stockmeyer [Sto83] considers slicing floorplans in which all leaf cells are inset cells. It also proves that the sizing problem for floorplans in general is NP-complete. The analysis of the time complexity of the sizing algorithm presented in the text can be found in [Len90]. Shi has shown that the sizing of slicing floorplans can always be performed with a time complexity of $\mathcal{O}(n \log n)$ irrespective of the fact whether the slicing tree is balanced or not and that no faster algorithm can be designed [Shi95] (the assumption made is that each leaf cell is an inset cell).

The idea to use a genetic algorithm for floorplan sizing in a way similar to the one described in the text is presented in [Reb96]. A special-purpose algorithm is described in [Pan95]. It has the nice property that it reduces to the Otten/Stockmeyer algorithm when applied to slicing floorplans. Algorithms that find the nonredundant shapes in $\mathcal{O}(k^2 \log k)$ time when composing with an order 5 operator, are the topic of [Che93] and [Pan94].

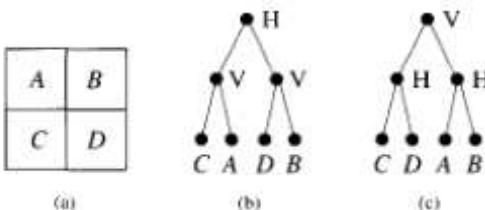


Figure 8.13 A floorplan (a) and two alternatives for its slicing tree (b,c).

8.4 Exercises

- 8.1** Consider the floorplan of Figure 8.13(a), consisting of four leaf cells: A , B , C and D . The leaf cells are inset cells that can be rotated and mirrored. Their dimensions are:

$A : 2 \times 2$

$B : 1 \times 3$

$C : 1 \times 4$

$D : 2 \times 4$.

Give the shape functions of these cells. Compute the optimal shape of the circuit, applying the sizing algorithm given in Section 8.2 using the slicing tree of Figure 8.13(b). Repeat the computation, now using the slicing tree of Figure 8.13(c). Compare the results.

9

Routing

The *routing* problem consists of interconnecting the cells that have been assigned positions as a solution of the placement problem. The specification of a routing problem will consist of the position of the *terminals*, the *netlist* that indicates which terminals should be interconnected and the area available for routing in each layer.

Routing is normally performed in two stages. The first stage, called *global* or *loose routing*, determines through which wiring channels a connection will run. The second stage, called *local* or *detailed routing*, fixes the precise paths that a wire will take (its position inside a channel and its layer). In this chapter, the two problems will be discussed in the reverse order. Already within the class of local routing problems a multitude of specific problems can be distinguished as will become clear in the first section of this chapter. Two of these, viz. *area routing* (by means of Lee's maze routing algorithm) and *channel routing*, will then be discussed in more detail. Area routing techniques can be used by global routing algorithms as well. This is the reason why the presentation of global routing techniques is postponed until the second part of this chapter. The discussion on global routing pays considerable attention to the construction of minimum rectilinear Steiner trees.

9.1 Types of Local Routing Problems

There are many types of local routing problems. They can be characterized by some "parameters", the most important of which are listed below. Different "settings" of these parameters define different routing problems that may or may not require special techniques for solving them. The parameters are:

- The number of wiring *layers*. The number of layers available depends on the technology and the design style (some layers might already be used for other purposes: one cannot e.g. route across a cell in diffusion or polysilicon). Traditionally, most technologies offered only two wiring layers, which is a minimal requirement to realize crossing connections. Recent developments have led to technologies that make several more layers available. A contact cut that realizes a connection between two layers is often called a *via* in the context of routing.

- The *orientation* of wire segments in a given layer. *Reserved-layer* models of routing use either horizontal or vertical segments in one layer, but not both. This restriction does not always apply. Sometimes it is also allowed to use segments with an orientation that is a multiple of 45° .
- *Gridded* or *gridless* routing. In gridded routing, all wire segments run along lines of an orthogonal grid with uniform spacing between the lines. In gridless routing, wires of different widths as well as contacts (that normally are wider than minimum-width wires) are explicitly represented.
- The presence or absence of *obstacles*. Sometimes the complete *routing area* is available for routing, sometimes part of the area in one or more layers is blocked.
- The *position* of the terminals. Terminals are sometimes located on the boundary of the routing area, but might sometimes as well be located anywhere inside the area.
- Terminals with a *fixed* or *floating* position. In some problems the position of the terminals is fixed, but in other problems the router can move the terminal inside a restricted area.
- *Permutability* of terminals. Sometimes the router is allowed to interchange terminals because they are *functionally equivalent* (e.g. the two inputs of an NAND-gate are functionally equivalent).
- *Electrically equivalent* terminals. In some situations, a group of terminals belonging to the same net may already be connected to each other (e.g. in a layer not available for routing). Then the router should connect the rest of the net to only one of the terminals in this group, whichever is the most suitable.

The *area routing* problem to be discussed in the next section is characterized by a single wiring layer, a grid, the presence of obstacles, and fixed terminals in all the routing area. The *channel routing* problem to be discussed in Section 9.3 can be characterized by two reserved layers, a grid, no obstacles, a rectangular area, two rows of fixed terminals on parallel boundaries and floating terminals on the other boundaries.

9.2 Area Routing

Routing problems in which terminals are allowed anywhere in the area available for routing are normally classified as *area routing* problems. In this section, an algorithm will be presented for a simple version of area routing, namely the version for a single wiring layer. This algorithm is the "path connection" or "maze routing" algorithm published by Lee in 1961. It can be considered the first significant contribution to routing. It is mentioned here because many current-day algorithms for routing still incorporate this algorithm in some way.

The basic algorithm is meant to realize a connection between two points in a plane, in an environment that may contain obstacles. One of these points is called the

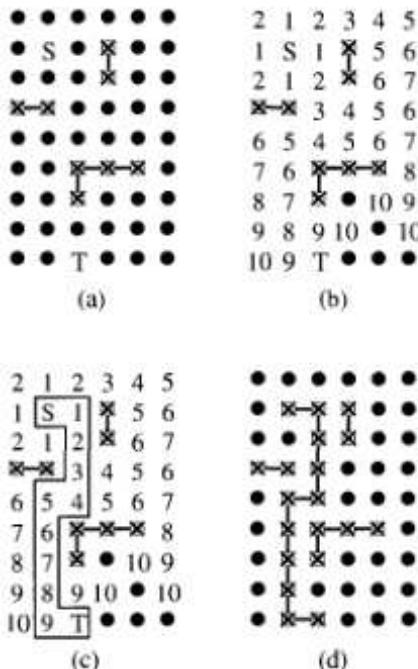


Figure 9.1 The basic version of Lee's algorithm: the points to be interconnected, S and T, together with the obstacles (a); the wave propagation from S to T (b); the path found (c); the situation to be used for new interconnections (d).

"source" terminal and the other the "target" terminal. If a path exists, the algorithm always finds the shortest connection, going around obstacles. It operates on a grid which means that the distance between two horizontally or vertically neighboring grid points corresponds to the shortest possible wire segment (the grid distance is normally derived from the design rules for a specific technology). Obstacles are modeled by grid points through which no wire segments can pass. Figure 9.1(a) shows an example problem that will be used for illustrating the different aspects of Lee's algorithm. In the figure, the dots indicate the grid points available for routing and the crosses the obstacles (e.g. wires resulting from earlier routing). The symbols "S" and "T" correspond to the source and target terminals respectively. The algorithm will try to find a connection from the source to the target.

The algorithm consists of three steps: *wave propagation*, *backtracing*, and *cleanup*. Its description in pseudo-code is given in Figure 9.2. Informally, in the first step a *wave front* of points is expanded, starting from the source terminal S. Points belonging to the same wave front receive the same integer-valued label, which is increased by one for subsequent wave fronts. This front goes around the obstacles in the routing area. The first step finishes when the front hits upon the target terminal T. This has been illustrated in Figure 9.1(b) that shows the successive wave fronts.

```

struct grid_point {
    int value; /* zero for unused, positive for a label, -1 for obstacle */
    ...
}

lee(struct grid_point S,T)
{
    set of struct grid_point wave_front, new_wave_front;
    struct grid_point neighbor, element, path_element;
    int label;
    /* Step 1: wave propagation */
    new_wave_front ← {S};
    label ← 0;
    while (T ∉ new_wave_front) {
        label ← label + 1;
        wave_front ← new_wave_front;
        new_wave_front ← ∅;
        for each element ∈ wave_front
            for each "neighbor of" element
                /* A neighbor is located above, below, at the left or at the right. */
                if (neighbor.value == 0) {
                    neighbor.value ← label;
                    new_wave_front ← new_wave_front ∪ {neighbor};
                }
        }
        /* Step 2: backtracing */
        path_element ← T;
        for (i ← label; i ≤ 1; i ← i - 1) {
            path_element ← "the neighbor of path_element such that
                           neighbor.value= i";
            /* In case of multiple possibilities use a heuristic to make a choice. */
            path_element.value ← i - 1
        }
        /* Step 3: clean up */
        for each point "on the grid"
            if (point.value > 0)
                point.value ← 0;
    }
}

```

Figure 9.2 The pseudo-code of Lee's path-connection algorithm.

Because the i th wave front contains all points at distance i from the source, it is guaranteed that the label that the target receives equals the length of the shortest path (in this case: 10). The second step consists of determining the shortest path itself by selecting a sequence of points in the grid with decreasing labels starting from the target and going back to the source. Note that in this backtracing step, sometimes the neighbor with label i is not unique: a heuristic should be used to make a choice. An example of a heuristic is not to change the orientation of the path unnecessarily, minimizing the number of corners in the solution in this way. A possible path for the considered example is shown in Figure 9.1(c). Once a path has been found, it will act as an obstacle for the next connections to be made. In order to be able to run the algorithm for other pairs of source-target points, the labels of those points that did not take part in the solution have to be cleaned up. This is done in the third step of the algorithm. The situation for the example problem after the cleanup phase and after the conversion of the path found into obstacles is shown in Figure 9.1(d). It is easy to see that the worst-case time complexity of Lee's algorithm operating on an $n \times n$ grid is $\mathcal{O}(n^2)$. Its space complexity is also $\mathcal{O}(n^2)$.

Lee's algorithm has been adapted for different purposes, like operating with multiple layers, using various cost models, working with nets having more than two terminals, etc. Also many improvements have been made on the efficiency of the basic algorithm. In the case that there are multiple layers, the algorithm operates on a three-dimensional grid, where the size of the third dimension equals the number of layers available. Especially in this three-dimensional version of the algorithm, the use of a more sophisticated cost function becomes necessary. In the basic algorithm, the cost of a path is simply the path length: a connection between two neighboring points contributes a unity cost to the total cost. One can introduce higher than unity costs for vias, for horizontal wire segments in a layer that is meant for vertical connections, etc.

When a net has three or more terminals first a path between two terminals should be found and then a generalization of the algorithm has to be used where a path can either act as a source or a target for the wave propagation. In this way the terminals are added one by one to the routing of the net. As opposed to the version for two terminals, Lee's algorithm does not guarantee the shortest possible path, when more than two terminals have to be interconnected. Note: in the absence of obstacles, the shortest-interconnection problem for more than two terminals is equivalent to the minimum rectilinear Steiner tree problem, which has been mentioned to be NP-complete in Section 4.5.

As mentioned above, when routing several nets, the paths of the nets which already have been routed act as obstacles for those still to be routed. Actually, the fact that nets have to be routed sequentially is the weak point of Lee's algorithm. Routing the nets in a different order strongly influences the final result. For some problems, a careful ordering of the nets decides between failure and success. Heuristics have, for this reason, been proposed to determine an appropriate ordering. However, as problems are known that cannot be solved by routing in any possible order, the merit of maze routing (even with ordering heuristics) is limited. On the other hand,

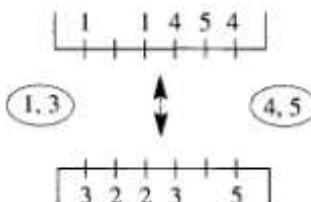


Figure 9.3 An instance of the channel routing problem.

it turns out that the principles of maze routing can be used with success within more sophisticated "iterative improvement" style algorithms where the routing of most nets is already known and a few nets are rerouted using the unoccupied grid points.

9.3 Channel Routing

Lee's algorithm is good in finding paths in an environment with many obstacles (when there are only a few paths with minimal length). However, it behaves poorly when there are few obstacles: it does not have the capability to choose a good path among the many possible paths. This section deals with such a situation without initial obstacles.

Channel routing occurs as a natural problem in standard cell and building block layout styles, but also in the design of printed circuit boards (PCBs). It consists of routing nets across a rectangular channel, as in Figure 9.3. In the figure, all terminals belonging to the same net have the same number as is customary in the field of routing. As opposed to Figure 9.1, the grid points are not shown; the grid distance is equal to the horizontal separation between the terminals. The nets have *fixed* terminals at the top and bottom of the channel and *floating* terminals at the "open" sides, at the left and right. A floating terminal is known to enter the channel on the left or on the right side, but it is up to the router to determine the exact position. In the example, the nets 1 and 3 have floating terminals at the left side and the nets 4 and 5 at the right. As a matter of fact, the height of the channel, the separation between top and bottom, is not fixed either. The main goal of channel routing is the minimization of the height, while a secondary goal is the minimization of the total wire length and the number of vias.

A routing problem that has some similarity with channel routing is *switchbox routing*. In this problem, fixed terminals can be found on all four sides of the rectangular routing area. So, opposite to channel routing, the minimization of the area is not an optimization goal. Switchbox routing is rather a *decision* problem: the goal is to find out whether a solution exists. When a solution can be found, a secondary goal is to minimize the total wire length and the number of vias.

In the rest of this section, first some more attention is paid to the channel routing models and then the problem is analyzed by means of the so-called *vertical* and

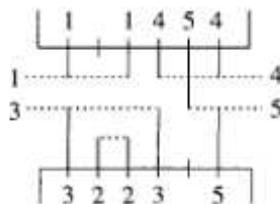


Figure 9.4 The solution of the channel routing problem of Figure 9.3: three rows are necessary.

horizontal constraints. Finally, an algorithm is presented that can solve the channel routing problem.

9.3.1 Channel Routing Models

Several models can be used for channel routing. The "classical" model is as follows:

- All wires run along orthogonal grid lines with uniform separation.
- There are two wiring layers.
- Horizontal segments are put on one layer and vertical segments on the other one.
- For each net, the wiring is realized by a single horizontal segment, with vertical segments connecting it to all terminals of the net. An exception is made when cycles in the *vertical constraint graph* (see later on) occur. In this case the problem cannot be solved unless at least one net in each cycle is realized with two horizontal segments.

The solution using the classical model of the example problem presented in Figure 9.3 is given in Figure 9.4. The vertical wire segments in the first wiring layer are shown as solid lines while the horizontal segments in the second layer are shown as dashed lines. Many variations on the classical model are possible. Routers have been designed for working without a grid. The so-called *gridless* routing model allows that each wire has a specific width (e.g. power wires that carry relatively large currents can be made wider to have less resistance). Other models use e.g. nonorthogonal grids, where 45° turns are allowed. As technology is progressing, more layers are becoming available for routing and algorithms are developed that can deal with more than two layers.

The model for routing where each layer has only wires in one direction is called the *reserved-layer* model. The use of this model can be motivated by referring to physical effects at the realization level: capacitive coupling between two wire segments running one on top of the other can lead to crosstalk. It cannot be denied, however, that this simple model with a relatively small solution space also simplifies the task of the designer of routing algorithms. Many modern algorithms work with a nonreserved layer model that has a larger solution space. The effects of using

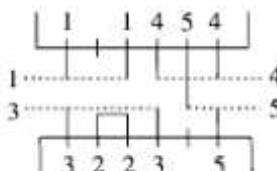


Figure 9.5 The solution of the channel routing problem of Figure 9.3 in the nonreserved layer model: two rows are sufficient.

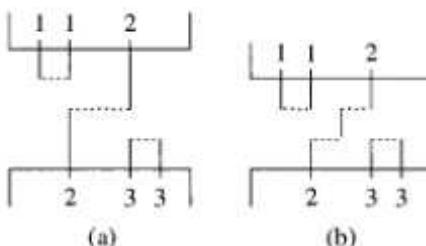


Figure 9.6 When the routing of Net 2 can only have one horizontal segment, the channel needs three rows (a); when doglegging is allowed, one row less is sufficient (b).

a nonreserved layer model are illustrated in Figure 9.5 where the solution of the problem in Figure 9.3 requires only two rows as compared to the three rows in the classical model. In the figure, the segment of Net 3 that runs on top of a segment of Net 2 has a slight offset with respect to the grid for the sake of clarity.

The requirement that all nets should be realized by a single horizontal segment, although again facilitating the task of the algorithm designer, may be too restrictive for the quality of the solution. The use of *doglegs*, i.e. of more than one horizontal segment per net, often offers the possibility of channel height reduction. Figure 9.6 shows an example of a channel routing problem where the introduction of a dogleg for Net 2 leads to a reduction of the channel height from 3 to 2.

9.3.2 The Vertical Constraint Graph

Consider a pair of terminals located in the same column and entering the channel in the same layer (see Figure 9.7(a)). It is obvious that in any solution of the problem, the endpoint of the segment coming from the top has to finish at a position higher than the endpoint of the bottom segment (otherwise, there would be a shortcircuit). This restriction is called a *vertical constraint*. Each column having two terminals in the same layer gives rise to a vertical constraint. The constraints are often represented in a *vertical constraint graph* (VCG). In this directed graph, the vertices represent the endpoints of the terminal segments and the directed edges represent the relation "should be located above" (see Figure 9.7(b)).

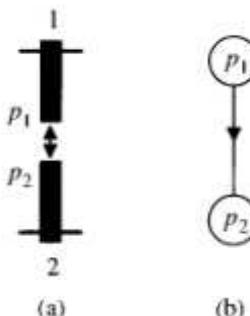


Figure 9.7 The vertical constraint between the interior endpoints p_1 and p_2 of terminal segments belonging to nets 1 and 2 (a) and its representation in the vertical constraint graph (b).

If nothing else is assumed, the VCG is rather uninteresting: it consists of pairs of vertices, one pair for each column that has two terminals in the same layer, each pair connected by a single directed edge from one vertex to the other, and unconnected vertices for the other columns. However, when the classical model for channel routing is used (one horizontal segment per net), for each net, all terminal segments end at the same row. This fact can be incorporated in the VCG by merging all vertices of the same net into a single vertex. Figure 9.8(a) shows an example problem that will be used to illustrate this. The vertical constraints considered separately for each of the columns a to h are shown in Figure 9.8(b). If one merges all vertices associated with the same net, the constraint graph of Figure 9.8(c) is obtained. This graph imposes a unique solution to the problem that is shown in Figure 9.8(d).

Up to now two extreme forms of the VCG have been presented, one of them is *fully merged* and the other one is *fully separated*. Many intermediate forms are also possible. They all have in common that one vertex corresponds to one row position for the horizontal segment that interconnects all interior endpoints of the terminal segments associated with the vertex. The main problem with the fully merged form is the possible existence of cycles, in which case the corresponding layout cannot be realized: a segment cannot be at the same time above and below another one. The problem can be solved by splitting one of the vertices in the cycle into two vertices (see Figure 9.9 for an example). Doglegging, which can lead to a reduction of the channel height as was shown in Figure 9.6, also involves splitting vertices in the VCG.

If the vertical constraints were the only constraints to be satisfied in the search of solution for channel routing, the problem would be relatively easy. In the absence of cycles in the VCG, a solution with a single horizontal segment per net would amount to finding the *longest path* in the graph. However, the presence of *horizontal constraints* that will be introduced below makes the problem considerably more complicated.

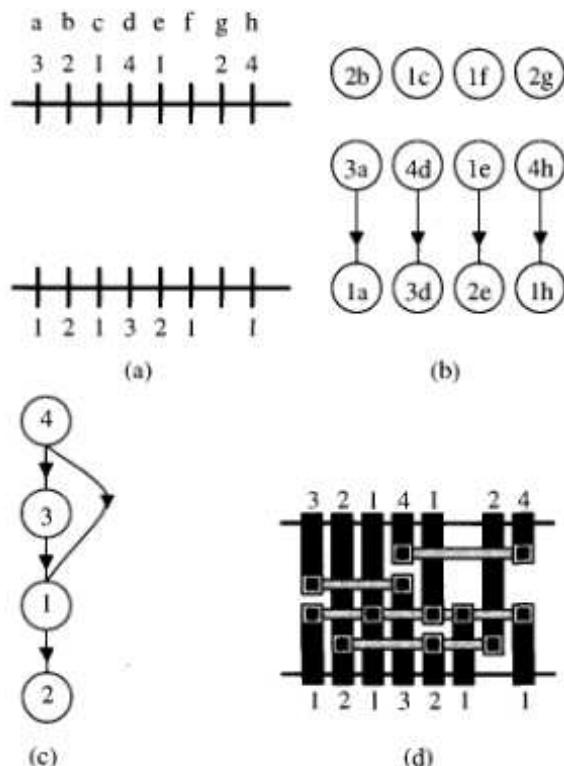


Figure 9.8 A simple channel routing problem (a), its VCG based on individual columns (b), the VCG when one horizontal segment per net is used (c) and the unique solution within the classical model (d).

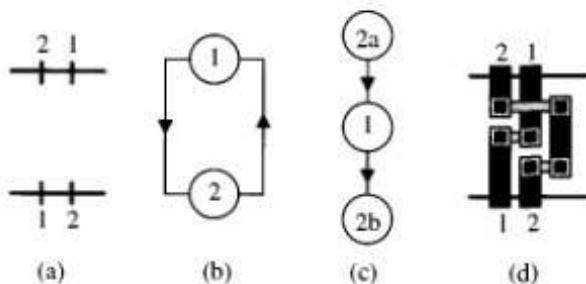


Figure 9.9 A simple problem (a) with a circular VCG (b); the cycle is broken by splitting one of the vertices (c), which makes it possible to solve the problem (d).

9.3.3 Horizontal Constraints and the Left-edge Algorithm

If, in the classical model for channel routing, horizontal segments belonging to different nets are put on the same row (implying that they will be in the same layer too), the segments cannot overlap (otherwise there would be a shortcircuit). This restriction is called a *horizontal constraint*.

If an instance of the channel routing problem does not have any vertical constraints (this happens e.g. when on each vertical grid line there is either a terminal at the top or at the bottom but not at both positions), the so-called *left-edge algorithm* solves the problem optimally. It was introduced by Hashimoto and Stevens for printed circuit board routing.

A net i in a channel routing problem without vertical constraints can be characterized by an interval $[x_{i_{\min}}, x_{i_{\max}}]$, corresponding to the left-most and right-most terminal positions of the net. The goal of channel routing is then reduced to assign a row position in the channel to each interval. No information is lost by the characterization of the net by an interval because it is known that there are no vertical constraints, and that the terminals of the net can therefore reach the horizontal segment corresponding to this interval irrespective of the row on which it will be located. An optimal solution combines those nonoverlapping intervals on the same row that will lead to a minimal number of rows.

The number of intervals that contain a specific x-coordinate x is called the *local density* at column position x and will be denoted by $d(x)$. The maximum local density in the range of all column positions is called the channel's *density* and is denoted by d_{\max} :

$$d_{\max} = \max_x d(x)$$

Obviously, the density is a lower bound on the number of necessary rows: all intervals that contain the same x-coordinate must be put on distinct rows.

The left-edge algorithm always finds a solution with a number of rows equal to the lower bound. Its pseudo-code is given in Figure 9.10. It is assumed that structures for the representation of intervals and linked lists of intervals, called `interval` and `list_of_interval` respectively, have been declared elsewhere. The code contains some standard "list processing" function calls: `first(l)` gives the first element of a list l ; `rest(l)` gives the list that remains when the first element is removed from l ; finally, `remove(e, l)` constructs a list of the elements of l in their original order leaving out those elements equal to e .

The algorithm operates on the list `i.list` that contains the intervals in order of increasing left coordinate. The main part of the algorithm consists of two nested loops. Each execution of the outer loop fills one row of the channel. The filling is done by the inner loop that scans `i.list` and removes from it the first interval that fits in the row. The algorithm is an example of a *greedy* algorithm, i.e. there is no sophisticated searching involved in the algorithm. The first "candidate" interval is directly inserted in the solution. Remarkably, the algorithm is always able to find a

```

left.edge(struct list.of.intervals i.list)
{
    /* the intervals in i.list are sorted by their left coordinate */
    set of set of struct interval solution;
    set of interval row;
    struct interval f;
    solution ← ∅;
    while ("i.list is not empty") {
        f ← first(i.list);
        i.list ← rest(i.list);
        row ← ∅;
        do { row ← row ∪ {f};
              f ← "first element in i.list nonoverlapping with f"
              i.list ← remove(f,i.list);
            } while ("such an f can be found");
        solution ← solution ∪ {row};
    }
    return(solution);
}

```

Figure 9.10 The left-edge algorithm.

global optimum, as opposed to most greedy algorithms that are likely to generate a local optimum (see Exercise 9.4).

Consider the instance of a channel routing problem without vertical constraints given in Figure 9.11(a). The seven nets can be represented by the following set of intervals: $i_1 = [1, 4]$, $i_2 = [12, 15]$, $i_3 = [7, 13]$, $i_4 = [3, 8]$, $i_5 = [5, 10]$, $i_6 = [2, 6]$ and $i_7 = [9, 14]$. The problem has a density of 3: $x = 3$ is for example contained in the intervals i_1 , i_4 and i_6 (there are many more columns that are covered by three intervals). The solution found by the left-edge algorithm is: $\{i_1, i_5, i_2\}$, $\{i_6, i_3\}$, $\{i_4, i_7\}$. The routing pattern corresponding to this solution is shown in Figure 9.11(b).

The time complexity of the algorithm can easily be expressed in terms of the number of intervals n and the density of the problem d (the number of rows in the solution). Sorting the set of intervals by their left coordinate can be done in $\mathcal{O}(n \log n)$. The outer loop will be executed d times and at most n intervals from the sorted list will be inspected in the inner loop. This leads to a total worst-case time complexity of $\mathcal{O}(n \log n + dn)$. This complexity can be improved if the algorithm is slightly rewritten (see the Bibliographic Notes at the end of this chapter).

The problem of assigning nonoverlapping intervals to rows can also be described in graph-theoretical terms. A set of intervals defines a so-called *interval graph* $G(V, E)$: for each interval i , there is a vertex $v \in V$ and there is an edge (v_k, v_l) if the corresponding intervals i_k and i_l overlap. Conversely, a graph is an interval graph if it is possible to associate an interval with each vertex such that the intervals associated with adjacent vertices overlap. One should realize that a graph can be

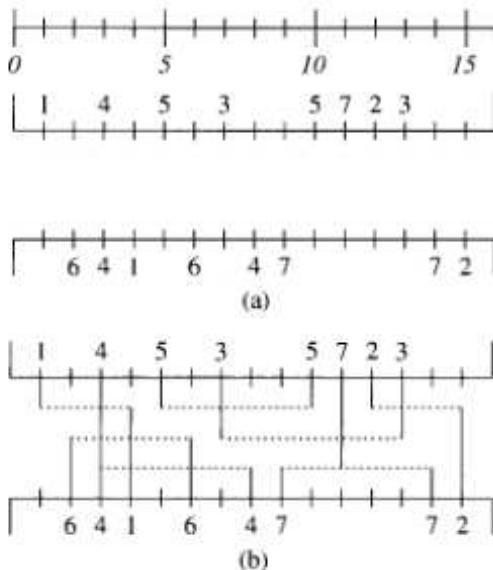


Figure 9.11 An instance of the channel routing problem without vertical constraints (a) and its solution as found by the left-edge algorithm (b).

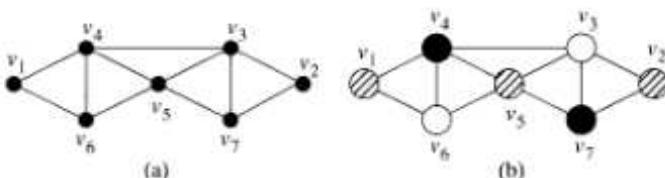


Figure 9.12 The interval graph constructed from the example problem instance in Figure 9.11 (a) and its coloring that corresponds to the solution (b).

constructed from any set of intervals, but that no set of intervals can be found for any graph. The set of interval graphs is therefore a strict subset of the set of all possible undirected graphs. The interval graph corresponding to the example of Figure 9.11 is given in Figure 9.12(a).

The problem of finding the minimum number of rows for the channel routing problem without vertical constraints is equivalent to finding a *vertex coloring* of the corresponding interval graph with a minimal number of colors. The vertex coloring problem for graphs is the problem of assigning a “color” to all vertices of the graph such that adjacent vertices have different colors and a minimal number of distinct colors are used. The problem is NP-complete for graphs in general. However, the existence of the left-edge algorithm clearly shows that the problem can be solved in polynomial time for the category of interval graphs. The coloring that corresponds

to the routing solution of Figure 9.11(b) is given in Figure 9.12(b).

9.3.4 Channel Routing Algorithms

The channel routing problem is NP-complete. As mentioned above, the problem can be solved optimally in polynomial time only in the case that there are no vertical constraints. One has, therefore, to be satisfied with heuristic algorithms in the more general case with vertical constraints.

In the course of time, many algorithms have been proposed for channel routing. Each of them has its strong and weak points. Below, one of these algorithms, viz. the *robust channel router* as published by Yoeli, will be presented in some detail. It has been selected here because it performs well in practice and is relatively easy to understand (there are more algorithms with similar properties).

A description in pseudo-code of the robust channel routing algorithm is given in Figure 9.13. The main loop of the algorithm iterates as often as there are rows in the channel (the number of rows is given by the variable `height`) and removes a group of nets from the problem description in each iteration. One can also say that channel routing problems of decreasing size (stored in the variable `N`) are solved in subsequent iterations. The selected nets will be located on the same row alternatingly either on the top or the bottom of the remaining channel (the top row is chosen when the variable `top` has a nonzero value). Each iteration consists of two parts: the assignment of weights to the nets and the selection of a maximal-weight subset of these nets. These two parts will be discussed below in more detail. It may be that the solution obtained contains vertical constraint violations. The algorithm tries to eliminate these by maze routing. Also this point will receive more attention later on.

The weight w_i of a net i expresses the desirability to assign the net to either the top or bottom row (depending on the value of the variable `top`). The side (top or bottom) that is selected in some point of the iteration will be called the "current side". Yoeli has chosen the following rules to compute the weights:

1. For all nets i whose intervals contain the columns of maximal density, add a large number B to the weights w_i . This stimulates the decrease of the density of the channel routing problem in the next iteration.
2. For each net i that has a current side terminal at the column positions x , add to w_i the local density $d(x)$ for all x . This encourages the selection of nets with terminals at the current side.
3. For each column x for which an assignment of some net i to the current side will create a vertical constraint violation, subtract $Kd(x)$ from w_i ; K is a parameter that should have a value between 5 and 10. This discourages the creation of vertical constraint violations.

Rules like these are typical "rules of thumb" that are characteristic for *heuristics*. It turns out that the performance of the algorithm is relatively insensitive to slight modifications of these rules, hence the name "robust" router.

```

robust_router (struct netlist  $N$ )
{
    set of int row;
    struct solution  $S$ ;
    int total[channel_width + 1], selected_net[channel_width + 1];
    int top, height,  $c$ ,  $r$ ,  $i$ ;

    top  $\leftarrow 1$ ;
    height  $\leftarrow \text{density}(N)$ ;
    for ( $r \leftarrow 1$ ;  $r \leq \text{height}$ ;  $r \leftarrow r + 1$ ) {
        for all "nets  $i$  in netlist  $N$ " {
             $w_i \leftarrow \text{compute\_weight}(N, top)$ ;
            total[0]  $\leftarrow 0$ ;
            for ( $c \leftarrow 1$ ;  $c \leq \text{channel\_width}$ ;  $c \leftarrow c + 1$ ) {
                selected_net[ $c$ ]  $\leftarrow 0$ ;
                total[ $c$ ]  $\leftarrow \text{total}[c - 1]$ ;
                if ("some net  $n$  has a top terminal at position  $c$ ")
                    if ( $w_n + \text{total}[x_{n_{min}} - 1] > \text{total}[c]$ ) {
                        total[ $c$ ]  $\leftarrow w_n + \text{total}[x_{n_{min}} - 1]$ ;
                        selected_net[ $c$ ]  $\leftarrow n$ ;
                    }
                if ("some net  $n$  has a bottom terminal at position  $c$ ")
                    if ( $w_n + \text{total}[x_{n_{min}} - 1] > \text{total}[c]$ ) {
                        total[ $c$ ]  $\leftarrow w_n + \text{total}[x_{n_{min}} - 1]$ ;
                        selected_net[ $c$ ]  $\leftarrow n$ ;
                    }
            }
        }
        row  $\leftarrow \emptyset$ ;
         $c \leftarrow \text{channel\_width}$ ;
        while ( $c > 0$ ) {
            if (selected_net[ $c$ ]) {
                 $n \leftarrow \text{selected\_net}[c]$ ;
                row  $\leftarrow \text{row} \cup \{n\}$ ;
                 $c \leftarrow x_{n_{min}} - 1$ ;
            }
            else
                 $c \leftarrow c - 1$ ;
            solution  $\leftarrow \text{solution} \cup \{\text{row}\}$ ;
            top  $\leftarrow !\text{top}$ ;
             $N \leftarrow "N \text{ without the nets selected in } \text{row}"$ 
        }
        "apply maze routing to eliminate possible vertical constraint violations"
    }
}

```

Figure 9.13 The pseudo-code of the “robust” channel routing algorithm.

Once all nets have received a weight, the robust routing algorithm finds the maximal-weight subset of nets that can be assigned to the same row. Obviously, the nets selected for the subset should not have horizontal constraints. This problem can be formulated in graph-theoretical terms by constructing the interval graph implied by the intervals $[x_{i_{\min}}, x_{i_{\max}}]$ associated with the nets i as was discussed in Section 9.3.3.

For any graph, a set of vertices that does not contain pairs of adjacent vertices is called an *independent set*. The problem of finding the maximal-weight subset of the nets could therefore be formulated as the maximal-weight independent set problem of the corresponding interval graph. However, the maximal-weight independent set problem is NP-complete.

Fortunately, it is not necessary to solve the problem for general graphs here. An efficient algorithm is available for the maximal-weight independent set problem for interval graphs based on *dynamic programming* (see also Section 5.3). A characteristic of dynamic programming is the fact that the optimal solution of some problem instance can efficiently be defined in terms of problem instances of smaller size. In the case of the problem of obtaining the group of nonoverlapping intervals with maximal total weight, the subinstances can be identified by a single parameter γ , with $1 \leq \gamma \leq \text{channel.width}$. To obtain the subinstance with $\gamma = c$, one should remove all intervals that extend beyond column position c from the original set of intervals. Consider e.g. the set of intervals $i_1 = [1, 4]$, $i_2 = [12, 15]$, $i_3 = [7, 13]$, $i_4 = [3, 8]$, $i_5 = [5, 10]$, $i_6 = [2, 6]$ and $i_7 = [9, 14]$. Then the subinstance for $\gamma = 0$ is empty, which is also true for the subinstances $\gamma = 1$, $\gamma = 2$ or $\gamma = 3$. The subinstances for $\gamma = 4$ and $\gamma = 5$ only contain i_1 , while the subinstance for $\gamma = 6$ contains i_1 and i_6 , and so on.

The costs of the optimal solutions for the subinstances with $\gamma = c$ are stored in the array locations `total[c]`. A crucial observation is then that the optimal cost for the subinstance with $\gamma = c$ can be derived from the optimal costs of the subinstances with $\gamma < c$ and the weights of the nets that have their right-most terminals at position c . There are at most two such nets (one with a terminal at the channel's top and the other at the bottom). Consider one of these nets identified with the integer n . Net n may or may not be part of the optimal solution for the subinstance with $\gamma = c$. It is part of the optimal solution if $\text{total}[c - 1] < w_n + \text{total}[x_{n_{\min}} - 1]$. Stated more informally, n is part of the optimal solution if n 's weight added to the optimal solution for the subinstance that did not include any nets that overlapped with n , is larger than the optimal solution for the subinstance with $\gamma = c - 1$. In the pseudo-code of Figure 9.13, such a test is performed twice: once for the net with a right-most terminal at the top and once at the bottom of column c . If a net is selected for some c , the net's identification is stored in the array `selected.net`.

Using the dynamic programming approach just described, it should be obvious that the arrays `total` and `selected.net` can be filled starting with the subinstance with $\gamma = 1$ (subinstances with an empty set of nets obviously have an optimal cost of zero) and moving from left to right in the channel. In order to find which nets are part of the optimal solution, one moves from right to left in the channel and makes

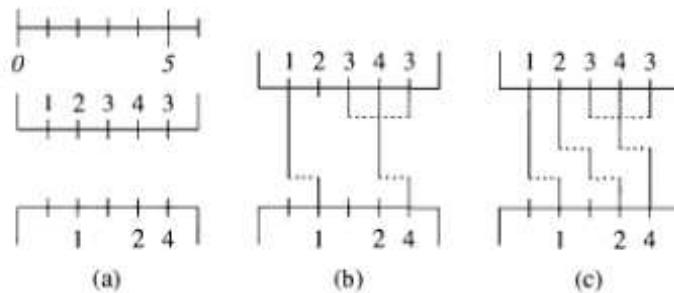


Figure 9.14 An example channel routing problem (a), the result of the robust router after the dynamic programming part (b) and after the rip-up and reroute part (c).

use of the array `selected.net` as described in the pseudo-code of Figure 9.13. Obviously, the right-most selected net is in the solution. Then one must skip any overlapping nets by jumping to the left and take again the right-most selected net, etc.

The selection of maximal-weight subsets of the nets is repeated as often as given by the variable `height` that is normally equal to the channel's density d_{max} . Although the weights are chosen such that vertical constraint violations should be avoided, it is not guaranteed that they will not occur. The robust channel routing algorithm uses a restricted maze routing algorithm to repair these violations by selectively undoing the assignments of nets to rows and rerouting these nets. Such a strategy is often called *rip up and reroute*. This will be illustrated in the example that follows after the next paragraph.

It may happen that applying maze routing in combination with rip up and reroute still does not succeed in finding a solution. In such a case, the variable `height` can be increased by one and a new attempt to solve the problem can be started. There also exist instances of the channel routing problem, like the one shown in Figure 6.12, where a solution can only be found by adding an additional *column* to the channel. This "outer loop" for height and width adjustment of the channel has not been included in the pseudo-code of Figure 9.13.

The rather lengthy description of the robust channel routing algorithm will now be illustrated by applying it to the problem instance with four nets and five columns given in Figure 9.14(a). The local densities are: $d(1) = 1$, $d(2) = 2$, $d(3) = 2$, $d(4) = 3$, and $d(5) = 2$. In the following weight calculations, the two constants in the heuristics mentioned above will be chosen as: $B = 1000$ and $K = 5$. The net weights in the first iteration ($r = 1$) are as follows (the contributions of the three heuristic rules are grouped by parentheses):

$$\begin{aligned} w_1 &= (0) + (1) + (-5 \times 2) = -9 \\ w_2 &= (1000) + (2) + (-5 \times 3) = 987 \\ w_3 &= (1000) + (2+2) + (0) = 1004 \\ w_4 &= (1000) + (3) + (-5 \times 2) = 993 \end{aligned}$$

This results in the following values for the arrays `total` and `selected.net`:

<code>total[1] = 0</code>	<code>selected.net[1] = 0</code>
<code>total[2] = max(0, 0 - 9) = 0</code>	<code>selected.net[2] = 0</code>
<code>total[3] = 0</code>	<code>selected.net[3] = 0</code>
<code>total[4] = max(0, 0 + 987) = 987</code>	<code>selected.net[4] = 2</code>
<code>total[5] = max(987, 0 + 1004, 0 + 993) = 1004</code>	<code>selected.net[5] = 3</code>

Clearly, Net 3 is selected as the only net to occupy the top row.

In the second iteration, the current side switches to the bottom and Net 3 is left out of the new netlist N . In this situation, $d_{max} = 2$ ($d(1) = 1$, $d(2) = 2$, $d(3) = 1$, $d(4) = 2$, and $d(5) = 1$). This leads to the following net weights:

$$\begin{aligned} w_1 &= (1000) + (2) + (0) = 1002 \\ w_2 &= (1000) + (2) + (-5 \times 2) = 992 \\ w_4 &= (1000) + (1) + (-5 \times 2) = 991 \end{aligned}$$

Making use of these, one finds:

<code>total[1] = 0</code>	<code>selected.net[1] = 0</code>
<code>total[2] = max(0, 0 + 1002) = 1002</code>	<code>selected.net[2] = 1</code>
<code>total[3] = 1002</code>	<code>selected.net[3] = 0</code>
<code>total[4] = max(1002, 0 + 992) = 1002</code>	<code>selected.net[4] = 0</code>
<code>total[5] = max(1002, 1002 + 991) = 1993</code>	<code>selected.net[5] = 4</code>

The conclusion after scanning the array `selected.net` from right to left is that Nets 4 and 1 should be put on the bottom row of the channel.

The third and final iteration is trivial: a single net, Net 2, is left for a single row. When combining the solutions of the three iterations, one discovers that the solution is not valid: putting Net 2 in the middle row of the channel would create a vertical constraint violation. This is shown in Figure 9.14(b). This means that maze routing in combination with rip up and reroute has to be applied. Clearly, Net 4 is preventing Net 2 from entering the channel and is ripped up. Net 4 can then be rerouted in the middle row. Now Net 2 can be routed with the maze router which is able to find a solution with a dogleg, as shown in Figure 9.14(c).

9.4 Introduction to Global Routing

As was mentioned at the beginning of this chapter, *global routing* is a design action that precedes local routing and follows placement. After the termination of placement, one will know the cell positions and the location of wiring channels

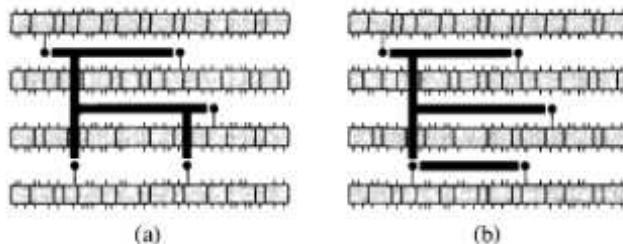


Figure 9.15 Rectilinear Steiner trees to determine a global route in standard-cell layout: a solution with minimal length (a) and one with a minimal number of cell-row crossings (b).

in between them. Global routing decides about the distribution across the available routing channels of the interconnections as specified by a netlist. Then, all required connections can be established by solving the local routing problem in each channel separately. Of course, global routing should contribute to the overall optimization goals, the most relevant of which are area minimization and observance of timing constraints.

In this section some issues that are relevant to global routing will be reviewed in general terms. In the next section, some techniques that are beneficial to the solution of global routing for standard-cell layout will be presented in more detail.

9.4.1 Standard-cell Layout

One of the most popular layout design methods for VLSI is standard cell layout. As was explained in Section 7.3, this type of layout is characterized by rows of cells separated by wiring channels. If all terminals of a net are connected to cells facing the same channel, the entire net can be routed by local routing only. If, on the other hand, the terminals of a net are connected to cells on more than two adjacent rows, the global router should split the net into parts that can be handled each by local routing.

Obtaining a wiring pattern that roughly (i.e. without performing detailed routing) interconnects all terminals of a net, amounts to constructing a *minimum rectilinear Steiner tree* (see Section 4.5). An example for a five-terminal net is shown in Figure 9.15(a). To illustrate that the exact positions within a channel are not known, the terminals have been positioned at the centers of the channels.

The rectilinear Steiner tree contains vertical segments that cross the rows of standard cells. They can be realized in different ways:

- By simply using a wiring layer that is not used by the standard cells.
- By making use of *feedthrough cells*; these are cells that are inserted between functional cells in a row of standard cells with the purpose of realizing vertical connections. As their insertion causes the cells to shift, the best results are

achieved by integrating placement and global routing and fixing the feedthrough-cell positions in this process.

- By making use of *feedthrough wires* that may be available within standard cells. These are wires that simply cross the cell without making a connection to the functional circuitry. They are sometimes incorporated in the cell layout with the purpose of facilitating the routing task.

All three variants will be called "feedthrough wires" in the following text whenever differentiation is not relevant.

In the same way that channel routing will be used to fix the exact wiring patterns in the channels, some kind of detailed routing should be used to fix the exact position of the vertical segments of the Steiner tree. First of all, it may be necessary to slightly shift the segments in order to align with feedthrough wire positions. Second, segments at approximately the same location can be permuted to reduce the densities in the channels above and below the row that they cross.

If feedthrough resources are scarce, their use can be minimized by building a Steiner tree for which vertical connections have a higher cost than horizontal ones. In the example of Figure 9.15(a), such an approach may lead to the alternative solution of Figure 9.15(b).

Another reason why a minimal-length Steiner tree may not be the optimal solution for a single net, is the existence of timing constraints. Connections that are part of a design's *critical path*, the path that determines the system's operation speed, receive special attention in *timing-driven* (or *performance-driven*) layout synthesis, both during placement and routing. Given the fact that longer wires roughly correspond to larger delays, cells connected to *critical nets* (nets that are part of the critical path) will receive a higher priority to be placed close to each other during placement.

It turns out that a more refined delay model than the simple total wire length of a net is required for the purpose of timing-driven global routing. Considering just the total wire length corresponds to a lumped electrical model composed of one resistance and one capacitance. In reality, a long wire in an IC behaves more like a *transmission line* and more accurate models should be used that partition the wire into multiple segments, each segment with its own resistance and capacitance. A widely-used model based on this principle, that shows sufficient accuracy for most purposes, is the *Elmore* delay model. It will not be further discussed here.

Assuming that the signal flow in a net is unidirectional starting from a *source* terminal and propagating to multiple *sink* terminals, signal changes will not arrive simultaneously at all sinks. Under such circumstances, properties of the chosen delay model will influence the Steiner-tree construction algorithms. It may e.g. be necessary to optimize the length of the connection from the source to the *critical sink* (this is a connection that is part of the critical path) rather than the overall tree length.

Of course, constructing Steiner trees alone is not enough to solve the global routing problem. As it is the case in local routing, the many nets in the layout interact. Finding the optimal pattern for one net may prevent another from being laid out

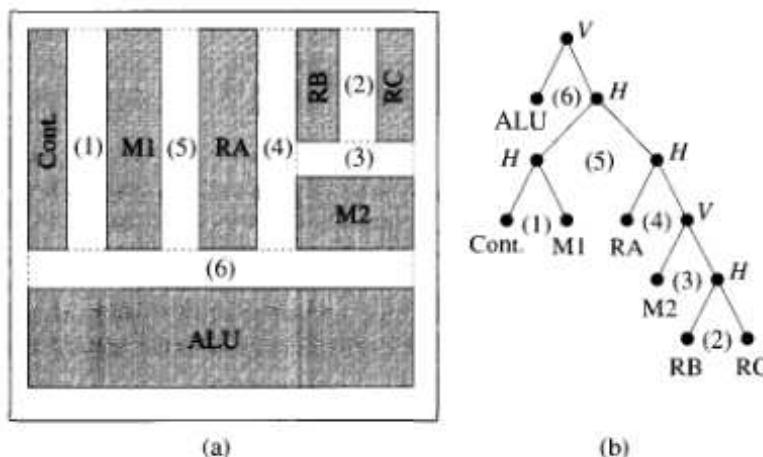


Figure 9.16 Channel definition and ordering for a building-block layout (a) and the corresponding slicing tree (b).

optimally. In one way or the other, the global router should choose the wiring patterns that minimize the overall cost function. This issue is discussed in more detail in Section 9.5.

In standard-cell layout, global routing minimizes the overall area if it minimizes the sum of all channel widths. It is not feasible to solve the channel routing problem for all channels for each tentative solution of global routing. Instead, the width of a channel is often estimated by computing the channel *density*.

9.4.2 Building-block Layout and Channel Ordering

Global routing for *building-block layout* is somewhat more complex than for standard-cell layout as a consequence of a higher degree of irregularity of the layout. One of the issues is the definition of routing channels. Area for routing is reserved around the cells, but it is not always obvious how this area can be partitioned into channels that can be handled by channel routers, and in which order these channels should be routed. These are respectively the *channel definition* and *channel ordering* problems. These problems are quite intricate in general, but almost trivial to solve if a *slicing floorplan* (see Section 8.1.1) of the layout is given. Consider e.g. the floorplan of Figure 8.2(b) and its slicing tree of Figure 8.3. If one assumes that the layout is composed of building blocks rather than flexible cells, one gets the layout of Figure 9.16(a). Figure 9.16(b) shows the corresponding slicing tree.

Both the layout and the tree are annotated with a number between parentheses that indicates a possible correct order for routing the channels. This order can be obtained by a *depth-first traversal* of the tree. This amounts to going down the tree starting from the root in a way similar to the depth-first search function discussed in Section 3.4.1. However, rather than processing a vertex before making the recursive

calls, the vertex is processed after the calls return. Then, a number one higher than the highest number used by the subtrees is assigned to the vertex. In Figure 9.16(b), this number is shown right below the vertex it belongs to, just between the roots of the subtrees.

The same number can be found in Figure 9.16(a) between the (composite or leaf) cells that correspond to the subtrees. The order also defines the channels. In the case of horizontal composition in the slicing tree, the channels are delimited by the top and bottom edges of the two cells involved in the composition. In the case of vertical composition, the left and right edges determine the channel borders. Note that the cells in Figure 9.16(a) always have matching top/bottom or left/right borders; this is not true in general, in which case the cell with the most extreme edges determines the channel size.

All detailed routing for a slicing floorplan can be performed with channel routing only by processing the channels in the right order. In the example of Figure 9.16(a), once Channel (2) has been routed, its floating terminals at its "bottom" side are fixed by the channel router and become fixed terminals for the top side of Channel (3). The floating terminals at the left side of Channel (3), on their turns, receive a fixed position after completing the routing of the channel and become fixed terminals for the right side of Channel (4), etc. If the floorplan is nonslicing, channel routers alone cannot complete all local routing. Switchbox routers or so-called *three-sided channel routers* are necessary for some parts of the routing (see also Exercise 9.5). A three-sided channel is a channel that has floating terminals only on one of the four borders of the rectangular routing area. It is comparable to a switchbox because the routing area cannot be stretched by the router if necessary.

Once the channels have been defined, global routing can take place. Global routing for building-block layout is not much different from the variant for standard-cell layout. However, if no additional wiring layer is available to route on top of the cells, all connections should run along the wiring channels. This means that rectilinear Steiner trees should be constructed that are entirely embedded in the routing area. The structure of the routing area, consisting of channels and the locations where they meet, can be modeled by a graph (an obvious choice is to use vertices for the meeting points and edges for the channels). Instead of looking for Steiner trees in the plane, one then needs to solve the *graph version* of the Steiner tree problem. In this version a subset of the vertices of the graph have to be interconnected by a tree using edges of the graph. Vertices of the graph not included in the subset may act as Steiner points.

9.5 Algorithms for Global Routing

This section presents a possible approach for the global routing of standard-cell layouts. The main ideas are based on the work by Lee and Sechen on sea-of-gates global routing, although the algorithm for rectilinear Steiner trees originates from Griffith *et al.* (see the Bibliographic Notes at the end of this chapter). Sea-of-gates layout has many similarities with standard-cell layout; the main differences are that the number of routing tracks in channels is fixed and that vertical routing channels

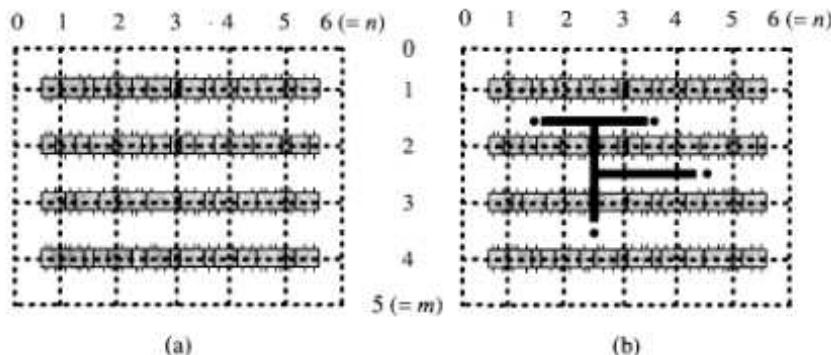


Figure 9.17 The partitioning of the standard-cell layout area by a grid for the sake of global routing (a) and a rectilinear Steiner tree embedded in this grid (b).

can be used instead of feedthrough wires. The ideas are, therefore, presented below in the context of standard-cell routing.

9.5.1 Problem Definition and Discussion

In order to achieve the appropriate degree of abstraction, the layout is covered by a grid. The horizontal grid lines are chosen such that they run across the centers of the cell rows. Vertical grid lines can be chosen such that the horizontal and vertical resolutions are roughly equal. Note that the exact distance between horizontal lines is not known in advance and depends on the results of channel routing. This idea is illustrated in Figure 9.17(a). The grid divides the routing area into elementary rectangles. The abstraction effected by the use of the grid implies that all terminals located in such a rectangle will be thought of as having the same coordinates. The points to be interconnected by rectilinear Steiner trees will then all be considered to lie at the center of these unit rectangles as is shown in Figure 9.17(b). It is, of course, the task of local routing to refine the global solution results by assigning exact positions to all wires.

Suppose that there are $m + 1$ horizontal grid lines numbered $0, 1, \dots, m$ from top to bottom and $n + 1$ vertical grid lines numbered $0, 1, \dots, n$ from left to right. The *local vertical density* $d_v(i, j)$ ($1 \leq i \leq m$; $1 \leq j \leq n - 1$; the border of the layout is left out of consideration) is then defined as the number of wires crossing the vertical grid segment located on vertical grid line j between the horizontal lines $i - 1$ and i . Analogously, the *local horizontal density* $d_h(i, j)$ ($1 \leq i \leq m - 1$; $1 \leq j \leq n$) is defined as the number of wires crossing the horizontal grid line i between the vertical grid lines $j - 1$ and j . The density $D_v(i)$ ($1 \leq i \leq m$) of the channel between grid lines $i - 1$ and i is then given by:

$$D_v(i) = \max_{j=1}^{n-1} d_v(i, j)$$

The goal of global routing is to minimize the *total channel density* given by:

$$\sum_{i=1}^m D_v(i)$$

subject to $d_h(i, j) \leq M_{ij}$ ($1 \leq i \leq m - 1; 1 \leq j \leq n$), where the M_{ij} are the parameters that give the maximum number of feedthroughs that can be accommodated per horizontal grid segment. All the values M_{ij} may be equal and depend on the technology and the properties of the available standard-cell library. They may also be distinct to reflect the number of feedthrough cells assigned by a placement tool (as part of integrated placement and global routing).

There are many ways to solve the problem as defined above. One could follow a sequential approach and process the nets one after the other in a similar way as was explained for local area routing (see Section 9.2). One could e.g. use a variant of Lee's algorithm that increases the segment weights crossed by a wire before routing the next segment. Such an approach has the same disadvantages that maze routing has for local routing (the dependency on net ordering, the wrong choices made when there are many equivalent solutions, etc.).

One could sequentially construct rectilinear Steiner trees for all nets. This would require a Steiner tree algorithm that is able to avoid congested areas, e.g. by using some weighted distance where the weight is derived from the connections that have already been established. One could, however, construct Steiner trees for all nets independently, examine the result for congested areas and try to modify the shapes of those trees that are the cause of overcongestion, or those trees that contribute to the reduction of the total channel density after reshaping. The algorithms presented here follow such an approach.

Instead of using the same grid during the complete routing process, one could start with a very coarse grid, say a 2×2 grid, perform global routing on this grid by assuming that all terminals covered by an elementary rectangle are located at the rectangle's center, and construct Steiner trees that evenly distribute the wires crossing the grid segments. One then gets four smaller routing problems that can be solved recursively following the same approach. The recursion stops when a sufficient degree of detail has been reached for handing the problem over to a local router. One can also continue and directly solve the local routing problem in this way. Such a *divide-and-conquer* algorithm has the advantage of limited problem size at each level of recursion. On the other hand, the subproblems to be solved are not completely independent. The decision on the ordering of wires crossing a boundary for one subproblem will constrain the search space of the neighboring one. Such a *hierarchical routing* approach may be especially attractive when it is combined with min-cut placement (see Section 7.4.1) and the floorplan-based design methodology (see Chapter 8).

Below, an algorithm for the construction of Steiner trees is presented that can operate in the (nonhierarchical) grid for the global-routing of standard cells as presented above. The algorithm constructs a Steiner tree based on a rectilinear

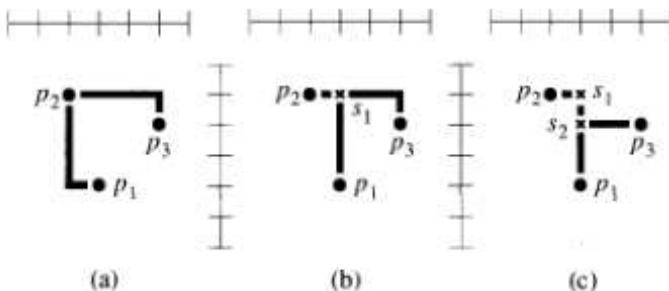


Figure 9.18 The interconnection of three points based on the minimal spanning tree (a), the introduction of a Steiner point after flipping and merging (b), and the final Steiner tree (c).

distance metric. Timing issues that may influence the tree shape are not taken into consideration. The trees are independently constructed for each net. The interaction of nets is discussed later on.

9.5.2 Efficient Rectilinear Steiner-tree Construction

The input of the *rectilinear Steiner-tree problem* is a set of n points P located in the two-dimensional plane: $P = \{p_1, p_2, \dots, p_n\}$. The rectilinear distance between a pair of points $p_i = (x_i, y_i)$ and $p_j = (x_j, y_j)$ is equal to $|x_i - x_j| + |y_i - y_j|$. The goal of the problem is to find a minimal-length *tree* that interconnects all points in P and makes use of new points in the plane not included in P if these new points contribute to the tree-length reduction. The set of new points will be denoted by S .

It was already mentioned in Section 4.5 that the rectilinear Steiner tree problem is NP-complete, but that the optimal tree can be approximated by a spanning tree that can e.g. be computed in polynomial time by Prim's algorithm. It is known that the length of the tree approximated in such a way is at most a factor of $\frac{3}{2}$ longer than the minimum Steiner tree. This fact has inspired many researchers to design heuristics that improve a given minimum spanning tree. One can e.g. exploit the two possible L-shaped connections between two points that are not located on the same horizontal or vertical line and choose for those shapes that allow the merging of wire segments. Consider e.g. the situation shown in Figure 9.18(a)¹ that shows a spanning tree for the three points p_1 , p_2 and p_3 . "Flipping" the L-shaped connection between p_1 and p_2 leads to the merging with the L-shaped connection between p_2 and p_3 and the creation of Steiner point s_1 as is shown in Figure 9.18(b). A similar "flipping" of the newly created connection between s_1 and p_3 creates a second Steiner point s_2 as shown in Figure 9.18(c). In the new situation, s_1 is no longer a Steiner point: the connections (p_1, s_1) and (s_1, s_2) can be replaced by the direct connection (p_1, s_2)

¹ For the discussion of rectilinear Steiner trees it is more convenient to draw the net terminals on the crossings of the grid lines rather than at the centers of the elementary grid rectangles as was done in the problem definition; clearly, this does not affect the validity of the presented theory.

```

(set of struct vertex, set of struct edge) steiner(set of struct vertex  $P$ )
{
    set of struct vertex  $T$ ;
    set of struct edge  $E, F$ ;
    int gain;

     $E \leftarrow \text{prim}(P)$ ;
     $(T, F, \text{gain}) \leftarrow \text{l-steiner}(P, E)$ ;
    while ( $\text{gain} > 0$ ) {
         $P \leftarrow T$ ;
         $E \leftarrow F$ ;
         $(T, F, \text{gain}) \leftarrow \text{l-steiner}(P, E)$ ;
    }
    return  $(P, E)$ ;
}

```

Figure 9.19 The iterated 1-Steiner heuristic for Steiner-tree construction.

without affecting the tree length. Stated more generally, s_1 is no longer a Steiner point because its *degree* is two (it has two incident edges). This phenomenon of a point acting only temporarily as a Steiner point also occurs in the algorithm explained below.

The algorithm to be presented here also makes use of a spanning-tree construction algorithm, but does this in a different way than the heuristic approach sketched above. It is guaranteed that the tree lengths of the solutions produced by it are strictly less than $\frac{3}{2}$ times the optimal length. This means that the algorithm outperforms a spanning-tree algorithm in solution quality for all cases when a spanning-tree algorithm would have worst-case performance.

In order to explain the algorithm, first the *1-Steiner tree problem* should be defined. This is the problem of finding a spanning tree in the set of points $P \cup \{s\}$ where the point s is chosen such that the length of the resulting spanning tree is minimal. One may say that this is a version of the general minimal Steiner-tree problem with the restriction $|S| = 1$. As will be explained below, efficient algorithms exist to find such a point s . The actual Steiner-tree heuristic amounts to repetitively solving the 1-Steiner problem. The pseudo-code of this procedure is shown in Figure 9.19. The algorithm makes use of graph-theoretical data structures: a point is represented by a *vertex* and a connection between two points by an *edge*. Apart from the information required for graph-theoretical computations, the structure *vertex* should, of course, contain information on the coordinates of its location for the purpose of distance computation.

The function *prim* computes the minimal spanning tree using Prim's algorithm (see Section 3.4.4) and is supposed to return the edge set that represents the tree. The function *l-steiner* takes the vertex and edge sets of a spanning tree as input and returns three values corresponding to the vertex and edge sets of the constructed 1-Steiner tree, and the decrease in tree-length that was the result of adding one Steiner

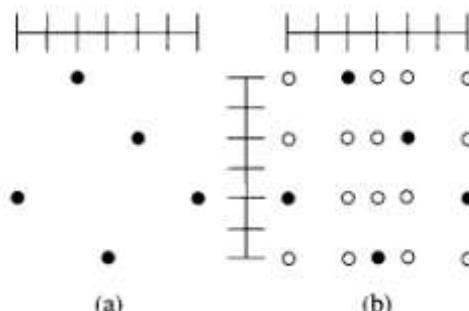


Figure 9.20 A set of points P for which the Steiner-tree should be constructed (a) and its Hanan points (b).

point. If no Steiner point could be found that improves the existing tree length, the third value is zero. This situation indicates to the calling environment that its main loop can terminate.

The principle for solving the 1-Steiner problem is quite simple: all candidate points s are visited and the spanning tree for the points in $P \cup \{s\}$ is computed each time. The point that leads to the cheapest tree is then selected. Two issues are important here. The first is to identify the set of candidate points and the second is to find a method for the spanning-tree construction that is more efficient than calling the function `prim` each time a new s is considered in the set $P \cup \{s\}$.

The first issue was settled by Hanan in 1966. He proved that an optimal rectilinear Steiner tree can always be embedded in the grid composed of only those grid lines that carry points of the set P . So, all grid lines on which no point of P is located can be left out of consideration. The candidate points are commonly called *Hanan points*. Figure 9.20(a) shows an example set of points P (the same set as in Figure 4.5(a)) and Figure 9.20(b) the Hanan points associated with the problem instance (white circles).

The best-known solution for the second issue was proposed by Griffith *et al.* It involves the *incremental* computation in linear time of the minimum spanning tree for the set $P \cup \{s\}$ given the minimum spanning tree for the set P . This is done by the function `spanning.update` in the pseudo-code of the function `1-steiner` shown in Figure 9.21. The code of the function `spanning.update` itself is shown in Figure 9.22.

The linear-time update algorithm is based on the following principles that are presented without proof. Restricting all points in a spanning tree to have at most degree four does not prevent the tree from having minimal length. The four points to which point s may be connected are the closest ones in each of the four regions obtained by partitioning the plane by two lines crossing s at angles of $+45$ and -45 degrees. In the pseudo-code, these four regions are called *north*, *east*, *south* and *west*, while the closest point to s from a point set V (excluding s itself) in a region r is computed by the function `closest.point`.

```

(set of struct vertex, set of struct edge, int)
  1-steiner(set of struct vertex V, set of struct edge E)
  {
    set of struct vertex W;
    set of struct edge F;
    struct vertex maxpoint;
    int gain, maxgain;

    maxgain ← 0;
    for each s ∈ “Hannan points of V” {
      (W, F, gain) ← spanning_update(V, E, s);
      if (gain > maxgain) {
        maxgain ← gain;
        maxpoint ← s;
      }
    }
    if (maxgain > 0) {
      (W, F, gain) ← spanning_update(V, E, s);
      return (W, F, maxgain);
    }
    else return (V, E, 0);
  }
}

```

Figure 9.21 The pseudo-code of the 1-Steiner algorithm.

```

(set of struct vertex, set of struct edge, int)
  spanning_update(set of struct vertex V, set of struct edge E, struct vertex s)
  {
    int delta;
    struct vertex u, v, w;

    delta ← 0;
    V ← V ∪ {s};
    for each d ∈ {north, east, south, west} {
      u ← closest_point(V, s, d);
      delta ← delta - distance(s, u);
      E ← E ∪ {(s, u)};
      if (cycle(V, E)) {
        (v, w) ← largest_cycle_segment(V, E);
        E ← E \ {(v, w)};
        delta ← delta + distance(v, w);
      }
    }
    return (V, E, delta);
  }
}

```

Figure 9.22 The function that incrementally computes a spanning tree when a new point is added to the original point set.

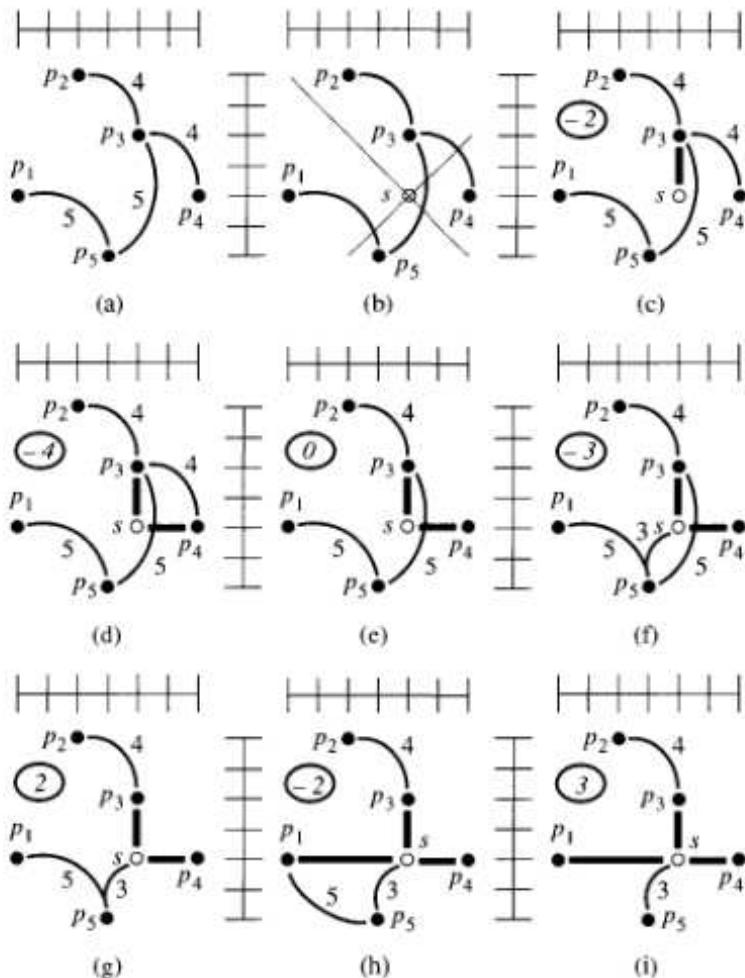


Figure 9.23 The different steps in the incremental computation of a spanning tree.

Figure 9.23 illustrates the functions `1-steiner` and `spanning_update`. The initial spanning tree for the point set P is shown in Figure 9.23(a). The connections between the points are shown as arcs rather than rectilinear segments to emphasize that no decision has been taken on the actual path that the connection follows (such as either of the L-shaped patterns). The length of the connection is shown by the label next to the arc. Figure 9.23(b) shows the new point s for which the spanning tree $P \cup \{s\}$ should be computed incrementally and the two lines that divide the area into four regions. Following the pseudo-code of Figure 9.22 the closest point to s in the north, east, south and west regions are p_3 , p_4 , p_5 and p_1 respectively.

The spanning tree for $P \cup \{s\}$ is constructed by connecting the point s with the

four points in each region. If the connection gives rise to a cycle (this condition is checked by the function `cycle` in the pseudo-code), the cycle should be identified and its largest segment removed. The identification is done by the function `largest_cycle_segment`. By definition of a tree, a cycle will be created for all connections except the first one. Figures 9.23(c)–(i) show how the algorithm progresses. In all figures, straight line segments are used whenever a connection between points on the same grid line is made (they are guaranteed to have the unique shortest pattern) and arcs otherwise. The value of the variable `delta` that indicates how much shorter the tree has become is shown in an ellipse at the upper left side (the function `distance` obviously computes the rectilinear distance between two points).

It is not difficult to see that the function `update_spanning` operates in linear time. Finding the four closest points can simply be done by visiting all points and saving the closest ones in each direction. The same is true for cycle detection (use depth-first search and signal detection of a cycle when coming back to a vertex that was visited earlier) and the identification of the longest segment in a cycle (keep track from where you arrive during depth-first search in order to trace the cycle). Given the fact that the number of Hanan points is $\mathcal{O}(n^2)$, the worst-time complexity of the function `1-steiner` becomes $\mathcal{O}(n^3)$.

Because the function `1-steiner` will be called at most $\mathcal{O}(n^2)$ times, the time complexity of the main function `steiner` can be stated to be $\mathcal{O}(n^5)$. This is, however, a very pessimistic estimation because it is known from the theory that a point set with n points has at most $n - 2$ Steiner points (this result follows from the fact that a Steiner point has at least degree three while the number of edges in the tree is bounded by the fact that a tree cannot have cycles by definition). On the other hand, points that have been selected as Steiner points may later become obsolete as was the case in Figure 9.18. A significant speedup can be obtained for large point sets using a *batched* version of the function `steiner`. This version takes advantage of the fact that the function `1-steiner` visits all candidate Hanan points and constructs a spanning tree for each point. It accepts a batch of incremental updates rather than a single one. This is allowed if the incremental updates are independent. Practical experiments show that the number of calls of `1-steiner` is a small constant (with an average value of about 2 for $n = 40$) that hardly grows with the problem size which justifies the claim that the overall average time complexity for constructing Steiner trees in this way is $\mathcal{O}(n^3)$.

Coming back to the example problem of Figure 9.20(a), it turns out that point s in Figure 9.23 is actually the optimal solution of the 1-Steiner tree problem (exercise: check this). In a second iteration, a second Steiner point can be found, after which no further improvement is possible. The final solution, which is optimal, is shown in Figure 9.24. The solution differs from the one of Figure 4.5(b) that is also an optimal solution for the same problem instance. In fact, it happens very often that a minimum rectilinear Steiner tree problem instance has many distinct optimal solutions. It may also happen that solutions exist with Steiner points that are not Hanan points. The next section shows how these facts can be exploited.

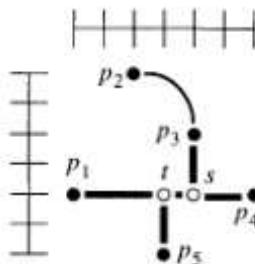


Figure 9.24 The solution found by the iterated 1-Steiner heuristic for the problem instance of Figure 9.20.

9.5.3 Local Transformations for Global Routing

Once Steiner trees for all nets have been generated independently, congested areas in the grid introduced in Section 9.5.1 can be identified. The trees of the nets contributing to the congestion can be reshaped by applying local transformations. Examples of such transformations are shown in Figure 9.25. Figure 9.25(a) shows the flip of an L-shaped connection discussed earlier. The more refined transformation of Figure 9.25(b) allows any Z-shaped connection between two points that are not on the same grid line. The transformation of Figure 9.25(c) replaces straight-line connections by U-shaped connections. Clearly, such a transformation increases the tree length but may decrease the global cost function because it reduces congestion. Figure 9.25(d) finally shows a much more sophisticated “segment shift” transformation which cannot be reversed with a single segment shift.

Algorithms for global routing may choose to use any of the types of transformation just mentioned. These transformations may be controlled by a general purpose optimization strategy such as local search or simulated annealing.

If the chosen optimization strategy fails to find a solution, maze routing may be used as a last resort. The nets crossing overcongested grid segments are then ripped up and rerouted using an appropriate variant of Lee’s algorithm.

9.6 Bibliographic Notes

The textbooks [Len90], [She93], [Sai95] and [Sar96], that deal with physical design automation in general, pay quite some attention to both global and local routing. A book that is dedicated to multilayer routing is [She95].

The original publication of Lee on the path connection algorithm is [Lee61]. Overviews of work that elaborate further on this algorithm concerning generalizations and techniques to make it more efficient can be found in [Ake72], [Rub74] and [Oht86]. The importance of Lee’s algorithm for routing can also be seen from the fact that special hardware has been constructed to implement it (see e.g. [Hon83] and [Suz86]).

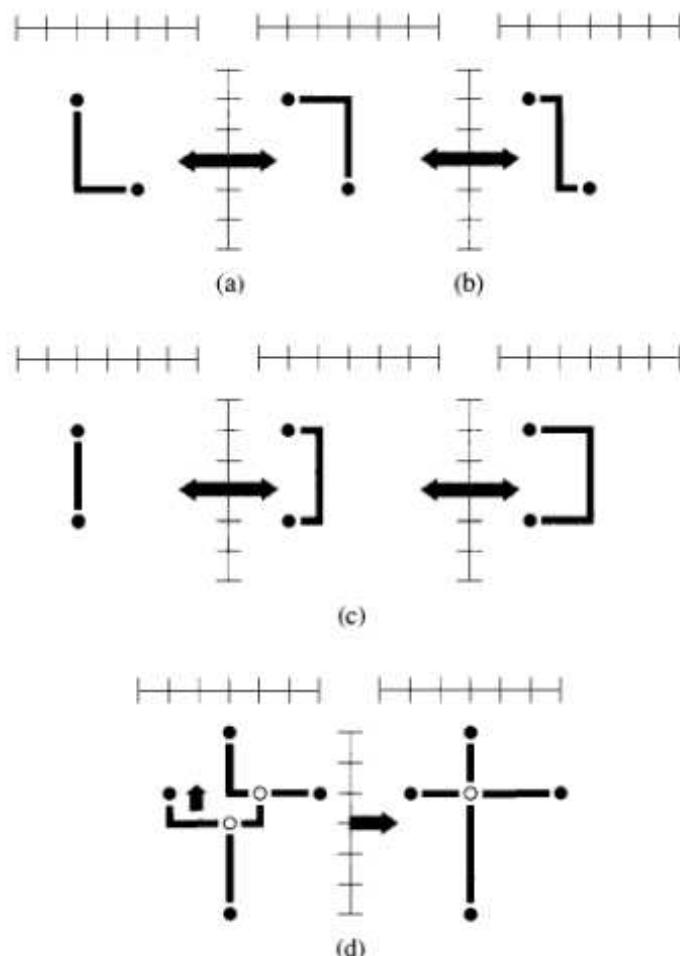


Figure 9.25 Local transformations for Steiner trees.

An example of a problem that cannot be solved by Lee's algorithm irrespective of the net ordering is given in [Oht86]. Examples of "iterative improvement" style algorithms that make use of the Lee's algorithm as a subroutine are described in [Shi87] and [Lin89].

There exist review papers on channel routing by Burstein [Bur86] and on switch-box routing by Marek-Sadowska [MS92]. A famous early paper on channel routing for printed circuit board layout is [Has71]. One of the pioneering papers on channel routing that makes use of the "classical model" described in Section 9.3.1 is [Ker73]; it proposes a branch-and-bound style algorithm. An example of a gridless channel router is described in [Che86] and an example of a router using 45° segments in [Enb87]. Examples of channel routers that can deal with more than two

wiring layers are presented in [Bra88] and [Con88]. Examples of algorithms that use the nonreserved-layer model are [Joo86] and [Shi87]. The use of "doglegs" was introduced by Deutsch [Deu76].

The left-edge algorithm was originally presented in [Has71]. The paper also contains a proof that the algorithm always finds an optimal solution (with a number of rows equal to the density). The NP-completeness of graph vertex coloring for general graphs is mentioned in [Gar79]. The time complexity of the left-edge algorithm can be improved by sorting both endpoints of all intervals in increasing order and putting them in a list. The assignment to rows of the intervals can then be accomplished in a single scan of this list [Gup79]. The sorting step determines the overall time complexity in such a case, which means that the time complexity becomes $\Theta(n \log n)$ or even $\Theta(n)$ if *radix sort* (see e.g. [Aho74] or [Sed88]) is used.

The NP-completeness of channel routing is proved in [Szy85]. An example of an algorithm that uses a combination of the left-edge algorithm and maze routing is described in [Ree85]. A description of Yoeli's "robust router" can be found in [Yoe91]. The NP-completeness of the maximal-weight independent set problem follows from the NP-completeness of the maximal independent set problem which is mentioned in [Gar79]. The latter is a special case of the first in which all vertex weights are equal to one.

A review paper on global routing is [Kuh86]. It was mentioned in the text that a local reordering of feedthrough wires can affect the densities in neighboring channels. Optimization algorithms for this problem are discussed in [Che97].

A key paper on algorithms for rectilinear Steiner trees in the context of timing-driven layout is [Boe95]. It gives a justification of the Elmore delay model based on appropriate experiments and then presents Steiner-tree heuristics that directly optimize the tree cost for the Elmore delay. Some other sources for timing-driven layout are [Kuh91], [Sap93] and [Kah95].

A detailed study of the channel definition and ordering problem can be found in [Cai89].

The two papers that are the basis for Section 9.5 are [Lee91b] (for the problem formulation and the local transformations) and [Gri94] (for the Steiner-tree construction).

The hierarchical routing approach was originally proposed by Burstein for local routing [Bur83b, Bur83c] and for simultaneous placement and routing [Bur83a]. Generalizations of these ideas have been elaborated in [Hac89].

At the end of Chapter 4 a number of references on rectilinear Steiner trees were already provided. Hanan's paper in which he shows that the Steiner points in a minimum rectilinear Steiner tree always can be chosen to have a coordinate in common with the original point set, is [Han66]. The result that a minimum Steiner tree (not only a rectilinear one) for a set of n points has at most $n - 2$ Steiner points is proved in [Gil68].

Recently, new approximation algorithms have been proposed that guarantee a solution that has a tree length which is at most $\frac{11}{8}$ times longer than the optimal one. One can consult [Föß97] and its reference list for more information on this topic.

As mentioned in the main text of this chapter, many algorithms have been published for the transformation of a rectilinear spanning tree into a rectilinear Steiner tree. Polynomial-time algorithms that find the optimal Steiner tree under the restriction that a spanning tree has been taken as a starting point are presented in [Ho90].

The idea to construct a rectilinear Steiner tree based on a heuristic that repetitively solves the 1-Steiner problem is presented in [Kah92b]. This paper also proves that the heuristic deviates at most by a factor of $\frac{4}{3}$ from the optimum length for all cases that a spanning-tree would be exactly $\frac{3}{2}$ times longer. The contribution of [Gri94] is the improvement in the time complexity for the 1-Steiner problem leading to the algorithm that has been presented in the text. It contains the proofs of many statements made in this text without proof.

The local transformations for global routing mentioned in [Lee91b] are quite similar to those proposed in [Ger89] for local routing.

9.7 Exercises

- 9.1** It was mentioned in Section 9.2 that Lee's algorithm can be generalized for nets with more than two terminals. Indicate how the code of Figure 9.2 should be modified for this goal. Invent an example problem with three terminals for which the solution found by the modified Lee algorithm is suboptimal.
- 9.2** Lee's algorithm and Dijkstra's algorithm discussed in Section 3.4.3 both compute shortest paths. Show how Lee's algorithm can be interpreted to be a version of Dijkstra's algorithm.
- 9.3** Consider the following intervals: [1,3], [4,8], [2,5], [10,12], [11,12], [7,9], [7,11], and [3,8]. What is the density of this set of intervals? Use the left-edge algorithm (see Figure 9.10) to group these intervals in rows, such that intervals on the same row do not overlap. Draw also the interval graph corresponding to this set of intervals and give the vertex coloring of this graph corresponding to the solution obtained by the left-edge algorithm.
- 9.4** Show that the left-edge algorithm (see Figure 9.10) produces a solution with a number of rows exactly equal to the density of the problem. Hint: show that in each iteration of the `while` loop the density of the subproblem formed by the intervals remaining in the list `i.list` decreases by one.
- 9.5** Consider a building-block layout with a "wheel" floorplan (see Figure 8.4) and with routing regions around each cell. Partition the routing region in such a way that the area to be routed by conventional channel routers is maximized and identify the areas that should be handled by a switchbox and/or three-sided-channel router.

10

Simulation

Simulation of VLSI circuits and systems is a very broad topic and complete books could be written to discuss all the different aspects involved. As the goal of this book is to cover a wide range of VLSI design automation tools, the treatment of the topic has to be limited to a few aspects. Therefore, after a general introduction, attention is paid to gate-level and switch-level simulation only.

Obviously, simulation is used for the purpose of *design verification* (see Chapters 1 and 2). As it is extremely costly to repair design errors after the fabrication of an integrated circuit, one needs to have sufficient confidence that all design errors have been eliminated before delivering the design to the foundry. As mentioned in Section 2.5, there are mainly two ways to verify a design: by simulation and by formal verification. The first possibility is the topic of this chapter.

10.1 General Remarks on VLSI Simulation

Simulation involves the construction of a computer model of the hardware that is being designed and executing the model to analyze its behavior. This should be done at the correct *level of abstraction* (see Chapter 1). On one hand, all issues relevant to the abstraction level in question should be part of the model in order for the simulation to be valuable. On the other hand, all issues that are not part of the abstraction level should be left out such that the computation time required for simulation remains reasonable.

In VLSI, a *signal* always belongs to the *electrical* domain by the nature of the realization medium. What differs, depending on the level of abstraction, is the interpretation of the signal. If all values in the *continuous* range of voltages are relevant, one considers the signal to belong to the *analog* domain. If the range of voltages is partitioned into *discrete* ranges, one operates in the *digital* domain.

The most important abstraction levels for which specific simulation tools have been developed are listed below, starting from the lowest level:

- *Device-level simulation* generally involves a single semiconductor device (e.g. a MOS transistor) of which a detailed model is used in order to follow aspects like the evolution in time of charge distributions in three dimensions as a function of

material parameters. Simulator techniques based on e.g. finite-element methods are used for this purpose. This abstraction level belongs to the domain of the technologists and a VLSI design engineer does not need to deal with simulations at this level.

- *Circuit-level simulation* deals with small groups of transistors modeled in the analog domain: the circuit is seen as an interconnection of resistors, capacitors and (controlled) voltage and current sources. The variables computed are currents and voltages. The computations involved are based on numerical methods.
- *Timing-level and macro-level simulation* also deal with signals in the analog domain. However, the models have been simplified in order to be able to deal with larger circuits at the expense of some loss in accuracy. Simplification can e.g. be achieved by replacing nonlinear characteristics of devices by piecewise-linear equations. Another possibility is to represent the behavior of a group of devices by a functional equivalent, also called a *macro*. A macro gives the relations between input and output variables without the individual descriptions of the constituent devices necessarily being distinguished in the functional description.
- *Switch-level simulation* models MOS transistors as switches that pass signals which are more or less digital in nature: the values of the signals are discrete, but the model includes features that can be associated to analog notions as resistance and capacitance. This topic is covered in more detail in Section 10.3.
- *Gate-level or logic-level simulation* distinguishes itself from the switch level in the fact that signal flow is unidirectional (from the input to the output of a gate, such as a NAND or XOR gate), whereas the signal flow at the switch level is bidirectional. Also, instead of transistors, a complete gate is now the basic element of which circuits are composed. A more detailed discussion of the topic follows in Section 10.2.
- *Register-transfer-level (RTL) simulation* is used in *synchronous* circuits where all registers are controlled by a system clock signal. One can say that the registers store the state of the system, while *combinational logic* computes the next state and the output based on the current state and the input. At this level, one is mainly interested in state transitions and the precise timing of intermediate signals in the computation of the next state is disregarded.
- *System-level simulation* deals with hardware described in terms of primitives that do not necessarily correspond with hardware building blocks. An example of a popular hardware description language that can be used for system-level simulation is VHDL. When used in the initial stages of a design, it can describe the *behavior* of a circuit as a "process" or as a set of communicating processes. Note, by the way, that VHDL is a language that can be used for description and simulation from the system level down to the gate level.

As indicated above, descriptions of hardware can cross several abstraction levels.

Simulators that operate on more than one level are quite desirable, as they allow the designer to simultaneously simulate low-level descriptions of critical parts of a design with higher-level descriptions of the less critical parts or parts whose descriptions are not yet available in detail. Simulators that handle different abstraction levels, especially in the analog and digital domains, are called *mixed-level* simulators. If the simulator uses different mechanisms for the different levels (e.g. a time-driven numerical integration technique and the event-driven technique to be discussed later) it is called a *mixed-mode* simulator.

Another issue that is becoming more and more popular for the design of digital systems is *hardware-software co-simulation*. The solution for a considerable number of digital design problems contains programmable hardware. It may e.g. turn out that a set of dedicated processors is the best solution for some problem. In such a case, it is desirable to compare a system-level description, in which no decisions on hardware and software partitioning have yet been made, with a combined description of the network of processors and the programs run by them.

The software modules that generally constitute a simulator consist of the following:

- *The simulator kernel.* This is the part that performs the actual simulation (it computes how signals propagate through a circuit).
- *The processing of the input description.* A circuit to be simulated has to be specified in some way. This can be done textually by means of some *hardware description language* or the simulator can accept a description provided by some *schematic entry tool* (see Chapter 2). Normally, a simulator has an *internal format* that is well-suited to be processed by the simulator kernel. The external description should be translated to the internal description as a preparatory step for simulation.
- *The processing of the stimuli.* The circuit or system that has to be simulated will generally require input signals from the “outside world”. These are called *stimuli* and describe the subsequent values that inputs have throughout the simulation period. Simulators normally allow the specification of the stimuli either interactively or by means of a special language.
- *The presentation of the results.* Running a simulator results in the computation of changes in signal values during the simulation period. There are many ways to present them to the user: tables of time-value pairs, value-time plots or various types of animation.

The rest of this chapter will mainly deal with the simulator kernel, as it is the most essential part of a simulator.

10.2 Gate-level Modeling and Simulation

This section discusses a number of important issues related to gate-level simulation. Attention is paid to *signal modeling* (the correspondence between voltage and current

values in the real circuit and the discrete signals in the model), *gate modeling* (the representation of a gate's behavior), *delay modeling* (the various types of delays encountered at the gate level), *the connectivity modeling* (the representation of the interconnections between the gates) and *the simulation mechanisms*.

10.2.1 Signal Modeling

Digital circuits are designed to process *binary* or *Boolean signals* to be denoted here by '0' and '1'. On the other hand, the realization of the circuit carries voltages and currents with continuous values that often have "high" or "low" levels, but sometimes have values in between these levels, notably when a transition occurs. For this reason, in some simulators one or more values are used to represent signals in transition. A signal that is neither '0' or '1', because the signal is undergoing a transition from one of these values to the other could be modeled with the value 'X', representing the *unknown* value. Another issue that is important in digital simulation is the detection of an *uninitialized* memory. Often a value 'U' is used for this purpose. So, the memory bits receive the value 'U' when the simulation starts. Improperly designed hardware will lead to the propagation of 'U' through the combinational logic. Still another point is the *strength* of a signal, reflecting the fact that a signal can become weaker after passing through a transistor. A signal can become weaker due to a transistor's resistance. The result can be that more time is needed for a transition, because a parasitic capacitance is charged with a weaker current. In Section 10.3 the *signal strength* is presented as a different attribute than the *signal level* (high, low, unknown, etc.) and a pair of these attributes forms the signal value. In some simulators, however, the combination of the strength and level is expressed as a single value.

Clearly, the modeling requirements impose signals to have multiple values, a different set of values depending on what one wants to model. Simulators offering up to 99 values have been reported. More sophisticated simulators, like simulators for VHDL, allow the user to define his or her own data types, which means that, in principle, he or she can work with the signal values that are most appropriate for the circuit being designed.

The more values are used for a signal, the more complex becomes the modeling of a gate's behavior. Obviously, the gate model should specify the output value for each possible combination of input values. If the gate has k input signals, with each signal having one of N values, the output for N^k combinations should be specified.

The logic involved in dealing with a circuit modeled using multiple-valued discrete signals is called *multiple-valued logic*. It has operators that map a set of multiple-valued signals to another set of multiple-valued signals. These operators can be used to model the behavior of a gate as explained in the next section.

in_1	in_2	out
'0'	'0'	'1'
'0'	'1'	'1'
'0'	'X'	'1'
'1'	'0'	'1'
'1'	'1'	'0'
'1'	'X'	'X'
'X'	'0'	'1'
'X'	'1'	'X'
'X'	'X'	'X'

Table 10.1 A truth-table modeling of the behavior of a two-input NAND gate.

10.2.2 Gate Modeling

Given the fact that a simulator is supposed to compute the propagation and change of signals in a circuit of interconnected gates, it should be able to model the behavior of a single gate in the first place.

The model should be such that signal values at the gate's outputs are efficiently computed as a function of the gate's inputs. There are several possibilities:

- A *truth table* representation. The table has a row for each combination of input values, as is illustrated in Table 10.1 for the case of a two-input NAND gate, assuming that signals have one of the three values '0', '1' or 'X', where 'X' means an unknown signal value.
- A *subroutine* representation. An efficient evaluation is obtained in the case that signals have only binary values by using the hardware instructions of the computer on which the simulation is running (most instruction sets have single instructions for many operations performed by Boolean gates). In the case of multiple-valued logic, similar techniques can be employed, using an appropriate combination of machine instructions.

10.2.3 Delay Modeling

The passage of time is an essential part of a simulation. At the gate level, time is modeled in a discrete way and all delays in the circuit are expressed as an integer multiple of some time unit.

The output of any physical gate will take some time to switch after the moment that an input has switched. The delays involved can affect the correct functioning of the circuit, especially when the circuit has asynchronous parts. Therefore, an accurate modeling of the delays is important. The most important models are listed here:

- *The propagation delay model.* This model associates a fixed delay with the gate's output. So any effect of switching inputs is seen at the output after this fixed delay.

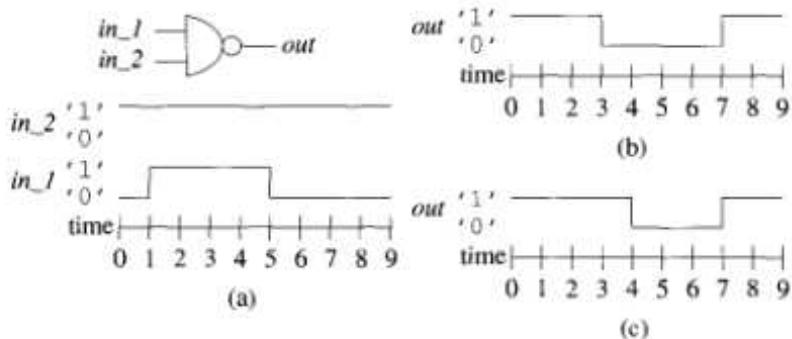


Figure 10.1 A 2-input NAND gate and its switching input signals (a); the corresponding output signal in the propagation delay model (b); the corresponding output signal in the rise/fall delay model (c).

Special cases of the model occur when the delay is equal to zero or one, resulting in the *zero-delay model* and the *unit-delay model* respectively. Figure 10.1(a) shows a 2-input NAND gate and its switching input signals; the corresponding output assuming a propagation delay of 2 time units is shown in Figure 10.1(b). A more refined model replaces the constant delay by a delay that depends on the number n of gates that an output drives (one also says: "the output's *fanout* equals n "). The delay can be expressed as $a + nb$, where a is a constant delay and b is a delay associated with the parasitic capacitance of the next gate's input and the wiring leading to it. Another refinement consists of introducing distinct delays between each input-output pair of a gate.

- *The rise/fall delay model.* In this model, different delays are used when an output signal rises and when it falls. The refinements mentioned for the propagation delay model can also be used here. Figure 10.1(c), shows the signal behavior of the output of the 2-input NAND gate when the delay for a falling signal is 3 time units and the delay for a rising signal is 2 time units.
- *The inertial delay model.* Inertial delay models the property observed in physical circuits that an input pulse (negative or positive) should have a minimal width in order to have any effect at the output. This can be explained by the fact that capacitances in a gate have to be charged before the gate's outputs can switch. If no sufficient energy is provided by the input pulse, no switching takes place. Inertial delay can be combined with both the propagation and the rise/fall delay models.

10.2.4 Connectivity Modeling

In order to compute the propagation of signals through the gates in the circuit, the simulator should have a suitable data structure to represent the connectivity of all

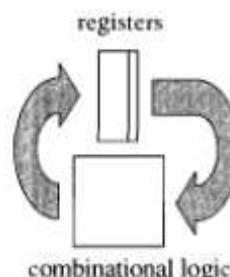


Figure 10.2 The core of a synchronous circuit.

gates in the network. Data structures for the representation of a circuit have already been discussed in Section 7.1. They can easily be extended for the incorporation of attributes typically required for simulation, such as the current signal value of a net, delays in a gate, etc.

10.2.5 Compiler-driven Simulation

In the previous section, different modeling issues related to gate-level simulation were discussed. Some of these, and especially the delay models, have direct consequences for the simulation mechanism, the method by which the evolution of the signals in the circuit are computed by the simulator kernel. Two basic mechanisms to simulate a circuit at the the gate level, viz. the *compiler-driven* and the *event-driven* methods, will be discussed in this and the next sections respectively.

The most obvious situation in which compiler-driven simulation is the best choice occurs in the context of *synchronous circuits*. The core of such a circuit consists of registers that store the *state* of the system and combinational logic that computes the next state as is shown in Figure 10.2. If one simply wants to verify that the patterns generated by the combinational logic are correct without caring about the delays occurring in the combinational part, one can neglect the delays in the gates and set them to zero. This results in the *zero-delay model* that was mentioned in Section 10.2.3. An efficient way to compute the signal propagation is to generate machine code that reflects the behavior of the combinational logic and then execute it.

A simple example circuit that could constitute the combinational part of a synchronous circuit is given in Figure 10.3. It will be used to illustrate the concepts of compiler-driven simulation. The circuit consists of the five inputs A to E and one output F. The inputs provide the signals to the five nets n_1 to n_5 . These signals determine the signals generated in the other nets n_6 to n_9 of the circuit.

The first step in code generation is *leveling*, which is the process of determining the order in which the signals carried by a net will be computed. Obviously, one should only compute the output of a gate in the circuit if its inputs are known. Using a method that is essentially the same as the longest-path algorithm discussed

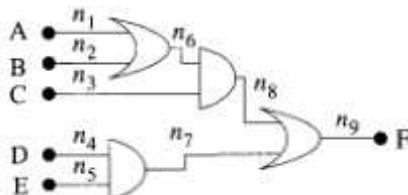


Figure 10.3 A simple circuit composed of logic gates.

```

 $n_1 \leftarrow A;$ 
 $n_2 \leftarrow B;$ 
 $n_3 \leftarrow C;$ 
 $n_4 \leftarrow D;$ 
 $n_5 \leftarrow E;$ 
 $n_6 \leftarrow \text{OR}(n_1, n_2);$ 
 $n_7 \leftarrow \text{AND}(n_4, n_5);$ 
 $n_8 \leftarrow \text{AND}(n_6, n_3);$ 
 $n_9 \leftarrow \text{OR}(n_7, n_8);$ 
F  $\leftarrow n_9;$ 

```

Figure 10.4 Code for the zero-delay simulation of the circuit of Figure 10.3.

in Section 6.4.1, one assigns *level 0* to all nets connected to inputs. The net connected to an output of a gate receives a level one higher than the maximum level of the gate's input nets. In the case of the example of Figure 10.3, n_1 to n_5 have Level 0, n_6 and n_7 have Level 1, n_8 has Level 2, and n_9 has Level 3. The level number is then used as a sorting criterion for code generation (lowest level first) as shown in Figure 10.4. Although the code is presented here as "high-level" pseudo-code, the idea is that this code will be translated to machine instructions. Depending on the instruction format of the target machine and on the type of signal carried by the nets (Boolean or multiple-valued) each line of pseudo-code will require a single or a few instructions. The symbols **OR** and **AND** represent the logic operation in the chosen value system. Note that they should not be interpreted as function calls but rather as *macro* calls (they are supposed to be replaced by *inline* code). Functions calls are expensive in terms of the number of machine instructions required and should be avoided in compiler-driven simulation. The first five lines of code seem to be redundant: they have, however, been included to indicate that the input values are retrieved through the interface of the simulator kernel with the "stimuli processing module". In the same way, the last line refers to the interface with the result presentation module.

A model that is slightly more realistic than the zero-delay model is the *unit-delay model* in which signals take one unit of time to propagate from the inputs to the outputs of any gate. In this way, the evolution in time of signal values can be followed. This is especially important for the detection of *glitches*, temporary changes in circuit outputs resulting from different delay paths. Figure 10.5 shows some signal

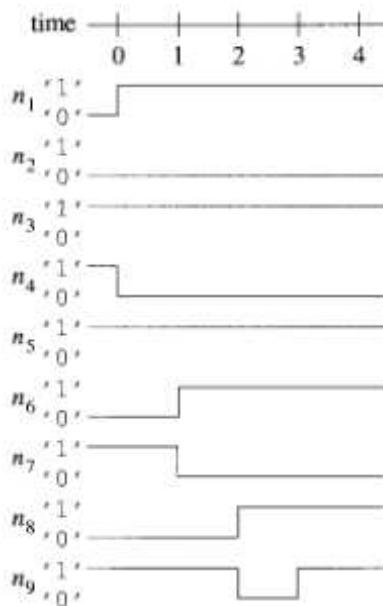


Figure 10.5 The appearing of a glitch on net n_9 in the circuit of Figure 10.3 when using unit-delay simulation.

patterns for the unit-delay simulation in the example circuit of Figure 10.3. At the inputs, the signals on nets n_1 and n_4 change simultaneously at time $t = 0$. As a result n_9 first falls from '1' to '0' at $t = 2$ and then rises again to '1' at time $t = 3$. If the zero-delay model had been used, n_9 would have always remained at '1'.

It is possible to generate code intended for compiled execution for the unit-delay model as well. In the most straightforward approach, this can be done by introducing separate variables for the signals carried by a net for each time instant that the simulation covers. For the example circuit of Figure 10.3, this would mean that variables $n_{i,t}$ have to be used in the code instead of variables n_i , with i ranging over all net numbers and t ranging over the time instants. The aspect of unit-delay is then expressed in statements that compute a gate's output at time t in terms of the gate's inputs at time $t - 1$, like in " $n_{6,2} \leftarrow \text{OR}(n_{1,1}, n_{2,1})$ ".

The above approach is, of course, naive in the sense that much more memory than necessary is used as the code never refers from a time t to an instant earlier than $t - 1$. So, a better solution would be to only reserve storage space for the signal values at times t and $t - 1$, for example by having two arrays, each with as many locations as there are nets in the circuit. This is illustrated in Figure 10.6, where the two arrays have been called `new` and `old`. The code has been embedded in a loop to be able to show that after each iteration the contents of `new` are copied to `old` (in practice it is more efficient just to interchange the pointers to the memory locations reserved for the two arrays). Similarly to the code presented in Figure 10.4, the code contains

```

for ( $t \leftarrow t_{start}; t \leq t_{end}; t \leftarrow t + 1$ ) {
    new[1]  $\leftarrow A$ ;
    new[2]  $\leftarrow B$ ;
    new[3]  $\leftarrow C$ ;
    new[4]  $\leftarrow D$ ;
    new[5]  $\leftarrow E$ ;
    new[6]  $\leftarrow OR(old[1], old[2])$ ;
    new[7]  $\leftarrow AND(old[4], old[5])$ ;
    new[8]  $\leftarrow AND(old[6], old[3])$ ;
    new[9]  $\leftarrow OR(old[7], old[8])$ ;
    F  $\leftarrow new[9]$ ;
    old  $\leftarrow new$ ;
}

```

Figure 10.6 Pseudo-code meant for the compiler-driven simulation of the circuit of Figure 10.3 using the unit-delay model.

statements for the interface with the stimuli processing and the result presentation modules of the simulator.

Clearly, the concept just presented can be generalized to delay models where each delay is a small multiple of unity by using arrays that store signal values at $t - 2, t - 3$, etc. The number of arrays required is equal to one plus the maximum delay present in the circuit. The fact that the signals involved can be represented with either a single *bit* when the Boolean signal model is used, or a few bits in the case of multiple-valued signals, can be exploited by using the same memory *word* to store a signal's value at time $t, t - 1$, etc. The assignment of *new* to *old* can then be realized by a "bit shift" instruction.

10.2.6 Event-driven Simulation

The *event-driven* method is motivated by the fact that normally very few gates are switching simultaneously and that computing signal propagation through all gates in the network over and over again at each time instant, as in compiler-driven simulation, amounts to many unnecessary calculations. It seems to be more economical to only recompute those signals that are actually changing. A signal change is called an *event* which explains the name "event-driven simulation". However, the economy argument is not completely valid as the computational overhead related to keeping track of all events and propagating signals in the correct order can become quite high, sometimes as high as 100 times the time that compiler-driven simulation needs for a single signal propagation. The main reason why event-driven simulation is very popular is that different delay models can be incorporated in the simulator relatively easily.

The central data structure for event-driven simulation is the *event queue* or *event list*. A pseudo-code description of this data structure and its associated functions are given in Figure 10.7. The first structure described is *event*. An event should have at

```
struct event {
    int time;
    struct net *node;
    struct signal_value value;
    ...
};

struct event_queue {
    ...
};

struct event_queue *new_queue();
{
    "create an empty event queue";
}

struct event *first_event(struct event_queue *queue)
{
    "remove the earliest event from an event queue and return it";
}

insert_event(struct event_queue *queue, struct event *new_event)
{
    "add a given event to an event queue";
}
```

Figure 10.7 The data structures and functions associated with an event queue.

least three attributes: the time at which the event is supposed to happen, the circuit net that will change value at that time and the new value that this net will assume then. The `event_queue` has to store events in one way or the other while optimizing the following actions:

- returning the earliest of the events still to be processed and removing it from the queue,
- adding a new event at an arbitrary time in the queue.

One can say that the `event_queue` data structure and the functions `new_queue`, `first_event` and `insert_event` form an *abstract data type*. This means that the actual implementation of an event queue is not very relevant as long as the three functions are available to the user of the data type.

A commonly made assumption is that there exists a minimum unit of time Δt and that all delays occurring in the circuit can be expressed as an integer multiple $k\Delta t$ of Δt . In addition, most of these multiples are assumed to be small integers. Under these conditions, an array indexed by the time seems to provide an efficient implementation. The array should be organized in such a way that each entry contains

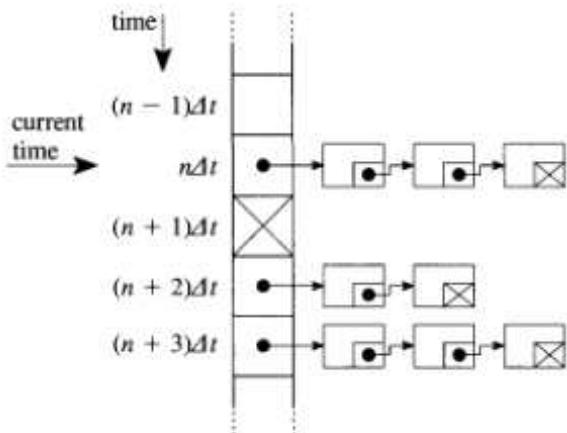


Figure 10.8 The array implementation of the event queue.

a linked list of events that occur simultaneously at the time instant corresponding to the index. This idea is illustrated in Figure 10.8. This data structure can both return the earliest event and add new events in constant time (independently of the number of events already stored). Of course, the solution given has one huge disadvantage: it requires that the array is as big as the number of time steps that the simulation should take.

This disadvantage can be overcome by reusing the array locations that would otherwise remain unused once the current time had become larger than their indices. This can be accomplished by using an index $t \bmod L$ instead of t , where L is the total number of array locations available. The fact that the modulo operator is used in addressing the array can be visualized by a cyclic arrangement of the subsequent array locations as shown in Figure 10.9. Such a data structure is called a *time wheel*. Unfortunately, a time wheel has a disadvantage as well: because of the fixed number of locations L , events that have to be inserted at more than L positions away from the current time cannot be stored in the time wheel. These events have to be stored in an *overflow list*. From time to time (at least each time that the current time has increased by L) the events in the overflow list have to be inspected and moved to the appropriate entry in the time wheel. More sophisticated solutions are possible: e.g. by organizing the overflow list as a time wheel itself. In such a case, this second time wheel has a resolution of $L\Delta t$ instead of Δt . Each time that all L locations of the first time wheel have been processed, all events stored at a single location of the second wheel can be moved to the first time wheel. Of course, the second time wheel will now need an overflow list.

The assumptions that made a time-wheel implementation efficient do not always hold. Then, so-called *priority queues* can be considered instead of the time wheel for the realization of the event queue. A priority queue can deal with events spanning any time scale (which means that time instances are not limited to small integer multiples

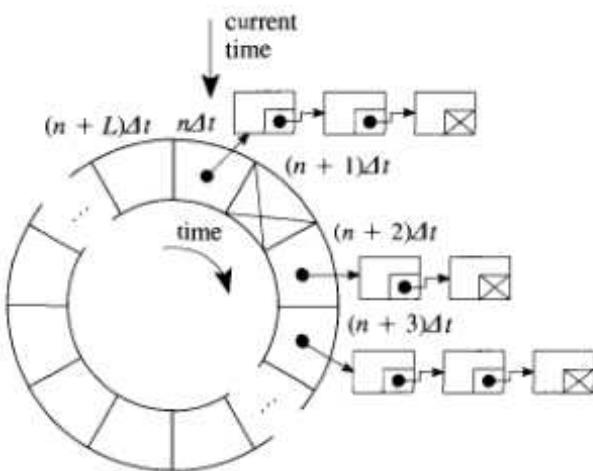


Figure 10.9 The time-wheel implementation of the event queue.

```

event_driven_simulation ()
{
    struct event_queue *Q;
    Q ← new_queue();
    "insert stimuli in Q";
    "initialize: all network nodes connected to a memory to 'U' and
    all other nodes to 'X'";
    for ( $t \leftarrow t_{start}; t < t_{end}\right)$  {
        current_event ← first_event(Q);
         $t \leftarrow \text{current\_event->time}$ ;
        "process current_event and add new events to Q at
        time  $t + \text{appropriate delay}$ ";
    }
}

```

Figure 10.10 Pseudo-code of the event-driven simulation function.

of Δt). Removing and adding events costs, however, $\mathcal{O}(\log n)$ time, where n is the number of events present in the event queue, while they cost $\mathcal{O}(1)$ time for the time wheel (see the Bibliographic Notes at the end of this chapter for more information on priority queues).

The event queue is only one part of event-driven simulation. The other part is the simulation algorithm itself that adds and retrieves events from the queue. The pseudo-code of such an algorithm is given in Figure 10.10. First, all events derived from the stimuli are added to the event queue. This statement is included in the pseudo-code just to indicate that stimuli give rise to events. In a practical implementation it is probably more efficient to create these events gradually in the

main loop rather than processing all of them before entering the main loop. The simulation in the main loop starts at a time t_{start} and continues until a time t_{end} performing a simulation in a time interval provided by the user. In the main loop, the earliest event in the event queue is fetched, the signal change associated with the event is processed (and possibly stored for future use when the results of the simulation have to be presented) and new events caused by the current event are created and stored in the event queue at the appropriate time.

The two basic mechanisms that are widely used for gate-level simulation, viz. *compiler-driven* and *event-driven* simulation, have been presented here. The choice for either of these techniques depends on the user requirements: compiler-driven simulation is quite fast but can only deal with limited delay models, whereas event-driven simulation can deal with very general delay models at the expense of more computer time (due to the overhead of the event queue processing). In the case that the delay model required is supported by both mechanisms, the circuit activity should determine the choice as compiler-driven simulation recalculates many or all signals for each time instant, whereas event-driven simulation only recomputes those signals that are actually changing.

Apart from refinements of the compiler-driven and event-driven mechanisms, more advanced simulators also use combinations of both techniques. Compile-time analysis can e.g. supply some information on when signal changes can be expected. This information can be used to prepare the data structures as much as possible in advance, such that inserting events to the event queue and removing them later can be performed more efficiently.

A completely different approach is *demand-driven simulation*. Instead of propagating stimuli forward through the circuit, it starts with the signals that the user wants to observe in some time interval and goes backwards towards the circuit inputs. Each time the algorithm traverses a gate from output to input, the boundaries of the time interval are appropriately adjusted. When an input is reached, only the stimuli for the relevant interval are processed and propagated. The demand-driven technique performs even fewer gate evaluations than event-driven simulation; in event-driven simulation, not all signal changes resulting from the stimuli may be interesting for the user. On the other hand, the method has e.g. the disadvantage that it cannot deal well with circularities in the circuit.

10.3 Switch-level Modeling and Simulation

The switch level is a level between the circuit level, where signals are analog, and the gate level, where signals with discrete values are propagated through unidirectional elements. At the switch level, signals are discrete, but signal flow is bidirectional due to the "switches" that model the transistors. Transistors are the only electrical components of which a circuit at the switch level is composed except for resistances and capacitances that may also be included in the circuit model for a better approximation of real circuit behavior.

The main reason why switch-level simulation is interesting is that the models used are more accurate than in gate-level simulation, while the computational effort required is far smaller than in circuit-level simulation. Obviously, appropriate techniques are required for performing simulations at this level. They are discussed here in two parts dedicated respectively to modeling issues and simulation mechanisms.

10.3.1 Connectivity and Signal Modeling

The modeling of connectivity at the switch level can be done by the general "cell-port-net" model introduced in Section 7.1. The only restriction is that all the cells are either switches, inputs or outputs.

Switch models were already used in the 1940's for relay circuits and received new attention in the late 1970's with the wide use of the MOS transistor. Two main contributors to this field have been Bryant and Hayes who independently proposed similar switch-level models. The MOS transistor mainly acts as a switch: the value of the *gate* signal determines whether or not the source and the drain are electrically connected (see also Appendix A). Besides the aspect of switching, the switch-level model also takes into account capacitances and resistances in the circuit. However, this is normally not done by means of linear differential equations known from circuit theory. Capacitances and resistances enter the model indirectly as *strengths*.

A *signal* is represented by a pair (s, v) , where:

- s is a *strength* (one could think of the impedance of the voltage source providing the signal), and
- v is a *level*, a voltage with discrete values at least including '1', '0' and 'X', where 'X' represents the *unknown* signal value.

Nets (or *nodes*¹) are divided into two groups:

- *storage nets*, that are able to store charge; they have a strength (capacitance value) which is discrete.
- *input nets*, that carry a signal with a fixed level and the highest-possible strength (they can provide the network with an arbitrary current; think of an ideal voltage source); an input net is always connected to an input (cell), which is often a power supply terminal.

Transistors also have a strength, a discrete value related to their conductance.

In Bryant's model, all strength values s are integers in the range $1, 2, \dots, k, \dots, w$, with the following subdivision:

- $s = w$: s is the strength of an input signal;

¹ The term *node* is mainly used in circuit theory and simulation, whereas the term *net* is more common in the field of layout. As both refer to the same notion, it has been chosen to always use *net* in this chapter.

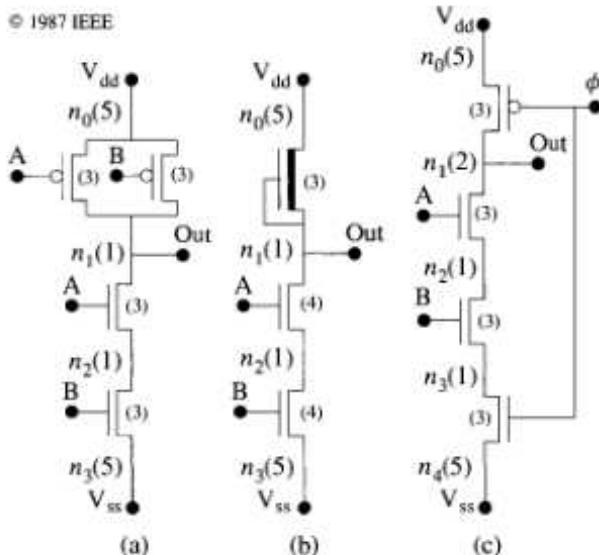


Figure 10.11 Switch-level modeling with 5 strength values: a static CMOS NAND gate (a); an nMOS NAND gate (b); and a NAND gate in CMOS domino logic (c).

- $k < s < w$: s is the strength of a transistor;
- $1 \leq s \leq k$: s is the strength of a storage net.

A transistor's strength s should be interpreted as a maximum strength that can pass through the transistor. So, signals with a strength larger than s are reduced to strength s , while signals with strength less than or equal to s maintain their strength when passing through a transistor. The strength s of a storage net has a similar function: when all transistors connected to the net are switched off, the strength of the signal remaining in the net is at most s .

The model will now be explained by means of examples of indexNAND gate-NAND gates in different logic circuit styles as shown in Figure 10.11. The strength of a net is shown between parentheses behind the net name and the strength of a transistor is given between parentheses next to the transistor symbol. It turns out that with $w = 5$ and $k = 2$, three different logic circuit styles can be modeled effectively: static CMOS, nMOS and domino CMOS. They are discussed separately below.

Static CMOS is the logic circuit style that consists of two complementary series-parallel networks as discussed in Appendix A. The term "static" refers to the fact that the gate behavior is independent of any clocking strategy: as long as the gate inputs are stable, the gate output will be stable as well. Only a single strength value for the transistors (3) and a single strength value for the storage nets (1) is sufficient for the adequate modeling of static CMOS as can e.g. be seen in the NAND gate presented in Figure 10.11(a). This is a direct consequence of the fact that for any combination

of input signals, there is either a conducting path from V_{dd} to the output or one from V_{ss} (but not both). This means that the output value is either (3, '1') or (3, '0') (the strength of the input nets is always reduced to 3, the only transistor strength in the circuit).

nMOS technology is a technology that historically preceded *CMOS technology*. As the name already suggests, this technology is based on n-channel MOS transistors (nMOSes) only. Two types of nMOSes are used in nMOS technology: the *enhancement* nMOS, which behaves exactly in the same way as the nMOS used in CMOS technology (see Appendix A) and the *depletion* nMOS that basically behaves as a resistance (the transistor is fabricated in such a way that its source and drain are always electrically connected). The nMOS NAND gate of Figure 10.11(b) functions as follows. When either of the signals A or B is '0', the output should have level '1', which is achieved because of the existence of a conducting path through the depletion transistor. When both A and B have value '1', the depletion transistor continues to conduct, but its resistance is higher than the resistance of the two enhancement nMOSes in series, which means that the output gets level '0'. The fact that the depletion transistor has a higher resistance (a lower conductance) than the enhancement transistors implies for the switch-level model that it has a lower strength (3) than the strength values of the enhancement transistors (4). So, when both inputs have level '1', the output value is (4, '0'); when either of the inputs have level '0', the output value is (3, '1').

The domino logic CMOS NAND gate of Figure 10.11(c) functions as follows: when the clock ϕ is low, net n_1 , feeding the gate's output, is *precharged*, i.e. charge flows from the power supply to n_1 via the pMOS transistor; when ϕ is high, charge stored on net n_1 should flow to V_{ss} only when both signals A and B have level '1' (NAND function). A correct functioning requires that n_1 can store considerably more charge than net n_2 , otherwise, when A has level '1' and B level '0', charge could flow from n_1 to n_2 , degrading the gate's output level unintentionally. This effect is called *charge sharing*. Charge sharing is avoided in the model by giving net n_1 a strength of (2) and n_2 a strength of (1). The output of this gate is either (3, '0') or (2, '1').

10.3.2 Simulation Mechanisms

The main issue in switch-level simulation is, of course, the simulation algorithm itself. Normally, the circuit is not simulated as a whole at the switch level. It is first partitioned into subcircuits that only have unidirectional communication with each other. Unidirectionality occurs when a signal drives the gate of a transistor. The exchange of signals between these subcircuits can be simulated using an event-driven mechanism. Therefore, the discussion in this section concentrates on the simulation of a single subcircuit after paying some attention on how the subcircuits can be obtained from the large circuit. Figure 10.12 illustrates how a network of transistors can be partitioned.

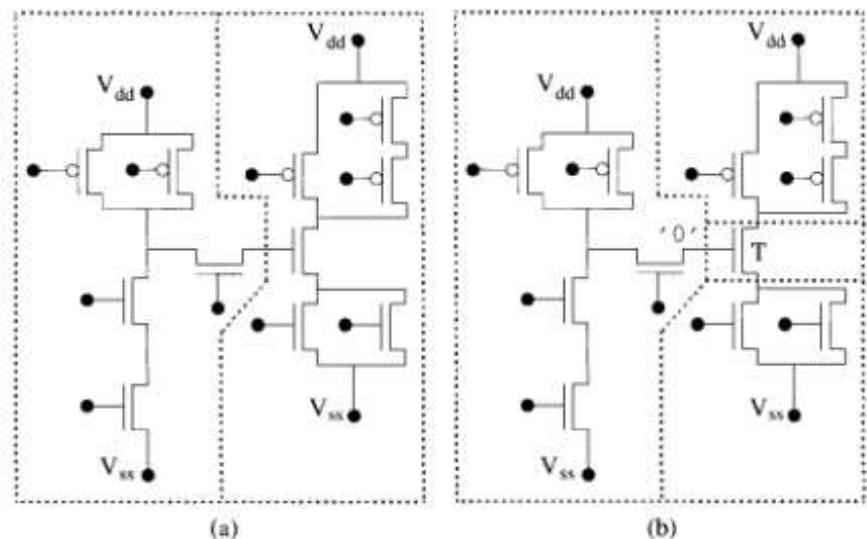


Figure 10.12 The static (a) and dynamic (b) partitioning of a CMOS network into unidirectional subcircuits.

Figure 10.12(a) shows a so-called *static* partitioning in which connections to the gate of a transistor determine subcircuit boundaries irrespective of the signals carried by the nets. *Dynamic* partitioning, on the other hand, takes signal values into account which can result in a further partitioning of the subcircuits. In Figure 10.12(b) for example, the fact that transistor T is switched off due to the presence of a signal '0' on its gate, means that the upper and lower part of the right subcircuit are electrically isolated and can therefore be evaluated separately by a simulator. Dynamic partitioning has the disadvantage that the partitioning of the circuit has to be repetitively recomputed as the signals change. The advantage is that the circuits that have to be simulated at the switch-level are smaller.

From a computational point of view, the static partitioning a circuit of transistors is quite straightforward (see also Exercise 10.1). Assuming that the "cell-port-net" model introduced in Section 7.1 is used for the circuit representation and that a list of all transistors is available, one can simply traverse the circuit starting from an arbitrary transistor checking the nets to which its *source* and *drain* ports are connected. Using a procedure similar to depth-first search, as discussed in Section 3.4.1, one can recursively visit those transistors whose *source* and *drain* ports are also connected to these nets. The search does not visit transistors of which the *gate* port is connected to the net. The search also stops when hitting an input net, as these nets *block* the propagation of signals arriving through a transistor. All transistors that have been visited as the result of one original call to the procedure, end up in one partition, sometimes called a *channel-connected component*. Transistors from the list that have not been yet visited, should be used

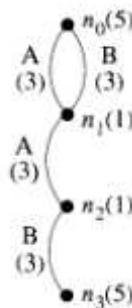


Figure 10.13 The representation of the static CMOS NAND gate of Figure 10.11(a) using the multigraph model.

as the starting point for the search for new partitions. All this can be done in time that linearly depends from the number of transistors and nets in the circuit. Looking for dynamic partitions can be done in a similar way except that transistors that are switched off should not be traversed by the search.

Instead of using the “cell-port-net” model, it is possible to use a simpler representation model for channel-connected components because of the specific circuit structure of such a component. The cell-port-net model had three disjoint vertex sets in order to express that a cell has many ports and that nets are connected to many ports as well. In a channel-connected component, only the *source* and *drain* ports of a transistor participate in the bidirectional signal flow, which means that the *gate* port does not need to be represented explicitly. This has the consequence that all transistors (the only “cells” in the circuit) would be represented by a vertex with exactly two outgoing edges (to the source and drain ports). Then, no information is lost if the vertex representing a transistor is removed and the transistor is represented by a single *edge* connecting the two ports. On the other hand, because net vertices are either connected to source or drain ports of different transistors and the two ports are functionally equivalent, again no relevant information is lost by merging all port vertices connected to a net vertex with the net vertex. What remains is a graph model with a single type of vertex for the nets and one type of edge for the transistors: the edge runs between the net connected respectively to the source and gate ports of the transistor. The resulting graph is a *multigraph*, a graph in which parallel edges are allowed (see also Section 3.1). Figure 10.13 shows the multigraph representation of the static CMOS NAND gate given in Figure 10.11(a). The figure shows, for each vertex, the net name corresponding to it together with the net strength and for each edge the transistor strength of the corresponding transistor together with the input signal controlling the transistor gate. The multigraph model will be used in the rest of this section to present the algorithms used in switch-level simulation.

Given a multigraph $G(V, E)$, the signal present on vertex $u \in V$ can be denoted by (σ_u, λ_u) , where σ_u is the signal’s strength and λ_u is the signal’s level. For an edge $(u, v) \in E$, $\epsilon_{u,v}$ has as its value the strength of the transistor corresponding to the

edge if the transistor is conducting. If the transistor is nonconducting, the value of $\epsilon_{u,v}$ is zero. In the discussion below, two effects that are mutually dependent, will be considered:

- the signal flow along an edge $(u, v) \in E$ given the signals present on the vertices u and v ;
- the signal present on a vertex $v \in V$ given all signals flowing along the edges $(u, v) \in E$.

Let $\sigma_{u \rightarrow v}$ denote the "strength of the signal flowing from vertex u to vertex v through edge $(u, v) \in E$ ". The semantics of the switch-level model are such that:

$$\sigma_{u \rightarrow v} = \min(\sigma_u, \epsilon_{u,v}) \quad (10.1)$$

So, as stated earlier, a transistor limits the strength of a signal passing through it to its own strength. The level of such a signal remains equal to λ_u . Note that signals passing through a nonconducting transistor will have a strength zero, which actually means that no signal at all passes the transistor. The value zero was not mentioned as a valid strength value in Section 10.3.1 because it is not a property of a circuit component. It is only introduced to facilitate the description of the simulation algorithms. Another remark to be made is that if the signal at v is stronger than the signal at u , no signal actually flows from u to v , but a signal flows from v to u . However, pretending that signals always flow both from u to v and from v to u along an edge $(u, v) \in E$ only facilitates the description of the simulation algorithms below without affecting its results.

In a switch-level network the signals present on the input nets will flow through the conducting transistors to determine the signals in other nets. It may be that some nets do not have a path through conducting transistors to any input net. The signals on these nets are determined by the charge initially present on them (this charge may be redistributed as a consequence of the strength of the storage nets). So, it is useful to subdivide the set of storage nets into two groups: *driven* nets of which the signal values are determined by a conducting path from input nets, and *charged* nets that do not have such a path. Consider e.g. net n_2 of the domino logic NAND gate of Figure 10.11(c): the net is driven when signal ϕ is low and it is charged when ϕ is high while either of the signals A or B are low.

Suppose that a driven net $v \in V$ has edges $(u_1, v), \dots, (u_m, v) \in E$, then the strength σ_v of the resulting signal will obey:

$$\sigma_v = \max_{1 \leq i \leq m} \sigma_{u_i \rightarrow v} \quad (10.2)$$

The level of the resulting signal will be equal to the level of the maximal strength signal if there is only one maximal strength signal or if all maximal strength signals have the same level. In the case of multiple maximal strength signals with distinct levels, the resulting level becomes 'X'.

```

struct signal {
    int strength;
    voltage level;
    ...
};

struct vertex {
    set of struct edge edges;
    int strength; /* net strength */
    struct signal state;
    ...
};

struct edge {
    struct vertex *to, *from;
    int strength; /* transistor strength */
    int on; /* equals zero when transistor is off */
    ...
};

```

Figure 10.14 The data structures to be used for the description of a switch-level simulation algorithm.

The strength and the level of a charged net are determined in a similar way. The only difference is that the strength of the net itself may turn out to be the maximal one. For a charged net Equation (10.2) has to be modified into:

$$\sigma_v = \max(\sigma_v, \max_{1 \leq i \leq m} \sigma_{u_i \rightarrow v}) \quad (10.3)$$

Now that the mechanisms governing the propagation of signals at the switch level have been explained, an algorithm can be presented to compute the signal values for each storage net given the signals on the input nets. The circuit is represented by the multigraph model. Figure 10.14 shows possible data structures for the algorithm. They contain separate structures for vertices and edges called `vertex` and `edge` (although an adjacency-list representation could also be used). Also, a structure `signal` is used to represent signals as a pair of `strength` and `level` in accordance with the theory presented here. The data type `voltage` that has not been declared is the enumeration of the three possible level values '0', '1' and 'X'. The `state` attribute for a vertex stores intermediate signals as computed by the algorithm and will eventually contain the final signal value for the vertex.

Of course, the algorithm needs to incorporate in some way the two effects mentioned above: signal propagation through the edges of the graph (the transistors of the network) and the combination mechanism for signals arriving at a vertex (net). However, Equations (10.1), (10.2) and (10.3) cannot be directly used in an algorithm. The equations are valid for a solution of the problem, but they do not tell how such a

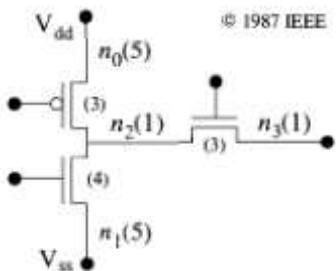


Figure 10.15 A small circuit of CMOS transistors.

propagate from → to	state of n_2	state of n_3
"initial state"	(1, 'X')	(1, 'X')
$n_0 \rightarrow n_2$	(3, '1')	(1, 'X')
$n_2 \rightarrow n_3$	(3, '1')	(3, '1')
$n_1 \rightarrow n_2$	(4, '0')	(3, '1')
$n_2 \rightarrow n_3$	(4, '0')	(3, 'X')

Figure 10.16 A "depth first" signal propagation for the circuit of Figure 10.15 leading to the wrong results.

solution could be obtained. It is certainly not the case that replacing the equal signs by assignments in these equations always leads to a correct solution.

One of the many methods for obtaining a correct solution will be pursued further here. The method is very simple in the sense that it only considers a single edge $(u, v) \in E$ at a time and updates the state of v given the state of u :

$$\sigma_v \leftarrow \max(\sigma_v, \sigma_{u \rightarrow v}) \quad (10.4)$$

Only a careful application of this rule leads to a correct solution as is shown below.

The initial state of the vertices should be initialized to the input signals for input nets and to signals with the net strength and appropriate initial level for storage nets. Consider e.g. the circuit shown in Figure 10.15 with $k = 2$ and $w = 5$ as the values of the "strength parameters" (the same values as in Figure 10.11). Also, assume that all three transistors are conducting. The state of net n_0 is stable at the value of (5, '1') throughout the execution of the algorithm because it is an input net. Similarly, n_1 has the stable value of (5, '0'). Propagating the value of n_0 to n_2 changes the state of n_2 to the signal value (3, '1'). This propagation step and the next ones for the case that a "depth first" strategy is followed, are shown in Figure 10.16. The "depth first" strategy amounts to propagating the effects of the latest signal change on some net in a similar way as the depth-first search algorithm discussed in Section 3.4.1. Unfortunately, the final value of (3, 'X') for net n_3 in the example is wrong. Because the final value of the signal on n_2 equals to (4, '0') (which is correct), propagation through the transistor connecting to n_3 should result in the signal value of (3, '0')

for n_3 . Clearly, some more analysis is required to design a correct signal propagation strategy.

The reason why the "depth first" strategy gave the wrong result for the example is that the value of n_2 was propagated to n_3 before it had reached its final stronger value. This wrong propagation put vertex n_3 into state $(3, '1')$, after which it was impossible to correct this state into $(3, '0')$ because a newly arriving signal is always combined with the old state in the computation of the new state.

The problem encountered in the example can be avoided if all nets are allowed to reach their final strength value before they can propagate their values to their neighbors. This can be achieved by giving priority to the vertex with the strongest signal for the propagation of its signal value to its neighbors. For example, a vertex v with strength 3 should not propagate its signals to its neighbors as long as there are vertices with stronger signals that still have to propagate their signals; they may change the strength of the state of v from 3 to 4. This method can be justified by realizing that signals traveling through a channel-connected component either maintain their strengths or become weaker; they are never amplified. Therefore, it is guaranteed that a signal that is propagated has always reached its final strength value. It is, however, possible that a vertex is selected twice for propagation for the same strength value but different level value (see later).

The strategy proposed asks for a queuing mechanism that stores the vertices whose signals need to be propagated, sorted by the strength of their signals. In the pseudo-code of the switch-level simulation algorithm given in Figure 10.17 this queuing is done by means of an array of sets of vertices, one set for each strength level. The pseudo-code summarizes the approach as discussed in the text above. Two auxiliary functions that have not been declared are called in the code. The first, `add_to_queue`, adds a vertex to the "queue" at the given strength value. It also removes the vertex from a queue position at a lower strength value if present. The function `combine` computes the combination of two signals according to the rules already explained (the result is the strongest signal; if the two signals have the same strength, the level of the result becomes 'X' when two distinct signal levels are involved).

It is not difficult to compute the time complexity of this algorithm. A vertex is selected for signal propagation at most twice. All vertices propagate their signals at least once, when they reach their final strength value. It may be, however, that a vertex is added to the queue for the second time at the same strength value but a signal level of 'X'. So, the number of propagations from a vertex is at most $2|V|$. Because each edge $(v, u) \in E$ in the graph is inspected both when propagating signals from v and u and from v to u , there will be at most $4|E|$ edge inspections. This gives a time complexity of $\mathcal{O}(|V| + |E|)$ which can be simplified to $\mathcal{O}(|E|)$ because all vertices in the graphs considered have at least one outgoing edge. Note that the removal of an already queued vertex in the function `add_to_queue` can be done in constant time by maintaining an extra attribute for each vertex that indicates its position in the queue.

```

switch_level_simulation (set of struct vertex V)
{
    set of struct vertex Q[w + 1]; /* Q[0] is not used */
    struct signal new_signal;

    for (i ← 1; i ≤ w; i ← i + 1)
        Q[i] ← ∅;

    for each v ∈ V {
        if ("v is an input vertex") {
            v.state ← "input signal value";
            add_to_queue(v, Q, w);
        }
        else /* v is a storage vertex */
            v.state ← "signal initially stored on v";
            add_to_queue(v, Q, v.strength);
    }

    for (i ← w; i ≥ 1; i ← i - 1)
        while (Q[i] ≠ ∅) {
            v ← "any element removed from Q[i]";
            for each e = (v, u) ∈ v.edges {
                if (e.on) {
                    new_signal.strength ← σv→u;
                    new_signal.level ← v.level;
                    if (new_signal "is not weaker than" u.state) {
                        u.state ← combine(u.state, new_signal);
                        if (u.state "has changed after calling combine")
                            add_to_queue(u, Q, new_signal.strength);
                    }
                }
            }
        }
    }
}

```

Figure 10.17 The pseudo-code of a switch-level simulation algorithm.

The evolution of the algorithm when applied to the example circuit of Figure 10.15 is illustrated in Figure 10.18. The first column shows the propagations of signals from a vertex v to a vertex u along the edges of the graph. An exclamation mark means that the propagation step did not result in adding u to the queue because the signal on u did not change. The second column shows the contents of the queue after propagation, while the third and fourth columns show the state of the vertices n_2 and n_3 (the states of n_0 and n_1 are not shown because they are constant throughout the computation).

Above, the basics of switch-level simulation have been discussed. This section ends with mentioning briefly some issues related to the topic that cannot be discussed in detail.

The approach to switch-level simulation described above is a *static* one: it assumes

prop. from → to	contents of Q	state of n_2	state of n_3
"initial state"	$Q[5] = \{n_0, n_1\}, Q[1] = \{n_2, n_3\}$	(1, 'X')	(1, 'X')
$n_0 \rightarrow n_2$	$Q[5] = \{n_1\}, Q[3] = \{n_2\}, Q[1] = \{n_3\}$	(3, '1')	(1, 'X')
$n_1 \rightarrow n_2$	$Q[4] = \{n_2\}, Q[1] = \{n_3\}$	(4, '0')	(1, 'X')
$n_2 \rightarrow n_0 !$	$Q[1] = \{n_3\}$	(4, '0')	(1, 'X')
$n_2 \rightarrow n_1 !$	$Q[1] = \{n_3\}$	(4, '0')	(1, 'X')
$n_2 \rightarrow n_3$	$Q[3] = \{n_3\}$	(4, '0')	(3, '0')
$n_3 \rightarrow n_2 !$		(4, '0')	(3, '0')

Figure 10.18 The illustration of the switch-level simulation algorithm when applied to the circuit of Figure 10.15.

that all input signals including those connected to the transistor gates are constant. For each change in any of these signals the algorithm has to be executed all over again. A *dynamic* approach, on the other hand, stores some additional information regarding how the results of the simulation were obtained, such that only a subset of the vertex states need to be recomputed when an input signal has changed.

Another issue that has not been discussed here is how to deal with transistor gate signals whose level value equals 'X' which means that it is unknown whether a transistor is conducting or not. The goal is in this case to detect those vertices whose signal values will not be affected by the "on" or "off" state of the transistors with unknown gate signals. All other vertices will receive a signal level 'X'. It seems that 2^n runs of the simulation algorithm are necessary if there are n transistors with unknown gate signals. However, it turns out that the problem can be solved with 2 runs only.

The switch-level model and the simulation algorithm based on this model, as has been described here, has the disadvantage that no information on switching delays can be inferred. The model is too abstract to relate strength values of transistors and nets to resistances and capacitances respectively. More refined models that better approximate the electrical properties of transistor circuits have been proposed. One can e.g. use continuous signal models instead of discrete ones, while still modeling a transistor as a switch that is either in a conducting, nonconducting or unknown state. If the model works with realistic values for resistances (R) and capacitances (C), RC products can be used to predict switching delays. A simulator that makes use of these principles is called a *switch-level timing simulator*.

10.4 Bibliographic Notes

A general introduction to VLSI simulation is given in Chapter 6 of [Rus85]. A more detailed discussion can be found in the textbook by Gosling [Gos93] that is dedicated to the topic of simulating digital circuits in general and gate-level simulation in particular. A paper that deals with theoretical issues related to the modeling and simulation of VLSI systems is [Lig87].

Textbooks on techniques used in circuit-level simulation are: [Vla83] or [Ogr94].

An enumeration of techniques used for macro and timing modeling and simulation that also covers the circuit and switch levels can be found in [Kon95]. A detailed discussion of timing simulation is given in [Ter85]. Mixed-mode simulation is the topic of [Sal94]. As mixed-mode simulation deals with the integration of analog and digital simulation models, the book also gives sufficient attention to both domains separately. A review of register-transfer level simulation is given in [Hem75]. Regarding VHDL, it can be said that it is strongly simulation-oriented and that the semantics of the language as e.g. described in [Lip89] indicate the techniques to be used in a simulator. An example of a system that is able to perform hardware-software co-simulation is *Ptolemy* [Kal93].

The notation of '0', '1', 'X', etc. for signal values has been borrowed from the standardized VHDL data type `std_logic` [Nay97].

Introductory texts on gate-level simulation are Chapter 4 of [Bre76], [d'A85] and Chapter 6 of [Rus85]. The topic is also discussed in [Mar93], which is a German text covering many aspects of VLSI design automation. A fast compiler-driven zero-delay simulation tool is presented in [Bar87]. Techniques that can be used in unit-delay compiled simulation are described in [Mau90].

An early paper that presents the event-driven simulation mechanism for digital circuits including the time wheel is [Ulr69]. More on the priority queue, which was mentioned as an alternative to the time wheel, can be found in many books on algorithms, including [Sed88] (and [Sed90]). A technique that describes a combination of compiler-driven and event-driven simulation is presented in [Wan90]. A demand-driven simulator has been reported in [Smi87].

The two researchers that independently have proposed the switch-level model for simulation, Hayes and Bryant, have authored two companion review papers: [Hay87] and [Bry87a]. The paper by Hayes is more on modeling and also contains some historical remarks. The paper by Bryant covers modeling too, but discusses possible simulation mechanisms as well. The notation used in this text has been taken from Bryant's paper.

The switch-level simulation algorithm presented in this chapter is a graph theoretical one. A similar algorithm, albeit for a slightly different switch-level model, is described in [Adl91]. An algebraic approach based on matrix calculus for Bryant's model is described in [Bry84]. Approaches partially based on compiler-driven methods have been reported in [Bry87b] and [Bar88]. A paper that concentrates on the fundamentals of switch-level simulation by showing its relation to circuit-level simulation is [Byr85].

Techniques for switch-level timing simulation are described in [Rao89], [Gen90] and [Gen91].

10.5 Exercises

- 10.1** Consider a transistor network meant for switch-level simulation, that has been described using the "cell-port-net" model of Section 7.1. Give the pseudo-code of a procedure that performs the static partitioning of the circuit and refer in

this code to the data structures of Figure 7.2 (the main lines of such a procedure are given in the text of Section 10.3.2).

- 10.2** The transistor strength of the switch-level model presented in Section 10.3 is a measure for the degree of conductance of the transistor (when the transistor is conducting). In the model, the series connection of two conductances is equivalent to the maximal conductance and the parallel connection of two conductances is equivalent to the minimal conductance. Verify this. Compare this behavior to the behavior in a real electric circuit and show that the switch-level model only approximates the electric model when the two conductances involved have different orders of magnitude.

11

Logic Synthesis and Verification

This chapter deals with *logic synthesis*, the automatic generation of circuitry starting from bit-level descriptions, and the related topic of *logic verification*, the comparison of a synthesis solution (especially those obtained by “manual design”) with a specification for the purpose of checking the solution’s correctness. As was briefly mentioned in Chapter 2, three categories for logic synthesis can be distinguished: two-level combinational logic, multilevel combinational logic and sequential logic. One can roughly claim that the complexity of the problems grows in the given order; two-level logic synthesis can be considered a special case of multilevel synthesis (add the constraint that no more than two logic levels can be used) and multilevel combinational synthesis can be seen as a special case of sequential synthesis (with zero states).

Due to the complexity of the matter and the limited space available, this chapter will mainly deal with two-level logic synthesis. Even the other two problems are more relevant to practice, they cannot be properly understood without some knowledge of two-level synthesis. The goal of this chapter is to provide some basic training in thinking about logic synthesis that goes beyond what can be found in introductory books on digital design.

After an introductory section, detailed attention is paid to the *binary-decision diagram* (BDD). This is a useful and often compact representation mechanism for Boolean functions that has applications in logic synthesis, verification as well as in many problems outside the field of logic design. The last part of this chapter presents an algorithm for two-level logic synthesis based on BDDs.

11.1 Introduction to Combinational Logic Synthesis

11.1.1 Basic Issues and Terminology

Logic circuits that do not possess an internal state are called *combinational*. Combinational logic circuits are built from elementary logic gates, such as NAND and NOR gates that are combinational themselves. Also, such circuits cannot have directed cycles (feedback loops) that may implicitly create memory elements.

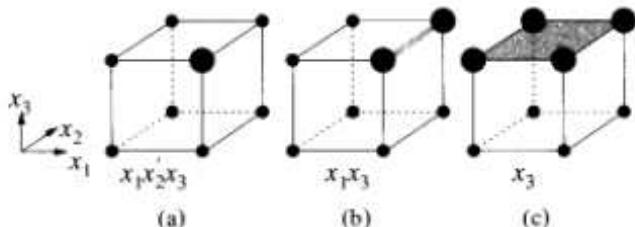


Figure 11.1 A minterm (a), a one-dimensional (b), and a two-dimensional cube (c).

The behavior of a combinational logic circuit consisting of m inputs and n outputs, can be described by a Boolean function $f : B^m \rightarrow B^n$, where $B = \{0, 1\}$ is the set of Boolean values. For each output, the function assigns either of the values '0' or '1' to any possible combination of input signals. The combinational logic synthesis problem could, therefore, be stated as the problem of generating some circuitry that will implement the behavior given by a Boolean function while minimizing some cost function.

In many practical situations, the behavior of a circuit is not fully specified by a Boolean function. The function is *incompletely* specified because some combinations of input signals are known not to occur or because the values of some outputs do not matter for some specific input patterns. Specifications of such a behavior require a third output value called the *don't care* value and denoted by '-'. In order to deal with don't care values, a Boolean function will be redefined as $f : B^m \rightarrow Y^n$, where $Y = \{0, 1, -\}$. For each output of the circuit, the space B^m of input signals can be partitioned into three sets: the *on-set* that contains all points in B^m for which the output should be '1', the *off-set* containing all points for which the output is '0' and the *dc-set* for the remaining points for which the output is don't care. The use of incomplete specifications has the main advantage that the logic synthesis tool has the freedom to decide which points of the dc-set will eventually be assigned an output value '0' and which a value '1'. This freedom will in general lead to cheaper solutions. A Boolean function whose dc-set is empty is called *fully* specified.

The m Boolean variables x_1, \dots, x_m will be used to specify any of the points in the input space B^m . A point is specified by assigning either '1' or '0' to each of the variables x_i ($1 \leq i \leq m$). Consider e.g. the point '(1, 0, 1)' in B^3 . It can also be given by the Boolean expression: $x_1 = 1 \wedge x_2 = 0 \wedge x_3 = 1$ or by the expression $x_1 \cdot \bar{x}_2 \cdot x_3$, if $x_i = 1$ is denoted by x_i and $x_i = 0$ by \bar{x}_i (the *complement* or *negation* of x_i) while the "and" operator (' \wedge ') is replaced by the smaller symbol ' \cdot '. The location of the point $x_1 \cdot \bar{x}_2 \cdot x_3$ in B^3 is shown in Figure 11.1(a).

The term *literal* denotes a Boolean variable or its complement. Any point in B^m can be identified by a product of the m distinct literals. Such a product is called a *minterm*. So, the expression $x_1 \cdot \bar{x}_2 \cdot x_3$ mentioned above is a minterm for B^3 .

A product of literals can be used to define a set of points rather than a single point

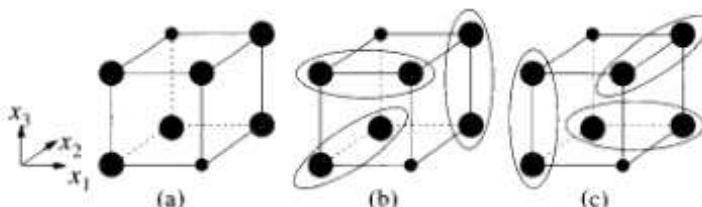


Figure 11.2 The on-set of a Boolean function f in B^3 (a) and two possible coverings by prime implicants (b, c).

by dropping one or more literals from the product. This means that the Boolean variable associated with the absent literal can have both possible Boolean values. The set of points defined in this way is called a *cube*. Figures 11.1(b) and (c) show examples of cubes that form a one and a two-dimensional subspace of B^3 respectively.

Given the facts that minterms uniquely identify single points in Boolean space and that a fully-specified Boolean function is described by the points in its on-set (the off-set is the on-set's complement), a Boolean function can be specified by a *sum of minterms*. The expression has a minterm for each point in the on-set. Consider e.g. the on-set of the Boolean function given in Figure 11.2(a). The sum of minterms corresponding to this function is:

$$f = \bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}_3 + \bar{x}_1 \cdot x_2 \cdot \bar{x}_3 + \bar{x}_1 \cdot \bar{x}_2 \cdot x_3 + x_1 \cdot \bar{x}_2 \cdot x_3 + x_1 \cdot x_2 \cdot \bar{x}_3 + x_1 \cdot x_2 \cdot x_3 \quad (11.1)$$

For each Boolean function, the sum of minterms is a *canonical form* which means that it represents the function in a unique way. To be more precise, the minterms in the expression should be sorted (e.g. using the position of the complemented Boolean variables as a sorting criterion) for the form to be truly canonical. The availability of a canonical form is especially important for purposes of *verification*: one may want to compare the specification of a (fully specified) Boolean function prior to (manual) synthesis with the Boolean function corresponding to the designed hardware and make sure that they are equal. If the sum-of-products canonical form of both are available, the comparison task is trivial and amounts to scanning the two expressions minterm by minterm. The main problem of using the sum-of-products canonical form in practice is the size of the representation. The input space for a Boolean function of m variables has 2^m points which means that a sum-of-products canonical form will consist of $\mathcal{O}(2^m)$ minterms. This prohibits the use of this canonical form in many practical situations.

The specification of a Boolean function by its sum-of-products canonical form is very similar to a specification by means of a *truth table*. The truth table enumerates all points in Boolean space and states for each point whether it belongs to the on-set, off-set or de-set. The truth table for the fully-specified function of Equation (11.1) (and Figure 11.2(a)) is given in Figure 11.3. By definition, any truth table for a

x_1	x_2	x_3	f
'0'	'0'	'0'	'1'
'0'	'0'	'1'	'1'
'0'	'1'	'0'	'1'
'0'	'1'	'1'	'0'
'1'	'0'	'0'	'0'
'1'	'0'	'1'	'1'
'1'	'1'	'0'	'1'
'1'	'1'	'1'	'1'

Figure 11.3 The truth table for the function of Figure 11.2(a).

function of m Boolean variables will have exactly 2^m entries.

When looking for a more compact canonical form for Boolean functions, one can think of using cubes instead of minterms. A cube whose points are either in the on-set or the dc-set is called an *implicant* of a Boolean function. An implicant that is not included in any other implicant and that has at least one point in the on-set, is called a *prime implicant*. Note that the definitions just given remain valid in the case of fully-specified functions whose dc-sets are empty (as is assumed below). Obviously, it is possible to have a sum-of-products expression in which the products are all prime implicants of the Boolean function. The function f of the example of Figure 11.2(a) can be written as the sum of its six prime implicants:

$$f = \overline{x_1} \cdot \overline{x_2} + \overline{x_1} \cdot \overline{x_3} + \overline{x_2} \cdot x_3 + x_1 \cdot x_2 + x_2 \cdot \overline{x_3} + x_1 \cdot x_3$$

Although each product is prime, the expression is redundant. The product $\overline{x_1} \cdot \overline{x_2}$ e.g. can be left out as the two points that it defines are contained in $\overline{x_1} \cdot \overline{x_3}$ and $\overline{x_2} \cdot x_3$. In the case of this example, it turns out that two *irredundant prime covers* are possible, corresponding to the expressions:

$$f = \overline{x_1} \cdot \overline{x_3} + \overline{x_2} \cdot x_3 + x_1 \cdot x_2 \quad (11.2)$$

and:

$$f = \overline{x_1} \cdot \overline{x_2} + x_2 \cdot \overline{x_3} + x_1 \cdot x_3$$

They are illustrated in the Figures 11.2(b) and (c). Although these last two expressions are both minimal, the existence of two different expressions shows that minimization alone does not lead to a canonical representation. The complete set of prime implicants of a function, on the other hand, is a canonical form, called *Blake's canonical form*. This canonical form has a limited practical relevance, however, as a consequence of the effort necessary to compute the complete set and the possibly large size of the set.

Another issue is that a representation with general cubes instead of minterms does not always lead to a reduction of the representation size. A well-known fully-specified function with this property is the *parity function*. Its on-set consists of

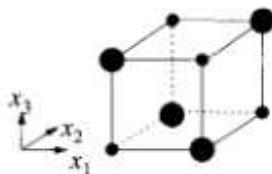


Figure 11.4 The on-set of the *parity* function in B^3 .

those minterms in which the number of noncomplemented Boolean variables is odd. Figure 11.4 shows the on-set of the parity function in B^3 . None of the points in the on-set of the parity function are adjacent (two points in Boolean space are *adjacent* if they can be transformed into each other by complementing a single literal in their minterms); this means that all prime implicants are minterms.

The topics that have just been described introduce two problems that will each be discussed in a separate section below. The first problem is to have a compact canonical representation of a Boolean function. An alternative to the sum of minterms is the *binary-decision diagram* that is presented in Section 11.2. The second problem is to obtain a minimal sum-of-products expression; this problem is called *two-level logic synthesis* and is discussed in Section 11.3. Before continuing the theory, however, a short exposition on the VLSI designer's view of logic synthesis is given below.

11.1.2 A Practical Example

Although a Boolean function of m variables gives rise to 2^m possible input combinations, most practical problems are such that not all possible combinations need to be specified explicitly. Besides, the language used as an input for logic synthesis will often allow the use of Boolean operators (such as `and` and `not`) and `if-then-else` constructs such that the designer can express the behavior of the desired function in a compact way. Figure 11.5 shows the specification in the popular VHDL language of a simple Boolean function. It is beyond the scope of this text to explain the syntax and semantics of VHDL in some detail. What matters in this example is that the declared circuit called `example` has 5 Boolean input variables x_1 to x_5 and two Boolean outputs y_1 and y_2 . The outputs are computed according to a specification in which the first two variables appear in the condition of an `if` statement and the other three in the assignment statements in the `then` and `else` branches of the `if` statements.

A logic synthesis system will parse this textual description and convert it to expressions that specify the on-sets and dc-sets of the two outputs. The on-sets are given by:

$$\begin{aligned} y_1 &= x_1 \cdot \overline{x_2} \cdot x_3 \cdot x_4 + \overline{x_1 \cdot \overline{x_2}} \cdot x_2 \cdot \overline{x_3} \cdot (x_4 + x_5) \\ y_2 &= x_1 \cdot \overline{x_2} \cdot (x_3 + x_4) \end{aligned} \tag{11.3}$$

```

library ieee;
use ieee.std_logic_1164.all;

entity example is
    port (x1, x2, x3, x4, x5: in std_logic;
          y1, y2: out std_logic);
end example;

architecture behavioral of example is
begin
    react: process (x1, x2, x3, x4, x5)
    begin
        if x1 = '1' and x2 = '0'
        then
            y1 <= x3 and x4;
            y2 <= x3 or x4;
        elsif x2 = '1'
        then
            y1 <= not (x3 and (x4 or x5));
            y2 <= '1';
        else
            y1 <= '1';
            y2 <= '0';
        end if;
    end process react;
end behavioral;

```

Figure 11.5 The VHDL code for a simple combinational logic circuit.

and the dc-sets by:

$$y_1 = \overline{x_1 \cdot \overline{x_2} \cdot \overline{x_2}}$$

$$y_2 = x_1 \cdot \overline{x_2} \cdot x_2$$

These expressions have been obtained by rewriting the `if` statements into Boolean expressions (by “anding” the condition with the expressions in the `then` branches and “anding” the complement of the condition with the expressions in the `else` branches). No further simplifications or transformations have been applied.

Note that expressions of this type are by no means canonical. They may be suitable as a starting point for multilevel logic synthesis. They may also be directly used to build BDD representations (see Section 11.2.2). If necessary, converting them to a sum-of-products form is not difficult by repetitively applying elementary laws of

Boolean algebra such as:

$$\begin{array}{ll}
 \overline{a \cdot b} = \overline{a} + \overline{b} & \overline{a + b} = \overline{a} \cdot \overline{b} \\
 (a + b) \cdot c = a \cdot b + a \cdot c & a \cdot b + c = (a + c) \cdot (b + c) \\
 a \cdot a = a & a + a = a \\
 a \cdot \overline{a} = '0' & a + \overline{a} = '1' \\
 a \cdot '0' = '0' & a + '0' = a \\
 a \cdot '1' = a & a + '1' = '1'
 \end{array} \tag{11.4}$$

Note that the first line contains the two versions of *De Morgan's Rule* and the second the two laws of *distributivity* (the second law of distributivity can be used to construct a *product-of-sums* form rather than a sum-of-products form).

One can also *evaluate* the expressions for all possible values of the input variables and construct a truth table. In such a case, one has already performed a number of calculations that grows exponentially with respect to the number of input variables already before starting the synthesis process.

11.2 Binary-decision Diagrams

As mentioned in Section 11.1, the fact that a Boolean function of m variables defines a Boolean space of 2^m points makes Boolean functions difficult to deal with from a computational complexity point of view. This exponential growth of Boolean space with respect to the number of variables may lead to algorithms with an exponential worst case time complexity.

Before thinking of algorithms, one should first make sure that the data structure to represent a Boolean function is compact on one hand and easy to manipulate on the other. The sum-of-minterms representation that was mentioned in Section 11.1 suffers from not being compact. This section presents an alternative called the *reduced ordered binary-decision diagram* (ROBDD). It has been proposed by Bryant and is now widely used for logic synthesis and verification as well as many other areas both inside and outside VLSI design automation. Its use leads to compact representations for most Boolean functions, although functions exist for which the size of the data structure still can grow exponentially. Also, many common manipulations on the data structure required for logic synthesis and verification can be performed efficiently.

11.2.1 ROBDD Principles

The first notion that should be introduced for the understanding of ROBDDs is *restriction*. This means the substitution of a constant value for one of the Boolean variables of a Boolean function. Substituting the value ' 1 ' for the variable x_i

($1 \leq i \leq m$) in a function f will be denoted by f_{x_i} and substituting the value '0' by $f_{\bar{x}_i}$. So:

$$\begin{aligned}f_{x_i} &= f(x_1, \dots, x_{i-1}, '1', x_{i+1}, \dots, x_m) \\f_{\bar{x}_i} &= f(x_1, \dots, x_{i-1}, '0', x_{i+1}, \dots, x_m)\end{aligned}$$

The two restrictions are also called the positive and respectively negative *cofactors* of f with respect to x_i . Consider e.g. the function f given earlier in Equation (11.1):

$$f = \bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}_3 + \bar{x}_1 \cdot x_2 \cdot \bar{x}_3 + \bar{x}_1 \cdot \bar{x}_2 \cdot x_3 + x_1 \cdot \bar{x}_2 \cdot x_3 + x_1 \cdot x_2 \cdot \bar{x}_3 + x_1 \cdot x_2 \cdot x_3$$

For this function:

$$\begin{aligned}f_{x_1} &= \bar{x}_2 \cdot x_3 + x_2 \cdot \bar{x}_3 + x_2 \cdot x_3 \\f_{\bar{x}_1} &= \bar{x}_2 \cdot \bar{x}_3 + \bar{x}_2 \cdot x_3 + x_2 \cdot \bar{x}_3\end{aligned}$$

The two restrictions with respect to a single Boolean variable are used to express the identity which is known as the *Shannon expansion* of a Boolean function f (although the principle was already known to Boole¹):

$$f = x_i \cdot f_{x_i} + \bar{x}_i \cdot f_{\bar{x}_i} \quad (11.5)$$

Note that the restriction of a Boolean function is a Boolean function with one variable less than the original function. The recursive application of Equation (11.5) to the two functions in its right-hand side will eventually lead to a fully-expanded expression. The recursion stops when all variables have been expanded and the restrictions f_{x_i} and $f_{\bar{x}_i}$ are equal to either of the constant functions '0' or '1' (the number of variables, m , is zero for a constant function). The step-by-step full expansion of the example function mentioned above, taking the variables in the order x_1 , x_2 and x_3 , results in:

$$\begin{aligned}f &= x_1 \cdot (\bar{x}_2 \cdot x_3 + x_2 \cdot \bar{x}_3 + x_2 \cdot x_3) + \bar{x}_1 \cdot (\bar{x}_2 \cdot \bar{x}_3 + \bar{x}_2 \cdot x_3 + x_2 \cdot \bar{x}_3) \\&= x_1 \cdot (x_2 \cdot (x_3 + \bar{x}_3) + \bar{x}_2 \cdot (x_3)) + \bar{x}_1 \cdot (x_2 \cdot (\bar{x}_3) + \bar{x}_2 \cdot (x_3 + \bar{x}_3)) \\&= x_1 \cdot (x_2 \cdot (x_3 \cdot '1' + \bar{x}_3 \cdot '1') + \bar{x}_2 \cdot (x_3 \cdot '1' + \bar{x}_3 \cdot '0')) + \\&\quad \bar{x}_1 \cdot (x_2 \cdot (x_3 \cdot '0' + \bar{x}_3 \cdot '1') + \bar{x}_2 \cdot (x_3 \cdot '1' + \bar{x}_3 \cdot '1'))\end{aligned} \quad (11.6)$$

Irrespective of the many possible initial specifications of a Boolean function, the full Shannon expansion of the function will lead to a unique representation provided that the expansion processes the variables in a fixed order. Full Shannon expansion is therefore a canonical form, which is not so surprising as the form enumerates all points in the input space of the Boolean function (compare the form to the sum of minterms and the truth table). The interesting aspect of the fully expanded form

¹ George Boole (1815–1864) is the founder of what is known today as *Boolean algebra*. Claude Shannon (born in 1916) is especially known as the father of *information theory*.

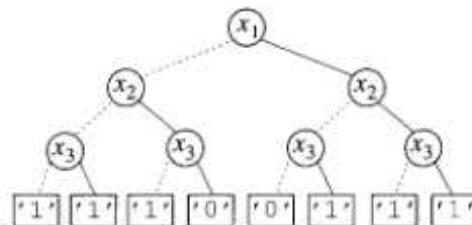


Figure 11.6 The OBDD for the example function of Equation (11.6).

is that it has a graph representation and that transformations can be applied to this graph representation which often considerably reduce the size of the graph without sacrificing the property of having a canonical form. Below, the graph representation called the *ordered binary-decision diagram* (OBDD) will be introduced first. An explanation on the transformations that reduce the OBDD into an ROBDD follows next.

The OBDD is a directed tree $G(V, E)$. All vertices $v \in V$, except for the root and the leaf vertices, have one edge incident to them and two edges incident from them. The two edges incident from a vertex point to the children vertices are called *high* and *low* and are respectively denoted by $\eta(v)$ and $\lambda(v)$. Also, each vertex has an attribute called *variable* and denoted by $\phi(v)$. The root vertex does not have an edge incident to it. The leaf vertices do not have any edges incident from them. The OBDD has a nonleaf vertex v for each application of Shannon expansion as given in Equation (11.5). The mapping is as follows: $\phi(v) = x_i$, $\eta(v)$ points to an OBDD (a subtree) that represents f_{x_i} and $\lambda(v)$ to an OBDD that represents $f_{\bar{x}_i}$. Leaf vertices are used when no more expansions are possible and the subtrees, therefore, correspond to one of the two constant Boolean functions '0' or '1'. In the case of a leaf vertex v , $\phi(v)$ gives the value of the appropriate constant function. The OBDD for the Boolean function as expanded in Equation (11.6) is shown in Figure 11.6. As usual, the edges are supposed to be directed from top to bottom. Edges that point to the "low" vertices are given by dashed lines while those that point to the "high" vertices are given by solid lines.

Note that it is essential that the expansion uses the variables in a fixed order for the representation to be canonical (in the case of the example, the ordering is x_1 followed by x_2 and x_3). This explains the adjective "ordered" in the name "ordered binary-decision diagram". The ordering of variables will be given by the function π that maps integers in the range 1 to m to variables. In the example: $\pi(1) = x_1$, $\pi(2) = x_2$ and $\pi(3) = x_3$. As any variable can only occur on one position, the inverse of π exists. Here: $\pi^{-1}(x_1) = 1$, $\pi^{-1}(x_2) = 2$ and $\pi^{-1}(x_3) = 3$.

Size reduction of an OBDD in order to obtain an ROBDD is achieved by means of the following transformations:

1. Replace all leaf vertices v with identical $\phi(v)$ by a single vertex and redirect all edges incident to the original vertices to this single vertex.

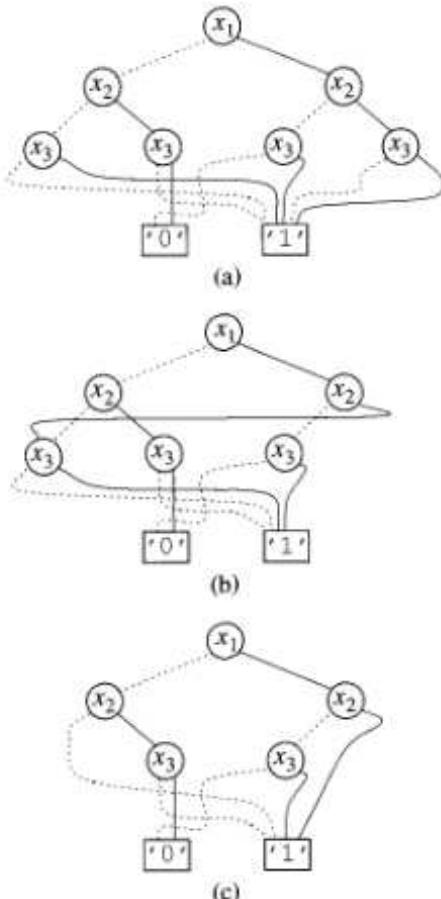


Figure 11.7 The application of Transformation Steps 1 (a), 2 (b), and 3 (c) to the OBDD of Figure 11.6 in order to obtain an ROBDD.

2. Process all vertices from bottom to top. If two vertices u and v are found for which $\phi(u) = \phi(v)$, $\eta(u) = \eta(v)$ and $\lambda(u) = \lambda(v)$, remove v and redirect to u all edges originally incident to v .
3. If edges v exist for which $\eta(v) = \lambda(v)$, remove v and redirect to $\eta(v)$ all edges originally incident to v .

Figure 11.7 illustrates how the subsequent application of these three steps to the OBDD of Figure 11.6 results in an ROBDD.

Note that any assignment of one of the two Boolean values '0' and '1' to all Boolean variables selects a unique path from the root vertex to any of the leaf vertices in the ROBDD. The value of the leaf vertex is the value of the represented Boolean function for the combination of input values chosen by the assignment.

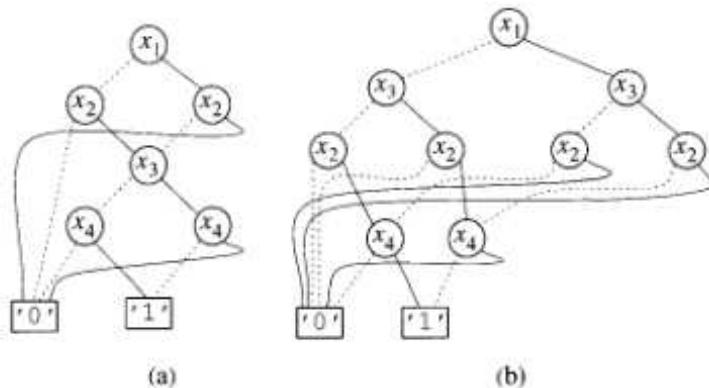


Figure 11.8 Two possible ROBDDs for the function of Equation (11.8) using a favorable (a) and a less favorable (b) variable ordering.

The ordering of variables strongly affects the size of an ROBDD. Families of Boolean functions are known that can be characterized by some complexity parameter k for which the size of the ROBDD can vary from a linear to an exponential function of k depending on the chosen variable ordering. Consider e.g. the family of functions given below:

$$f = \prod_{j=1}^k x_{2j-1} \oplus x_{2j} \quad (11.7)$$

(the ' \oplus ' symbol denotes the EXCLUSIVE-OR operator). When $k = 2$, one gets the function:

$$f = (x_1 \oplus x_2) \cdot (x_3 \oplus x_4) \quad (11.8)$$

The ROBDDs of this function for two different variable orderings are given in Figure 11.8. In the general case of the function of Equation (11.7), processing the variables in increasing index order (x_1, x_2, x_3, \dots) will result in a compact ROBDD containing $3k + 2$ vertices. Processing the variables with an odd index before the variables with an even index ($x_1, x_3, \dots, x_2, x_4, \dots$), on the other hand, will result in an exponentially growing ROBDD with $3 \cdot 2^k - 1$ vertices (exercise: check these expressions). This difference in size is due to the fact that the EXCLUSIVE-OR operation can be directly "evaluated" for the first ordering whereas, for the second ordering, all possible value combinations of the variables with an odd index have to be stored before the evaluation of the EXCLUSIVE-ORs can start.

Although a favorable variable ordering can be found for most Boolean functions, families of functions exist that have an exponentially growing number of vertices in their ROBDDs irrespective of the variable ordering. The multiplication function is such a function. The multiplication of two words of k bits, gives a result of $2k$ bits. Bryant has proven that for each ordering at least one of the $2k$ outputs needs an ROBDD whose size is an exponential function of k .

11.2.2 ROBDD Implementation and Construction

Now that the main principles regarding ROBDDs have been introduced and it has become clear that their use results in compact representations in most cases, the moment has come to discuss the algorithms that can operate on them. In this respect, ROBDDs can best be interpreted as an *abstract data type* (or a *class* in terms of object-oriented programming), a data structure with some procedures through which the user communicates and that hide the actual details of the data structure. The implementation of the data structures and procedures is generally called a *BDD package*. The general ideas behind such a package will be presented below. However, not all possible details that contribute to the package's efficiency will be covered in order to keep the presentation clear. The functionality provided by the procedures in a BDD package includes:

- The conversion the description of some Boolean function in an external format into an ROBDD. For reasons of efficiency, ROBDDs of multiple functions will share as many vertices as possible.
- The combination of functions that are already available, to create new functions (by means of operations such as AND, OR, XOR, etc.).
- The conversion of a stored function into an external format for use outside the BDD package.

A pseudo-code description of the data structure for an ROBDD vertex is shown in the first part of Figure 11.9. Obviously, it needs to contain pointers to the variable (ϕ) and the two subtrees (η and λ) that the vertex is pointing at. An actual implementation will need to store some more data that are not relevant to the discussion here. However, the notation " (ϕ, η, λ) " will be used in the following text to uniquely identify a vertex.

A crucial issue in the efficient implementation of a BDD package is that the ROBDDs are built directly rather than starting with a binary decision tree (that always has an exponentially growing size with respect to the number of variables) and then reducing it as was illustrated in Figure 11.7. The *unique table* is used for this purpose. For each triple (ϕ, η, λ) , the table points to the ROBDD vertex pointing to the triple elements if such a vertex exists. The table can be implemented by means of a *hash table* (see the Bibliographic Notes at the end of this chapter if you are unfamiliar with hash tables). A hash table has the nice property that it can return an entry in (almost) constant time without needing to reserve storage space for all possible entries. In the rest of the discussion, it is sufficient to know that the interface to the unique table is taken care of by the function `old-or-new` depicted in the bottom part of Figure 11.9. Given a triple (ϕ, η, λ) , it first checks whether a vertex associated with the triple exists. If so, it returns the vertex. Otherwise, it creates a new vertex, adds it to the unique table and then returns it. It can be assumed that each call to `old-or-new` takes constant time.

```

/* The basic data structure */

struct vertex {
    char * $\phi$ ;
    struct vertex * $\eta$ , * $\lambda$ ;
    ...
}

/* The interface to the unique table */

struct vertex *old_or_new(char * $\phi$ , struct vertex * $\eta$ , * $\lambda$ )
{
    if ("a vertex  $v = (\phi, \eta, \lambda)$  exists")
        return  $v$ ;
    else {
         $v \leftarrow$  "new vertex pointing at  $(\phi, \eta, \lambda)$ ";
        return  $v$ ;
    }
}

```

Figure 11.9 The vertex data structure and the function old-or-new.

A Boolean function described in some external format (such as a symbolic expression) can be converted into an ROBDD by the repetitive application of Shannon expansion as given in Equation (11.5). The recursive function `robdd.build` described in pseudo-code in Figure 11.10 takes care of the conversion. It is assumed that the two possible leaf vertices are already available before any call to the function: the vertex v_1 representing the constant function '1' and v_0 representing the constant function '0'. The top-level call to `robdd.build` takes two arguments: the Boolean function f that should be converted into an ROBDD and the integer 1 corresponding to the first variable $\pi(1)$ to be used for Shannon expansion.

Another thing that is assumed to be available is a simple *symbolic computation* system related to the external Boolean function format. It provides the data type `struct expr` for expressions, the function `equal` to compare two expressions and is able to perform the restriction operation to obtain f_ϕ and $f_{\bar{\phi}}$ from f .

When the algorithm is applied to the Boolean function given in Equation (11.2), it evolves as shown in Figure 11.11. In the figure, the successive calls of `robdd.build` are shown together with the result of the call. If a recursive call is necessary before reaching the result, the calls are shown using the same convention with a small indentation. The symbols ' $\xrightarrow{\eta}$ ' and ' $\xrightarrow{\lambda}$ ' are used to indicate whether the recursive call was made for the η or λ subtree. As in the previous examples, it has been assumed that $\pi(1) = x_1$, $\pi(2) = x_2$ and $\pi(3) = x_3$. The tree that has been constructed for the example is shown in Figure 11.12. It is, not surprisingly, identical to the tree of Figure 11.7(c). However, this figure also shows the names of each vertex.

```

struct vertex *robdd_build(struct expr f, int i)
{
    struct vertex * $\eta$ , * $\lambda$ ;
    struct char * $\phi$ ;

    if (equal(f, '0'))
        return v0;
    else if (equal(f, '1'))
        return v1;
    else {
         $\phi \leftarrow \pi(i)$ ;
         $\eta \leftarrow \text{robdd\_build}(f_{\phi}, i + 1)$ ;
         $\lambda \leftarrow \text{robdd\_build}(f_{\bar{\phi}}, i + 1)$ ;
        if ( $\eta = \lambda$ )
            return  $\eta$ ;
        else
            return old_or_new( $\phi, \eta, \lambda$ );
    }
}

```

Figure 11.10 The pseudo-code for the construction of the ROBDD of Boolean functions provided in an external format.

When building the ROBDD for a Boolean function *f*, the function `robdd_build` will create fewer ROBDD vertices than the number of vertices in the final ROBDD structure (it will not need to create *v*₀, *v*₁ and possibly other vertices created as a consequence of an earlier function call). The function is efficient in this sense. However, the number of recursive function calls can be larger than the number of vertices in the ROBDD. This is due to the fact that `robdd_build` needs to apply restrictions down to the lowest level before it can discover equivalent subfunctions. This effect could be circumvented by storing the root ROBDD vertex for each subfunction processed (e.g. by using an extra hash table).

11.2.3 ROBDD Manipulation

Once externally provided Boolean functions have been converted into ROBDDs, they are available for different sorts of manipulation. One can e.g. create new functions by applying binary operators to existing functions or by function composition (using a function's output as an input for another). A discussion on algorithms for the efficient manipulation of ROBDDs is given below.

Although Boolean functions obey the same algebraic rules irrespective of their representations, it is convenient to use upper-case letters, such as *F*, when referring to functions represented as ROBDDs, while continuing to use lower-case letters for the abstract functions or those given in an external format. Symbols for functions with an ROBDD representation will on one hand be used in a way similar to abstract functions (e.g. F_x and $F_{\bar{x}}$ for restriction). On the other hand, they will denote the

```

robdd.build( $\overline{x_1} \cdot \overline{x_3} + \overline{x_2} \cdot x_3 + x_1 \cdot x_2$ , 1)
     $\xrightarrow{\eta}$  robdd.build( $\overline{x_2} \cdot x_3 + x_2$ , 2)
         $\xrightarrow{\eta}$  robdd.build('1', 3)
             $v_1$ 
             $\xrightarrow{\lambda}$  robdd.build( $x_3$ , 3)
                 $\xrightarrow{\eta}$  robdd.build('1', 4)
                     $v_1$ 
                     $\xrightarrow{\lambda}$  robdd.build('0', 4)
                     $v_0$ 
                     $v_2 = (x_3, v_1, v_0)$ 
                     $v_3 = (x_2, v_1, v_2)$ 
                 $\xrightarrow{\lambda}$  robdd.build( $\overline{x_3} + \overline{x_2} \cdot x_3$ , 2)
                     $\xrightarrow{\eta}$  robdd.build( $\overline{x_3}$ , 3)
                         $\xrightarrow{\eta}$  robdd.build('0', 4)
                         $v_0$ 
                         $\xrightarrow{\lambda}$  robdd.build('1', 4)
                         $v_1$ 
                         $v_4 = (x_3, v_0, v_1)$ 
                     $\xrightarrow{\lambda}$  robdd.build( $\overline{x_3} + x_3$ , 3)
                         $\xrightarrow{\eta}$  robdd.build('1', 4)
                         $v_1$ 
                         $\xrightarrow{\lambda}$  robdd.build('1', 4)
                         $v_1$ 
                         $v_5 = (x_2, v_4, v_1)$ 
                     $v_6 = (x_1, v_3, v_5)$ 

```

Figure 11.11 The step-by-step evolution of the robdd.build algorithm when applied to the function as given in Equation (11.2).

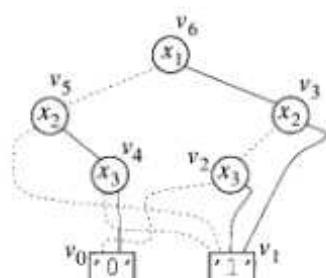


Figure 11.12 The tree corresponding to the example of Figure 11.11.

root vertex in the ROBDD representation, such that notations such as $\phi(F)$, $\eta(F)$ and $\lambda(F)$ or $F = (\phi, \eta, \lambda)$ make sense.

It turns out that the so-called *if-then-else operator ‘ite’* can be used as the basis for all ROBDD manipulations. The interpretation of $z = \text{ite}(f, g, h)$ is straightforward. It means that z equals g when f is true and to h otherwise. This can be written as:

$$z = \text{ite}(f, g, h) = f \cdot g + \bar{f} \cdot h$$

The ‘ite’ operator can be used to express all possible Boolean functions of two variables. Here are some examples:

$$\begin{aligned} z &= f \cdot g = \text{ite}(f, g, '0') \\ z &= f + g = \text{ite}(f, '1', g) \\ z &= \bar{f} = \text{ite}(f, '0', '1') \end{aligned}$$

An important property of the ‘ite’ operator is that it can be mapped on an ROBDD vertex v with $\phi(v) = x$ according to the following rule:

$$v = \text{ite}(F, G, H) = (x, \text{ite}(F_x, G_x, H_x), \text{ite}(F_{\bar{x}}, G_{\bar{x}}, H_{\bar{x}})) \quad (11.9)$$

In accordance with the notational convention mentioned above, it is assumed that F , G and H are the root vertices of respectively the functions f , g and h and that $\text{ite}(F, G, H)$ is the root vertex of the resulting function.

Note that the rule given in Equation (11.9) can be used as the basis for a recursive function that manipulates Boolean functions available as ROBDDs. The pseudo-code description of this function called `apply.ite` is given in Figure 11.13. The code consists of checks to see whether the recursion can be terminated followed by the recursive calls corresponding to Equation (11.9).

An important observation is that all functions represented by ROBDDs have the same variable ordering given by the function π . The variables should be processed in increasing order of the index i as can be seen from the pseudo-code where i is increased for each new level of recursion ($i = 1$ for the top-level call). Given this approach, the computation of the restrictions required for the recursive calls becomes easy. Suppose that i is the index value at the current level of recursion and that a function F should be restricted with respect to variable $x = \pi(i)$. F is the root vertex of an ROBDD. Because all variables up to $\pi(i - 1)$ have been processed at previous levels of recursion, $\pi^{-1}(\phi(F)) \geq i$. Then:

$$F_x = \begin{cases} F & \text{if } \pi^{-1}(\phi(F)) > i \\ \eta(F) & \text{if } \pi^{-1}(\phi(F)) = i \end{cases} \quad (11.10)$$

Analogously:

$$F_{\bar{x}} = \begin{cases} F & \text{if } \pi^{-1}(\phi(F)) > i \\ \lambda(F) & \text{if } \pi^{-1}(\phi(F)) = i \end{cases} \quad (11.11)$$

```

struct vertex *apply_ite(struct vertex *F, *G, *H, int i)
{
    char x;
    struct vertex *η, *λ;

    if (F = v1)
        return G;
    else if (F = v0)
        return H;
    else if (G = v1 && H = v0)
        return F;
    else {
        x ← π(i);
        η ← apply_ite(Fx, Gx, Hx, i + 1);
        λ ← apply_ite(F¬x, G¬x, H¬x, i + 1);
        if (η = λ)
            return η;
        else
            return old_or_new(x, η, λ);
    }
}

```

Figure 11.13 The function `apply_ite` that applies the '`ite`' operator on three argument functions represented by ROBDDs.

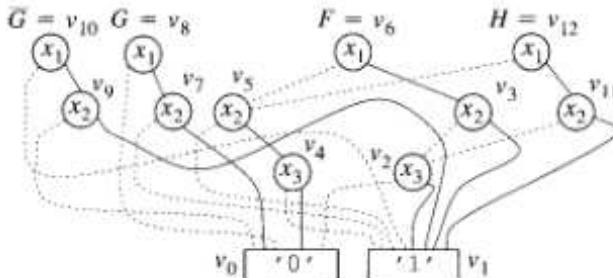


Figure 11.14 The ROBDDs for the functions F , G , \bar{G} and H used as examples in the text.

The use of the function `apply_ite` will be illustrated by elaborating further on the example ROBDD of Figure 11.12. Consider first the second Boolean function:

$$g = x_1 \cdot \overline{x_2}$$

that should be added to the existing ROBDD structure. Constructing the ROBDD of g by means of `robdd.build` will result in the root vertex v_8 shown in Figure 11.14 (see Exercise 11.3). Suppose now that one wants to use the ROBDDs of F and G to construct the function $H = F \oplus G$. In terms of the '`ite`' operator, this means that $H =$

```

apply_ite(v8, v0, v1, 1)
    ↳ apply_ite(v7, v0, v1, 2)
        ↳ apply_ite(v0, v0, v1, 3)
            ↳ apply_ite(v1, v0, v0, 3)
                ↳ apply_ite(v1, v0, v0, 3)
                    ↳ v0
                    ↳ v9 = (x2, v1, v0)
                ↳ apply_ite(v0, v0, v1, 2)
                    ↳ v1
                    ↳ v10 = (x1, v9, v1)

```

Figure 11.15 The trace of the function `apply_ite` when computing \overline{G} from $G = v_8$.

```

apply_ite(v6, v10, v8, 1)
    ↳ apply_ite(v3, v9, v7, 2)
        ↳ apply_ite(v1, v1, v0, 3)
            ↳ v1
            ↳ apply_ite(v2, v0, v1, 3)
                ↳ apply_ite(v1, v0, v1, 4)
                    ↳ v0
                    ↳ apply_ite(v0, v0, v1, 4)
                        ↳ v1
                        ↳ v4 = (x3, v0, v1)
                        ↳ v11 = (x2, v1, v4)
                    ↳ apply_ite(v5, v1, v0, 2)
                        ↳ v5
                        ↳ v12 = (x1, v11, v5)

```

Figure 11.16 The trace of the function `apply_ite` when computing H from $F = v_6$, $G = v_8$ and $\overline{G} = v_{10}$.

$ite(F, \overline{G}, G)$. \overline{G} on its turn requires the computation of $\overline{G} = ite(G, '0', '1')$ ². The step-by-step evolution of the call to `apply_ite` for the latter computation is shown in Figure 11.15. The notational convention used in this figure is similar to the one of Figure 11.11. The computation of H that can take place now is illustrated in Figure 11.16. The final ROBDD structure for this example is given in Figure 11.14.

If the number of vertices for an ROBDD having F as its root is denoted by $|F|$, then the time complexity of the `apply_ite` function is given by $\mathcal{O}(|F| \times |G| \times |H|)$. The fact that $|F| \times |G| \times |H|$ is an upper bound for the number of calls to `apply_ite`

² Efficiently implemented BDD packages will use so-called *negative* edges which make function complementation a trivial action that can be performed in constant time.

is not difficult to see: the function is called at most once for each combination of vertices originating from each of the three trees. As each call of `apply_ite` will create at most one ROBDD vertex, the number of vertices in the resulting ROBDD will also be $\mathcal{O}(|F| \times |G| \times |H|)$. This time complexity is interesting when one realizes that the size of an ROBDD may be a linear up to an exponential function of the number of variables. The size of the resulting ROBDD is bounded by the sizes of the original functions: small original functions will lead to a relatively small result function. Many calls of `apply_ite` are required to create functions that are drastically larger than the original functions. This *graceful degradation* property is another advantage of using ROBDDs for Boolean function manipulation.

A frequently encountered problem is to construct the ROBDD of a combinational circuit built of discrete components: the so-called *composition problem*. Assuming that the ROBDDs of all components are available (they can e.g. be obtained by the application of the `robdd.build` function discussed in Section 11.2.2), the issue is to use them for constructing the ROBDD of the overall circuit. The problem actually amounts to using the output of a Boolean function g as one of the inputs x_i of a function f . The expression for the resulting function h is:

$$h = f(x_1, \dots, x_{i-1}, g, x_{i+1}, \dots, x_n)$$

Substituting $x_i = g$ in the expression of Equation (11.5) for the Shannon expansion, one gets:

$$h = g \cdot f_{x_i} + \bar{g} \cdot f_{\bar{x}_i} = \text{ite}(g, f_{x_i}, f_{\bar{x}_i}) \quad (11.12)$$

The fact that composition can be expressed in terms of the ‘`ite`’ operator means that the required computations can be carried out by means of the function `apply_ite`. However, the restrictions f_{x_i} and $f_{\bar{x}_i}$ that are the arguments of the ‘`ite`’ operator should be computed first.

Deriving the restriction of a function represented by an ROBDD with respect to some variable x_i is quite straightforward. It amounts to remove all vertices v with $\phi(v) = x_i$ and redirecting the edges incident to v either to $\eta(v)$ (for the positive cofactor) or $\lambda(v)$ (for the negative cofactor). Doing this directly in the data structure would destroy the original function’s ROBDD. It is, however, not very difficult to design an algorithm for restriction that is not destructive. Figure 11.17 shows the pseudo-code of a recursive algorithm, which is somewhat similar to `apply_ite`, that computes the positive cofactor. The algorithm has three arguments: the root vertex F of the function that should be restricted, the index r of the variable $\pi(r)$ with respect to which the restriction should take place, and the recursion index i ($i = 1$ for the top-level call). When i reaches the value r the recursion stops and the algorithm returns the appropriate subtree $\eta(F)$. For smaller values of i the same mechanism as in `apply_ite` takes care of directly constructing the reduced tree. An algorithm to compute the negative cofactor would look exactly the same except for returning $\lambda(F)$ instead of $\eta(F)$ when $r = i$.

```

struct vertex *positive_cofactor(struct vertex *F, int r, i)
{
    char x;
    struct vertex *η, *λ;

    if (F = v1)
        return v1;
    else if (F = v0)
        return v0;
    else if (r = i)
        return η(F);
    else {
        x ← π(i);
        η ← positive_cofactor(Fx, r, i + 1);
        λ ← positive_cofactor(F¬x, r, i + 1);
        if (η = λ)
            return η;
        else
            return old_or_new(x, η, λ);
    }
}

```

Figure 11.17 The pseudo-code of an algorithm for computing the positive cofactor of an ROBDD.

Consider once more the Boolean function F as given in Figure 11.14. The computation of F_{x_3} by means of the function `positive_cofactor` is illustrated in Figure 11.18. The overall example ROBDD structure after the calculation of F_{x_3} is shown in Figure 11.19.

The discussion above makes it clear that the combination of the '*ite*' and restriction operators is sufficient to solve the function composition problem. By repetitively applying this principle to each of the connections in a circuit built from combinational logic components, one can obtain the ROBDD of the entire circuit. Note that many intermediate ROBDD structures may be created in this process and that the overall computational effort involved may vary with the order in which the connections are processed.

Computing restrictions first and then using them as argument for '*ite*' may create unnecessary vertices that are used for the representations of the restrictions only. This can be avoided by designing a recursive algorithm that combines the functionality of '*ite*' and restriction. The creation of vertices for temporary use can occur, by the way, in many other situations. BDD packages use so-called *garbage collection* mechanisms to automatically remove vertices that are no longer required.

```

positive_cofactor(v6, 3, 1)
 $\xrightarrow{\eta}$  positive_cofactor(v3, 3, 2)
 $\xrightarrow{\eta}$  positive_cofactor(v1, 3, 3)
 $v_1$ 
 $\xrightarrow{\lambda}$  positive_cofactor(v2, 3, 3)
 $v_1$ 
 $v_1$ 
 $\xrightarrow{\lambda}$  positive_cofactor(v5, 3, 2)
 $\xrightarrow{\eta}$  positive_cofactor(v4, 3, 3)
 $v_0$ 
 $\xrightarrow{\lambda}$  positive_cofactor(v1, 3, 3)
 $v_1$ 
 $v_7 = (x_2, v_0, v_1)$ 
 $v_{13} = (x_1, v_1, v_7)$ 

```

Figure 11.18 The trace of the function `positive_cofactor` when computing the restriction F_{x_3} .

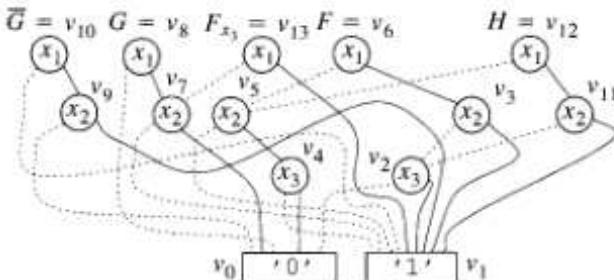


Figure 11.19 The ROBDD structure of Figure 11.14 after its extension with the function F_{x_3} .

11.2.4 Variable Ordering

It was already mentioned in Section 11.2.1 that the size of an ROBDD can vary strongly depending on the variable ordering chosen. An interesting problem is, of course, to find the ordering that results in the smallest ROBDD. Unfortunately, this is an intractable problem (the problem is *co-NP-complete* to be precise; the difference between NP-complete and co-NP-complete problems is outside the scope of Chapter 4). The heuristics that have been proposed for variable ordering can be divided into *static* and *dynamic* methods. They will be briefly discussed here.

Static variable ordering methods are based on an analysis of the Boolean function as provided in an external format, such as a Boolean expression or a network of gates. The goal is to try to “evaluate” a subexpression or gate’s function as soon as possible without the interference of variables that are not directly involved (think of

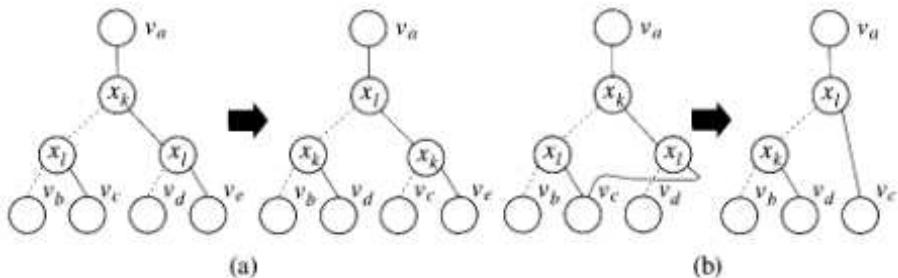


Figure 11.20 The correctness-preserving transformation that interchanges adjacent vertices (a) and a situation in which such a transformation reduces the ROBDD size (b).

the example illustrated in Figure 11.8).

Static heuristics are of limited use for a BDD package because multiple externally provided functions are simultaneously stored and new functions are created by manipulation, while the same variable is used for all functions. An initial choice for the ordering can turn out to result in large ROBDDs for functions created later.

Static methods fix the variable ordering prior to ROBDD construction, whereas dynamic methods can operate in the presence of ROBDDs already stored. Besides, heuristic dynamic ordering heuristics can be called several times in the course of Boolean function manipulation such that the ROBDD structure can adapt itself to situations where new functions have been created or obsolete functions have been removed. Both garbage collection and dynamic variable ordering are functions of a BDD package about which a user does not need to bother. The package itself activates them, e.g. after having used its memory space and before claiming more memory from the operating system.

Successful dynamic ordering heuristics are based on the principle that interchanging ROBDD vertices with variables $\pi(i) = x_k$ and $\pi(i + 1) = x_l$ (i, k, l are all in the range 1 to m), i.e. adjacent variables in the ordering, only has a local effect on the stored ROBDDs. This is illustrated in Figure 11.20(a). In order to prove the correctness of the transformation, one should show that a path in the ROBDD that has been selected by assigning a value to each Boolean variable ends in the same terminal vertex both before and after the transformation (see also Section 11.2.1). It is not difficult to see that the vertex v_a with $\pi^{-1}(\phi(v_a)) < i$ is connected to the vertex v_b with $\pi^{-1}(\phi(v_b)) > i + 1$ only when $x_k = '0'$ and $x_l = '0'$ both before and after the transformation. A similar reasoning can be carried out for the three other possible assignments to the variables and the three other vertices v_c, v_d and v_e . Of course, the ROBDD also remains correct for the trivial cases when vertices with either of the variables x_k or x_l do not occur in a path (in such a case, the graph is not modified in spite of a change in the variable ordering).

Even when the transformation of Figure 11.20(a) preserves correctness, it does not lead to any reduction of the ROBDD size; the number of vertices before and after the transformation is the same. The transformation becomes interesting when

two vertices with variable x_i point to the same subtree for the same value assignment to x_i . A variable reordering will then lead to an ROBDD with one vertex less as is illustrated in Figure 11.20(b).

The availability of a variable reordering transformation for adjacent variables means that any pair of variables can be made to swap positions by applying a series of neighbor swaps. However, one also needs to know which variables could be swapped in order to reduce the ROBDD size. This issue is often tackled by means of brute-force heuristics like moving a variable through all positions and remembering the optimal position. Suppose e.g. that the ordering for a five-variable ROBDD at a certain moment is $x_3x_1x_4x_5x_2$; variable x_4 can be made to move to the left and right-most positions by means of the following transformations: $x_3x_1x_4x_5x_2 \rightarrow x_3x_4x_1x_5x_2 \rightarrow x_4x_3x_1x_5x_2 \rightarrow x_3x_4x_1x_5x_2 \rightarrow x_3x_1x_4x_5x_2 \rightarrow x_3x_1x_5x_4x_2 \rightarrow x_3x_1x_5x_2x_4$. Once the optimal position is known, a few more neighbor swaps may be necessary to restore it, starting from the final ordering where x_4 occupies the right-most position.

11.2.5 Applications to Verification

One of the problems for which ROBDDs can be used is *logic verification*. The issue is to check whether a combinational logic circuit that resulted from manual or automatic (but possibly erroneous) logic synthesis, the so-called *implementation*, obeys the *specification* of the circuit.

The simplest situation occurs when the specification consists of a fully-specified function f . The ROBDD with root vertex F of this function can be built from the external representation. The ROBDD of the implementation can be constructed from the ROBDDs of the components in the circuit using the composition technique explained in Section 11.2.3. Suppose that the root vertex of the ROBDD of the implementation is G . The facts that the implementation and specification should show an identical behavior (the same output value for the same combination of input values) and that both F and G are represented within the same ROBDD structure, actually means that F and G should coincide. This is a consequence of the way ROBDDs are constructed and manipulated. So, the verification step amounts to the simple (pointer) test $F = G$. One says that the representation of Boolean functions using a single ROBDD structure has the property of *strong canonicity* which means that identical functions are not only represented in the same way, but that the representations are one and the same. The equality test can therefore be performed in constant time.

The situation becomes slightly more complex when the specification is incomplete, i.e. the dc-set of the specification function is not empty. When incompletely specified functions were introduced in Section 11.1.1, a three-element set $Y = \{'0', '1', '-'\}$ of output values was used. The use of such a set would require the modification of the ROBDD data structure and associated algorithms to incorporate a third type of leaf vertex. However, the ROBDD concept can be used without

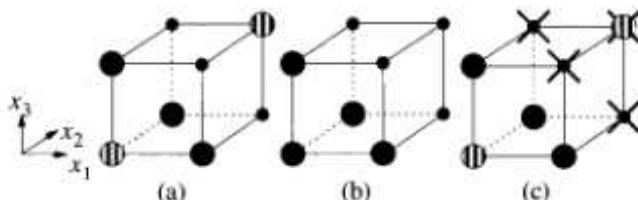


Figure 11.21 An example of a specification (a), its implementation (b) and their verification using Equation (11.13).

adaptation if two Boolean functions f and d are used for the on-set and dc-set respectively. f equals '1' for those points in Boolean space that belong to the on-set, and equals '0' otherwise. d equals '1' for those points in Boolean space that belong to the dc-set, and equals '0' otherwise.

Suppose that the Boolean function of the implementation is given by g . Note that an implementation does not have a dc-set; the fact that it is built of real components makes it fully specified. Then, the following expression should be a *tautology*, i.e. it should evaluate to the constant function '1', in order for the implementation to satisfy the specification:

$$d + f \cdot g + \bar{f} \cdot \bar{g} \quad (11.13)$$

This means that at least one of the following statements should be true for each point in Boolean space (keep in mind that the on-set and dc-set of the specification are disjoint):

- the point belongs to the dc-set of the specification;
- if the point belongs to the on-set of the specification, it is also part of the on-set of the implementation;
- if the point does not belong both to the off-set of the implementation, it neither belongs to the off-set of the specification.

Clearly, the expression of Equation (11.13) can be easily evaluated using the methods presented in Section 11.2.3. If the final result equals the constant function '1' (the vertex returned is v_1), it can be asserted that the implementation is correct.

Equation (11.13) is illustrated in Figure 11.21. Figure 11.21(a) shows an example specification in B^3 consisting of an on-set indicated by gray circles and a dc-set indicated by black-and-white circles. The implementation found by synthesis is given in Figure 11.21(b) where again gray circles are used for the on-set. Note that one of the two points in the specification's dc-set is now in the implementation's on-set. Figure 11.21(c) shows the tree terms of Equation (11.13): the black-and-white circles show d , the gray circles show $f \cdot g$ and the crosses $\bar{f} \cdot \bar{g}$. All points in B^3 are covered by at least one term which means that the implementation obeys the specification.

In the discussion above, Boolean functions were used for the representation of sets in Boolean space. It is sometimes convenient to use these functions as operands of set operators. Equation (11.13) can e.g. be better understood if written as:

$$(g \subset (f \cup d)) \cdot (f \subset g) \quad (11.14)$$

So, the correctness of the implementation means that the implementation's on-set is contained in the union of the on-set and dc-set of the specification while all elements of the specification's on-set should be contained in the implementation's on-set.

Using the equivalences $a \cup b \equiv a + b$ and $a \subset b \equiv \bar{a} + b$ and the fact that f and d are disjoint, it is not difficult to derive Equation (11.13) from Equation (11.14). The "intersection" operator that does not occur here, also has a Boolean equivalent: $a \cap b \equiv a \cdot b$.

11.2.6 Applications to Combinatorial Optimization

One often needs to know whether a Boolean function is *satisfiable*, i.e. whether there exists a combination of values for the input variables for which the function has the value '1'. This is a trivial task when the function is represented as an ROBDD. If the function is not satisfiable, this means that there are no paths from the root to leaf vertex v_1 which implies in turn that the root coincides with leaf vertex v_0 (this is a consequence of the reduction rules presented in Section 11.2.1). If the function is satisfiable, a solution can be derived from a path starting at the root and ending at v_1 as follows:

- if an edge $(v, \eta(v))$ occurs in the path, the variable $\phi(v)$ has a value of '1' in the solution;
- if an edge $(v, \lambda(v))$ occurs in the path, the variable $\phi(v)$ has a value of '0' in the solution;
- if a variable has not been assigned any value after processing all edges in the path, its value can be chosen arbitrarily to obtain a valid solution.

Obviously, ROBDDs can be used to solve the *satisfiability* problem introduced in Chapter 4. It involved establishing whether a Boolean expression consisting of a *product of sums* could be satisfied. Converting the expression into an ROBDD solves the problem. The problem remains, of course, NP-complete and exponential run times with respect to the number of variables may occur. However, the use of ROBDDs can be of practical importance as problem instances which were insolvable by other methods may now be tackled successfully.

Suppose that there is a cost c_i associated with each Boolean variable x_i ($1 \leq i \leq m$) of the satisfiability problem. Then the *minimum-cost satisfiability* problem asks for a solution that minimizes:

$$\sum_{i=1}^m c_i \mu(x_i) \quad (11.15)$$

The expression makes use of an auxiliary function μ : $\mu(x_i) = 0$ if x_i has value '0', and $\mu(x_i) = 1$ if x_i has value '1'. This problem can easily be solved if the Boolean function for which minimum-cost satisfiability is sought is represented by an edge-weighted ROBDD. This is an ordinary ROBDD extended with edge weights: each edge $(v, \eta(v))$ has a weight c_j related to the variable $x_j = \phi(v)$ and each edge $(v, \lambda(v))$ has a weight zero. Finding the minimum-cost assignment then amounts to finding the *shortest path* from the root vertex to v_1 in the ROBDD. As an ROBDD is a DAG, this can be done in linear time with respect to the graph size using an algorithm similar to the longest-path algorithm presented in Figure 6.8.

Note that the expression of Equation (11.15) is almost the same as the cost function of linear programming (LP, see Section 5.4.1), the only difference being that the variables in LP are numbers rather than of Boolean type. The special case of linear programming called *zero-one ILP* which was discussed in Section 5.4.2, can be reduced to a minimum-cost satisfiability problem by converting its constraints into a Boolean function. Here only one example will be given. Consider the following zero-one ILP constraint:

$$x_1 + x_2 + x_3 + x_4 = 3 \quad (11.16)$$

Given the correspondence of the integer values 0 and 1 that the variables x_1 to x_4 can have, with the respective Boolean values '0' and '1' for the Boolean variables x_1 to x_4 used below, the constraint can be represented by:

$$(x_1 + x_2) \cdot (x_1 + x_3) \cdot (x_1 + x_4) \cdot (x_2 + x_3) \cdot (x_2 + x_4) \cdot (x_3 + x_4) \cdot (\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4)$$

The first six sums in the product have been chosen such that at least three of the four variables should have the value '1' in order for the expression to be satisfiable. The last sum states that at least one of the variables has value '0'. Together they state that exactly three variables should be '1'. All constraints in the *standard form* of LP (and of zero-one ILP) are equality constraints similar to the example given in Equation (11.16) (see Section 5.4.1). As a solution of zero-one ILP should obey all constraints, the Boolean expression required to solve the problem by means of ROBDDs will consist of the product of all products of sums derived from the separate constraints.

It can be stated that minimum-cost satisfiability combined with a solution method based on ROBDDs is a general-purpose optimization method. Apart from the fact that the zero-one ILP problem, which is itself a general-purpose optimization method, can be solved in this way, many combinatorial optimization problems can be directly formulated in terms of minimum-cost satisfiability (see the Bibliographic Notes at the end of this chapter).

A combinatorial optimization problem with applications in logic synthesis is *set covering*. It involves a set $S = \{s_1, \dots, s_m\}$ and a set $K = \{K_1, \dots, K_n\}$ the elements of which are subsets of S . Also, a cost c_j is associated with each element K_j of K ($1 \leq j \leq n$). The goal of set covering is to select a minimal-cost subset Γ of K that the union of all subsets in Γ equals S . It is said that Γ *covers* S . The cost

	K_1	K_2	K_3	K_4	K_5	K_6
s_1	1	1	0	0	0	1
s_2	1	0	0	1	1	1
s_3	0	1	1	0	1	0
s_4	0	1	0	1	0	1

Figure 11.22 The covering matrix of the example used in the text.

of the cover is the sum of the costs of the subsets included in Γ . Choosing $c_j = 1$ for each subset K_j will minimize the number of subsets selected for Γ . The set covering problem with this cost function is NP-complete.

Choosing $c_j = |K_j|$ as the cost of a subset will minimize the number of elements of S that are covered multiple times. Consider the following example: $S = \{s_1, s_2, s_3, s_4\}$, $K = \{K_1, K_2, K_3, K_4, K_5, K_6\}$, $K_1 = \{s_1, s_2\}$, $K_2 = \{s_1, s_3, s_4\}$, $K_3 = \{s_3\}$, $K_4 = \{s_2, s_4\}$, $K_5 = \{s_2, s_3\}$, and $K_6 = \{s_1, s_2, s_4\}$. Using the cost model mentioned, the minimal solution consists of $\{K_3, K_6\}$ and has a cost of 4. An example of a solution that is not optimal is $\{K_1, K_2\}$; it has a cost of 5. $\{K_1, K_2, K_3\}$ is an example of a *redundant* solution as omitting the subset K_3 still leaves a valid solution.

Set covering problems can be visualized by a *covering matrix A* which has a row for each element in S and a column for each element of K . A matrix element A_{ij} equals 1 when the subset K_j contains the set element s_i , and equals 0 otherwise. The covering matrix for the example is shown in Figure 11.22.

A solution of the set covering problem can be described by associating a Boolean variable x_j with each element $K_j \in K$. $x_j = '1'$ means that Γ contains K_j , while $x_j = '0'$ implies that K_j is not part of the solution. In this situation, the cost function for set covering is given by Equation (11.15).

The Boolean variables can be used to formulate set covering in terms of satisfiability of a product-of-sums expression. Each element of the set S contributes a sum to the product composed of those variables that are associated to the subsets in which the element is contained. In the case of the example, this results in the expression:

$$(x_1 + x_2 + x_6) \cdot (x_1 + x_4 + x_5 + x_6) \cdot (x_2 + x_3 + x_5) \cdot (x_2 + x_4 + x_6) \quad (11.17)$$

The first sum states that either of K_1 , K_2 or K_6 should be chosen in order to cover s_1 . It corresponds to the first row of the covering matrix as given in Figure 11.22. The next sums are related to the remaining elements of the set S . Note that none of the variables in the expression are complemented. Complemented variables are required when the set covering problem is constrained in the sense that the occurrence of one subset in the solution excludes the selection of some other subsets. Consider again the example, now with the constraint that the subsets S_3 and S_6 are not allowed to be part of the solution simultaneously. Then, the corresponding satisfiability expression

becomes:

$$(x_1 + x_2 + \bar{x}_3 + x_6) \cdot (x_1 + \bar{x}_3 + x_4 + x_5 + x_6) \cdot \\ (x_2 + x_3 + x_5 + \bar{x}_6) \cdot (x_2 + \bar{x}_3 + x_4 + x_6) \quad (11.18)$$

Set covering problems that can be described by product-of-sums expressions of the type of Equation (11.17) are called *unate*. If the expression also contains complemented variables as in Equation (11.18), the problem is called *binate*. Clearly, ROBDDs can be used to solve the two types of problems. Both unate and binate covering are encountered in the field of logic synthesis. An example of unate covering will be given in the next section that deals with two-level synthesis. Binate covering is e.g. encountered in the synthesis of sequential logic, a topic that is outside the scope of this book.

11.3 Two-level Logic Synthesis

Any Boolean function can be realized in two levels e.g. as an AND-OR (sum of products) OR-AND (products of sums), NAND-NAND or NOR-NOR network (the latter two can be obtained from the former two by application of De Morgan's Rule). The minimal realization of logic circuits in two levels has traditionally received considerable attention not only because of theoretical reasons but as well because actual realizations in two levels were practically relevant. *Programmable logic arrays* (PLAs) were e.g. often used in nMOS technology. Now that CMOS technology has replaced nMOS as the mainstream technology, multilevel realizations composed of standard cells are almost uniquely used. The reason that two-level logic synthesis is still important is that its result is often a suitable starting point for multilevel synthesis algorithms.

Two aspects of two-level logic synthesis will be discussed here. First the problem is defined and analyzed along the lines of the classical Quine-McCluskey algorithm without going too much into details. Then a heuristic based on ROBDDs is presented in more detail. It has been chosen because it is relatively easy to understand.

11.3.1 Problem Definition and Analysis

Most of the necessary terminology to formulate the problem of two-level logic synthesis was already introduced in Section 11.1.1. The goal is to generate a minimal sum-of-products expression that is equivalent to some completely or incompletely specified Boolean function. The "products" in this expression are cubes. Optimality is often based on a cost function that consists of the sum of literals in each cube (the number of transistors in the realization is more or less proportional to this sum). Then, it is clear that all cubes in the solution should be prime implicants as the number of literals in a prime implicant is always smaller than those of a cube strictly contained in it. Also, the set of cubes of a minimal solution should be an

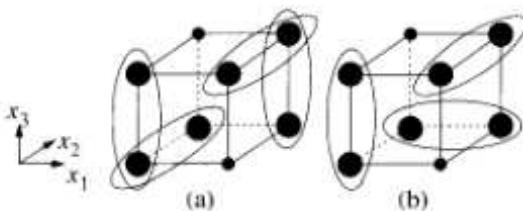


Figure 11.23 A locally (a) and a globally (b) minimal irredundant prime cover.

irredundant prime cover as the omission of redundant cubes will always decrease the cost function.

An irredundant prime cover is not necessarily minimal. This will be shown using the on-set of Figure 11.2(a) for which two irredundant prime covers are shown in Figure 11.23. The expression corresponding to Figure 11.23(a) is:

$$\overline{x_1} \cdot \overline{x_2} + \overline{x_1} \cdot \overline{x_3} + x_1 \cdot x_2 + x_1 \cdot x_3$$

It consists of four cubes with two literals each leading to a cost of 8. The expression corresponding to Figure 11.23(b), on the other hand, is:

$$\overline{x_1} \cdot \overline{x_2} + x_2 \cdot \overline{x_3} + x_1 \cdot x_3$$

It has one cube less which gives a cost of 6. The existence of two solutions shows that an irredundant prime cover is not necessarily a globally optimal solution.

A well-known algorithm for two-level logic synthesis is the *Quine-McCluskey algorithm*. It is based on contributions to the theory by Quine and McCluskey made in the 1950s. The algorithm first generates all prime implicants of the union of the on-set and dc-set omitting those prime implicants that only cover points of the dc-set. It then finds the minimum-cost cover of all minterms in the on-set by prime implicants from the set of all primes.

Note that the problem to be solved is the *set covering* problem presented in Section 11.2.6: the set of minterms (or the on-set) is the set S ; the set of prime implicants is the set K . The two-level logic synthesis problem could in principle be solved using the technique based on ROBDDs for solving set covering problems described in Section 11.2.6. The difficulty with this approach is that the set of prime implicant can grow excessively large with respect to the number of variables. It may already be impossible to obtain the set of all prime implicants as the set size can grow exponentially with respect to the number of Boolean variables. As each prime implicant contributes a variable for the ROBDD and the ROBDD can grow exponentially itself with respect to its variables, this approach may only be feasible for small problem instances.

Even when they do not affect the theoretical time complexity, the Quine-McCluskey algorithm first applies some simplifications to the problem instance at hand to reduce the search space. These simplifications will be discussed here. If a

	K_1	K_2	K_3	K_4	K_5		K_1	K_2	K_4	K_5	
s_1	1	0	0	1	1		s_1	1	0	1	1
s_2	1	1	0	0	0		s_2	1	1	0	0
s_3	1	1	0	1	0		s_3	1	1	1	0
s_4	0	0	1	0	0		s_5	0	1	1	1
s_5	0	1	0	1	1		s_7	1	0	0	1
s_6	1	0	1	1	0						
s_7	1	0	0	0	1						

Figure 11.24 The simplification of a covering matrix (a) because of essentiality (b), row dominance (c), and column dominance (d).

prime implicant is the only one to cover one or more minterms, it is called *essential*. As it will occur in any solution, this implicant and all the minterms covered by it can be removed from the covering matrix of the problem instance yielding a smaller instance. Consider the problem instance specified by the covering matrix shown in Figure 11.24(a). K_3 is an essential column for this instance because it is the only column to cover element s_4 . Because K_3 has to be part of any solution, it is guaranteed that s_4 and s_6 are already covered. After removing the column and the two rows involved, the reduced problem specification shown in Figure 11.24(b) is obtained.

If for some row s_i in a covering matrix all columns by which it is covered cover another row s_j , it is said that s_i dominates s_j . s_j can be removed from the covering matrix due to *row dominance*. In the case of Figure 11.24(b), it can be seen that the only two columns covering s_2 are K_1 and K_2 . At least one of these columns will be part of the solution. As both columns cover s_3 , the coverage of s_3 is guaranteed and the row can be removed from the matrix resulting in the matrix of Figure 11.24(c).

If all rows covered by some column K_j are also covered by a column K_i , it is said that column K_i dominates K_j . K_j can be removed from the matrix provided that $c_j > c_i$. This simplification rule is called *column dominance*. Consider the covering matrix of Figure 11.24(c) in which the rows covered by K_4 (s_1 and s_5) are also covered by K_5 . Column K_4 can be removed from the matrix when $c_4 > c_5$. It is not difficult to see that a solution that includes K_4 can never be optimal as replacing K_4 by K_5 will decrease the solution's cost without violating the condition that all rows should be covered. Assuming that $c_4 > c_5$, one gets the new covering matrix shown in Figure 11.24(d).

Once all three possible simplifications have been applied to the covering matrix,

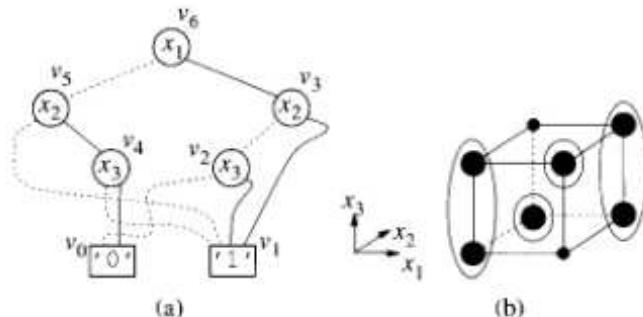


Figure 11.25 An example ROBDD (a) and the cover that can be derived from it directly (b).

one is left with the so-called *cyclic core* of the covering problem instance. In the cyclic core, none of the rows or columns is dominant. Apart from solving the problem via satisfiability and ROBDDs, one can also use a branch-and-bound algorithm to find a solution (see Section 5.2). Such an approach may still suffer from excessive computation times. If the estimation function used for pruning the search space is chosen such that it is “too optimistic”, the computation time can be reduced at the expense of possibly overlooking parts of the search space containing the optimal solution. The resulting algorithm is then a heuristic instead of an exact algorithm.

The most powerful techniques that find an exact solution for two-level logic synthesis have been proposed by Coudert and his colleagues (see the Bibliographic Notes at the end of this chapter). These techniques do not explicitly enumerate the set of all prime implicants. An implicit representation by means of ROBDDs is used instead.

11.3.2 A Heuristic Based on ROBDDs

ROBDDs have been introduced as a compact canonical representation for Boolean functions. One can argue that some kind of minimization has taken place during the construction of the compact representation which can be useful for synthesis. The most direct way to generate a two-level expression from an ROBDD is to consider all paths from the root to the leaf vertex v_1 . A product in a sum-of-products expression is generated by considering all edges in each path as follows: if the edge $(v, \eta(v))$ incident from some vertex v is in the path, the literal $\phi(v)$ is a factor in the product; if the edge is of the type $(v, \lambda(v))$, the literal $\overline{\phi(v)}$ is the factor to be used. Consider the example ROBDD of Figure 11.25(a) that is a copy of Figure 11.12. It has four paths from the root to the v_1 leaf vertex from which the following sum-of-products expression can be derived:

$$\overline{x_1} \cdot \overline{x_2} + \overline{x_1} \cdot x_2 \cdot \overline{x_3} + x_1 \cdot \overline{x_2} \cdot x_3 + x_1 \cdot x_2$$

The four cubes by which the on-set is covered are shown in Figure 11.25(b). It can be seen that not all cubes are prime implicants: the two one-element subsets can be

expanded to become two-element subsets. On the other hand, the subsets from which the cover is composed are disjoint, making the cover irredundant. The subsets of a cover derived in this way are always disjoint as each subset corresponds to a path and no two paths in an ROBDD can be simultaneously activated for some assignment to the Boolean variables.

Clearly, some more effort should be made to generate irredundant prime covers from an ROBDD. This happens in the *isop* (irredundant sum of products) algorithm proposed by Minato to be presented here. It is based on the following expansion theorem attributed to Morreale for a Boolean function f with respect to one of its variables x :

$$f = \bar{x} \cdot f_0 + x \cdot f_1 + f_d \quad (11.19)$$

When this expansion is compared to the Shannon expansion given in Equation (11.5), the following statements can be made: $f_0 \subset f_{\bar{x}}$, $f_1 \subset f_x$, and $f_d \subset (f_{\bar{x}} \cap f_x)$. In other words, f_d covers those points of the on-set that are contained in cubes in which the literal x does not occur. The expansion is not unique and different expansions exist for different f_d ($f_d = '0'$ is e.g. an extreme case). The algorithm to be presented below attempts to find a solution with the largest possible f_d because a cube in f_d always covers at least two points of the on-set which would otherwise need to be covered by two separate cubes from f_0 and f_1 . The algorithm constructs f_0 , f_1 and f_d in such a way that f is an irredundant prime cover provided that f_0 , f_1 and f_d are irredundant prime covers themselves. The latter is assured by the recursive nature of the algorithm.

The algorithm can deal with incompletely specified functions. At all levels of recursion, the possibility to manipulate a dc-set is essential and no additional effort is necessary to handle a nonempty dc-set at the top level. A pseudo-code description of the algorithm is given in Figure 11.26. The function *isop* is called with three arguments: the ROBDD root vertex pointers F and D that respectively represent the on-set and dc-set³ of the incompletely specified Boolean function to be synthesized, and the index i that selects a variable through the function π (see Section 11.2.1; $i = 1$ for the top-level call). It returns an expression that is restricted to a sum of products. It is assumed that the data type *expr* is available for the storage and manipulation of expressions.

The on-set and dc-set are disjoint at all levels of recursion. The algorithm consists of different steps that are illustrated in Figure 11.27. The figure attempts to illustrate in two dimensions structures that normally involve many more dimensions (imagine that some kind of projection has been performed). The parts (a) through (h) of the figure correspond to the parts with the same label in the pseudo-code. So, Figure 11.27(a) shows the initial situation. The entire Boolean space is the rectangle. The on-set, dc-set and off-set are respectively represented by F , D and $\overline{F \cup D}$.

³ Minato uses a different way to represent the two sets with two ROBDDs. His way is more advantageous from the computational complexity point of view. The "direct" encoding used here, however, gives a better insight in the principles of the algorithm.

```

/* (a) */
struct expr *isop(struct vertex *F, *D, int i)
{
    struct vertex *F0, *F1, *F'0, *F'1, *F''0, *F''1;
    struct vertex *D0, *D1, *D'0, *D'1, *D''0, *D''1;
    struct vertex *G0, *G1, *Fd, *Dd;
    struct expr *g0, *g1, *gd;
    char x;

    if(F = v0)
        return '0';
    else if (F + D = v1)
        return '1';
    else
        /* (b) */
        x ← π(i);
        F0 ← Fx;
        D0 ← Dx;
        F1 ← F¬x;
        D1 ← D¬x;
        /* (c) */
        F'0 ← F0 ·  $\overline{F_1 + D_1}$ ;
        D'0 ← D0 + F0 · (F1 + D1);
        F'1 ← F1 ·  $\overline{F_0 + D_0}$ ;
        D'1 ← D1 + F1 · (F0 + D0);
        /* (d) */
        g0 ← isop(F'0, D'0, i + 1);
        G0 ← robdd.build(g0);
        g1 ← isop(F'1, D'1, i + 1);
        G1 ← robdd.build(g1);
        /* (e) */
        F''0 ←  $\overline{G_0} \cdot F_0$ ;
        D''0 ← D0 + G0;
        F''1 ←  $\overline{G_1} \cdot F_1$ ;
        D''1 ← D1 + G1;
        /* (f) */
        Fd ← F''0 · F''1 + F''0 · D''1 + F''1 · D''0;
        Dd ← D''0 · D''1;
        /* (g) */
        gd ← isop(Fd, Dd, i + 1);
        /* (h) */
        return x · g0 + x · g1 + gd;
}

```

Figure 11.26 The pseudo-code of Minato's algorithm to generate a irredundant prime cover from an ROBDD.

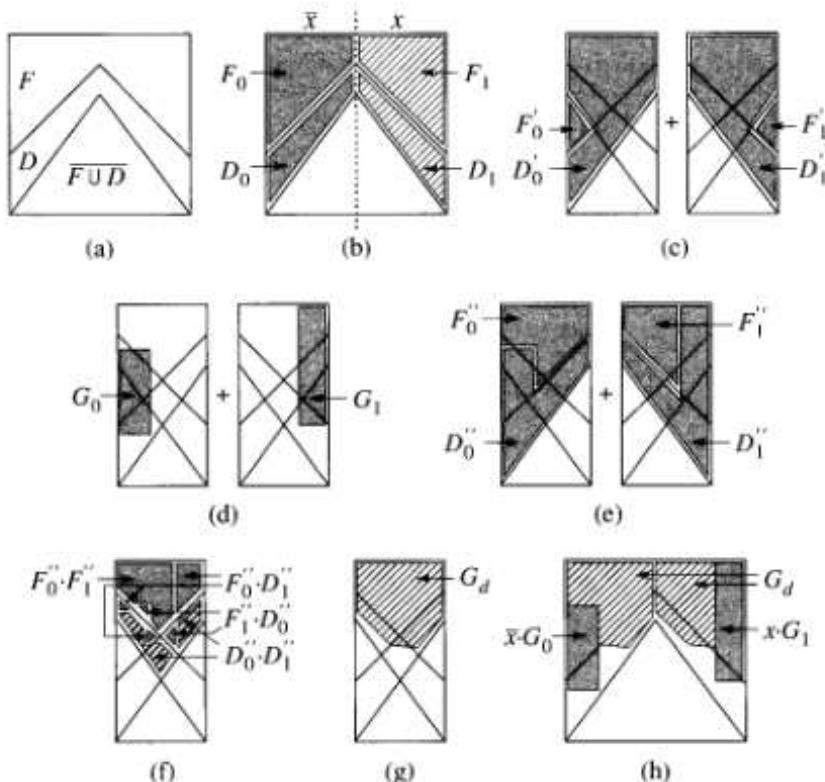


Figure 11.27 A visualization of the different steps involved in the *isop* algorithm as described in Figure 11.26.

In Step (b) of the algorithm, the two restrictions of F and D with respect to variable $x = \pi(i)$ are computed. This can be done in the manner of Equations (11.10) and (11.11) (see Section 11.2.3) because the algorithm descends the ROBDD according to the variable order given by π . Note that the variable x bisects Boolean space and that the bisection of the on-set and dc-set as shown in Figure 11.27(b) corresponds to the restrictions to be computed.

If the variable x is no part of the problem instance, the two subspaces corresponding to x and \bar{x} will coincide. It is in this reduced space that the points to be covered by f_d in Equation (11.19) can be found. They are located in the area given by $(F_0 \cup D_0) \cap (F_1 \cup D_1)$. In its search for these points, the algorithm first looks for those points that will certainly be covered by f_0 and f_1 of Equation (11.19) (as opposed to the rest of the text, there is no equivalence here between f_0 and f_1 on one hand and F_0 and F_1 on the other). Note: due to the symmetry in the reasoning, everything that can be claimed for f_0 and all functions derived from it have a counterpart for f_1 ; the rest of the explanation will concentrate on issues related to f_0 only. The set

of points that certainly should be covered by f_0 is given by the ROBDD root F'_0 . This is computed in Step (c) of the algorithm. F'_0 will be used as the on-set for a recursive call of the algorithm. The associated dc-set D'_0 consists of the original dc-set D_0 extended with the points that will potentially be covered by f_d . This extension is allowed because these points will still be either in the on-set or dc-set of the original function after "multiplication" of the cover with \bar{x} (think of Equation (11.19)). Figure 11.27(c) that illustrates the preparatory steps for the recursive calls, has two parts for the sake of clarity: both parts refer to the same Boolean space.

The recursive call of `isop` in Step (d) results in a sum-of-products expression that is assigned to g_0 . As all Boolean manipulation in the algorithm is made with ROBDDs, g_0 also needs to be available as an ROBDD. The function `robdd.build` introduced in Section 11.2.2 is called for this purpose (a more efficient solution could be used in practice) to obtain G_0 .

In Step (e), further preparations are made to compute f_d of Equation (11.19). F''_0 contains those points of F_0 that are not yet covered by G_0 . The points already covered by G_0 may be covered multiple times when it is convenient and are therefore contained in D''_0 .

The specification for the synthesis of f_d from Equation (11.19) is based on the potentially available area $(F_0 \cup D_0) \cap (F_1 \cup D_1)$ that was mentioned before. This area is carefully partitioned to form the on-set and dc-set for a recursive call of `isop` in Step (f). The result of this call is available in Step (g) as the sum-of-products expression g_d .

The expression $\bar{x} \cdot g_0 + x \cdot g_1 + g_d$ that is the overall result of the algorithm covers the original specification as is illustrated in Figure 11.27(h). Note that the ' \cdot ' and ' $+$ ' operators used in this expression operate on symbolic expressions as opposed to the same operators used earlier in the program where they operate on ROBDDs.

The proof that the result returned by `isop` forms an irredundant prime cover can be given by induction on the variable involved (the variable x in the algorithm description). The two results that can be returned at the lowest levels of recursion, the constant functions '`0`' and '`1`', are irredundant prime covers themselves (their associated Boolean spaces are empty; any statement about elements of an empty set is true). What remains to be proved is that $\bar{x} \cdot g_0 + x \cdot g_1 + g_d$ forms an irredundant prime cover, given the fact that g_0 , g_1 and g_d are irredundant prime covers themselves.

Consider primality first. Suppose that g_0 contains some cube c which contributes the cube $\bar{x} \cdot c$ to the final result. Can the \bar{x} be dropped from the solution? This either means that the cube $x \cdot c$ with c originating from g_1 is in the solution or that c itself with c originating from g_d is in the solution. None of the two cases can be true because c should cover at least one point of F'_0 to be in g_0 , but c should as well cover a point in F'_1 or F_d in order to be in g_1 or g_d . This condition cannot be satisfied as the three sets F'_0 , F'_1 and F_d are disjoint.

Irredundancy is proved in a similar way. The cubes in $\bar{x} \cdot g_0$ and in $x \cdot g_1$ are located in separate subspaces and their union cannot be redundant. One may think that a cube c originating from g_d may be jointly covered by cubes in $\bar{x} \cdot g_0$ and $x \cdot g_1$.

However, in order to be part of g_d , c should at least cover one point in F_d . F_d does not contain points that are simultaneously in g_0 and g_1 (points in $g_0 \cap g_1$ end up in D_d) which means that c cannot be entirely located in $\bar{x} \cdot g_0 + x \cdot g_1$.

In spite of its relative complexity, one should keep in mind that the isop algorithm is a heuristic. Different variable orderings in the ROBDD may result in different solutions. For each variable, the goal is to have g_d as large as possible even when this may not be favorable for the variables to be considered next. The algorithm is greedy in this sense. The strong point of the algorithm is that neither all minterms nor all prime implicants need to be represented explicitly.

The application of the algorithm on the ROBDD of Figure 11.25(a) is illustrated in Figure 11.28. It shows parts of a trace of the algorithm. For each call of the algorithm, the call itself is shown, the arguments, the local variables, the recursive calls, and the return value are shown. The return value figures in the last line of the text covering the function call. Although the algorithm operates on ROBDDs, the minterms covered by the ROBDDs are shown for the sake of readability rather than the graphs. Each part of the trace that has been omitted is indicated by an ellipsis ('...'). It can be seen that the result is indeed an irredundant prime cover but not a globally optimal one. For the globally optimal solution, the algorithm should not have maximized the coverage by g_d (see Figure 11.23). One should keep in mind that the algorithm presented here is especially interesting for large problem instances for which exact solution methods fail.

11.4 Bibliographic Notes

The basics of logic synthesis, including the Quine-McCluskey method for two-level minimization, can be found in elementary books on logic design. A large variety of books of this type exist. The following form an "arbitrary selection": [Hil81], [Puc90], [Hay93], [Kat94], and [Man97]. More advanced books on logic synthesis have been written as well. An in-depth treatment that deals with most aspects of logic design and verification, is given in [Hac96]. Other books on logic synthesis are: [Edw92], [DM94] and [Dev94]. A tutorial discussion that concentrates on issues that are of practical rather than academic importance can be found in [Rud96]. A detailed review of techniques that can be used in multilevel logic synthesis is given in [Bra90b]. Two books dedicated to various aspects of sequential synthesis are [Kam97] and [Vit97].

As stated in Section 11.1.2, VHDL is a popular language for the input specification of circuits that are to be synthesized automatically. The following books explain the details of how such specifications should be written: [Bha96], [Cha97], and [Nay97].

Although the binary-decision diagram [Ake78] and the related concept of the binary-decision program [Lee59] were known before, it was Bryant [Bry86] who pointed out that the use of BDDs leads to canonical forms provided that a fixed ordering of the variables is used. He also formulated the rules for reduction and showed how the resulting ROBDDs could efficiently be manipulated. The same paper contains the proof of the multiplication function having an exponentially

```

isop( $F, D, 1$ )
 $F = \{\overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3}, \overline{x_1} \cdot x_2 \cdot \overline{x_3}, \overline{x_1} \cdot \overline{x_2} \cdot x_3, x_1 \cdot \overline{x_2} \cdot x_3, x_1 \cdot x_2 \cdot \overline{x_3}, x_1 \cdot x_2 \cdot x_3\}; D = v_0$ 
 $F_0 = \{\overline{x_2} \cdot \overline{x_3}, x_2 \cdot \overline{x_3}, \overline{x_2} \cdot x_3\}; D_0 = v_0$ 
 $F_1 = \{x_2 \cdot x_3, x_2 \cdot \overline{x_3}, x_2 \cdot x_3\}; D_1 = v_0$ 
 $F'_0 = \{\overline{x_2} \cdot \overline{x_3}\}; D'_0 = \{x_2 \cdot \overline{x_3}, \overline{x_2} \cdot x_3\}$ 
 $F'_1 = \{x_2 \cdot x_3\}; D'_1 = \{\overline{x_2} \cdot x_3, x_2 \cdot \overline{x_3}\}$ 
 $\stackrel{0}{\rightarrow} \text{isop}(F'_0, D'_0, 2)$ 
 $F = \{\overline{x_2} \cdot \overline{x_3}\}; D = \{x_2 \cdot \overline{x_3}, \overline{x_2} \cdot x_3\}$ 
 $F_0 = \{\overline{x_3}\}; D_0 = \{x_3\}$ 
 $F_1 = v_0; D_1 = \{\overline{x_3}\}$ 
 $F'_0 = v_0; D'_0 = v_1$ 
 $F'_1 = v_0; D'_1 = \{\overline{x_3}\}$ 
 $\stackrel{0}{\rightarrow} \text{isop}(F'_0, D'_0, 3)$ 
 $\cdots$ 
 $\stackrel{0}{\rightarrow}$ 
 $g_0 = '0'; G_0 = v_0$ 
 $\stackrel{1}{\rightarrow} \text{isop}(F'_1, D'_1, 3)$ 
 $\cdots$ 
 $\stackrel{0}{\rightarrow}$ 
 $g_1 = '0'; G_1 = v_h$ 
 $F''_0 = \{\overline{x_3}\}; D''_0 = \{x_3\}$ 
 $F''_1 = v_0; D''_1 = \{\overline{x_3}\}$ 
 $F_d = \{\overline{x_3}\}; D_d = v_0$ 
 $\stackrel{d}{\rightarrow} \text{isop}(F_d, D_d, 3)$ 
 $\cdots$ 
 $\stackrel{0}{\rightarrow}$ 
 $\overline{x_3}$ 
 $\stackrel{0}{\rightarrow}$ 
 $g_0 = \overline{x_3}; G_0 = [\overline{x_2} \cdot \overline{x_3}, x_2 \cdot \overline{x_3}]$ 
 $\stackrel{1}{\rightarrow} \text{isop}(F'_1, D'_1, 2)$ 
 $\cdots$ 
 $x_3$ 
 $\stackrel{0}{\rightarrow}$ 
 $g_1 = x_3; G_1 = [\overline{x_2} \cdot x_3, x_2 \cdot x_3]$ 
 $F''_0 = \{\overline{x_2} \cdot x_3\}; D''_0 = [\overline{x_2} \cdot \overline{x_3}, x_2 \cdot \overline{x_3}]$ 
 $F''_1 = \{x_2 \cdot \overline{x_3}\}; D''_1 = [\overline{x_2} \cdot x_3, x_2 \cdot x_3]$ 
 $F_d = \{\overline{x_2} \cdot x_3, x_2 \cdot \overline{x_3}\}; D_d = v_0$ 
 $\stackrel{d}{\rightarrow} \text{isop}(F_d, D_d, 2)$ 
 $\cdots$ 
 $\overline{x_2} \cdot x_3 + x_2 \cdot \overline{x_3}$ 
 $g_d = \overline{x_2} \cdot x_3 + x_2 \cdot \overline{x_3}$ 
 $\overline{x_1} \cdot \overline{x_3} + x_1 \cdot x_3 + \overline{x_2} \cdot x_3 + x_2 \cdot \overline{x_3}$ 

```

Figure 11.28 The trace of the function `isop` when computing an irredundant prime cover of the ROBDD of Figure 11.25(a).

growing ROBDD. The developments until 1992 that have led to applications of BDDs beyond logic design are summarized in [Bry92]. A more recent survey paper that lists many variants on the BDD is [Dre97].

The ideas presented here for an efficient implementation of a BDD package can be found in [Bra90a]. Many books on algorithms, including [Sed88], pay attention to hash tables, the data structure used to implement the unique table. A comparison between different popular BDD packages is provided in [Sen96].

It is mentioned in this chapter that the right order in the application of *composition* may have an influence on the computational effort involved. An approach to optimizing this effort is described in [Jai96]. The fact that the variable ordering problem for ROBDDs is co-NP-complete is mentioned without providing a proof in [Bry86]. Heuristics for static variable ordering based on the circuit topology are proposed in [Fuj88] and [Mal88]. Dynamic variable ordering heuristics are described in [Fuj91] and [Rud93]. The latter paper also provides valuable information on efficiently implementing a BDD package.

The fact that minimum-cost satisfiability amounts to finding the shortest path in an ROBDD is mentioned in [Lin90]. The relation between ILP, satisfiability, the binate-covering problem and ROBDDs is discussed in [Jeo93]. The formulation of VLSI layout optimization problems in terms of satisfiability and their subsequent solution using ROBDDs is presented in [Dev89]. The NP-completeness of set covering is mentioned in [Gar79].

Regarding two-level synthesis, [Bra84] is a monograph that explains the algorithms used in the popular program Espresso. A recent review of two-level minimization techniques can be found in [Cou94]. It especially concentrates on the innovative methods proposed by its author to solve the problem exactly without enumerating all prime implicants. The methods extensively use BDDs for implicit representations of sets. The algorithm by Minato as presented in this chapter originates from [Min96]. The expansion theorem on which the algorithm is based is mentioned in [Mor70].

11.5 Exercises

- 11.1 Convert the equations of Equation (11.3) into sum-of-products expressions using the rules of Equation (11.4).
- 11.2 Find a family of Boolean functions different from the one given in Equation (11.7) whose ROBDD representation may have a size that varies from a linear to an exponential function of a complexity parameter k depending on the chosen variable order.
- 11.3 Show by means of a trace how the function `robdd.build` constructs the ROBDD for the function G in Figure 11.14.
- 11.4 Suppose that the output of function G in Figure 11.19 is connected to the x_3 input of F . Perform all required computations to obtain the ROBDD of the

composite function. Give traces of all function calls in the style used in the text.

- 11.5** Provide the missing parts of the trace of the function `isop` as shown in Figure 11.28.

12

High-level Synthesis

High-level synthesis is the process of mapping a behavioral description at the algorithmic level to a structural description in terms of functional units, memory elements and interconnections (e.g. multiplexers and buses). The functional units normally implement one or more elementary operations like addition, multiplication, etc. This step in the design process can be made visible in Gajski's Y-chart (see Figure 1.2) as illustrated in Figure 12.1.

This chapter pays attention to several aspects of high-level synthesis (but not all of them). Section 12.1 is dedicated to *hardware models*: before mapping an algorithm to hardware, one should define the type of the target hardware. Another important point in high-level synthesis is the formal description of the algorithm to be mapped. Most high-level synthesis systems use some sort of *data-flow graph* for this purpose. These graphs are discussed in Section 12.2. Prior to the actual mapping, transformations can be applied to the input description. However, they are better understood with more knowledge of the mapping process and are therefore discussed at the end of the chapter in Section 12.6. The actual mapping consists of the tasks *allocation*, *assignment* and *scheduling*. The terms are introduced in Section 12.3. A number of popular scheduling algorithms are presented in Section 12.4 while a short introduction to the assignment problem is given in Section 12.5.

12.1 Hardware Models for High-level Synthesis

It is assumed that the reader is familiar with the basic notions of digital hardware design. Here only those aspects that are relevant for high-level synthesis are briefly reviewed.

Logic circuits interact with their environments by means of input and output signals. If the outputs of a circuit are observed for different patterns of input signals and it turns out that the outputs only depend on the current inputs (but not on the previous inputs), the circuit is called *combinational*. The function of such a circuit can be completely described by a *truth table* that presents the value of the output signals for each possible combination of input signal values. If, on the other hand, the output signals are not uniquely defined for a combination of input signal values, the circuit is called *sequential*. This means that the *state* of the circuit influences

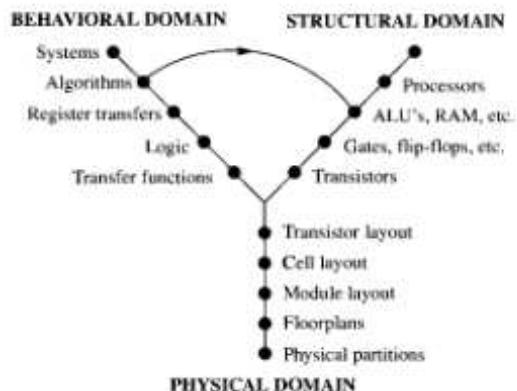


Figure 12.1 High-level synthesis as a transition in Gajski's Y-chart.

the output values, implying that the circuit has an internal memory. Sequential circuits are divided into two groups: *synchronous* or *clocked* circuits, where the state transitions can only happen on regular moments defined by one or more *clock* signals and *asynchronous* circuits, where state transitions can occur on arbitrary moments.

If no restrictions are put on the type of hardware, it becomes difficult, if not impossible, to clearly define the high-level synthesis problem. In addition, even if one succeeded to define the problem, the search space for a solution would become too large. Therefore, almost all high-level synthesis systems restrict the hardware to synchronous circuits. Also, restrictions are put on the type of hardware components used, the way they are interconnected, and the clocking strategy. All restrictions together define a high-level synthesis system's *hardware model*, the type of hardware that the system is able to generate.

12.1.1 Hardware for Computations, Data Storage, and Interconnection

A very essential hardware component is the *functional unit* (FU). This is a combinatorial or sequential logic circuit that realizes some Boolean function, such as an adder, a multiplier or an arithmetic logic unit (ALU). Figure 12.2(a) presents the symbol to be used for an FU throughout this chapter. Another essential component inherent to synchronous logic is the *register* which makes it possible to store data in the circuit (see Figure 12.2(b)). Both FUs and registers normally operate on *words*, which means that each input or output is actually realized by a number of signals carrying a *bit*. The number of bits in a word is called the *word length*.

A register is the simplest form of a *memory element*. Sometimes, using many separate registers is not very efficient, as individual wires have to be connected to each register. Registers are then combined to form so called *register files*. Registers in a register file share their input and output signals. *Address* signals indicate to which of the registers in the file the input should be written to or from which register

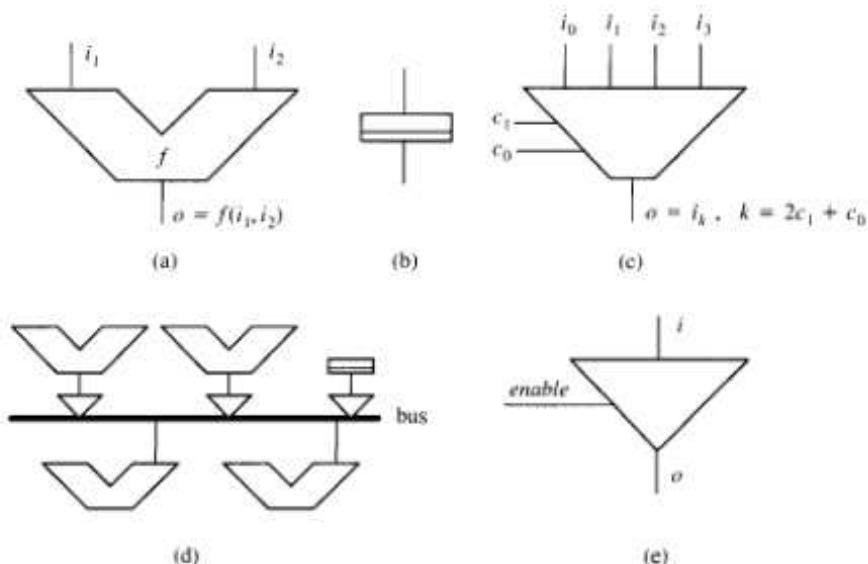


Figure 12.2 Hardware components that can be used by a high-level synthesis system: a functional unit (a), a register (b), a 4-input multiplexer (c), a configuration with a bus (d), and a three-state driver (e).

the output should be read. A register file is, in a sense, a small RAM (random access memory). In some applications, constant data are required to be present in the circuit. They can be stored in a ROM (read-only memory).

The simplest way to make connections between hardware components is by using *wires*. It often happens, however, that hardware can be used much more efficiently if it is possible to change the connections between components during a computation (e.g. an adder taking one of its inputs from register x at some moment and from register y at a later moment). One way to realize this is to use *multiplexers*: depending on the value of control signals, the output becomes equal to one of the inputs. In general n control signals can select one out of 2^n inputs. Figure 12.2(c) shows a 4-input multiplexer that requires 2 control signals. Even more efficient hardware design is possible if *buses* and *three-state drivers* (also called *tri-state* drivers) can be used. As can be seen in Figure 12.2(d), many components are connected to a bus for reading and writing. The use of buses becomes especially interesting when the word length increases. Of course, only one component at a time can write on a bus. This is accomplished by using three-state drivers to connect a component that needs to write to the bus. Depending on an *enable* signal, the three-state driver either connects its input to its output or puts its output in *high-impedance* state, which means that effectively the input and output become electrically disconnected (see Figure 12.2(e)).

12.1.2 Data, Control, and Clocks

It is a common practice to divide signals in a logic circuit into two groups: *data* signals and *control* signals. Informally, data signals carry the "operands" of the functional units. Control signals regulate the transfer of data signals between hardware components: the enable input of a three-state buffer, the control inputs of a multiplexer, the signals that select the function of an ALU, the address signals for a register file, etc. The hardware components interconnected by wires carrying data signals form the so-called *data path*. One can say that the data path is the part of the logic circuit where the actual computations are performed and their results are stored. Of course, it is also necessary to have hardware that generates the correct control signals at the correct moment. This part of the logic circuit is called the *control circuitry* or *controller*.

Most high-level synthesis systems concentrate on the generation of a data path: the term *data-path synthesis* is used in this context. The automatic synthesis of the control circuitry is normally not considered to be part of high-level synthesis. *Controller synthesis* is a field of its own that is often seen as part of logic synthesis (see also Section 2.2).

In synchronous circuits, the notion of the *system clock* is essential. The duration of a computation on an FU can be expressed in multiples of the system clock period. Such a period is also called a *control step* or a *cycle*. Some high-level synthesis systems work with a hardware model where all computations are done in one clock period. In a more general hardware model, computations are allowed to last several clock periods. These are so-called *mecycle* operations. The opposite, performing more than one computation in one clock period, is called (*operation*) *chaining*. Multicycle operations normally occupy an FU for the total duration of the computation. However, if the FU is *pipelined*, i.e. it has internal registers, the FU can receive a new input before the output belonging to the previous input has been produced. Clearly, pipelined FUs are not combinatorial circuits. Note that the designer often has the freedom of selecting the clock frequency but is limited in his or her choice of FUs. The FUs are included in some given library. Each element of the library is characterized by some operation delay expressed in absolute time (e.g. in nanoseconds). So, the choice of the clock frequency will influence the presence or absence of multicycle operations.

In the above description, it was assumed that all computations are performed during a single clock phase, which means that all computations and data transfers take place in one clock phase, while only the master-slave updates of the registers take place in the other. Sometimes *two-phase* clocks are part of the hardware model of a high-level synthesis system. Then, one part of the data transfers is performed during one clock phase and the other part is performed in the second clock phase.

Above, many possibilities have been sketched for the hardware model of a high-level synthesis system. When designing such a system, one should make motivated choices, being conscious that these choices have consequences for the actual synthesis algorithms. Examples of choices to be made are: to allow or disallow pipelined

FUs, to only have registers or register files as well, to make interconnections with or without buses, etc.

12.2 Internal Representation of the Input Algorithm

The algorithm to be synthesized by the high-level synthesis system has to be described in some way. The usual way of doing this is in textual form, by means of a formal language. This could be in a conventional programming language or a *hardware description language*. Researchers in the field have very different opinions about the language most suitable for high-level synthesis. VHDL, Verilog and Silage are examples of such languages, just to name some popular ones. It is, however, outside the scope of this text to pay attention to this point. Whatever language is used, the textual form is not appropriate for the representation of the algorithm during the process of synthesis. Clearly, text is simply a long string of symbols and the string representation does not contain any explicit structure. One would especially be interested in the representation of the parallelism present in the algorithm. It is therefore necessary to *parse* the text and transform it into a structured internal representation. It is almost generally agreed that this representation should be graph based. The graph that is used to represent an algorithm is called a *data-flow graph* (DFG).

The main goal of this section is to introduce the notion of a DFG. Before explaining the properties of a DFG, it is necessary to mention that many definitions of such a graph exist, although all of them roughly amount to the same. The different definitions may or may not be indicated by a specific name.

The distinction made in hardware between data path and control (see Section 12.1) has made some researchers define two graphs: *data-flow* and *control-flow* graphs. However, the term "data-flow graph" is also widely used when information on control is included in the graph. The latter will be the case in this text.

Data-flow graphs are closely related to *signal-flow graphs*, which are traditionally used in the field of digital signal processing (DSP). The special requirements of DSP, such as the repetitive application of the same algorithm to data arriving at fixed intervals, have led to the development of specific DFGs for the synthesis of DSP algorithms. An important class of DSP algorithms, characterized by the absence of computations that are controlled by data-dependent conditions (such as if-then-else and while constructs), is formed by the *synchronous data-flow graphs*.

12.2.1 Simple Data Flow

A data-flow graph is a directed graph $G(V, E)$. The set of nodes¹ V is subdivided in *computational* nodes, where actual computations are performed, *input* and *output*

¹ In this chapter, the term *node* will be used for data-flow graphs, whereas the term *vertex* will be used for other type of graphs.

$$\begin{aligned}x &\leftarrow a * b \\y &\leftarrow c + d \\z &\leftarrow x + y\end{aligned}$$

Figure 12.3 A short program.

nodes for the communications with the outer world, and *conditional* nodes, where data-dependent decisions are taken. Computational nodes are either *atomic* or *composite*. Atomic nodes perform elementary computations (e.g. additions), composite nodes are DFGs themselves and therefore allow the representation of hierarchy. If one wants to allow recursively defined computations at the data-flow level, recursive computations can be expressed using composite nodes.

An edge $(v_i, v_j) \in E$ transports a stream of so-called *tokens* from node v_i to v_j . One says that the edge indicates a *data dependency* between nodes v_i and v_j which means that v_j needs the result of the computation performed on v_i before it can start its own computation. A token carries data of some specific type. In the application area of high-level synthesis, data types such as bits, n -bit integers, or floating point numbers of a given precision are carried by a token. The principle of data-flow computing is that a computational node can only be active if tokens are present at each of its inputs. It *consumes* the input tokens, applies a specific computation on them and *produces* output tokens that carry the result of the computation. One says that a node *fires* when it consumes all required input tokens.

In a general data-flow model, a node may require multiple tokens to be present on an input edge before it can fire, and may produce multiple tokens at its output after firing. When multiple tokens are present on an edge, the ordering of the tokens has to be respected: a token that was produced first at one end of an edge is consumed first at the other end. The implementation of such a graph then requires *queues* or *buffers* for the edges. An important issue is that a buffer used for an edge has the right size; it certainly has to be avoided that an unbounded number of tokens are accumulated on an edge. For purposes of simplicity, it will be assumed in the rest of the text that all nodes consume a single token from each input edge and produce a single token on each output edge. Input nodes can only produce tokens, whereas output nodes can only consume tokens.

As an example, consider the DFG representation of the program given in Figure 12.3. There, the variables a , b , c and d are inputs, x and y are the labels of intermediate edges and z is an output. The DFG is shown in Figure 12.4(a). The rest of the figure illustrates the token flow. All inputs produce their tokens simultaneously at, say, time $t = 0$ (Figure 12.4(b)). As they arrive simultaneously, they are immediately consumed by the addition and multiplication nodes that will compute x and y respectively. Assuming that an addition is ready earlier than a multiplication, the input tokens for the addition node that will compute z arrive at distinct moments in time (Figure 12.4(c,d)). When both are present, they will be consumed and some time later the output token z will be computed (Figure 12.4(e)).

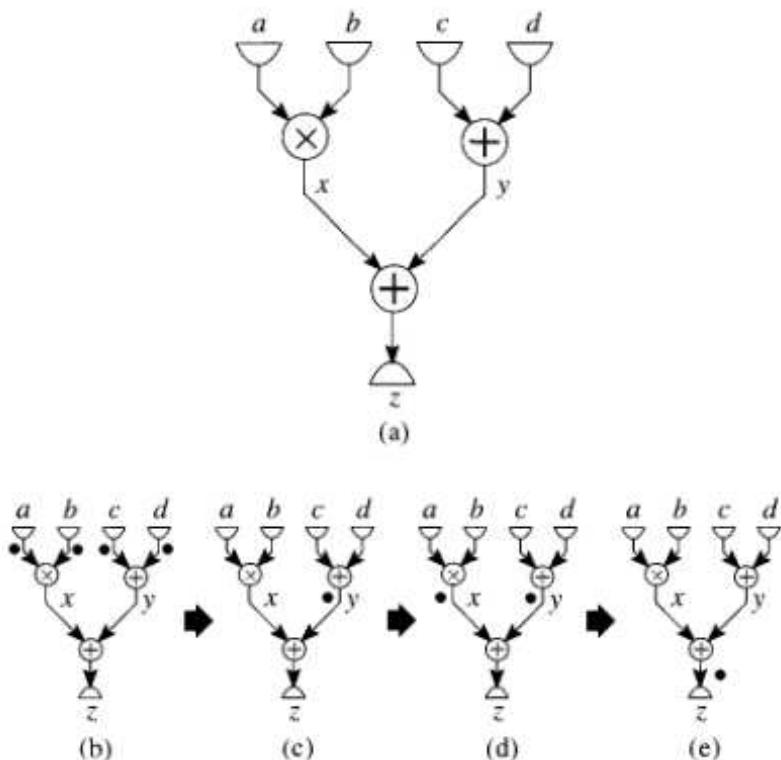


Figure 12.4 A simple DFG (a) and different stages of its execution (b-e).

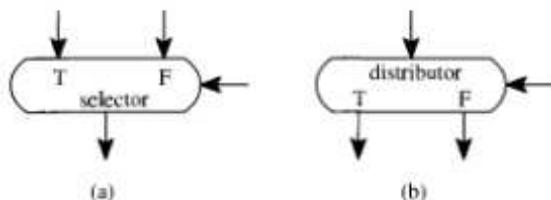


Figure 12.5 The selector node (a) and the distributor node (b).

12.2.2 Conditional Data Flow

Conditional computations require the use of two special *conditional* nodes: the *selector* node and the *distributor* node. They are shown in Figure 12.5. Both of them are characterized by a *horizontal* input that can only carry Boolean tokens. Boolean tokens can be produced by computational nodes that e.g. perform a comparison such as “less than or equal to” (\leq). A selector node has two inputs labeled *true* and *false*,

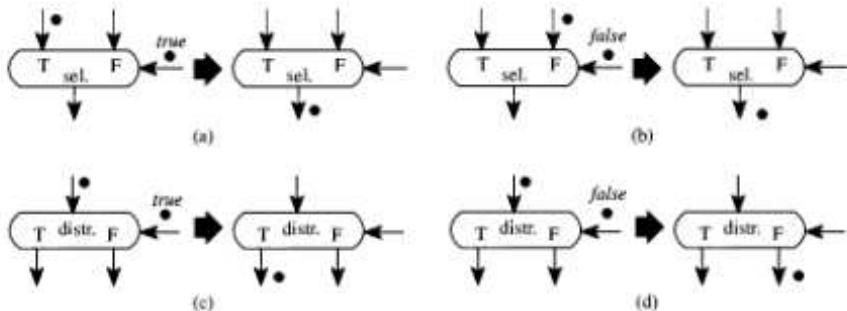


Figure 12.6 The firing rules for a selector node (a,b) and a distributor node (c,d).

```

if ( $a > b$ )
     $c \leftarrow a - b;$ 
else
     $c \leftarrow b - a;$ 

```

Figure 12.7 An example of a conditional construct.

one of which will be selected depending on the value of the token on the horizontal input. In a similar way, the horizontal input selects one of the two outputs labeled *true* and *false* of a distributor node.

Conditional nodes have firing rules that slightly differ from the firing rule for a computational node. In the case of a selector node, a node can fire if:

- there is a token with value *true* on the horizontal input and a token at the input labeled *true*; in this case, the latter token will be propagated to the output.
- there is a token with value *false* on the horizontal input and a token at the input labeled *false*; in this case, the latter token will be propagated to the output.

A distributor node can fire when both its horizontal and “vertical” inputs contain a token. However, a token is produced at only one output: if the value of the token on the horizontal input is *true* the input token is gated to the output labeled *true*; it is gated to the output labeled *false*, otherwise. The firing rules for conditional nodes have been illustrated in Figure 12.6.

Conditional nodes can be used to represent if-then-else constructs, by using combinations of distributor and selector nodes that receive the same horizontal input. An example of a conditional program fragment is shown in Figure 12.7. A corresponding DFG is shown in Figure 12.8(a). The same program fragment can also be represented in a DFG by using selector nodes only, as shown in Figure 12.8(b). The first graph makes it easier for the synthesis program to identify the conditional part of the DFG: the subgraph corresponding to the if-then-else construct is delimited by distributor nodes on one side and selector nodes on the other. Therefore, many

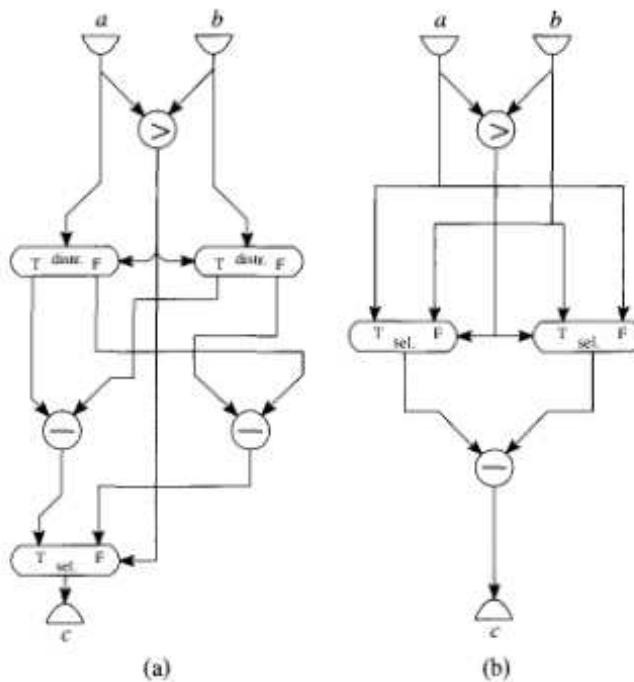


Figure 12.8 Two different DFG representations of the program fragment in Figure 12.7.

high-level synthesis systems may restrict the use of conditional nodes to structures as the one in Figure 12.8(a).

Note: as opposed to the simple DFG of Figure 12.4, the two DFGs of Figure 12.8 have nodes whose outputs are used by multiple nodes (the two input nodes and the “greater than” operator). The easiest way to represent such a situation is to have multiple edges leaving the node concerned. The consequence for the token flow is that as many tokens per computation and per output are produced as there are edges connected to the output. Each of these tokens carries identical values. For the sake of readability, the multiple outgoing edges in Figure 12.8 have been partially drawn on top of each other.

12.2.3 Iterative Data Flow

Combinations of selector and distributor nodes can also be used for the representation of iterative constructs. This can be done in many ways. However, a structured representation makes it easier to recognize these constructs in the DFG. A simple while loop is shown in Figure 12.9 as an example of an iterative construct. Its possible representation as a DFG is given in Figure 12.10. Note that an initial token with value *false* has to be present at the horizontal input of the selector node in order to allow

```
while ( $a > b$ )
     $a \leftarrow a - b;$ 
```

Figure 12.9 An example of an iterative construct.

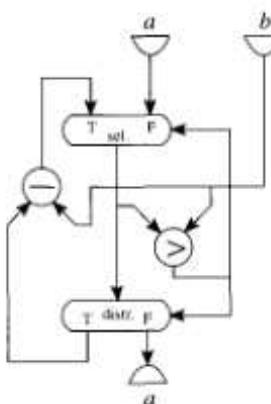


Figure 12.10 A DFG representation of the program fragment given in Figure 12.9.

the computation to start.

Note also that the representation of iteration in this way introduces cycles in the DFG. A cycle in a DFG can give rise to *deadlock*: none of the nodes in the cycle can fire as each node is waiting for its predecessor in the graph to produce a token. In the example here, the initial conditions for the conditional nodes have been chosen such that the computation can start and terminate properly. In a DFG without conditional nodes, a necessary condition for the graph to be free of deadlocks is that each cycle contains at least one edge carrying an initial token.

Apart from explicitly describing iteration as part of a DFG, there is another way of specifying a repetitive computation. This is done by having the input nodes fire repetitively: the computation is performed again each time that new inputs are available. This type of iteration is restricted to the outer loop of a computation. In spite of this restriction, it is often used in the context of digital signal processing. In many applications in this field, there is an infinite stream of input values to which the same computations should be applied yielding an infinite stream of output values.

Input values that arrive at regular time intervals introduce synchronous behavior in DFGs that are asynchronous by nature. For each set of input tokens that arrive at the start of a new iteration a corresponding set of output tokens is produced after some time. In such a synchronous system, tokens initially present on edges function as buffers that make that the output values not only depend on the current input values but also on previous ones. This is illustrated in Figure 12.11. In this figure the variables have an index related to the iteration number which they belong to. In

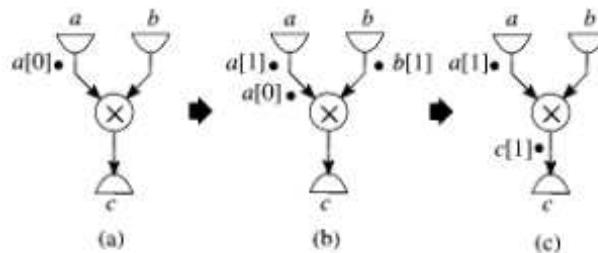


Figure 12.11 The first stages of iterative synchronous token flow in the presence of an initial token.

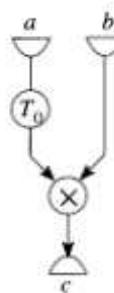


Figure 12.12 The representation of the DFG of Figure 12.11 by using a separate delay node.

DSP, it is sometimes customary to use separate *delay nodes* instead of initial tokens on the edges as shown in Figure 12.12. Such a node stores an incoming token for a period T_0 , the *iteration period* of the graph (the time interval between subsequent inputs). After this period the token is passed to the output. A delay node may be associated with a *state variable* most of the time. A small example of a DFG that is often used in the field of digital signal processing, a “second-order digital filter section”, is shown in Figure 12.13. In this example, all multiplications take one of their inputs from an input node. These input nodes are special in the sense that they should provide the same values in each iteration as they model multiplications with constant coefficients.

12.2.4 Data-flow Graph Representation

Above, it was stated that DFGs were very suitable to be used as the internal representation of the algorithm to be synthesized. It is therefore appropriate to consider the data structures by which DFGs can be modeled.

The first issue to note is that DFGs are more than ordinary graphs, although they are composed of nodes (vertices) and directed edges. The additional information to be modeled is the fact that the inputs and outputs of a node need to be distinguished.

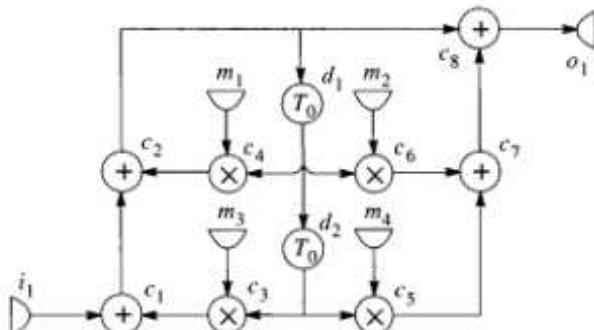


Figure 12.13 The DFG of a second-order digital filter section.

An arithmetic node like "subtraction", for example, has two inputs that cannot be interchanged without changing the algorithm represented by the DFG. Similar remarks can be made for the *true* and *false* inputs of a selector node and the *true* and *false* outputs of a distributor node. This issue can be handled by the introduction of *ports* as was done in the "cell-port-net" model for circuit representation discussed in Section 7.1. On the other hand, the "cell-port-net" is not very suitable either because edges in a DFG always connect two ports as opposed to nets that can connect to multiple ports. So, the best representation is a graph structure with two types of vertices: one for the nodes and the other for the ports respectively. Directed edges indicate the data flow between nodes as well as within the nodes. To illustrate this representation a part of the DFG of Figure 12.8(a) has been reproduced in Figure 12.14(a) while the corresponding "node-port" representation is shown in Figure 12.14(b).

As can be seen in the figure, each node can be identified by its type (input, distributor, "greater than" operation). Each port has a label (e.g. "left", "right", or "out") for unique identification. The data model should also contain additional information that is not shown in the figure. Essential information is e.g. the data type carried by an edge such as "two's complement 16-bit integer". Another thing that is useful is to internally represent "reverse edges" (for each edge a companion edge opposite to the data flow direction) such that the data structure can be traversed in any desired direction.

Note that this data model can easily be extended to deal with *hierarchical* DFGs, i.e. DFGs in which nodes can be complete DFGs themselves. Similarly to the "cell-port-net" model of Section 7.1, this can be achieved by contracting a DFG to a single "master" node. In this process, the input and output nodes of the DFG are converted into the ports of the new node.

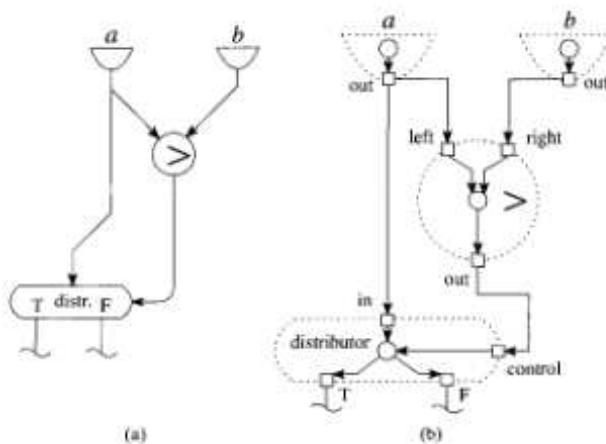


Figure 12.14 A fragment of a DFG (a) and its model for representation with data structures (b).

12.3 Allocation, Assignment and Scheduling

12.3.1 Goals and Terminology

The main issue in high-level synthesis is the mapping of the internal description of some algorithm to a hardware configuration that obeys the hardware model of the synthesis system. In the rest of this chapter, it will be assumed that the internal representation consists of a DFG. Also, it will be assumed that there are no conditional computations. Such applications are sometimes called *data dominated* as opposed to *control-dominated* applications that are characterized by conditional computations. The algorithms that are used for the synthesis of control-dominated problems are quite different from those used for data-dominated ones. They will not be discussed in this chapter.

In data-dominated problems, each operation in the DFG has to be assigned at least two entities: the functional unit (FU) on which it will be executed and the time step in which its execution will start. In addition, the remaining part of the hardware configuration has to be specified, viz. the memory elements with the values they will store at specific instants and the interconnections between all hardware units.

High-level synthesis is often divided into a number of subtasks. These tasks are normally interdependent. However, considering them as independent tasks makes it easier to define optimization problems and to design algorithms to solve them. Below, the most important subtasks are defined (actually, there is no general agreement by the community of researchers in this field on the terminology; the meanings of "assignment" and "allocation" are e.g. often interchanged with respect to those given below).

Scheduling is the task of determining the instants at which the execution of the operations in the DFG will start.

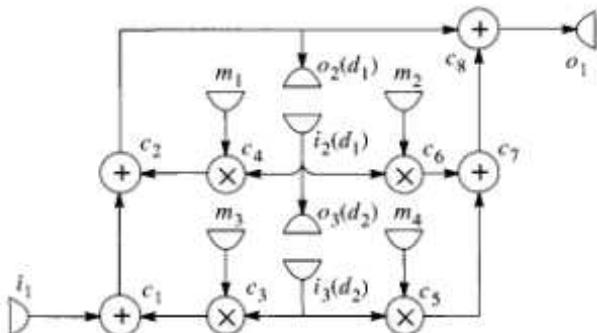


Figure 12.15 The acyclic DFG corresponding to the second-order filter section of Figure 12.13.

Assignment maps each operation in the DFG to a specific functional unit on which the operation will be executed. Assignment is also concerned with mapping *storage values* to specific memory elements and of data transfers to interconnection structures. A storage value is an intermediate result produced by the data path that needs to be stored until no operation will make use of the value anymore. Assignment is also called *binding*.

Allocation (or “resource allocation”) simply reserves the hardware resources that will be necessary to realize the algorithm. So, it determines that x units of resource type *A*, y units of resource type *B*, etc. will be used, without specifying which unit will execute which operation. Another term used for this task is *module selection*.

Before continuing with the optimization issues associated with these subtasks, the issues are introduced informally. This is done in the next subsection by means of the synthesis of a small example.

12.3.2 A Detailed Example

The example used here is the second-order filter section of Figure 12.13. Although it is possible to use the DFG as such for high-level synthesis, the synthesis becomes easier if the DFG is first transformed into an acyclic graph. To this end, a delay element is regarded as a pair of output and input nodes: the input of the delay element is an output node for the DFG and the output of the delay element is an input node. The assumption behind this transformation is that the output produced in an iteration will be stored in a register which will be read in the next one. The data is then delayed for the duration of one iteration, which is exactly the functional behavior of a delay element. The acyclic DFG obtained in this way from the DFG of Figure 12.13 is shown in Figure 12.15. Note that the acyclic graph actually models a single iteration whereas the original graph models all iterations. One says that only the *intra-iteration* parallelism, i.e. the parallelism between operations belonging to the same iteration, is represented in the acyclic DFG. In the original graph, however,

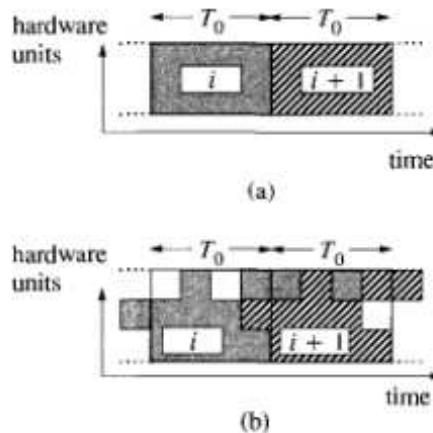


Figure 12.16 A nonoverlapped (a) and an overlapped (b) schedule for an iterative algorithm.

the *inter-iteration* parallelism or the parallelism between operations belonging to different iterations is also present.

Disregarding inter-iteration parallelism makes synthesis easier (the search space is smaller and the optimization algorithms can be simpler), but may lead to poorer results because part of the search space is not accessible to the optimization algorithm. When inter-iteration parallelism is taken into account, the search space is larger because so-called *overlapped* schedules can be generated. These are schedules where not all operations of a certain iteration need to have been executed before the next iteration can start. This is illustrated in Figure 12.16. Other names for overlapped scheduling are *loop folding* and *software pipelining*.

The synthesis of the second-order filter section based on a nonoverlapped schedule is considered first. It is assumed that only a single multiplier and a single adder are available as functional units. One could say that the decision to use this quantity of hardware was the result of the allocation subtask. The properties of the hardware are such that a multiplication takes two cycles of the internal clock and that an addition takes one cycle. Because there is only one FU of each type, the assignment task is trivial: all multiplications have to be assigned to the only multiplier present and all additions to the only adder.

One of the main issues in scheduling is that the *precedence relations* in the DFG should be respected. This means that for each edge $(v_i, v_j) \in E$ between the computational nodes v_i and v_j , v_j cannot be scheduled earlier than the time when the execution of v_i has been completed.

A schedule for the graph of Figure 12.15 is given in Figure 12.17. The figure shows a time axis of 10 clock cycles and the operations that are executed on those cycles on each functional unit. (Exercise: check that the graph cannot be scheduled in less than 10 cycles; check also that all precedence relations are respected.)

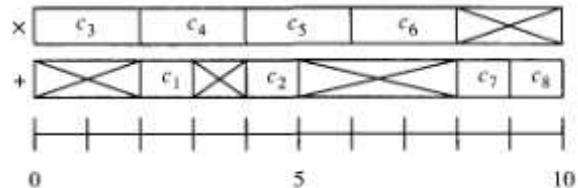


Figure 12.17 The scheduling of the additions and multiplications of the DFG of Figure 12.15.

operation	first input	second input	output
c_1	i_1	r_3	r_1
c_2	r_1	r_3	r_1, d_0
c_3	ROM	d_2	r_3
c_4	ROM	d_1	r_3
c_5	ROM	d_2	r_3
c_6	ROM	d_1	r_2
c_7	r_2	r_3	r_4
c_8	r_1	r_4	o_1

Figure 12.18 The register assignment for the example DFG given in Figure 12.15.

In order to complete the synthesis, the memory elements have to be specified and the design has to be completed by interconnecting all FUs and memory elements. In this example, it is assumed that the multiplier coefficients are stored in a ROM and that other intermediate results are stored in registers. So, specifying the memory elements amounts to performing the "register assignment" subtask. A solution is given in Figure 12.18. Apart from the registers representing the input (i_1), the output (o_1), the delay elements (d_1 and d_2), and the ROM, five registers have been introduced to store intermediate results (r_1 through r_4 and d_0). Register d_0 receives the new value to be stored in d_1 (exercise: check that the old value of d_1 still needs to be used when its new value is produced). At the end of the iteration, the value of d_0 should be copied to d_1 and the value of d_1 to d_2 .

After realizing the interconnections between the registers and the FUs and introducing multiplexers where necessary, one obtains the network as shown in Figure 12.19. (Verify the correctness of the register assignment and the network.) The actual result of the synthesis is this network augmented with the specification of the required control signals in this network. The control signals can be derived directly from the schedule and the register assignment. (Specify for each clock cycle the control signals: the control signals for the multiplexers, the write-enable signals for the registers, and the ROM addresses).

Consider now the synthesis of the second-order filter section based on an overlapped schedule. It should be based on the DFG of Figure 12.13 (the delay elements

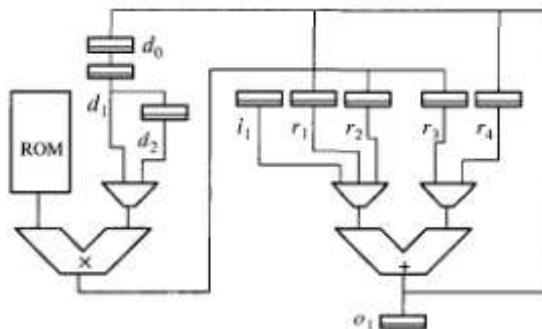


Figure 12.19 The network resulting from the synthesis of the DFG of Figure 12.15.

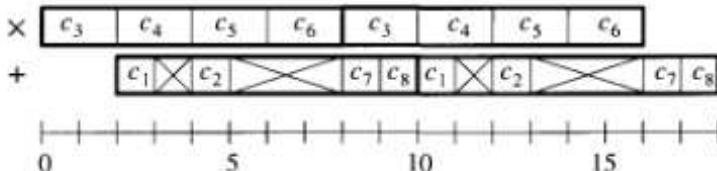


Figure 12.20 Two iterations of an overlapped schedule for the DFG of Figure 12.13.

should not be transformed into pairs of input and output nodes). Assuming that again only one multiplier and one adder are available, the solution to the scheduling problem will be as given in Figure 12.20. The figure actually shows two iterations to emphasize the overlapped nature of the schedule. From the schedule it can be seen that the computation can be repeated every 8 clock cycles compared to an iteration period of 10 in the case of the nonoverlapped schedule. The generation of the register assignment and interconnection hardware is similar to the nonoverlapped case and is not further considered here (see also Exercise 12.3).

12.3.3 Optimization Issues

The function to be optimized in high-level synthesis is in principle the same as for a VLSI design problem as a whole (see Section 1.1). In order to simplify the problem, however, all hardware units are characterized by two parameters only: *speed* and *cost*, where the cost normally mainly reflects the *area*. Another entity, the optimization of which is becoming more and more important, is *power consumption*. However, this optimization issue will be kept outside the discussion in this chapter.

By making use of these two parameters, two problems can be investigated:

- Given the time within which the hardware has to complete its computation, find the hardware configuration with minimal overall cost. The overall cost is simply

computed by adding the cost parameters of all hardware elements used. This problem is called *time-constrained* synthesis.

- Given an allocation of hardware, find a scheduling and assignment such that the total computation is completed in minimal time. This problem is called *resource-constrained* synthesis. In another version of this problem an allocation is not given, but a limit to the overall cost is specified.

There is actually a third problem that amounts to the decision version of the two optimization problems mentioned above: check whether a network can be constructed to execute an input algorithm within a given time, while limitations are given as well on either the use of individual hardware elements or the overall cost.

The problem is often further simplified by considering FUs as the only hardware elements contributing to the cost. This means that the use of memory elements and interconnection is not optimized in the first place. The optimization of these resources is done as a *postprocessing* step after a synthesis step in which only the FU cost is taken into account.

In the rest of this chapter, the following scheduling problem is given some deeper attention:

- The schedule should be nonoverlapping.
- The input for the scheduler consists of a DFG $G(V, E)$ and a library \mathcal{R} of FUs (\mathcal{R} is also called the set of resource types). $G(V, E)$ does not have conditional nodes. For each node $v \in V$, the input description also states on which library element $r \in \mathcal{R}$ this node will be executed. This mapping is denoted by a function $\rho : V \rightarrow \mathcal{R}$. Because of this unique mapping and the known characteristics of the library, a time $\delta(v)$ can be associated with each $v \in V$ indicating the number of control steps required to execute v . The cost of an element $r \in \mathcal{R}$ is given by $\omega(r)$.
- The problem is time-constrained. The hardware resulting from synthesis should complete the required computation at time T_0 at latest. This means that the time instants from the set $\mathcal{T} = \{0, 1, 2, \dots, T_0 - 1\}$ are available for an operation to start its execution (of course, an operation $v \in V$ with duration $\delta(v) > 1$, can start at time $T_0 - \delta(v)$ at latest).
- A schedule σ is a function that gives the starting time for the execution of each operation: $\sigma : V \rightarrow \mathcal{T}$. Obviously, a schedule can only be legal if the following inequality is satisfied for each edge $(v_i, v_j) \in E$ between computational nodes v_i and v_j :

$$\sigma(v_j) \geq \sigma(v_i) + \delta(v_i) \quad (12.1)$$

- Assuming that there is a "requirement function" $N_r(\sigma)$ that gives the minimal number of resources of type r that are necessary to execute the schedule σ , the goal of this problem is to find a σ such that the overall cost given by:

$$\sum_{r \in \mathcal{R}} \omega(r) N_r(\sigma) \quad (12.2)$$

is minimized.

N_r is actually a function that has to perform the assignment for each resource type separately. Depending on the variant of the synthesis problem involved, the assignment problem may be tractable or intractable, as will be explained in Section 12.5. As the scheduling problem preceding the assignment is NP-complete, the synthesis problem defined here is intractable in any case. One therefore has to be satisfied with heuristic scheduling algorithms. A number of them are discussed in the next section.

12.4 Some Scheduling Algorithms

12.4.1 ASAP Scheduling

ASAP (an abbreviation of “as soon as possible”) scheduling is one of the simplest ways to find a solution to a scheduling problem with precedence constraints. An operation in the DFG is scheduled exactly at the moment that all its predecessor nodes in the graph have completed their computations.

Finding the ASAP schedule of an acyclic DFG is equivalent to finding the longest path in a directed acyclic graph. An algorithm to find such a longest path was presented in Figure 6.8 in the context of layout compaction. In order to be able to apply this algorithm, a simple directed graph $H(\{v_0\} \cup V, F)$ derived from the DFG $G(V, E)$ should be constructed as follows:

- All edges $(v_i, v_j) \in E$ are also member of F .
- Each edge (v_i, v_j) such that v_i is a computational node, has to be assigned a weight $w((v_i, v_j)) = \delta(v_i)$, the computational delay of the computational node v_i that puts a token on the edge. Note that in this way all inequalities obeying Inequality (12.1), are represented in the graph in the same way as the inequalities of the layout compaction problem.
- An edge (v_i, v_j) , where v_i is an input node, receives weight $w((v_i, v_j)) = 0$. This actually models that all inputs are available at time 0. Inputs that arrive later can be represented by edges with an appropriate weight. Also, new edges (v_0, v_i) with weight $w((v_0, v_i)) = 0$ are added to F for each input node v_i . v_0 is a dummy source node that is required by the algorithm presented in Figure 6.8.

The values x_i found by the algorithm denote the ASAP scheduling times $\sigma(v_i)$ of the computational nodes v_i . The transformed graph of the example in Figure 12.15 is shown in Figure 12.21.

Note: the scheduling times found by ASAP scheduling are the earliest possible; the scheduling problem therefore only has a solution if for all computational nodes $\sigma(v_i) + \delta(v_i) < T_0$ (see also Subsection 12.3.3).

ASAP scheduling has the disadvantage that the algorithm nowhere refers to the resource usage. Therefore, no attempt is made to minimize the cost function of Formula (12.2). However, ASAP scheduling can be used as part of more sophisticated algorithms such as those discussed in the next subsections.

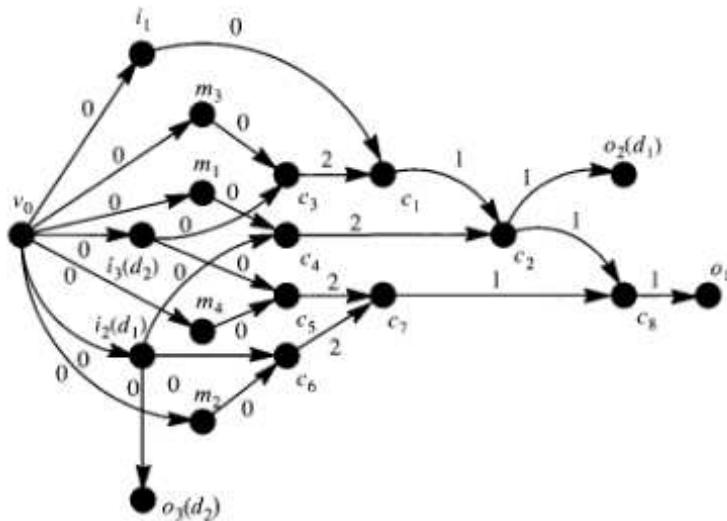


Figure 12.21 An adaptation of the graph of Figure 12.15 such that the longest-path algorithm of Figure 6.8 can operate on it.

12.4.2 Mobility-based Scheduling

In ASAP scheduling one computes the earliest time at which an operation can be scheduled. One can also do the opposite and compute the latest time at which an operation can be scheduled, knowing that all operations should be ready at time T_0 . It is not difficult to adapt the longest-path algorithm described above to work from the outputs backwards and find the “as late as possible” (ALAP) times in this way (see Exercise 12.4).

ALAP scheduling, as such, has the same disadvantages as ASAP scheduling. However, combining the information obtained by both ways of scheduling gives rise to more powerful heuristics. If the ASAP scheduling time of node v_i is denoted by $\sigma_S(v_i)$ and the ALAP time by $\sigma_L(v_i)$, the interval $[\sigma_S(v_i), \sigma_L(v_i)]$ contains all possible time instants at which v_i can be scheduled. This interval is sometimes called the *time frame* or the *scheduling range* of the operation. The length of the interval, i.e. $\sigma_L(v_i) - \sigma_S(v_i)$, is called the operation’s *mobility*.

Mobility can be used as the basis of several scheduling heuristics. Such a heuristic investigates the time instants within the scheduling range of an operation and chooses an instant in such a way that the usage of resources is optimized. Fixing the time of an operation can affect the mobility of the yet unscheduled operations. Their mobilities may decrease. Consider e.g. the simple DFG of Figure 12.22 in which the input and output nodes are connected by three nodes in series, each with a delay of 2 clock cycles and $T_0 = 10$. The initial scheduling ranges for the nodes c_1 , c_2 , and c_3 respectively are: [0,4], [2,6], and [4,8]. If the scheduling algorithm decides to fix c_2 at time 5, the ranges become: [0,3], [5,5], and [7,8].

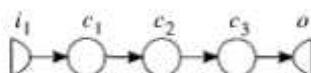


Figure 12.22 A simple DFG.

```

"determine  $\tilde{\sigma}^{(0)}$  by computing  $\sigma_S$  and  $\sigma_L$ ";  

 $k \leftarrow 0$ ;  

while ("there are unscheduled operations") {  

     $v \leftarrow$  "one of the nodes with lowest mobility";  

    "schedule  $v$  at some time that optimizes the current resource utilization";  

    "determine  $\tilde{\sigma}^{(k+1)}$  by updating the scheduling ranges  

    of the unscheduled nodes";  

     $k \leftarrow k + 1$   

}
  
```

Figure 12.23 A simple mobility-based scheduling algorithm.

When talking of scheduling ranges that are decreased in the course of an algorithm, the concept of a *partial schedule* is very useful. A partial schedule $\tilde{\sigma}$ is a function that gives a scheduling range for each computational node in a DFG: $\tilde{\sigma} : V \rightarrow [\mathcal{T}, \mathcal{T}]$. Let the lower bound of the interval for a node v be denoted by $\tilde{\sigma}_{min}(v)$ and the upper bound by $\tilde{\sigma}_{max}(v)$. Many mobility-based scheduling algorithms generate a sequence of partial schedules: $\tilde{\sigma}^{(0)}, \tilde{\sigma}^{(1)}, \dots, \tilde{\sigma}^{(n)}$. $\tilde{\sigma}^{(0)}$ gives the ASAP and ALAP schedules. In each step the mobility of at least one operation decreases, while no operation's mobility increases. The final partial schedule $\tilde{\sigma}^{(n)}$ is actually a complete schedule σ , as for all nodes $v \in V$ the following should be true: $\tilde{\sigma}_{min}^{(n)}(v) = \tilde{\sigma}_{max}^{(n)}(v) = \sigma(v)$.

The pseudo-code of a general algorithm based on these ideas is presented in Figure 12.23. This is a very general algorithm. Many details have to be elaborated before the algorithm can be applied in practice. The algorithm uses the heuristic of starting with operations of lowest mobility. This seems to be a good rule as it is easier to find a suitable scheduling time for operations of higher mobility later, when the resource usage of low-mobility operations is known. If, on the other hand, one starts with high-mobility operations, it is easier to make the wrong choice in the presence of more freedom. More criteria are needed to make a selection among the nodes with equal lowest mobility. Also criteria are needed to choose at which time an operation should be scheduled. Updating the ranges can be done using the longest-path algorithm, in a similar way as was done for computing the initial ranges (note that in Figure 6.8 the initialization occurs outside the procedure `longest-path`).

12.4.3 Force-directed Scheduling

A more sophisticated scheduling algorithm based on mobility is the *force-directed* method introduced by Paulin and Knight. It provides a heuristic solution to a problem already mentioned in Figure 12.23: how to find the best position in time for a node such that the overall resource utilization is optimized, while still many other nodes have to be scheduled. In force-directed scheduling, the resource utilizations of the nodes yet to be scheduled are estimated by assuming an "average" utilization within their scheduling ranges. For example, a unit-time operation with a mobility of three is assumed to use one third of a resource for the three time instants within its scheduling range. The combination of these averages results in a *resource requirement distribution function* that is extensively used by the scheduling algorithm. In this way, the unscheduled nodes also participate in the decision made during scheduling.

More precisely, the resource requirement distribution function for a resource r at time t , $\theta_r(\bar{\sigma}, t)$ is computed making use of all the information present in the partial schedule $\bar{\sigma}$. It is computed as follows:

$$\theta_r(\bar{\sigma}, t) = \sum_{v \in V} \alpha(v, r, \bar{\sigma}, t)$$

In the formula, the contributions of individual operations to the resource requirement distribution, given by the auxiliary function α , are added together. When $\rho(v) \neq r$, obviously $\alpha(v, r, \bar{\sigma}, t) = 0$. Otherwise, α is defined as follows:

$$\alpha(v, r, \bar{\sigma}, t) = \sum_{i=\bar{\sigma}_{\min}(v)}^{\bar{\sigma}_{\max}(v)} \sum_{j=0}^{\delta(v)-1} \xi(i+j, t) \times \frac{1}{\bar{\sigma}_{\max}(v) - \bar{\sigma}_{\min}(v) + 1}$$

where:

$$\xi(a, b) = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{if } a \neq b \end{cases}$$

The function α can be understood as follows: for a unit-delay operation, its requirement distribution is uniform within its scheduling range; for an operation of longer duration, the requirement distribution is computed as if the operation was composed of a concatenation of unit-delay operations. Figure 12.24 shows a simple DFG and the computation of the resource requirement distribution functions for the initial schedule $\bar{\sigma}^{(0)}$ when $T_0 = 4$, $\delta(+)=1$, and $\delta(\times)=2$.

Given a partial schedule $\bar{\sigma}^{(k)}$, force-directed scheduling makes a transition to schedule $\bar{\sigma}^{(k+1)}$ by selecting a node v which is scheduled at a time t , in a way similar to what is done in the simple algorithm of Figure 12.23. In order to find the best choices for v and t , force-directed scheduling evaluates all possibilities. This is done by computing the "force" associated with a possible transition.

Let $\bar{\sigma}|_{v \rightarrow t}$ denote the schedule obtained from $\bar{\sigma}$ by fixing operation v at a time t within its scheduling range. Then the force associated with the transition is given by:

$$F(\bar{\sigma}|_{v \rightarrow t}, \bar{\sigma}) = \sum_{s \in T} \theta_r(\bar{\sigma}, s) [\alpha(v, \rho(v), \bar{\sigma}|_{v \rightarrow t}, s) - \alpha(v, \rho(v), \bar{\sigma}, s)] \quad (12.3)$$

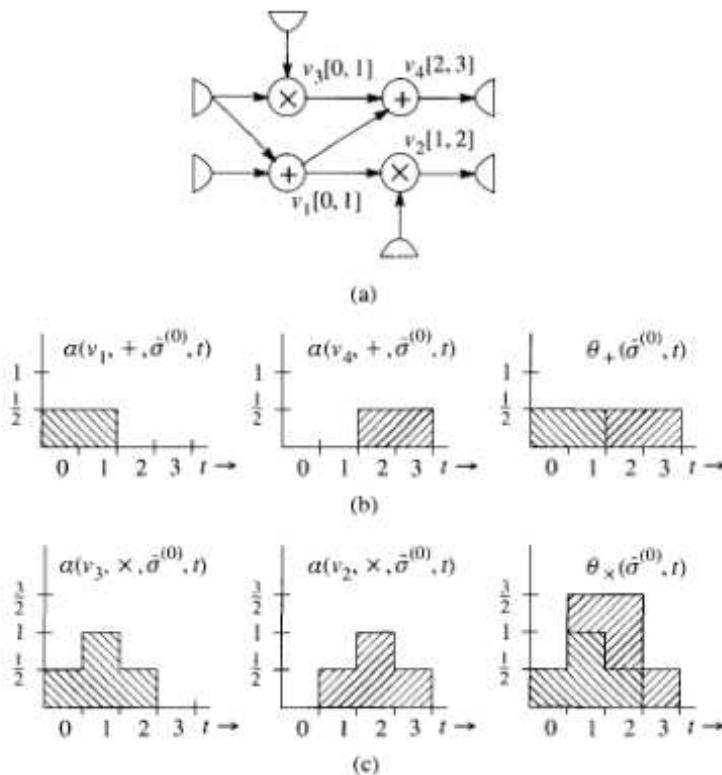


Figure 12.24 A simple DFG (a) and its initial resource requirement distribution functions for the resources + (b) and \times (c).

The transition chosen is the pair (v, t) that results in the minimal force.

Inspection of the terms that are summed in Equation (12.3) shows that, for each instant s , the force is composed of the product of the current distribution and the change of the distribution for the operation that is tentatively scheduled at time t . One could interpret each term as the product of a spring constant (the distribution) with a displacement (the change of the distribution for the operation under investigation). In analogy with Hooke's Law, one could call the product a force, hence the name of the method.

One can understand Equation (12.3) as follows: the change in the distribution of the operation v due to its fixing at time t is weighted with the total distribution. For this reason, the contribution to the force of those time instants s with a higher distribution value will be more significant: a decrease in the distribution of v at s due to the fact that v is fixed at a time instant $t \neq s$, will be more strongly rewarded, while an increase due to $t = s$ will be more strongly penalized. In this way, nodes will be scheduled at instants where the expected resource utilization is relatively low.

Consider the scheduling of the multiplications in the example of Figure 12.24,

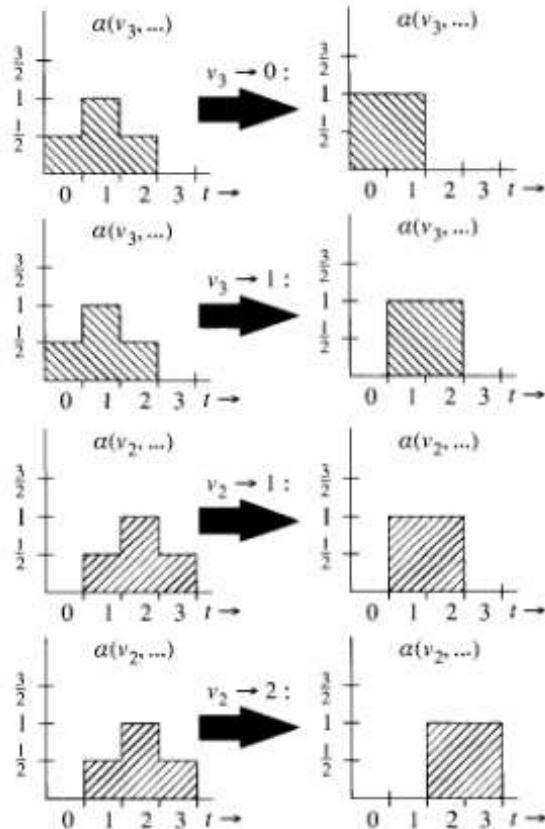


Figure 12.25 The changes in the values of the function α for the tentative scheduling to be considered when applying the force-directed method to the situation in Figure 12.24.

in which the following (v, t) pairs should be investigated: $(v_3, 0)$, $(v_3, 1)$, $(v_2, 1)$, and $(v_2, 2)$. The changes to the distributions of the nodes concerned are shown in Figure 12.25. The forces associated with the four tentative scheduling are given below:

$$\begin{aligned} F(\tilde{\sigma}^{(0)}|_{v_3 \rightarrow 0}, \tilde{\sigma}^{(0)}) &= \frac{1}{2} \times \frac{1}{2} + \frac{3}{2} \times 0 + \frac{3}{2} \times -\frac{1}{2} + \frac{1}{2} \times 0 = -\frac{1}{2} \\ F(\tilde{\sigma}^{(0)}|_{v_3 \rightarrow 1}, \tilde{\sigma}^{(0)}) &= \frac{1}{2} \times -\frac{1}{2} + \frac{3}{2} \times 0 + \frac{3}{2} \times \frac{1}{2} + \frac{1}{2} \times 0 = \frac{1}{2} \\ F(\tilde{\sigma}^{(0)}|_{v_2 \rightarrow 1}, \tilde{\sigma}^{(0)}) &= \frac{1}{2} \times 0 + \frac{3}{2} \times \frac{1}{2} + \frac{3}{2} \times 0 + \frac{1}{2} \times -\frac{1}{2} = \frac{1}{2} \\ F(\tilde{\sigma}^{(0)}|_{v_2 \rightarrow 2}, \tilde{\sigma}^{(0)}) &= \frac{1}{2} \times 0 + \frac{3}{2} \times -\frac{1}{2} + \frac{3}{2} \times 0 + \frac{1}{2} \times \frac{1}{2} = -\frac{1}{2} \end{aligned}$$

The force-directed scheduling algorithm will choose for either the pair $(v_3, 0)$ or the

```

    "determine  $\tilde{\sigma}^{(0)}$  by computing  $\sigma_S$  and  $\sigma_L$ ";  

     $k \leftarrow 0$ ;  

    while "there are unscheduled operations" {  

        for each  $r \in \mathcal{R}$   

            "compute  $\theta_r(\tilde{\sigma}^{(k)}, t)$ ";  

        for each "unscheduled"  $v \in V$  {  

            for ( $t \leftarrow \tilde{\sigma}_{min}^{(k)}(v); t \leq \tilde{\sigma}_{max}^{(k)}(v); t \leftarrow t + 1$ ) {  

                "compute  $F(\tilde{\sigma}^{(k)}|_{v \rightarrow t}, \sigma^{(k)})$ ";  

                if ("this force is smaller than previous ones") {  

                     $v_{best} \leftarrow v$ ;  

                     $t_{best} \leftarrow t$ ;  

                }  

            }  

             $\tilde{\sigma}^{(k+1)} \leftarrow \tilde{\sigma}^{(k)}|_{v_{best} \rightarrow t_{best}}$ ;  

             $k \leftarrow k + 1$ ;  

    }
}

```

Figure 12.26 The force-directed scheduling algorithm.

pair $(v_2, 2)$, which in both cases will avoid the allocation of a second multiplier.

The main ideas of force-directed scheduling, as presented above, have been summarized in the pseudo-code of Figure 12.26. From the code, the worst-case time complexity of the algorithm can be derived. The force computations dominate the calculation and require $\mathcal{O}(n^2 t_{max}^2)$ time, where n is the number of nodes in the DFG.

In Equation (12.3), only the change in the distribution of v resulting from the tentative scheduling of v at t is taken into account. This is only a rough approximation of changes in the resource requirement distribution function. In reality, such a tentative scheduling will also affect the scheduling ranges of other unscheduled operations. The original description of Paulin and Knight therefore not only considers the force as given in Equation (12.3), called by them the *self-force*, but also the changes in the distributions of the nodes incident to and incident from v . These changes result in the so-called *predecessor* and *successor* forces. It is beyond the scope of this text to discuss the details of these force computations.

Although force-directed scheduling has been used by many researchers, it has a higher time-complexity than other heuristics without guaranteeing better results. In particular, heuristics based on *list scheduling* should be considered as an alternative to force-directed scheduling. A short description of list scheduling follows in the next subsection.

12.4.4 List Scheduling

The scheduling methods described until now (ASAP, mobility-based and force-directed scheduling) are meant for time-constrained problems, although they could be adapted for resource-constrained problems as well. *List scheduling*, on the other

hand, is meant for resource-constrained problems in the first place, but can also be adapted for time-constrained problems.

The main idea is to process the available time instants in increasing order (starting at zero) and schedule as many as possible operations at a certain instant before moving to the next. This continues until all operations have been scheduled. In this way an attempt is made to minimize the total execution time of the operations in the DFG. Of course, the precedence relations also need to be respected in list scheduling. All operations, whose predecessors in the DFG have completed their executions at a certain time instant t are put in the so-called *ready list* L_t . The list contains exactly those operations that are available for scheduling.

If there are sufficient unoccupied resources at time t for all operations in L_t , these operations can be scheduled trivially. However, if an appropriate resource for each operation in L_t is not available, a choice has to be made on which operations will be scheduled at time t and which operations will be deferred to be scheduled at a later time. This choice is normally based on heuristics, each heuristic defining a specific type of list scheduling.

A popular version is *critical-path* list scheduling. In this context, the longest path from a node $v \in L_t$ to an output node of the DFG is called its critical path (note that the critical path of the DFG as a whole is the longest path starting from any of the input nodes and ending in any of the output nodes). The maximum over all $v \in L_t$ of the critical-path lengths gives a lower bound on the total time necessary to execute the remaining part of the schedule. Therefore, in critical-path list scheduling, nodes with the greatest critical-path lengths are selected to be scheduled at time t . The method is illustrated in Figure 12.27, where the DFG shown should be scheduled on two identical ALUs that require one time unit to compute an addition or subtraction and two time units to compute a multiplication. The critical-path length for each node is given between parentheses behind the node name. The intermediate values of L_t during the execution of the algorithm are given in Figure 12.28. Note that the elements in the lists are sorted according to their critical-path lengths and that therefore the first two elements are selected for scheduling whenever the list contains more than two elements.

A reason for the popularity of critical-path list scheduling is that often satisfactory results are obtained while the algorithm has a low time complexity. The critical-path lengths for each node need only to be computed once, which can be done in linear time traversing the graph from the output nodes to the inputs in a way similar to the longest-path algorithm presented in Figure 6.8, resulting in a time complexity of $\mathcal{O}(|E|)$. Sorting the list of all nodes by critical-path length can also be done only once, in $\mathcal{O}(n \log n)$ time. Composing the ready list and selecting its first elements requires $\mathcal{O}(nT_0)$ time in the worst case when implemented in a straightforward way. So, the overall worst-case time complexity is $\mathcal{O}(|E| + n \log n + nT_0)$.

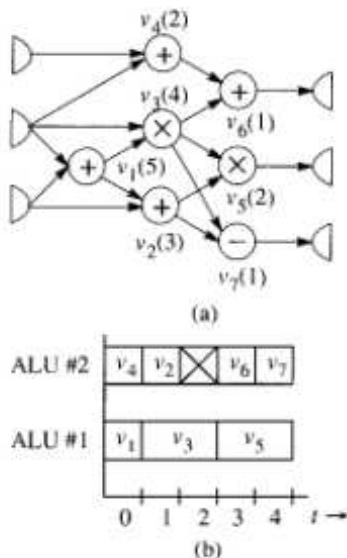


Figure 12.27 A DFG (a) and its schedule obtained by critical-path list scheduling when two ALUs are available (b).

t	L_t
0	{ $v_1(5)$, $v_4(2)$ }
1	{ $v_3(4)$, $v_2(3)$ }
2	\emptyset
3	{ $v_5(2)$, $v_6(1)$, $v_7(1)$ }
4	{ $v_7(1)$ }

Figure 12.28 The evolution of the set L_t in the critical-path list scheduling of the problem from Figure 12.27.

12.5 Some Aspects of the Assignment Problem

12.5.1 Optimization Issues

As mentioned earlier, the scheduling and assignment tasks are interrelated. For an optimal design they should be solved simultaneously. Some synthesis systems use algorithms that do so. However, most systems first solve the scheduling problem and then try to find a good assignment given a certain schedule. There also exist systems that first solve the assignment problem and schedule afterwards. In the rest of this text, it will be assumed that assignment follows scheduling.

The assignment problem itself consists of several subproblems:

- *operation-to-FU assignment*: this is the problem of mapping a computation to an FU of an appropriate type.

- *value grouping*: this is the problem of partitioning all storage values in such a way that a subset does not contain values that are read or written simultaneously. Then each subset can be realized as a register bank. In the case of *multiport memories*, the conditions for grouping should be adapted accordingly (a multiport memory has several ports that allow to perform more than one read or write action in a single clock cycle).
- *value-to-register assignment*: this is the problem of assigning a memory location to storage values in the same group. Values with nonoverlapping *life times* can share the same location. The life time of a storage value is the time interval starting at the instant that it is created, and ending at the moment that it no longer is required.
- *transfer-to-wire assignment*: a transfer is the actual transport of data from one hardware unit to another. In a bus-based architecture, one has the choice of which bus to write. The choice affects the number of three-state drivers connected to the unit from which the transfer originates and the type of multiplexers connected to the unit receiving the transfer.
- *wire to FU-port assignment*: in the case of commutative operations, one can choose one of the two equivalent input ports to feed the data to the functional unit. More in general, the problem exists when a functional unit has ports that are functionally equivalent.

Again, the complete solution space can only be explored by considering all problems simultaneously. The problem is quite complex, however, so that only heuristic methods can generate solutions for problem instances of practical size. The alternative of considering the different subproblems individually and possibly solving them exactly, may, therefore, be a better alternative.

Assuming that assignment is performed after scheduling, the number of clock cycles that the algorithm needs cannot be influenced anymore. The optimization goal is now primarily the total chip area. Another goal could be the performance (the length of the interconnections, the number of multiplexers connected to a bus, etc. can affect the length of the clock cycle). Interconnection cost is one of the items to be minimized during assignment. It includes the number of multiplexers and three-state drivers, but certainly also the bus length. As the length is difficult to estimate, most systems do not take it into account. There exist a few systems, however, that do some kind of *floorplanning* (see Chapter 8) during assignment.

12.5.2 Graph Theoretical Problem Formulation

Problems like operation-to-FU assignment and value-to-register assignment, when looked upon in a more abstract way, are variants of the same problem. This problem will be called *task-to-agent* assignment, where a *task* can be an operation or a value and an *agent* can be an FU or a register. Tasks are called *compatible* if they can be executed on the same agent. In case of values, for example, this means that they are compatible when their life times do not overlap. The set of tasks can be used as

the vertex set of a so-called *compatibility graph* $G_c(V_c, E_c)$. The graph has edges $(v_i, v_j) \in E_c$ if and only if the tasks corresponding to $v_i \in V_c$ and $v_j \in V_c$ are compatible.

Alternatively, one can say that two tasks are in *conflict* if they cannot be executed on the same agent. The set of tasks is then used as the vertex set of a *conflict graph* that has edges for those vertex pairs that are in conflict. The conflict graph is the *complement graph* of the compatibility graph. This means that the conflict graph can be obtained from the compatibility graph $G_c(V_c, E_c)$ by taking the complete graph of the vertex set V_c and removing all edges in E_c from this graph. Clearly, taking the complement of the conflict graph gives back the compatibility graph.

The goal of the assignment problem is to minimize the number of agents for the given set of tasks. This goal can be formulated either in terms of the compatibility graph or the conflict graph as is explained below.

The vertices of any complete subgraph of a compatibility graph correspond to a set of tasks that can be assigned to the same agent. The goal of the assignment problem is then to partition the compatibility graph in such a way that each subset in the partition forms a complete graph and the number of subsets in the partition is minimal. The subsets are pairwise disjoint and the union of the subsets forms the original set by definition of a partition. In the literature such a partitioning is called a *clique partitioning* even when the subsets in the partition are not necessarily cliques, but only complete subgraphs. A term that is mathematically more precise is a "covering by disjoint completely connected sets".

In a conflict graph, on the other hand, adjacent vertices in the graph have to be assigned to different agents. Minimizing the number of agents then amounts to finding the *minimal coloring* of the conflict graph. Vertices receiving the same color can be assigned to the same agent.

The notions presented in this section are illustrated in Figure 12.29. Figure 12.29(a) shows an example of a compatibility graph, while the corresponding conflict graph is shown in Figure 12.29(b). A possible minimal clique covering of the compatibility graph consists of the three vertex sets $\{v_1, v_2, v_6\}$, $\{v_3, v_4\}$ and $\{v_5\}$ as is shown in Figure 12.29(c). Not surprisingly, the minimal coloring of the conflict graph needs three colors as shown in Figure 12.29(d).

At first sight it may seem that there is no advantage of formulating one and the same problem as two different graph theoretical problems, especially because solving one of the problems automatically implies a solution for the second as well. Both the minimal clique covering problem and the graph vertex coloring problem are NP-complete in the case that the graphs concerned do not belong to special families. It may, however, be that the conflict graph is an interval graph, which leads to an optimal solution in polynomial time as explained in Section 12.5.3 following next. On the other hand, an algorithm such as the one presented in Section 12.5.4 can better be explained in terms of the compatibility graph.

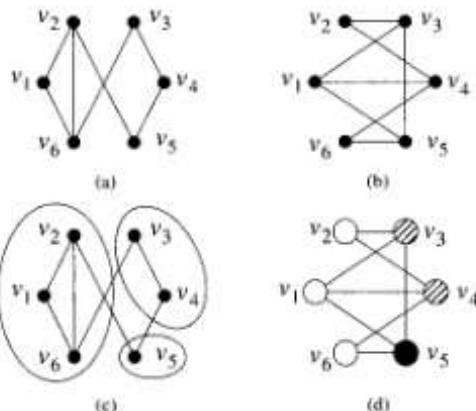


Figure 12.29 A compatibility graph (a), its corresponding conflict graph (b), a minimal clique partitioning of the compatibility graph (c) and a minimal coloring of the conflict graph (d).

12.5.3 Assignment by Interval and Circular-arc Graph Coloring

Consider the operator-to-FU assignment subproblem. If the goal of this subproblem were to minimize the number of FUs *without* considering the impact on register and interconnect cost, then the problem can optimally be solved in polynomial time in the case of *nonoverlapped* scheduling. As the times that an operation starts and terminates are known after scheduling and each operation thus has an execution interval, the problem amounts to “packing intervals” in as few as possible rows. This problem can be solved optimally in polynomial time by the *left-edge algorithm*, first proposed for the channel routing problem (see Section 9.3.3). Stated in graph theoretical terms, the intervals can be thought as vertices in an *interval graph*. This interval graph is a conflict graph, which means that overlapping intervals give rise to an edge in the graph. Finding the minimal number of FUs then amounts to coloring the graph with a minimal number of colors. The left-edge algorithm guarantees an optimal solution in polynomial time but cannot take into account “weights” that might be used to express the impact on interconnection of a specific assignment.

In the case of *overlapped* scheduling, however, the time axis is folded back on itself. This means that execution intervals at the end of the iteration period may overlap with those at the beginning (of the next iteration period). Intervals of this type can be visualized as arcs of a circle as has been done in Figure 12.30(a). In order to keep the figure readable, the arcs have been drawn on separate concentric circles. Again a conflict graph can be constructed based on the overlapping relation (in the figure, one should project the arcs radially on the inner circle to detect whether two arcs overlap). Such a graph is called a *circular-arc graph*. Figure 12.30(b) shows the circular-arc graph derived from the set of arcs in Figure 12.30(a). The minimal coloring of this graph is given in Figure 12.30(c). The minimal solution requires three agents: one for the tasks corresponding to v_1 and v_4 , one for those corresponding to

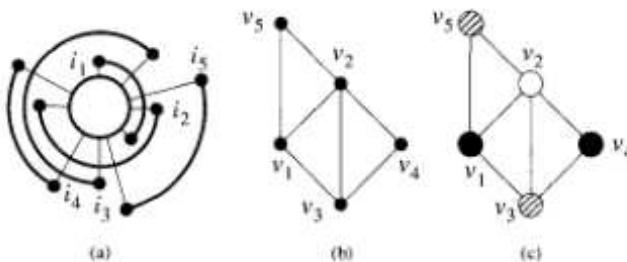


Figure 12.30 A set of circular arcs (a), its corresponding circular-arc graph (b) and the minimal coloring of this graph (c).

v_3 and v_5 and one for v_2 .

Note that a graph can be constructed from any set of circular arcs, but that the reverse is not true in general: a given graph may or may not have a set of corresponding circular arcs. A similar statement can be made about interval graphs. So, the set of all circular-arc graphs and the set of all interval graphs are strict subsets of the set of all possible graphs (and the set of all interval graphs is a subset of the set of all circular-arc graphs). Because of their specific properties it may be that, for some graph problems, algorithms can be designed that are more efficient than algorithms meant for all possible graphs.

Even when the coloring problem for circular-arc graphs looks similar to the coloring problem for interval graphs at first sight, it is an NP-complete problem. No specific algorithm for coloring circular-arc graphs will be discussed here. One possibility is to take the complementary compatibility graph and then use the “clique partitioning” approach presented in the next section.

The problem of value-to-register assignment is similar to the operator-to-FU assignment problem. However, one should be aware that a nonoverlapping schedule of an iterative algorithm still will require that some values will be alive across loop boundaries. So, the problem will still be a circular-arc graph coloring problem.

12.5.4 Assignment by Clique Partitioning

This section explains a technique called *clique partitioning* that can be applied to almost any subproblem of the assignment task (see Section 12.5.1). The problem of finding a clique partitioning of a graph is NP-complete. A heuristic algorithm to solve the problem was proposed by Tseng and Siewiorek and will be presented here. It is based on combining vertices in the compatibility graph step by step. The resulting vertices are called *supervertices*. The index i of a supervertex v_i represents the set of indices of the vertices from which the supervertex was formed. For example, combining vertices v_1 , v_3 and v_7 gives a supervertex $v_{1,3,7}$. By definition, the original vertices of the compatibility graph will also be called supervertices. If the intermediate graph after k steps of the algorithm is called $G_c^k(V_c^k, E_c^k)$, the initial graph $G_c^0(V_c^0, E_c^0)$ is equal to the compatibility graph.

```

 $k \leftarrow 0;$ 
 $G_c^k(V_c^k, E_c^k) \leftarrow G_c(V_c, E_c);$ 
while ( $E_c^k \neq \emptyset$ ) {
    "find  $(v_i, v_j) \in E_c^k$  with largest set of common neighbors";
     $N \leftarrow$  "set of common neighbors of  $v_i$  and  $v_j$ ";
     $s \leftarrow i \cup j;$ 
     $V_c^{k+1} \leftarrow V_c^k \cup \{v_s\} \setminus \{v_i, v_j\};$ 
     $E_c^{k+1} \leftarrow \emptyset;$ 
    for each  $(v_m, v_n) \in E_c^k$ 
        if ( $v_m \neq v_i \wedge v_m \neq v_j \wedge v_n \neq v_i \wedge v_n \neq v_j$ )
             $E_c^{k+1} \leftarrow E_c^{k+1} \cup \{(v_m, v_n)\};$ 
    for each  $v_n \in N$ 
         $E_c^{k+1} \leftarrow E_c^{k+1} \cup \{(v_n, v_s)\};$ 
     $k \leftarrow k + 1;$ 
}

```

Figure 12.31 A heuristic to compute the clique partitioning of a graph.

The algorithm looks for a pair of supervertices with the largest number of *common neighbors*. A supervertex $v_n \in V_c^k$ is a common neighbor of the supervertices $v_i, v_j \in V_c^k$, if both edges (v_i, v_n) and (v_j, v_n) are included in E_c^k . The two supervertices are combined into a new supervertex as described in the pseudo-code of Figure 12.31. The new supervertex remains connected to the common neighbors only. The algorithm goes on to combine supervertices until the graph $G_c^k(V_c^k, E_c^k)$ has an empty edge set.

A problem here is how to break ties when there are several pairs of supervertices with the largest set of common neighbors. One could work with priorities or edge weights to express a preference of one combination above another. Such a preference scheme is based on the actual problem that one wants to solve: a specific combination of tasks on the same agent might e.g. require more multiplexers than another combination. In the example shown in Figure 12.32, which illustrates the application of the algorithm to some example graph, ties have been broken by giving preference to the lowest vertex index.

Clique partitioning can be used for both nonoverlapped and overlapped scheduling. This is due to the global nature of the method: it simply considers conflicts between tasks, it does not process the resources in increasing order of time instants as e.g. the left-edge algorithm.

12.6 High-level Transformations

Until now, it has been assumed in this chapter that the DFGs used for the input description of high-level synthesis could not be manipulated. Although it was stated in Section 12.2 that a DFG explicitly represents *all* parallelism present in a computation, this does not mean that there is a unique DFG for each computation (an example

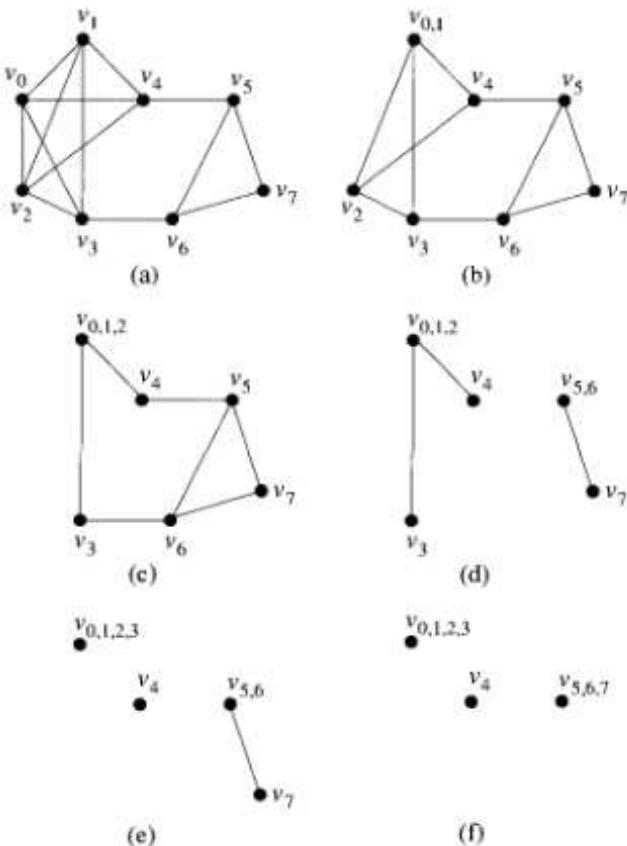


Figure 12.32 The successive graphs obtained in the application of the clique partitioning heuristic to an example graph.

of two different DFGs for the same computation was given in Figure 12.8). On the other hand, it should be clear from the discussion in the previous sections that modifications of the DFG may affect the result of scheduling and assignment and therefore the quality of the final design. For this reason, *transformations* are normally applied to a DFG before performing steps like scheduling and assignment.

There are many different types of transformation, of which a few that are applicable to data-dominated applications will be discussed here. First of all, there are simple arithmetic transformations based on “commutativity”, “associativity” and “distributivity”. Addition and multiplication are examples of *commutative* operators, which e.g. means that $a + b = b + a$. Such a transformation may lead to savings in interconnection hardware like multiplexers during assignment.

For an illustration of *associativity*, consider the addition of 8 operands given

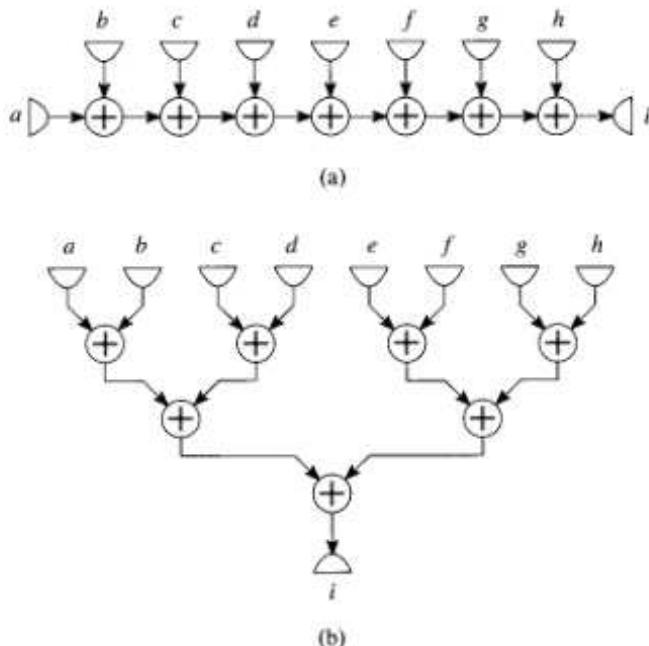


Figure 12.33 The most serial (a) and most parallel (b) DFG for an eight operand addition.

below:

$$a + b + c + d + e + f + g + h$$

When translating this addition into a DFG using addition nodes with two operands, one could interpret the expression as:

$$((((((a + b) + c) + d) + e) + f) + g) + h$$

or as:

$$((a + b) + (c + d)) + ((e + f) + (g + h))$$

This is the consequence of the associativity of addition. The DFGs corresponding to the two interpretations are shown in Figure 12.33. The "most serial" version of the computation shown in Figure 12.33(a) has a critical path of seven additions whereas the "most parallel" one of Figure 12.33(b) has a critical path of three additions. However, the latter form is not always the most preferable one. This is especially the case if the DFG of this addition is embedded in a DFG of a larger computation. It could then happen that the critical path of the overall computation becomes longer when transforming the serial form into the parallel form (suppose for example that the data at input h are available much later than the other inputs). In another situation, e.g.

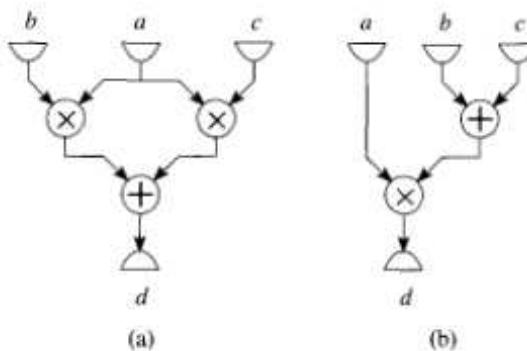


Figure 12.34 A DFG transformation corresponding to the distributivity law.

one with a resource constraint of a single adder, the parallel version will require more registers for the storage of intermediate results. The conclusion is that the application of associativity can lead to improvements in the quality of the final result, but that the choice between the most serial, the most parallel and any of the intermediate forms strongly depends on the precise synthesis circumstances.

Distributivity states that:

$$a \times b + a \times c = a \times (b + c) \quad (12.4)$$

as illustrated in Figure 12.34. At first sight the situation of Figure 12.34(b), that has one multiplication less than the situation of Figure 12.34(a), seems to be preferable. However, also here the embedding of this DFG in a larger one may make the DFG with the two multipliers a better choice (because no multiplier is available at the end of the iteration period whereas there is time and hardware to perform the two multiplications at the beginning).

Although Equation (12.4) is perfectly valid from a mathematical point of view, it does not necessarily hold at the moment that the computations occurring in the expressions are performed on hardware. This is due to the fact that numbers have to be represented by a finite number of bits and that the outputs of a multiplication or addition require more bits than the inputs if no precision is to be lost. In applications like digital signal processing where “state variables” are repetitively recomputed by additions and multiplications, it is inevitable to perform some kind of rounding or truncation after each arithmetic operation. It may therefore be that the so-called “finite word-length effects” are different before and after the application of the distributivity transformation (because the additions and multiplications are performed in a different order).

Because of the effects just mentioned, it is even doubtful whether distributivity belongs to the category of high-level transformations. It is rather an *algorithmic* transformation, a type of transformation that can be applied by the algorithm designer. Of course, some interaction between high-level design and algorithm design

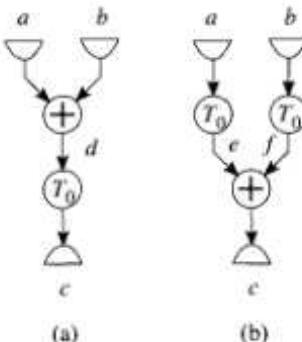


Figure 12.35 The retiming transformation.

is always possible to see whether a specific transformation like distributivity is acceptable. There exist many more algorithmic transformations that e.g. can be used to increase the parallelism in the computation in order to get faster implementations or to reduce the dissipated power. They are, however, outside the scope of this chapter.

In Section 12.3.2 it was explained that delay nodes should be split into pairs of input and output nodes when generating a nonoverlapped schedule for the algorithm. The transformation to be discussed next changes the position of the delay nodes in the DFG and therefore may affect the length of the critical path of the DFG (the longest path from any input to any output node). Clearly, the critical path gives the minimal iteration period of a nonoverlapped schedule in the absence of resource constraints. The transformation is called *retiming*. Simply stated, the transformation is based on the fact that performing some operation in the DFG and then delaying its output is equivalent to performing the same operation on the delayed inputs. This is illustrated in Figure 12.35, where the graphs used are in principle parts of larger graphs. It is not difficult to see that the transformation is valid. In the derivation, the signals will be indexed by a time instant as was done in Section 12.2.3. The following holds in Figure 12.35(a): $c[n] = d[n - 1]$ and $d[n] = a[n] + b[n]$, which means that $c[n] = a[n - 1] + b[n - 1]$. In Figure 12.35(b), the situation is: $c[n] = e[n] + f[n]$, $e[n] = a[n - 1]$, $f[n] = b[n - 1]$, and therefore $c[n] = a[n - 1] + b[n - 1]$. A similar derivation could be given for any node in the DFG with any number of inputs or outputs.

As stated before, retiming can be used to optimize the critical path of a DFG for nonoverlapping schedules. It can also be used to achieve a better distribution of the resources over the iteration period (e.g. to improve a situation where all multiplications are performed at the beginning of the iteration period and all additions at the end). Actually, retiming is only useful for nonoverlapped scheduling. Most overlapped scheduling methods (not discussed here) in a sense consider all retimed versions of a DFG simultaneously, because they do not split delay nodes into pairs of input and output nodes.

A final issue to discuss is the strategy for applying the transformations. Some

systems are interactive and let the user choose which transformations should be applied on which part of the DFG. If one wants to apply transformations without user interaction, one should first be aware of the computational complexity of the problems involved. An example of a problem that can be solved in polynomial time is "retiming for the optimization of the critical path". However, "retiming for the optimization of resources" turns out to be NP-complete. In such a case, one could use a method like simulated annealing (see Section 5.6) to search for a good solution. The "moves" for simulated annealing can be chosen from a set of transformations that can be as large as wanted. The cost function could e.g. be based on the resource requirement distribution function discussed in Section 12.4.3. It is relatively easy to calculate compared to the effort involved in complete scheduling and assignment.

12.7 Bibliographic Notes

[McF90] is a review article that is a good introduction to the topic of high-level synthesis and covers all important aspects. Less detailed but more recent review articles are [Gaj94] and [Lin97]. [Gaj92] and [DM94] are introductory books, mainly written with an educational goal. Books more oriented to the presentation of research results in the field include [Cam91b], [Wal91], [Ku92], [Mic92b], [Cat93], [Van93] and [Bay94].

This chapter assumes a basic familiarity with computer architecture. Those readers that lack this knowledge, are referred to [War90], one of the many good books on this topic. In the context of the hardware models a distinction was made between data and control signals. It is not always easy to make this distinction; a formal definition of control signals is given in [Gho92]. A clear explanation of "two-phase clocking" can be found in [Mea80]. An example of a high-level synthesis system that uses a two-phase hardware model is [Har89].

A topic that is just mentioned in this chapter, but not further covered, concerns the description language to be used as a starting point for high-level synthesis. One of the most used languages is VHDL, even when the language is primarily suitable for simulation rather than for synthesis. This language spans many levels of the hierarchy of design descriptions including the algorithmic one. [Lip89] and [Nav93] are two of the many books available on VHDL. Another popular language that is somewhat comparable to VHDL is Verilog; it is explained in [Tho91] and [Gol96]. A totally different language that was especially designed for the description and synthesis of digital signal processing algorithms is Silage; information on Silage can be found in [Hil85] and [Hil92].

A good introduction to the topic of data-flow graphs, including a section on the history of data-flow models for parallel computing, is given in [Dav82]. Similar ideas can be found in the work of Lee [Lee88, Lee91a, Lee95], and in the proposal for a DFG exchange standard for the purpose of synthesis from Eindhoven University of Technology [Eij92]. Examples of publications that use separate data and control flow graphs are [McF90] and [Cam91a]. Other researchers combine the two in one graph, calling the graph *control-data flow graph* [Pau89] or *data-control flow graph* [Pot92].

More on *signal-flow graphs* can be found in many publications in the field of digital signal processing, including [Fet76] and [Rob87]. Synchronous data flow graphs are explained in [Lee87]. They have for example been used in the Gabriel system as a graphic input language [Lee89]. The consistency problem for data-flow graphs (in a consistent graph, it is ensured that all nodes receive sufficient tokens to keep on computing, while no unbounded number of tokens is accumulated on any of the edges) has been studied by Lee both for synchronous [Lee87] and asynchronous [Lee91a] graphs.

Examples of high-level synthesis systems for control-dominated applications, a topic not covered in this text, are: [Cam91a], [Wol92] and [O'B93].

The definitions in the text for "scheduling", "assignment" and "allocation" follow those given in [Pot92].

More on the principles of "overlapped scheduling" can be found in the review paper [Ger98]. Key articles on the topic are [Par91] and [HdG92]. Examples of high-level synthesis systems that generate overlapped schedules are described in [Goo90], [Olá92], [Lee94], [Kos95] and [Wan95]. Overlapped scheduling can be performed for computations with multiple nested loops such as found in image processing systems. The Phideo high-level synthesis system can deal with the overlapping of nested loops [Mee95, Ver97].

A review paper dedicated to the scheduling problem only is [Wal95]. The notation used here for the description of the scheduling problem has been adapted from [Olá92] which in its turn was based on the one of [Ver91]. Many different variants of the scheduling problem exist and most of them are NP-complete. Those that have been discussed in this chapter can easily be reduced to the NP-complete problems listed in [Gar79].

Mobility-based scheduling has been investigated by many researchers. One of the earliest publications of this approach applied to high-level synthesis is [Par86] where "mobility" is called "freedom". The terms "time frame" and "scheduling range" originate respectively from [Pau89] and [HdG92]. Descriptions of Paulin's force-directed scheduling algorithm can be found in [Pau89] and [Pau91]. This algorithm has been used and adapted by many researchers as can e.g. be seen from the reference list of [Ver92]. A comparison between force-directed and list scheduling can be found in [Jai91].

Regarding the assignment problem, [Sto94] is a detailed review paper on different assignment problems occurring in data-path synthesis. An example of a system that aims at optimizing the performance is [Jia94], while [Rim94] is an example of a system that takes wire length into account by performing some kind of floorplanning. The term "task-to-agent" assignment originates from [Rim94].

A discussion on the use of compatibility and conflict graphs in high-level synthesis and their respective subclasses such as interval graphs is presented in [Spr94]. More on interval graphs, other special families of graphs, and their colorings can be found in [Gol80]. The NP-completeness of circular-arc graph coloring is proved in [Gar80]. An algorithm requiring exponential time in the worst-case for finding the optimal solution for the value-to-register assignment problem (which is equivalent to

```

gcd (int a, b)
{
    while (a ≠ b)
        if (a > b)
            a ← a - b;
        else
            b ← b - a;
    return a;
}

```

Figure 12.36 Euclid's Algorithm to compute the greatest common divider (GCD) of two integers.

circular-arc-graph coloring) is presented in [Sto92]. If special addressing hardware is available, also the polynomial-time algorithm proposed in [Mee93] could be used.

Tseng and Siewiorek were the first to propose the use of a “clique partitioning” heuristic to assignment [Tse86]. A description of their algorithm can also be found in [Gaj92]. More on the NP-completeness of the clique partitioning problem can be found [Gar79].

The application of transformations is often one of the tasks performed by a high-level synthesis system as is reported in e.g. [Har89]. Publications that deal with transformations as their main topic include [Pot94] and [Cha95b]. The NP-completeness of the optimization of resource utilization by retiming is discussed in [Pot94]. Retiming and algorithms to achieve an optimal retiming for the shortest critical path are presented in [Lei83] and [Lei91]. More on algorithmic transformations for digital signal processing can be found in the review papers [Par89] and [Par95].

12.8 Exercises

- 12.1 Suppose that the program of Figure 12.3 is written as “ $x \leftarrow a * b; z \leftarrow c + d + x$ ” and that the nodes allowed in the DFG are either two-input additions or two-input multiplications but not three-input additions. Give the different DFGs corresponding to this program.
- 12.2 Figure 12.36 shows *Euclid's Algorithm* to compute the greatest common divider of two integers. Give a DFG that represents this computation; structure the graph in such a way that the while-loop and if-statement are clearly delimited by conditional nodes.
- 12.3 Find a register assignment and a data path for the overlapped schedule given in Figure 12.20.
- 12.4 Describe how an acyclic DFG should be modified such that the longest-path algorithm of Figure 6.8 computes the ALAP scheduling times of all computational nodes.

- 12.5** Provide simple criteria for the missing details of the scheduling procedure in Figure 12.23. Use the algorithm obtained to find a schedule for the DFG of Figure 12.15, with $T_0 = 10$. Do you find a solution that only uses one multiplier and one adder?

Part III

Appendices

Appendix A

CMOS Technology

Most of the readers that study this book on VLSI design automation probably have sufficient knowledge of VLSI itself to be able to follow the text. This should especially be the case for students of electrical engineering. Students of computer science or physics may, however, never have been exposed to integrated circuit design itself. For this reason the minimal knowledge of *CMOS technology* necessary to understand the rest of the book is presented in this appendix.

CMOS (Complementary Metal-Oxide-Semiconductor) is the silicon-based technology used in almost all VLSI chips produced. The name refers to the materials from which a transistor, the key element of an integrated circuit, is built. A transistor is a physical device. Voltages are applied to it, currents flow through it and the properties of the materials and the way in which they have been combined make the device more or less behave like a "switch". In this appendix, the references to the "analog world" of voltages and currents will be limited to some qualitative remarks, such as the presence of parasitic capacitances. More is not necessary, as "analog CAD" is not covered by this text (see also Chapter 2).

By means of abstraction, it becomes possible to take distance from entities in the "analog world" and to talk about a transistor as a "digital" device. Digital abstraction assigns the Boolean value '0' to all voltages within a certain margin of the negative supply voltage which normally has a voltage of 0 V. It assigns the Boolean value '1' to the voltages within a margin of the positive supply voltage. The positive supply voltage has traditionally been 5 V, but has a tendency to drop: voltages around 3 V are already common. There is great interest in lowering the positive supply voltage because of the lower power consumption. The margins around the two supply voltages can, of course, not overlap. At the same time, the (almost) ideal devices to be discussed in this appendix are not supposed to generate intermediate voltage values outside the ranges covered by either of the two margins.

This appendix consists of two sections. First, the transistor is presented together with the way it can be used to build more complex circuits. Then an explanation of the layout of circuits in CMOS technology is given. This layout is directly related to the mask patterns used in the production of an integrated circuit.

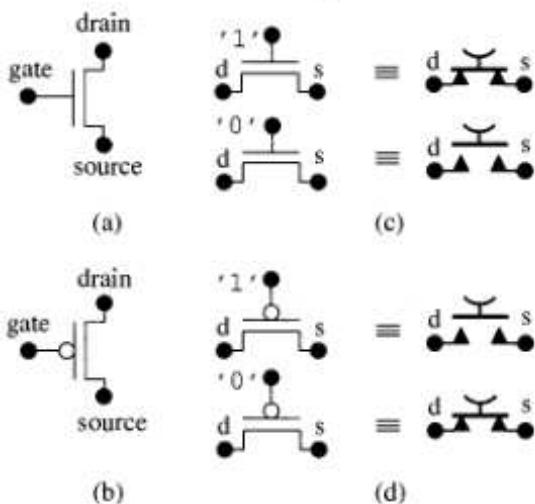


Figure A.1 The schematic symbols for the nMOS (a) and pMOS (b); the switching behaviors of the nMOS (c) and pMOS (d).

A.1 The MOS Transistor and CMOS Logic Design

The MOS transistor can be seen as a *voltage-controlled switch*. It is a device with three terminals; the value of the signal at one of these terminals, called the *gate*, determines whether there is a conducting connection between the other two terminals, called the *drain* and the *source*. The use of two types of MOS transistor is characteristic for CMOS technology. The so-called *n-channel* transistor or *nMOS* (an abbreviation of "n-channel metal-oxide-semiconductor transistor") behaves as a closed connection when its gate signal is '1', and as an open connection when its gate signal is '0'. Its dual, the *p-channel* transistor or *pMOS*, has the reverse behavior. The schematic symbols for these transistors and their behavior are illustrated in Figure A.1.

Basically, a circuit with any desired functionality can be built with these two types of transistor. The circuit's inputs will be connected to the gates of some transistors which may or may not change the value of internal nodes, which in their turn will control other transistors and possibly make them switch. Each change in value of an internal node can lead to more switching. Some of these internal nodes will be connected to the circuit's output.

When transistors are in a conducting state, they *propagate* a signal from their source terminal to their drain terminal or vice versa. Due to technological constraints, an nMOS can better propagate a signal with value '0' than a signal with value '1'. One can say that the electrical resistance of the switch is higher in the latter case. Conversely, a pMOS can better propagate a '1' than a '0'. Therefore, in general, pMOSs are used for the propagation of the positive supply voltage, whereas nMOSs are used for the propagation of the negative supply voltage.

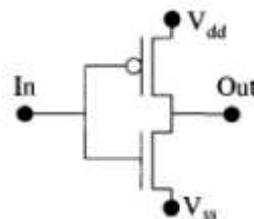


Figure A.2 The CMOS inverter.

The properties mentioned above have been taken into account in the design of the *inverter* shown in Figure A.2. A signal '1' applied to the input *In* will cause the upper transistor to open and the lower one to become conductive resulting in a connection from the negative supply voltage V_{ss} to the output *Out*. This means that the output will carry signal '0'. A signal '0' at the input will lead to the dual situation, creating the output signal '1' as a result of the connection between the positive supply voltage V_{dd} and the output. The fact that the signal *Out* is the inverse of the signal *In* explains the name "inverter" for the circuit. Note, by the way, that V_{ss} and V_{dd} are generally accepted names for the negative and positive supply voltages respectively.

The inverter is a simple *logic gate* (not to be confused with the *gate terminal* of a transistor). A logic gate is characterized by its *unidirectional signal flow*, which means that it has clearly identified input and output terminals and that changes at the input terminals may lead to changes at the output, but that the reverse is not true. On the other hand, a transistor in CMOS has *bidirectional signal flow* as signals can propagate both from the source to the drain terminal as from the drain to the source.

The essential point in the inverter circuit is to have a conducting path from either V_{dd} or V_{ss} to the output. Note that it should neither occur that there is a conducting path between V_{dd} or V_{ss} (a shortcircuit!) nor that the output is not connected to either of V_{dd} or V_{ss} leading to an *undefined* output value. This principle can be applied to build more sophisticated logic gates as the 2-input NAND and 2-input NOR gates shown in Figure A.3 with their truth tables. The output of the 2-input NAND gate is only '0' when both of its inputs *A* and *B* are '1'. This is achieved by the series connection of the two nMOSs shown in Figure A.3(a). The two pMOSs connected in parallel make sure that the output becomes '0' for the other signal combinations on inputs *A* and *B*. In the case of the 2-input NOR shown in Figure A.3(b), the transistors are connected in just the opposite way: the two pMOSs are now in series and the two nMOSs in parallel.

This principle can be generalized to make more general gates. Obviously, a circuit of pMOSs should handle those entries of the truth table that have output '1' and a circuit of nMOSs should take care of the entries that generate an output '0'. The two transistor networks deal with *complementary* parts of the truth table. This explains the word *complementary* in the acronym CMOS. A gate that is built in this way, but is not just a NAND or NOR gate, is called a *CMOS complex gate*. Its

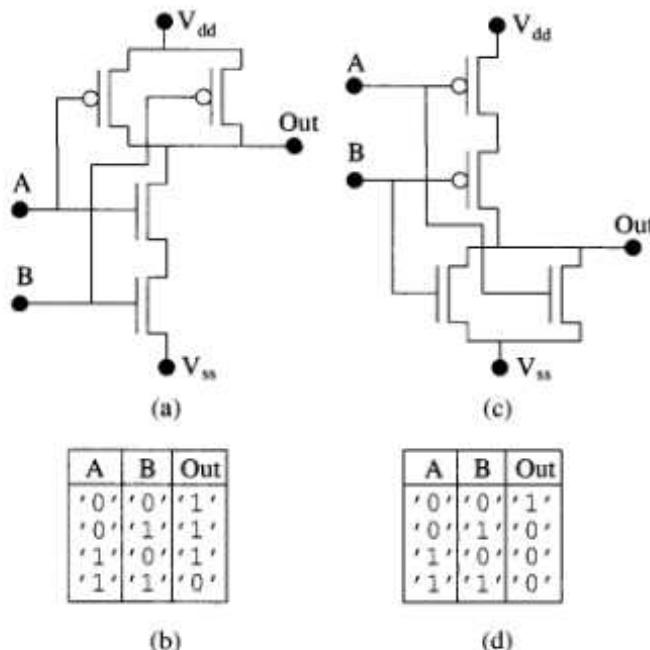


Figure A.3 A two-input NAND gate (a) and its truth table (b), and a two-input NOR gate (c) and its truth table (d).

structure is shown in Figure A.4. For any required Boolean function, the structure of its corresponding complex gate can systematically be derived from the Boolean equation describing the function. This will first be shown for the 2-input NAND given in Figure A.3(a).

The 2-input NAND gate obeys the Boolean equation $Out = \overline{A \cdot B}$. The fact that the output has value '1' when the expression $\overline{A \cdot B}$ is true, means that the output

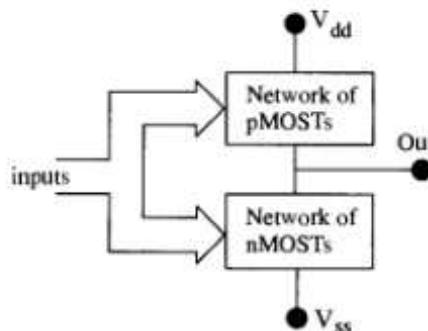


Figure A.4 The general structure of a CMOS complex gate.

will have value '0' when $A \cdot B$ is true. This behavior can be realized by a series connection of two nMOSes between Out and V_{ss} . Only when both A and B are '1', will there be a '0' at Out . As mentioned before, the output of the gate is '1' when the expression $\overline{A} \cdot \overline{B}$ is true. By the application of *De Morgan's Rule*, the expression becomes equivalent to $\overline{A} + \overline{B}$. As a negated input corresponds to the nonnegated signal controlling a pMOS, $\overline{A} + \overline{B}$ can be realized by the parallel connection of two pMOSes between V_{dd} and Out . It is not difficult to see that a NAND gate with more than two inputs can be realized by putting more transistors in series in the nMOS network and as many transistors in parallel in the pMOS network. Similarly, by using parallel connections in the nMOS network and series connections in the pMOS network, one gets a multiple-input NOR gate.

The fact that the Boolean AND operator (indicated by the symbol ".") corresponds to a series connection in the nMOS network and that the Boolean OR operator (indicated by the symbol "+") corresponds to a parallel connection, can be used to build more sophisticated gates. Because subnetworks obtained by series or parallel composition can be used themselves in a series or parallel connection, any nesting of AND and OR in a Boolean expression can be translated into a complex gate. Both the nMOS and pMOS networks then have the so-called *series-parallel* form. As is known from Boolean algebra, however, the operators AND and OR are not sufficient for building any Boolean function. One also needs the operator NOT (it is indicated in this text by a horizontal line above the expression to be negated). This operator cannot be in an arbitrary place in the expression to be realized by a complex gate. The only place where a negation *can* and *must* be present in the expression is at the top level (so, the expression can be an arbitrary composition with AND and OR followed by NOT). The consequence of a negation at any other place in the expression is that multiple complex gates must be used.

Consider e.g. the following Boolean equation: $Out = \overline{(A \cdot B + C) \cdot (D + E)}$. By removing the overall negation, one gets the expression $(A \cdot B + C) \cdot (D + E)$ which can be used for the direct realization of the network of nMOSes. The transistors controlled by the signals A and B are connected in series, this subnetwork is connected in parallel with a transistor controlled by C and this subnetwork on its turn is connected in series with the subnetwork consisting of the parallel connection of two nMOSes controlled by D and E . The network of nMOSes that has been obtained in this way is part of the network shown in Figure A.5. The network of pMOSes, also shown in the figure, follows from the subsequent application of De Morgan's Rule until all occurrences of the NOT operator have been moved to the lowest level:

$$\begin{aligned} Out &= \overline{(A \cdot B + C) \cdot (D + E)} \\ Out &= \overline{(A \cdot B + C)} + \overline{(D + E)} \\ Out &= \overline{A} \cdot \overline{B} \cdot \overline{C} + \overline{D} \cdot \overline{E} \\ Out &= (\overline{A} + \overline{B}) \cdot \overline{C} + \overline{D} \cdot \overline{E} \end{aligned}$$

As a direct consequence of the application of De Morgan's Rule, each series con-

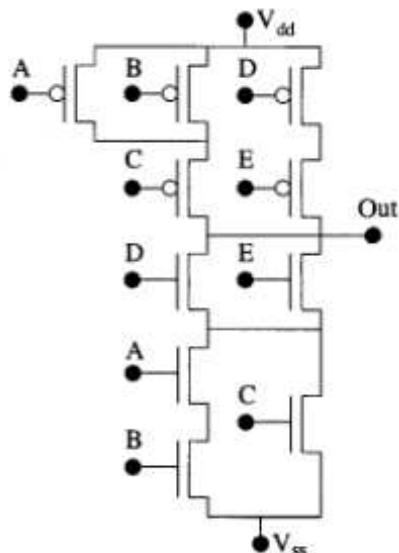


Figure A.5 A complex gate that uses complementary series-parallel networks.

nection in the nMOST network is a parallel connection in the pMOST network and vice versa. One says that the two series-parallel networks are each other's *dual*. As a consequence, the number of transistors in both networks is equal.

It is not necessarily the case that a CMOS complex gate is built from two dual transistor networks. It is not even the case that any IC built in CMOS technology solely consists of complex gates. Transistors can be used to switch signals in other configurations. These issues are, however, not discussed here. When necessary, they will be touched upon in the rest of the text.

A.2 Transistor Layout in CMOS and Related Issues

Figure A.6 gives stylized views in three dimensions of the two types of transistor used in CMOS. For both types the gate consists of a deposit of *polysilicon* (polycrystalline silicon), which is conducting material, on top of a layer of *silicon oxide*, which is nonconductive and therefore isolates the polysilicon from the underlying silicon substrate. Note that the triplet metal-oxide-semiconductor in the acronym MOS refers to an earlier technology in which a gate was made of metal instead of polysilicon. In the nMOST, the substrate is p-doped and the source and drain areas located on both sides of the gate are formed by diffusion of n-type dopants into the substrate. The source and drain areas are therefore called the *diffusion areas*. The terms "to dope" and "dopant" refer to the substances introduced in pure silicon in order to give it a shortage (p) or an excess (n) of electrons. When a voltage is applied to the gate, a thin layer of n-type silicon is created in the substrate just under the

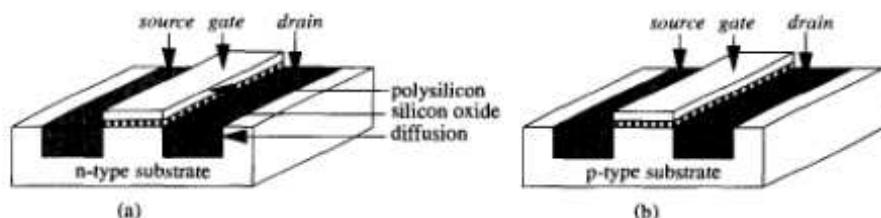


Figure A.6 A stylized 3-dimensional view of the pMOS (a) and nMOS (b).

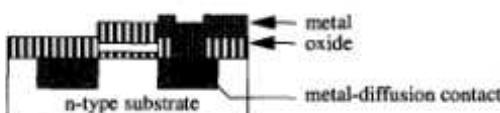


Figure A.7 The cross section of a metal-diffusion contact.

gate area and a current can flow between the two diffusion areas. This thin area is the *channel* which explains why one talks of an "n-channel" transistor. A pMOS is the dual of the nMOS: it requires an n-doped substrate and p-doped diffusion areas; conduction takes place through a p-channel.

From Figure A.6 it can be seen that both the gate and diffusion areas consist of conducting surfaces that are separated by an insulating layer from the substrate (the voltage across the p-n junction between diffusion and substrate is such that the junction forms an insulating layer). These surfaces obviously lead to *parasitic capacitances*. Because they need to be charged or discharged when signals switch between the values '0' and '1', they affect the switching speed of a circuit.

Of course, a single nMOS or pMOS does not make a VLSI chip. Millions of them can be put on a single chip and it is essential to interconnect them. Some local interconnection can be realized using polysilicon or diffusion. These two materials cannot cross, however, as a transistor is created at the crossing. One or more layers of *metal* (aluminum) are used for the purpose of interconnection. Before depositing metal on the chip, a layer of silicon oxide is deposited on top of the polysilicon and diffusion areas, isolating them electrically from the metal. Where necessary, *contact cuts*, holes in the oxide, are used for the interconnection of metal and either of the polysilicon or diffusion areas. If the technology has more than one layer of metal, a new layer of silicon oxide is used for isolation and contact cuts make it possible to connect to the new layer. Depending on the technology, restrictions can exist on the layers that can be interconnected by a contact cut (e.g. it may not be allowed to have a direct contact from the second layer of metal to polysilicon or diffusion). A cross section of a metal wire contacting a diffusion area is shown in Figure A.7.

Another question is how to combine nMOSes and pMOSes, which have differently doped substrates, on a single chip. This is achieved by the use of so-called *wells*. A well is a relatively large area in the substrate that is doped with the opposite

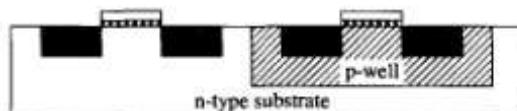


Figure A.8 The use of a well for the combination of transistors of the opposite type.

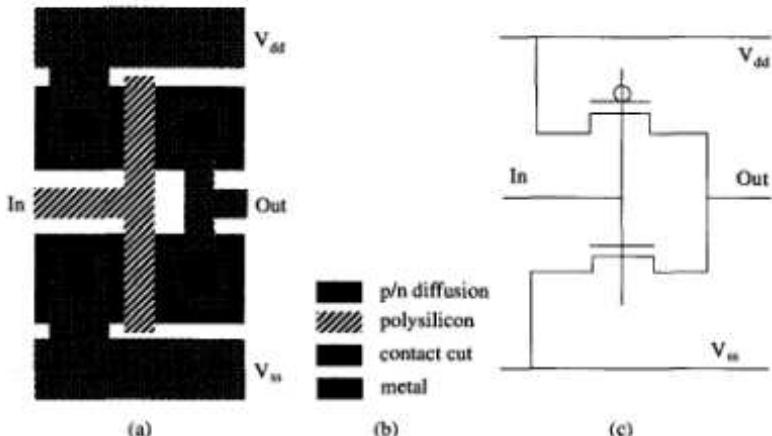


Figure A.9 The layout of an inverter (a), an explanation of the legend (b) and the inverter's schematic view for the same topology (c).

dopant as the substrate itself. If, for example, an n-type substrate is used, pMOS can directly be built on it, but nMOS require an *p-well* before they can be realized. This is shown in Figure A.8, that shows a cross section of two transistors of the opposite type located next to each other on the same chip. In order to minimize the number of wells, a CMOS designer will in general try to cluster transistors of the same type.

The most common way to picture the layout of a chip is by giving a top view showing all conducting layers. First the diffusion layer is drawn and then the polysilicon layer followed by the layers of metal. Figure A.9(a) shows the layout of an inverter while the shading patterns used for the different layers are given in Figure A.9(b). In order to make it easier to understand the layout, a schematic representation of the inverter is given in Figure A.9(c). Note that no wells are shown in the figure as is the case in the rest of this text.

Important parameters that a designer can influence when designing a transistor circuit are the *length* and *width* of a transistor. The length of a transistor is basically the length of the channel, the distance between the two diffusion areas for the source and drain. The width is, of course, the other dimension of the channel area (perpendicular to the length). In general, a wider transistor will have less resistance and will therefore switch faster. However, varying the length and width of a transistor

also affects the various parasitic capacitances in the circuit.

A.3 Bibliographic Notes

There are many books that partly or entirely deal with CMOS VLSI design. The following books are mainly "systems oriented" rather than "circuit oriented": [Muk86], [Puc88], [Mor90], [Wes93], [Wol94b], and [Rab96].

De Morgan's Rule and other elementary notions from the field of Boolean algebra can be found in any textbook on *logic design*. A number of them are listed at the end of Chapter 11.

Appendix B

About the Pseudo-code Notation

This text explains several algorithms related to VLSI design automation. A convenient way to present them is to use a more or less "formal" description in some kind of programming language accompanied by informal comments in English. It has been chosen to use the *C* programming language as a basis for the programming language, because more and more people are becoming familiar with this language. However, it is not the intention of this text to contain algorithmic descriptions that are directly executable. This would make it necessary to include many details that may hide the main ideas involved. It is instead supposed that the reader has some experience in programming and is able to turn the code presented into executable software.

The formalism used in this text is "pseudo-code based on *C*", which means that the formalism is based on *C* but is extended with constructions that make it easier to express specific actions more compactly without affecting the readability of the code. Sections B.1 and B.2 mainly deal with those elements of *C* that are used without modification in pseudo-code (these sections can be skipped by readers already familiar with *C*; those that are not are assumed to know Pascal or a similar language). These two sections, although remaining quite close to *C*, clearly constrain themselves to subsets of the language: they are solely meant to be able to read the pseudo-code in this text not to learn the basics of programming in *C*. Section B.3 gives an overview of extensions to *C* characteristic of the pseudo-code as used throughout this text.

A remark on typography: descriptions in pseudo-code are normally presented in a figure, separately from the main text. In such a case, the proportionally-spaced "roman" text font is mainly used, using "**bold**" font for the reserved keywords. Pseudo-code elements that belong to the "mathematical" domain (see also Section B.3) are typeset as mathematical formulae, which means that mainly "*italic*" font is used. However, when the *main* text quotes elements of the pseudo-code, "typewriter font" is used for those words that directly originate from the pseudo-code. Pseudo-code language elements from the mathematical domain are an exception: they are typeset identically in pseudo-code descriptions and the main text.

B.1 Data Structures and Declarations

The primitive data types known in C are: integers (designated by the keyword `int`), floating point numbers (`float`) and characters (`char`).¹ Variables of a specific type are declared by giving the keyword belonging to the data type desired followed by one or more variable identifiers separated by a comma. A declaration is terminated by a semicolon. Example:

```
float number1, number2;
int i, j, k;
```

Note that C does not have a separate Boolean data type. Integers are used for this purpose. A value of 0 (zero) is interpreted as "false" and any other value is interpreted as "true".

One kind of composite data type is the *array*. It is declared in a similar way as the scalar variables mentioned above with the difference that the number n of array elements is declared between square brackets after the variable identifier. The index of such an array ranges from 0 to $n - 1$. An array element is specified by giving the index in square brackets after the array name. Example (note that comments in C are enclosed between the delimiters `/*` and `*/`):

```
int numbers[100]; /* The array numbers has 100 entries indexed from 0 to 99. */
... numbers[i] ...; /* This is a reference to the ith element, somewhere in the code. */
```

A *string* is declared as an array of `char`. When the number of array elements is omitted in the declaration, the array name is a *pointer* to the string for which storage space has been reserved elsewhere. Example:

```
char name[15]; /* A string of 15 characters stored locally */
char id[ ]; /* A string of unknown length stored elsewhere; (i.d is just a pointer) */
```

Another kind of a composite data type is the *structure* (it corresponds to a *record* in Pascal). It is used to group a number of data items called its *members* (*fields* in Pascal). A structure declaration starts with the keyword `struct` followed by the name of the structure data type and the members enclosed in braces. The specification of a member within a structure is similar to the declaration of a variable. A structure-type variable is declared by the keyword `struct`, the structure name and the identifier of the variable. Access to a member of a structure-type variable is gained by concatenating a dot (".") and the member name to the variable. All this is illustrated below:

¹ No attention is paid here to C keywords that indicate different word lengths for one data type, e.g. short integers of 8 bit and long integers of 32 bit.

```
struct person {  
    char name[10]; /* An array of 10 characters to store the person's name */  
    int age;  
};  
  
/* Declare two variables of type struct person */  
struct person person1, person2;  
  
... person1.age ... /* Reference to the member age of person1 */
```

A data type that is closely related to a structure is a *pointer* to a structure. It makes possible the creation of complex data structures in which structures can point to each other. If *p* is a pointer, **p* gives the object pointed at. Conversely, the declaration of **p* being of a certain type actually declares *p* to be a pointer to that type. The example below shows the declaration and use of a data structure for the representation of a linked list of integers:

```
struct list_element {  
    int value;  
    struct list_element *next;  
};  
  
struct list_element *p; /* p points to a list_element */  
  
... p->value ... /* access to a member of an object pointed at;  
this notation is equivalent to (*p).value */
```

Note that the definition of a structure can contain references to its own data type. This is only allowed for the declaration of pointers.

B.2 C-language Constructs

Roughly stated, a C program consist of declarations and statements. In the previous section the declaration of data types and variables has been mentioned. The third type of declaration, that of functions, is explained in this section. Besides, attention is paid to the most important statement types, viz. the assignment, the conditional statement, the iterative statement and the function call.

An issue that is not covered here in detail is the *expression* that e.g. occurs at the right-hand side of an assignment statement or as the condition that controls a conditional statement. Similarly to a language like Pascal, expressions are composed of the combination of variables and function calls using operators such as + and *. The operators for Boolean operations, however, are indicated by symbols that

differ from those in Pascal. The "logical AND", "logical OR" and "logical NOT" are respectively indicated by `&&`, `||` and `!`. It was mentioned before that C does not have a Boolean data type and that integers are used instead; the value zero corresponds to "false" and any other value to "true".

The *assignment statement* indicates that an expression at the right-hand side of the assignment operator should be evaluated and stored at the location indicated by the left-hand side of the operator. In C, the assignment operator is "`=`", the equal sign. However, in order not to conflict with the mathematical notation used in the pseudo-code (see Section B.3) in which the equal sign is used as an equality predicate in Boolean expressions, the symbol "`←`" is used in pseudo-code to indicate assignment (in C, the symbol "`==`" is used for the equality predicate). Example (note that an assignment statement should be terminated by a semicolon):

```
int a;
a ← 1;
```

A *compound statement* or *block* is a group of statements that belong together, e.g. because they form the body of an iterative statement. The fact that they belong together is indicated by enclosing them in braces (the symbols "`{`" and "`}`").

Statements that should only be executed under some condition require the use of the *if* statement. An *if* statement has two appearances, with and without an *else* part. This is indicated below (as opposed to the previous examples, the next ones contain syntax descriptions and not fragments of pseudo-code):

```
if ((Boolean expression))
    ⟨statement⟩

if ((Boolean expression))
    ⟨statement⟩
else ⟨statement⟩
```

The *(Boolean expression)* should be an expression that evaluates to an integer value (remember that C does not have a Boolean data type). It is always written between parentheses. *(statement)* can be any type of statement including the compound statement.

In C there are three iteration statements all three of which are used in pseudo-code descriptions. The *while* statement obeys the following syntax:

```
while ((Boolean expression))
    ⟨statement⟩
```

First the `<Boolean expression>` is evaluated (enclosed in parentheses) to determine whether or not the loop body indicated by `(statement)` is executed. So, the number of iterations is zero or more.

A variation on the `while` statement is the `do` statement, the syntax of which is given below:

```
do (statement)
  while ((Boolean expression))
```

Here, the loop body given by `(statement)` is always executed at least once as it precedes the evaluation of the termination condition given by `(Boolean expression)` (enclosed in parentheses).

The `for` statement is mainly interesting because of its compact form in certain situations. Below its syntax is given and it is shown how it can be mapped to a `while` statement.

```
for ((statement 1); (Boolean expression); (statement 2))
  (statement 3)

  (statement 1);
  while ((Boolean expression)) {
    (statement 3);
    (statement 2);
  }
```

So, the three language elements following the keyword `for` (enclosed in parentheses and separated by semicolons) designate the initialization of loop variables, the termination condition and the updating of the loop variables.

As in most other programming languages, C offers the possibility to group a number of statements, give the group a name, parameterize some of the variables in the group and invoke these statements by using this name and appropriate values for the parameters. This is the function mechanism (this mechanism roughly corresponds to both the "function" and "procedure" in Pascal). A *function definition* defines the statements grouped together and a *function call* is a statement that invokes them. The example below gives a possible definition of the factorial function followed by a call:

```

int factorial (int n)
{
    int result;
    result ← 1;
    while (n > 1) {
        result ← result * n;
        n ← n - 1;
    }
    return result;
}

... factorial(k); ... /* function called from somewhere in the program */

```

In the function definition, the function name is preceded by the data type to be returned by the function. If the return type is not relevant, it may be omitted.² After the function name, the parameters of the function preceded by their data types are specified. The parentheses enclosing the parameters are compulsory even when the function does not have any parameters. Then follows the function body enclosed by compulsory braces. The body consists of declarations followed by statements. The `return` statement terminates the execution of the body and returns the value of the expression following the keyword `return` as the value of the function call.

In C, all statements are part of some function body. There is no "main program" as in Pascal. One function called `main` has a special meaning as it is the function called first in any C program (so all C programs should contain a function `main`).

B.3 Pseudo-code Constructs

The goal of using pseudo-code is to make the presentation of an algorithm more compact without sacrificing readability. This is achieved by extending the notation with constructs foreign to the language syntax. The assumption is that each extension has a (probably more verbose) counterpart in the language itself and that the reader is able to find this counterpart easily. The main features of the pseudo-code used in this text are as follows:

- The use of unspecified functions or data types. Functions or data types are used that have not been declared in the pseudo-code. The text explaining the pseudo-code tells in such a case what the function is supposed to compute or what the data structure is supposed to store.
- The use of English text enclosed in double quotes. This indicates that a specific operation is summarized informally using plain English. An example is:

² This is equivalent to defining the return type to be `void` which is an explicit way of stating that no value is returned.

```

int A[max_index];


---


min_A ← "the minimum value stored in array A";

```

This replaces the less informative code consisting of an iteration over all elements of array *A* and the comparison of each of its values with the minimum found up to that point.

- The use of mathematical typography. The pseudo-code might for example contain:

```

float a, b, c;


---


c ←  $\sqrt{a^2 + b^2}$ ;

```

The mathematical expression is more readable than its equivalent in plain C: *c* = *sqrt(a*a + b*b)*; . Note that the three variables are consistently typeset in italics, indicating that they belong to the mathematical domain.

- Extensions for dealing with sets. Sets occur quite often in the algorithms presented in this text. Therefore a “generic set data type” is part of the pseudo-code. Using the keywords **set** or before any data type in declarations means that the entire data type represents a set (this will often be a pointer to a linked list in a practical implementation). The mathematical notation mentioned above is often used in combination with sets, making use of the operators “ \cap ” for set intersection, “ \cup ” for set union, etc. In addition, a new iterative statement, the **for each** statement, is part of the pseudo-code: it allows the processing of each element of the set one by one in the body of the loop. The keywords **for** and **each** should be followed by an expression of the form $a \in A$ and a (possibly compound) statement for the loop’s body. In each iteration *a* has the value of a distinct element of *A*. All this is illustrated in the example below:

```

set of int A, B;


---


A ← Ø;
B ← Ø;


---


for each a ∈ A
  If (a ≥ 0)
    B ← B ∪ {a};


---


/* A has been filled with elements by code not shown here. */

```

- Multiple return values. It is sometimes convenient to pretend that a function returns multiple, say *k*, values that can be assigned to *k* variables in the calling

environment. This is indicated by declaring the successive data types of the returned values enclosed between parentheses and separated by commas before the function name in the function definition. The return statement in the function body should also group the values to be returned inside parentheses and separate them by commas. In the calling environment, the variables that will be assigned are also enclosed between parentheses and separated by commas. Here is an example:

```
(int, float) powers (int x)
{
    int y;
    float z;
    y ←  $x^2$ ;
    z ←  $\sqrt{x}$ ;
    return (x,y);
}

int m,n;
float p;
...
(n,p) ← powers(m);
```

B.4 Bibliographic Notes

The pseudo-code introduced here has been specially designed for this text. It is, however, based on the C programming language. Due to the popularity of C, a multitude of books exist on this language and many more appear on the market every year. Here only the one by Kernighan and Ritchie is mentioned [Ker88]. This second edition has been adapted for the standardized version ANSI C.

Appendix C

List of Acronyms

ALAP:	as late as possible
ALU:	arithmetic logic unit
ASAP:	as soon as possible
ASIC:	application-specific integrated circuit
ATPG:	automatic test-pattern generation
BDD	binary-decision diagram
CAD:	computer-aided design
CMOS:	complementary MOS
DAG:	directed acyclic graph
DC	don't care
DFG:	data-flow graph
DRC:	design-rule checking
DSP:	digital signal processing/digital signal processor
EDIF:	electronic design interchange format
FIFO:	first in first out
FPGA:	field-programmable gate array
FSM:	finite state machine
FU:	functional unit
HDL:	hardware description language
IC:	integrated circuit
ILP:	integer linear programming
LC:	least cost
LIFO:	last in first out
LP:	linear programming
MOS:	metal-oxide-semiconductor
nMOST:	n-channel MOS transistor
NP:	nondeterministic polynomial
NPC:	NP-complete
OBDD	ordered binary-decision diagram
P:	polynomial
PCB:	printed circuit board
PLA:	programmable logic array

pMOST:	p-channel MOS transistor
RAM:	random-access memory
ROBDD:	reduced ordered binary-decision diagram
ROM:	read-only memory
RTL:	register-transfer level
TSP:	traveling salesman problem
VCG:	vertical constraint graph
VHDL:	VHSIC hardware description language
VHSIC:	very high speed integrated circuit
VLSI:	very large scale integration

References

- [Aar89] E. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines, A Stochastic Approach to Combinatorial Optimization and Neural Computing*. John Wiley & Sons, Chichester, 1989.
- [Abr90] M. Abramovici, M.A. Breuer, and A.D. Friedman. *Digital Systems Testing and Testable Design*. Computer Science Press, 1990.
- [Adl91] D. Adler. Switch-Level Simulation Using Dynamic Graph Algorithms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(3):346–355, March 1991.
- [Aho74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.
- [Ake72] S.B. Akers. Routing. In M.A. Breuer, editor, *Design Automation of Digital Systems, Vol. I: Theory and Techniques*, pages 283–333. Prentice-Hall, Englewood Cliffs, New Jersey, 1972.
- [Ake78] S.B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.
- [Ake82] S.B. Akers. Clustering Techniques for VLSI. In *International Symposium on Circuits and Systems*, pages 472–476, 1982.
- [Alp95] C.J. Alpert and A.B. Kahng. Recent Directions in Netlist Partitioning: A Survey. *Integration, The VLSI Journal*, 19:1–81, 1995.
- [Ama94] H.P. Amann, P. Moeschler, F. Pellandini, A. Vachoux, C. Munk, and D. Mlynek. High-Level Specification of Behavioral Hardware Models with MODES. In *International Symposium on Circuits and Systems*, pages 1.387–1.390, 1994.
- [Ame94] S. Amellal and B. Kaminska. Functional Synthesis of Digital Systems with TASS. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(5):537–552, May 1994.
- [Asa86] T. Asano, M. Sato, and T. Ohtsuki. Computational Geometry Algorithms. In T. Ohtsuki, editor, *Advances in CAD for VLSI, Vol. 4: Layout Design and Verification*, pages 295–347. North-Holland, Amsterdam, 1986.
- [Baa78] S. Baase. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, Reading, Massachusetts, 1978. Reprinted with corrections, January 1983.
- [Bal97] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems, The POLIS Approach*. Kluwer Academic Publishers, Boston, 1997.
- [Ban94] P. Banerjee. *Parallel Algorithms for VLSI Computer-Aided Design*. PTR Prentice Hall, Englewood Cliffs, New Jersey, 1994.

- [Bar87] Z. Barzilai, L. Carter, B.K. Rosen, and J.D. Rutledge. HSS—A High-Speed Simulator. *IEEE Transactions on Computer-Aided Design*, CAD-6(4):601–617, July 1987.
- [Bar88] Z. Barzilai, D.K. Beece, L.M. Huisman, V.S. Iyengar, and G.M. Silberman. SLS—A Fast Switch-Level Simulator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(8):838–849, August 1988.
- [Bar92] T.J. Barnes, D. Harrison, A.R. Newton, and R.L. Spickelmeier. *Electronic CAD Frameworks*. Kluwer Academic Publishers, Boston, 1992.
- [Bay94] M.A. Bayoumi, editor. *VLSI Design Methodologies for Digital Signal Processing Architectures*. Kluwer Academic Publishers, Boston, 1994.
- [Ben98] L. Benini and G. De Micheli. *Dynamic Power Management, Design Techniques and CAD Tools*. Kluwer Academic Publishers, Boston, 1998.
- [Ber97] J.M. Bergé, O. Levia, and J. Rouillard, editors. *Hardware/Software Co-Design and Co-Verification*. Kluwer Academic Publishers, Boston, 1997.
- [Beu90] F.A. Beune. *Generalizing VLSI Layout Design, A Rule-Based Symbolic Approach*. PhD thesis, University of Twente, Department of Electrical Engineering, May 1990.
- [Bha96] J. Bhasker. *A VHDL Synthesis Primer*. Star Galaxy Publishing, Allerton, PA, 1996.
- [Boe95] K.D. Boese, A.B. Kahng, B.A. McCoy, and G. Robins. Near-Optimal Critical Sink Routing Tree Constructions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(12):1417–1436, December 1995.
- [Bon97] E.R. Bonsma and S.H. Gerez. A Genetic Approach to the Overlapped Scheduling of Iterative Data-Flow Graphs for Target Architectures with Communication Delays. In *ProRISC Workshop on Circuits, Systems and Signal Processing*, November 1997.
- [Bou90] D.E. van den Bout and T.K. Miller. Graph Partitioning Using Annealed Neural Networks. *IEEE Transactions on Neural Networks*, 1(2):192–203, June 1990.
- [Boy88] D.G. Boyer. Symbolic Layout Compaction Review. In *25th Design Automation Conference*, pages 383–389, 1988.
- [Bra84] R.K. Brayton, G.D. Hachtel, C.T. McMullen, and A.L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Boston, 1984.
- [Bra87] H.N. Brady and J. Blanks. Automatic Placement and Routing Techniques for Gate Array and Standard Cell Designs. *Proceedings of the IEEE*, 75(6):797–806, June 1987.
- [Bra88] D. Braun, J.L. Burns, F. Romeo, A. Sangiovanni-Vincentelli, K. Mayaram, S. Devadas, and H.-K.T. Ma. Techniques for Multilayer Channel Routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(6):698–712, June 1988.
- [Bra90a] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient Implementation of a BDD Package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.
- [Bra90b] R.K. Brayton, G.D. Hachtel, and A.L. Sangiovanni-Vincentelli. Multilevel Logic Synthesis. *Proceedings of the IEEE*, 78(2):264–300, February 1990.
- [Bre76] M.A. Breuer and A.D. Friedman. *Diagnosis and Reliable Design of Digital Systems*. Pitman, London, 1976.
- [Bre77] M.A. Breuer. Min-Cut Placement. *Journal of Design Automation and Fault-Tolerant Computing*, 1(4):343–362, October 1977.
- [Bry84] R.E. Bryant. A Switch-Level Model and Simulator for MOS Digital Systems. *IEEE Transactions on Computers*, C-33(2):160–177, February 1984.
- [Bry86] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
- [Bry87a] R.E. Bryant. A Survey of Switch-Level Algorithms. *IEEE Design and Test of Computers*, 4(4):26–40, August 1987.

- [Bry87b] R.E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler. COSMOS: A Compiled Simulator for MOS Circuits. In *24th ACM/IEEE Design Automation Conference*, pages 9–16, 1987.
- [Bry92] R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
- [Bur83a] M. Burstein, S.J. Hong, and R. Pelavin. Hierarchical VLSI Layout: Simultaneous Placement and Wiring of Gate Arrays. In F. Anceau and E.J. Aas, editors, *VLSI '83*, pages 45–60, Amsterdam, 1983. North-Holland.
- [Bur83b] M. Burstein and R. Pelavin. Hierarchical Channel Router. *Integration, The VLSI Journal*, 1:21–38, 1983.
- [Bur83c] M. Burstein and R. Pelavin. Hierarchical Wire Routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, CAD-2(4):223–234, 1983.
- [Bur86] M. Burstein. Channel Routing. In T. Ohtsuki, editor, *Advances in CAD for VLSI, Vol. 4: Layout Design and Verification*, pages 133–167. North-Holland, Amsterdam, 1986.
- [Byr85] R.H. Byrd, G.D. Hachtel, M.R. Lightner, and M.H. Heydemann. Switch Level Simulation: Models, Theory and Algorithms. In A. Sangiovanni-Vincentelli, editor, *Advances in Computer-Aided Engineering Design*, volume 1, pages 93–148. JAI Press, Greenwich, Connecticut, 1985.
- [Cai89] H. Cai. *Routing Channels in VLSI Layout*. PhD thesis, Delft University of Technology, Department of Electrical Engineering, 1989.
- [Cam91a] R. Camposano. Path-Based Scheduling for Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(1):85–93, January 1991.
- [Cam91b] R. Camposano and W. Wolf, editors. *High-Level VLSI Synthesis*. Kluwer Academic Publishers, Boston, 1991.
- [Car86] H.W. Carter. Computer-Aided Design of Integrated Circuits. *IEEE Computer*, pages 19–36, April 1986.
- [Cat93] F. Catthoor and L. Svensson, editors. *Application-Driven Architecture Synthesis*. Kluwer Academic Publishers, Boston, 1993.
- [Cha95a] A.P. Chandrakasan and R.W. Brodersen. *Low Power Digital CMOS Design*. Kluwer Academic Publishers, Boston, 1995.
- [Cha95b] A.P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R.W. Brodersen. Optimizing Power Using Transformations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(1):12–31, January 1995.
- [Cha97] K.C. Chang. *Digital Design and Modeling with VHDL and Synthesis*. IEEE Computer Society Press, Los Alamitos, California, 1997.
- [Che86] H.H. Chen and E.S. Kuh. Glitter: A Gridless Variable-Width Channel Router. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, CAD-5(4):459–465, October 1986.
- [Che93] C.H. Chen and I.G. Tollis. Area Optimization of Spiral Floorplans. *Journal of Circuits, Systems, and Computers*, 3(4):833–857, 1993.
- [Che97] Y.P. Chen and D.F. Wong. A Graph Theoretic Approach to Feed-Through Pin Assignment. *Integration, The VLSI Journal*, 24:147–158, 1997.
- [Cho85] Y.E. Cho. A Subjective Review of Compaction. In *22nd Design Automation Conference*, pages 396–404, 1985.
- [Chu84] K.C. Chu and R. Sharma. A Technology Independent MOS Multiplier Generator. In *21st Design Automation Conference*, pages 90–97, 1984.
- [Coh87] J.P. Cohoon and W.D. Paris. Genetic Placement. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):956–964, November 1987.

- [Con88] J. Cong, D.F. Wong, and C.L. Liu. A New Approach to Three- or Four-Layer Channel Routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(10):1094–1104, October 1988.
- [Cor90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.
- [Cou94] O. Coudert. Two-Level Logic Minimization: An Overview. *Integration, The VLSI Journal*, 17:97–140, 1994.
- [Cro88] K. Croes, H.J. De Man, and P. Six. CAMELEON: A Process-Tolerant Symbolic Layout System. *IEEE Journal of Solid-State Circuits*, 23(3):705–713, June 1988.
- [d'A85] M.A. d'Abreu. Gate-Level Simulation. *IEEE Design & Test of Computers*, pages 63–71, December 1985.
- [Dav82] A.L. Davis and R.M. Keller. Data Flow Program Graphs. *IEEE Computer*, pages 26–41, February 1982.
- [Dav91] L. Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, 1991.
- [Deu76] D.N. Deutsch. A “Dogleg” Channel Router. In *13th Design Automation Conference*, pages 425–433, 1976.
- [Dev89] S. Devadas. Optimal Layout via Boolean Satisfiability. In *International Conference on Computer-Aided Design*, pages 294–297, 1989.
- [Dev94] S. Devadas, A. Ghosh, and K. Keutzer. *Logic Synthesis*. McGraw-Hill, New York, 1994.
- [Dij59] E.W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [DM94] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, 1994.
- [DM96] G. De Micheli and M. Sami, editors. *Hardware/Software Co-Design*. Kluwer Academic Publishers, Boston, 1996.
- [Doe87] J. Doenhardt and T. Lengauer. Algorithmic Aspects of One-Dimensional Layout Compaction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-6(5):863–878, September 1987.
- [Don88] W.E. Donath. Logic Partitioning. In B.T. Preas and M.J. Lorenzetti, editors, *Physical Design Automation of VLSI Systems*, pages 65–88. Benjamin Cummings, Menlo Park, CA, 1988.
- [Dre97] R. Drechsler and B. Becker. Overview of Decision Diagrams. *IEE Proceedings on Computers and Digital Techniques*, 144(3):187–193, May 1997.
- [Dut90] N.D. Dutt and D.D. Gajski. Design Synthesis and Silicon Compilation. *IEEE Design and Test of Computers*, pages 8–23, December 1990.
- [Dut93] S. Dutt. New Faster Kernighan-Lin-Type Graph-Partitioning Algorithms. In *International Conference on Computer-Aided Design*, pages 370–377, 1993.
- [Edw92] M.D. Edwards. *Automatic Logic Synthesis for Digital Systems*. Macmillan, Hounds-mills, Basingstoke, Hampshire, 1992.
- [Eij92] J.T.J. van Eijndhoven and L. Stok. A Data Flow Graph Exchange Standard. In *European Conference on Design Automation, EDAC '92*, pages 193–199, 1992.
- [Enb87] R.J. Enbody and H.C. Du. General Purpose Router. In *24th Design Automation Conference*, pages 637–640, 1987.
- [Fet76] A. Fettweis. Realizability of Digital Filter Networks. *Archiv fuer Elektronik und Uebertragungstechnik*, 30(2):90–96, 1976.
- [Fid82] C.M. Fiduccia and R.M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *Proceedings 19th Design Automation Conference*, pages 406–413, 1982.

- [Föss97] U. Fössmeier, M. Kaufmann, and A. Zelikovsky. Faster Approximation Algorithms for the Rectilinear Steiner Tree Problem. *Discrete and Computational Geometry*, 18:93–109, 1997.
- [Fou84] L.R. Foulds. *Combinatorial Optimization for Undergraduates*. Springer-Verlag, New York, 1984.
- [Fuj88] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams. In *International Conference on Computer-Aided Design*, pages 2–5, 1988.
- [Fuj91] M. Fujita, Y. Matsunaga, and T. Kakuda. On Variable Ordering of Binary Decision Diagrams for the Application of Multi-Level Logic Synthesis. In *European Design Automation Conference, EDAC '91*, pages 50–54, 1991.
- [Gaj83] D.D. Gajski and R.H. Kuhn. New VLSI Tools, Guest Editors' Introduction. *IEEE Computer*, 16(12):11–14, December 1983.
- [Gaj88a] D.D. Gajski and Y.-L.S. Lin. Module Generation and Silicon Compilation. In B.T. Preas and M.J. Lorenzetti, editors, *Physical Design Automation of VLSI Systems*, pages 283–345. Benjamin Cummings, Menlo Park, CA, 1988.
- [Gaj88b] D.D. Gajski and D.E. Thomas. Introduction to Silicon Compilation. In D.D. Gajski, editor, *Silicon Compilation*, pages 1–48. Addison-Wesley, Reading, Massachusetts, 1988.
- [Gaj92] D.D. Gajski, N.D. Dutt, A.C.H. Wu, and S.Y.L. Lin. *High-Level Synthesis, Introduction to Chip and System Design*. Kluwer Academic Publishers, Boston, 1992.
- [Gaj94] D.D. Gajski and L. Ramachandran. Introduction to High-Level Synthesis. *IEEE Design and Test of Computers*, pages 44–54, Winter 1994.
- [Gar77] M.R. Garey and D.S. Johnson. The Rectilinear Steiner Tree Problem is NP-complete. *SIAM Journal of Applied Mathematics*, 32(4):826–834, June 1977.
- [Gar79] M.R. Garey and D.S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, 1979.
- [Gar80] M.R. Garey, D.S. Johnson, G.L. Miller, and C.H. Papadimitriou. The Complexity of Coloring Circular Arcs and Chords. *SIAM Journal on Algebraic and Discrete Methods*, 1(2):216–227, June 1980.
- [Gen90] A.J. van Genderen. SLS: An Efficient Switch-Level Timing Simulator Using Min-Max Voltage Waveforms. In G. Musgrave and U. Lauther, editors, *VLSI 89*, pages 79–88. North-Holland, Amsterdam, 1990.
- [Gen91] A.J. van Genderen. *Reduced Models for the Behavior of VLSI Circuits*. PhD thesis, Department of Electrical Engineering, Delft University of Technology, Delft, The Netherlands, 1991.
- [Ger89] S.H. Gerez and O.E. Herrmann. Switchbox Routing by Stepwise Reshaping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(12):1350–1361, December 1989.
- [Ger98] S.H. Gerez, S.M. Heemstra de Groot, E.R. Bonsma, and M.J.M. Heijligers. Overlapped Scheduling Techniques for High-Level Synthesis and Multiprocessor Realizations of DSP Algorithms. In J.C. Lopez, R. Hermida, and W. Geisselhardt, editors, *Advanced Techniques for Embedded System Design and Test*, pages 125–150. Kluwer Academic Publishers, Boston, 1998.
- [Gho92] S. Ghosh and P.A. Subrahmanyam. On the Notion of Control Signals in Digital Designs. *IEEE Circuits & Devices*, pages 47–52, July 1992.
- [Gib85] A. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, Cambridge, 1985.
- [Gil68] E.N. Gilbert and H.O. Pollak. Steiner Minimal Trees. *SIAM Journal of Applied Mathematics*, 16(1):1–29, 1968.

- [Gin84] L.P.P. van Ginneken and R.H.J.M. Otten. Stepwise Layout Refinement. In *International Conference on Computer Design*, pages 30–36, 1984.
- [Glo93] F. Glover, E. Taillard, and D. de Werra. A User's Guide to Tabu Search. In F. Glover, M. Laguna, E. Taillard, and D. de Werra, editors, *Annals of Operations Research*, volume 41, pages 3–28. J.C. Baltzer AG Science Publishers, Basel, Switzerland, 1993.
- [Gol80] M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.
- [Gol89] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, Massachusetts, 1989.
- [Gol96] U. Golze. *VLSI Chip Design with the Hardware Description Language Verilog. An Introduction Based on a Large RISC Processor Design*. Springer Verlag, Berlin, 1996.
- [Goo90] G. Goossens, J. Rabacy, J. Vandewalle, and H. De Man. An Efficient Microcode Compiler for Application Specific DSP Processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(9):925–937, September 1990.
- [Gos93] J.B. Gosling. *Simulation in the Design of Digital Electronic Systems*. Cambridge University Press, Cambridge, UK, 1993.
- [Got86] S. Goto and T. Masuda. Partitioning, Assignment and Placement. In T. Ohtsuki, editor, *Advances in CAD for VLSI, Vol. 4: Layout Design and Verification*, pages 55–97. North-Holland, Amsterdam, 1986.
- [Gri94] J. Griffith, G. Robins, J.S. Salowe, and T. Zhang. Closing the Gap: Near-Optimal Steiner Trees in Polynomial Time. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(11):1351–1365, November 1994.
- [Gup79] U.I. Gupta, D.T. Lee, and J.Y.T. Leung. An Optimal Solution for the Channel-Assignment Problem. *IEEE Transactions on Computers*, C-28(11):807–810, November 1979.
- [Gup92] A. Gupta. Formal Hardware Verification Methods: A Survey. *Formal Methods in System Design*, 1:151–238, 1992.
- [Gup95] R.K. Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded Systems*. Kluwer Academic Publishers, Boston, 1995.
- [Hac89] G.D. Hachtel and C.R. Morrison. Linear Complexity Algorithms for Hierarchical Routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(1):64–80, January 1989.
- [Hac96] G.D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, Boston, 1996.
- [Haj88] B. Hajek. Cooling Schedules for Optimal Annealing. *Mathematics of Operations Research*, 13(2):311–329, May 1988.
- [Han66] M. Hanan. On Steiner's Problem with Rectilinear Distance. *SIAM Journal of Applied Mathematics*, 14(2):255–265, March 1966.
- [Har87] D. Harel. *Algorithmics, The Spirit of Computing*. Addison-Wesley, Wokingham, England, 1987.
- [Har89] B.S. Haroun and M.I. Elmasry. Architectural Synthesis for DSP Silicon Compilers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(4):431–447, April 1989.
- [Har90a] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.
- [Har90b] D.S. Harrison, A.R. Newton, R.L. Spickelmier, and T.J. Barnes. Electronic CAD Frameworks. *Proceedings of the IEEE*, 78(2):393–417, February 1990.

- [Has71] A. Hashimoto and J. Stevens. Wire Routing by Optimizing Channel Assignment within Large Apertures. In *8th Design Automation Workshop*, pages 155–169, 1971.
- [Hay87] J.P. Hayes. An Introduction to Switch-Level Modeling. *IEEE Design and Test of Computers*, 4(4):18–25, August 1987.
- [Hay93] J.P. Hayes. *Introduction to Digital Logic Design*. Addison-Wesley, Reading, Massachusetts, 1993.
- [HdG92] S.M. Heemstra de Groot, S.H. Gerez, and O.E. Herrmann. Range-Chart-Guided Iterative Data-Flow-Graph Scheduling. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 39:351–364, May 1992.
- [Hea85] S.T. Healey and D.D. Gajski. Decomposition of Logic Networks into Silicon. In *22nd Design Automation Conference*, pages 162–168, June 1985.
- [Hea87] S.T. Healey and W.J. Kubitz. Abstract Routing of Logic Networks for Custom Module Generation. In *24th Design Automation Conference*, pages 230–236, 1987.
- [Hei95] M.J.M. Heijliger and J.A.G. Jess. High-Level Synthesis Scheduling and Allocation Using Genetic Algorithms Based on Constructive Topological Scheduling Techniques. In *International Conference on Evolutionary Computation*, pages 56–61, Perth, Australia, 1995.
- [Hei96] M.J.M. Heijliger. *The Application of Genetic Algorithms to High-Level Synthesis*. PhD thesis, Eindhoven University of Technology, Department of Electrical Engineering, October 1996.
- [Hem75] C.W. Hemming and S.A. Szygenda. Register Transfer Language Simulation. In M.A. Breuer, editor, *Digital System Design Automation: Languages, Simulation and Data Base*, pages 219–269. Computer Science Press, Woodland Hills, California, 1975.
- [Hil81] F.J. Hill and G.R. Peterson. *Introduction to Switching Theory and Logical Design*. John Wiley & Sons, New York, third edition, 1981.
- [Hil85] P.N. Hilfinger. A High-Level Language and Silicon Compiler for Digital Signal Processing. In *Custom Integrated Circuit Conference*, pages 213–216, 1985.
- [Hil89] D. Hill, D. Shugard, J. Fishburn, and K. Keutzer. *Algorithms and Techniques for VLSI Layout Synthesis*. Kluwer Academic Publishers, Boston, 1989.
- [Hil92] P. Hilfinger and J. Rabaey. DSP Specification Using the Silage Language. In R.W. Brodersen, editor, *Anatomy of a Silicon Compiler*, pages 199–220. Kluwer Academic Publishers, Boston, 1992.
- [Ho90] J.M. Ho, G. Vijayan, and C.K. Wong. New Algorithms for the Rectilinear Steiner Tree Problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(2):185–193, February 1990.
- [Hon83] S.J. Hong and R. Nair. Wire-Routing Machines - New Tools for Physical VLSI Design. *Proceedings of the IEEE*, 71(1):57–65, January 1983.
- [Hop86] J.J. Hopfield and D.W. Tank. Computing with Neural Circuits: A Model. *Science*, 233:625–633, 1986.
- [Hor78] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, Rockville, Maryland, 1978.
- [Hu69] T.C. Hu. *Integer Programming and Network Flows*. Addison-Wesley, Reading, Massachusetts, 1969.
- [Hwa76] F.K. Hwang. On Steiner Minimal Trees with Rectilinear Distance. *SIAM Journal of Applied Mathematics*, 30(1):104–114, January 1976.
- [Hwa92] F.K. Hwang, D.S. Richards, and P. Winter. *The Steiner Tree Problem*. North-Holland, Amsterdam, 1992. Annals of Discrete Mathematics, no. 53.
- [Jai91] R. Jain, A. Mujumdar, A. Sharma, and H. Wang. Empirical Evaluation of Some High-Level Synthesis Scheduling Heuristics. In *28th Design Automation Conference*, pages 686–689, 1991.

- [Jai96] J. Jain, A. Narayan, C. Coelho, S.P. Khatri, A. Sangiovanni-Vincentelli, R.K. Brayton, and M. Fujita. Decomposition Techniques for Efficient ROBDD Construction. In M. Srivastava and A. Camilleri, editors, *Formal Methods in Computer-Aided Design*. Springer, Berlin, 1996. Lecture Notes in Computer Science, nr. 1166.
- [Jeo93] S.W. Jeong and F. Somenzi. A New Algorithm for 0-1 Programming Based on Binary Decision Diagrams. In T. Sasao, editor, *Logic Synthesis and Optimization*, Boston, 1993. Kluwer Academic Publishers.
- [Jia94] Y.M. Jiang, T.F. Lee, T.T. Hwang, and Y.L. Lin. Performance-Driven Interconnection Optimization for Microarchitecture Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(2):137–149, February 1994.
- [Joh89] D.L. Johannsen. Silicon Compilation. In C.L. Seitz, editor, *Advanced Research in VLSI, Proceedings of the Decennial Caltech Conference on VLSI*, pages 17–36. The MIT Press, Cambridge, Massachusetts, 1989.
- [Joh96] F.M. Johannes. Partitioning of VLSI Circuits and Systems. In *33rd ACM/IEEE Design Automation Conference*, 1996.
- [Joo86] R. Joobani and D. Siewiorek. WEAVER: A Knowledge-Based Routing Expert. *IEEE Design & Test*, 3(1):12–23, February 1986.
- [Kah92a] H.J. Kahn and R.F. Goldman. The Electronic Design Interchange Format EDIF: Present and Future. In *29th Design Automation Conference*, pages 666–671, 1992.
- [Kah92b] A.B. Kahng and G. Robins. A New Class of Steiner Tree Heuristics with Good Performance. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(7):893–902, July 1992.
- [Kah95] A.B. Kahng and G. Robins. *On Optimal Interconnections for VLSI*. Kluwer Academic Publishers, Boston, 1995.
- [Kal93] A. Kalavade and E.A. Lee. A Hardware-Software Codesign Methodology for DSP Applications. *IEEE Design and Test of Computers*, pages 16–28, September 1993.
- [Kam97] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. *Synthesis of Finite State Machines: Functional Optimization*. Kluwer Academic Publishers, Boston, 1997.
- [Kat94] R.H. Katz. *Contemporary Logic Design*. Benjamin/Cummings, Redwood City, California, 1994.
- [Ker70] B.W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell Systems Technical Journal*, 49(2):291–307, February 1970.
- [Ker73] B.W. Kernighan, D.G. Schweikert, and G. Persky. An Optimum Channel-Routing Algorithm for Polycell Layouts of Integrated Circuits. In *10th Design Automation Workshop*, pages 50–59, 1973.
- [Ker88] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1988.
- [Keu96] K. Keutzer. The Need for Formal Methods for Integrated Circuit Design. In M. Srivastava and A. Camilleri, editors, *Formal Methods in Computer-Aided Design*. Springer, Berlin, 1996. Lecture Notes in Computer Science, nr. 1166.
- [Keu97] K. Keutzer and P. Vanbekbergen. Computer-Assisted Design. In D. Christiansen, editor, *Electronic Engineer's Handbook, Fourth Edition*, pages 7.1–7.25. McGraw-Hill, New York, 1997.
- [Kir83] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–690, May 1983.
- [Kli89] R.A. Kling and P. Banerjee. ESP: Placement by Simulated Evolution. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(3):245–256, March 1989.
- [Knu76] D.E. Knuth. Big Omicron and Big Omega and Big Theta. *ACM SIGACT News*, 8(2):18–24, April-June 1976.

- [Kon95] J.T. Kong and D. Overhauser. *Digital Timing Macromodeling for VLSI Design Verification*. Kluwer Academic Publishers, Boston, 1995.
- [Kos95] M.S. Koster and S.H. Gerez. List Scheduling for Iterative Data-Flow Graphs. In *GRONICS '95, Groningen Information Technology Conference for Students*, pages 123–130, February 1995.
- [Kri84] B. Krishnamurthy. An Improved Min-Cut Algorithm for Partitioning VLSI Networks. *IEEE Transactions on Computers*, C-33(5):438–446, May 1984.
- [Ku92] D.C. Ku and G. De Micheli. *High Level Synthesis of ASICs Under Timing and Synchronization Constraints*. Kluwer Academic Publishers, Boston, 1992.
- [Kuh86] E.S. Kuh and M. Marek-Sadowska. Global Routing. In T. Ohtsuki, editor, *Advances in CAD for VLSI, Vol.4: Layout Design and Verification*, pages 169–198. North-Holland, Amsterdam, 1986.
- [Kuh90] E.S. Kuh and T. Ohtsuki. Recent Advances in VLSI Layout. *Proceedings of the IEEE*, 78(2):237–263, February 1990.
- [Kuh91] E.S. Kuh, A. Srinivasan, M.A.B. Jackson, M. Pedram, Y. Ogawa, and M. Marek-Sadowska. Timing-Driven Layout. In R.W. Dutton, editor, *VLSI Logic Synthesis and Design*, pages 263–270. IOS Press, Amsterdam, 1991.
- [Laa87] P.J.M. van Laarhoven and E.H.L. Aarts. *Simulated Annealing: Theory and Applications*. D. Reidel, Dordrecht, 1987.
- [Law76] E.L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York, 1976.
- [Lee59] C.Y. Lee. Representation of Switching Circuits by Binary-Decision Programs. *The Bell System Technical Journal*, 38:985–999, July 1959.
- [Lee61] C.Y. Lee. An Algorithm for Path Connections and Its Applications. *IRE Transactions on Electronic Computers*, EC-10:346–365, September 1961.
- [Lee87] E.A. Lee and D.G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.
- [Lee88] E.A. Lee. Recurrences, Iteration and Conditionals in Statically Scheduled Block Diagram Languages. In R.W. Brodersen and H.S. Mosowitz, editors, *VLSI Signal Processing III*, pages 330–340. IEEE Press, New York, 1988.
- [Lee89] E.A. Lee, W.H. Ho, E.E. Goel, J.C. Bier, and S. Bhattacharyya. Gabriel: A Design Environment for DSP. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37(11):1751–1762, November 1989.
- [Lee91a] E.A. Lee. Consistency in Dataflow Graphs. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):223–235, April 1991.
- [Lee91b] K.W. Lee and C. Sechen. A Global Router for Sea-of-Gates Circuits. In *European Design Automation Conference (EDAC '91)*, pages 242–247, 1991.
- [Lee94] T.F. Lee, A.C.H. Wu, Y.L. Lin, and D.D. Gajski. A Transformation-Based Method for Loop Folding. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):439–450, April 1994.
- [Lee95] E.A. Lee and T.M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–799, May 1995.
- [Lei83] C.E. Leiserson, F.M. Rose, and J.B. Saxe. Optimizing Synchronous Circuitry by Retiming (Preliminary Version). In R. Bryant, editor, *Third Caltech Conference on VLSI*, pages 87–116. Springer Verlag, Berlin, 1983.
- [Lei91] C.E. Leiserson and J.B. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, 6:5–35, 1991.
- [Len90] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, Chichester, 1990.

- [Len93] T. Lengauer and R. Mueller. Robust and Accurate Hierarchical Floorplanning with Integrated Global Wiring. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(6):802–809, June 1993.
- [Leu97] R. Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, Boston, 1997.
- [Lia83] Y.Z. Liao and C.K. Wong. An Algorithm to Compact a VLSI Symbolic Layout with Mixed Constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-2(2):62–69, April 1983.
- [Lie98] C. Liem. *Retargetable Compilers for Embedded Core Processors, Methods and Experiences in Industrial Applications*. Kluwer Academic Publishers, Boston, 1998.
- [Lig87] M.R. Lightner. Modeling and Simulation of VLSI Digital Systems. *Proceedings of the IEEE*, 75(6):786–796, June 1987.
- [Lin89] Y.-L. Lin, Y.-C. Hsu, and F.-S. Tsai. SILK: A Simulated Evolution Router. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(10):1108–1114, October 1989.
- [Lin90] B. Lin and F. Somenzi. Minimization of Symbolic Relations. In *International Conference on Computer-Aided Design*, pages 88–91, 1990.
- [Lin97] Y.L. Lin. Recent Developments in High-Level Synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 2(1):2–21, January 1997.
- [Lip89] R. Lipsett, C.F. Schaeffer, and C. Ussery. *VHDL: Hardware Description and Design*. Kluwer Academic Publishers, Boston, 1989.
- [Mac96] G.A.S. Machado, editor. *Low Power HF Microelectronics, A Unified Approach*. The Institution of Electrical Engineers, London, 1996.
- [Mal88] S. Malik, A.R. Wang, R.K. Brayton, and A. Sangiovanni-Vincentelli. Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment. In *International Conference on Computer-Aided Design*, pages 6–9, 1988.
- [Man97] M.M. Mano and C.R. Kime. *Logic and Computer Design Fundamentals*. Prentice Hall International, London, 1997.
- [Mar93] P. Marwedel. *Synthese und Simulation von VLSI-Systemen, Algorithmen für den rechnerunterstützten Entwurf hochintegrierter Schaltungen*. Carl Hanser Verlag, Munich, 1993. In German.
- [Mar95] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, Boston, 1995.
- [Mau90] P.M. Maurer and Z. Wang. Techniques for Unit-Delay Compiled Simulation. In *27th ACM/IEEE Design Automation Conference*, pages 480–484, 1990.
- [May86] R.N. Mayo. Mocha Chip: A System for the Graphical Design of VLSI Module Generators. In *International Conference on Computer-Aided Design*, pages 74–77, 1986.
- [McF90] M.C. McFarland, A.C. Parker, and R. Camposano. High-Level Synthesis of Digital Systems. *Proceedings of the IEEE*, 78(2):301–318, February 1990.
- [McH90] J.A. McHugh. *Algorithmic Graph Theory*. Prentice Hall International, London, 1990.
- [Mca80] C. Mead and L. Conway. *Introduction to VLSI Systems*. Addison-Wesley, Reading, Massachusetts, 1980.
- [Mee93] J.L. van Meerbergen, P.E.R. Lippens, W.F.J. Verhaegh, and A. van der Werf. Relative Location Assignment for Repetitive Schedules. In *European Conference on Design Automation with the European Event on ASIC Design, EDAC/EUROASIC*, pages 403–407, 1993.
- [Mee95] J.L. van Meerbergen, P.E.R. Lippens, W.F.J. Verhaegh, and A. van der Werf. PHIDEO: High-Level Synthesis for High Throughput Applications. *Journal of VLSI Signal Processing*, 9:89–104, 1995.

- [Mic92a] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, Berlin, 1992.
- [Mic92b] P. Michel, U. Lauther, and P. Duzy, editors. *The Synthesis Approach to Digital System Design*. Kluwer Academic Publishers, Boston, 1992.
- [Mil94] G. Milne. *Formal Specification and Verification of Digital Systems*. McGraw-Hill Book Company, London, 1994.
- [Min96] S. Minato. *Binary Decision Diagrams and Applications to VLSI CAD*. Kluwer Academic Publishers, Boston, 1996.
- [Mly86] D.A. Mlynki and C.H. Sung. Layout Compaction. In T. Ohtsuki, editor, *Advances in CAD for VLSI, Vol. 4: Layout Design and Verification*, pages 199–235. North-Holland, Amsterdam, 1986.
- [Mor70] E. Morreale. Recursive Operators for Prime Implicant and Irredundant Normal Form Determination. *IEEE Transactions on Computers*, C-19(6):504–509, June 1970.
- [Mor90] M.J. Morant. *Integrated Circuit Design and Technology*. Chapman and Hall, London, 1990.
- [Mor91] B.M.E. Moret and H.D. Shapiro. *Algorithms from P to NP. Volume I: Design & Efficiency*. Benjamin Cummings, Redwood City, CA, 1991.
- [Mos87] R.C. Mosteller, A.H. Frey, and R. Suaya. 2-D Compaction, A Monte Carlo Method. In P. Losleben, editor, *Advanced Research in VLSI: Proceedings of the 1987 Stanford Conference*, Cambridge, Massachusetts, 1987. The MIT Press.
- [MS92] M. Marek-Sawowska. Switch Box Routing: A Retrospective. *Integration, The VLSI Journal*, 13:39–65, 1992.
- [Muk86] A. Mukherjee. *Introduction to nMOS & CMOS VLSI Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
- [Mur95] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani. Rectangle-Packing-Based Module Placement. In *International Conference on Computer-Aided Design*, 1995.
- [Nah89] S. Nahar, S. Sahni, and E. Shragowitz. Simulated Annealing and Combinatorial Optimization. *International Journal of Computer Aided VLSI Design*, 1(1):1–23, 1989.
- [Nav93] Z. Navabi. *VHDL. Analysis and Modeling of Digital Systems*. McGraw Hill, New York, 1993.
- [Nay97] D. Naylor and S. Jones. *VHDL: A Logic Synthesis Approach*. Chapman and Hall, London, 1997.
- [Neb97] W. Nebel and J. Mermet, editors. *Low Power Design in Deep Submicron Electronics*. Kluwer Academic Publishers, Dordrecht, 1997.
- [Nem88] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons, New York, 1988.
- [Nem89] G.L. Nemhauser and L.A. Wolsey. Integer Programming. In G.L. Nemhauser, A.H.G. Rinnooy Kan, and M.J. Todd, editors, *Handbooks in Operations Research and Management Science: Optimization*, volume 1. North-Holland, Amsterdam, 1989.
- [New81] A.R. Newton. Computer-Aided Design of VLSI Circuits. *Proceedings of the IEEE*, 69(10):1189–1199, October 1981.
- [New86] A.R. Newton and A.L. Sangiovanni-Vincentelli. Computer-Aided Design for VLSI Circuits. *IEEE Computer*, pages 38–60, April 1986.
- [O'B93] K. O'Brien, M. Rahmouni, and A. Jerraya. DLS: A Scheduling Algorithm for High-Level Synthesis in VHDL. In *European Design Automation Conference EDAC/EuroAsic*, pages 393–397, 1993.
- [Ogr94] J. Ogródzki. *Circuit Simulation Methods and Algorithms*. CRC Press, Boca Raton, 1994.

- [Oht86] T. Ohtsuki. Maze-Running and Line-Search Algorithms. In T. Ohtsuki, editor, *Advances for CAD in VLSI, Vol. 4: Layout Design and Verification*, pages 99–131. North-Holland, Amsterdam, 1986.
- [Olá92] A. Oláh, S.H. Gerez, and S.M. Heemstra de Groot. Scheduling and Allocation for the High-Level Synthesis of DSP Algorithms by Exploitation of Data Transfer Mobility. In *International Conference on Computer Systems and Software Engineering, CompEuro 92*, pages 145–150, May 1992.
- [Ott83] R.H.J.M. Otten. Efficient Floorplan Optimization. In *International Conference on Computer Design*, pages 499–502, 1983.
- [Ott88] R.H.J.M. Otten. Graphs in Floor-Plan Design. *International Journal of Circuit Theory and Applications*, 16:391–410, 1988.
- [Ott89] R.H.J.M. Otten and L.P.P.P. van Ginneken. *The Annealing Algorithm*. Kluwer Academic Publishers, Boston, 1989.
- [Ous85] J.K. Ousterhout, G.T. Hamachi, R.N. Mayo, W.S. Scott, and G.S. Taylor. The MAGIC VLSI Layout System. *IEEE Design & Test of Computers*, 2(1):19–30, 1985.
- [Pan94] P. Pan, W. Shi, and C.L. Liu. Area Minimization for Hierarchical Floorplans. In *International Conference on Computer-Aided Design*, pages 436–440, 1994.
- [Pan95] P. Pan and C.L. Liu. Area Minimization for Floorplans. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(1):123–132, January 1995.
- [Pap82] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization, Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, New Jersey, 1982.
- [Pap94] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Massachusetts, 1994.
- [Par86] A.C. Parker, J.T. Pizarro, and M. Mlinar. MAHA: A Program for Datapath Synthesis. In *23rd Design Automation Conference*, pages 461–466, 1986.
- [Par89] K.K. Parhi. Algorithm Transformation Techniques for Concurrent Processors. *Proceedings of the IEEE*, 77(12):1879–1895, December 1989.
- [Par91] K.K. Parhi and D.G. Messerschmitt. Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding. *IEEE Transactions on Computers*, 40(2):178–195, February 1991.
- [Par95] K.K. Parhi. High-Level Algorithm and Architecture Transformations for DSP Synthesis. *Journal of VLSI Signal Processing*, 9:121–143, 1995.
- [Pau89] P.G. Paulin and J.P. Knight. Force-Directed Scheduling for the Behavioral Synthesis of ASIC's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(6):661–679, June 1989.
- [Pau91] P.G. Paulin. Global Scheduling and Allocation Algorithms in the HAL System. In R. Camposano and W. Wolf, editors, *High-Level VLSI Synthesis*, pages 255–281. Kluwer Academic Publishers, Boston, 1991.
- [Ped96] M. Pedram. Power Minimization in IC Design: Principles and Applications. *ACM Transactions on Design Automation of Electronic Systems*, 1(1), January 1996.
- [Phi95] W.J.M. Philipsen. *Optimization with Potts Neural Networks in High Level Synthesis*. PhD thesis, Department of Electrical Engineering, Eindhoven University of Technology, January 1995.
- [Pig91] C. Piguet and E. Dijkstra. Design Methodologies and CAD Tools. *Integration, The VLSI Journal*, 10(3):219–250, 1991.
- [Pot92] M. Potkonjak and J.M. Rabaey. Scheduling Algorithms for Hierarchical Data Control Flow Graphs. *International Journal of Circuit Theory and Applications*, 20:217–233, 1992.
- [Pot94] M. Potkonjak and J. Rabaey. Optimizing Resource Utilization Using Transformations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(3):277–292, March 1994.

- [Pre88] B.T. Preas and P.G. Karger. Placement, Assignment and Floorplanning. In B.T. Preas and M.J. Lorenzetti, editors, *Physical Design Automation of VLSI Systems*, pages 87–155. Benjamin Cummings, Menlo Park, CA, 1988.
- [Pri57] R.C. Prim. Shortest Connection Networks and Some Generalizations. *Bell System Technical Journal*, 36:1389–1401, November 1957.
- [Puc88] D.A. Pucknell and K. Eshraghian. *Basic VLSI Design, Systems and Circuits*. Prentice-Hall, New York, second edition, 1988.
- [Puc90] D.A. Pucknell. *Fundamentals of Digital Logic Design: with VLSI Circuit Applications*. Prentice Hall, New York, 1990.
- [Rab95] J.M. Rabaey and M. Pedram, editors. *Low Power Design Methodologies*. Kluwer Academic Publishers, Boston, 1995.
- [Rab96] J.M. Rabaey. *Digital Integrated Circuits, A Design Perspective*. Prentice Hall, Upper Saddle River, New Jersey, 1996.
- [Ram88] J. Ramanujam and P. Sadayappan. Optimization by Neural Networks. In *IEEE International Conference on Neural Networks*, pages II-325–332, 1988.
- [Rao89] V.B. Rao, D.V. Overhauser, T.N. Trick, and I.N. Hajj. *Switch-Level Timing Simulation of MOS VLSI Circuits*. Kluwer Academic Publishers, Boston, 1989.
- [Reb96] M. Rebaudengo and M.S. Reorda. GALLO: A Genetic Algorithm for Floorplan Area Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(8):943–951, August 1996.
- [Ree85] J. Reed, A. Sangiovanni-Vincentelli, and M. Santomauro. A New Symbolic Channel Router: YACR2. *IEEE Transactions on Computer-Aided Design of Integrated Circuits, CAD-4(3)*:208–219, July 1985.
- [Ree95] C.R. (Ed.) Reeves. *Modern Heuristic Techniques for Combinatorial Optimization*. McGraw-Hill Book Company, London, 1995.
- [Rim94] M. Rim, A. Majumdar, R. Jain, and R. De Leone. Optimal and Heuristic Algorithms for Solving the Binding Problem. *IEEE Transactions on Very Large Scale Integration Systems*, 2(2):211–225, June 1994.
- [Rob87] R.A. Roberts and C.T. Mullis. *Digital Signal Processing*. Addison-Wesley, Reading, Massachusetts, 1987.
- [Row91] J. Rowson. Computer-Aided Design Tools and Systems. In N.G. Einspruch and J.L. Hilbert, editors, *Application Specific Integrated Circuit (ASIC) Technology*, pages 125–183. Academic Press, San Diego, 1991.
- [Rub74] F. Rubin. The Lee Path Connection Algorithm. *IEEE Transactions on Computers*, C-23(9):907–914, September 1974.
- [Rud93] R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *International Conference on Computer-Aided Design*, pages 42–47, 1993.
- [Rud96] R. Rudell. Tutorial: Design of a Logic Synthesis System. In *33rd Design Automation Conference*, 1996.
- [Rus85] G. Russell, D.J. Kinniment, E.G. Chester, and M.R. McLauchlan. *CAD for VLSI*. Van Nostrand Reinhold, Wokingham, England, 1985.
- [Rut89] R.A. Rutenbar. Simulated Annealing Algorithms: An Overview. *IEEE Circuits and Devices Magazine*, pages 19–26, January 1989.
- [Sai95] S.M. Sait and H. Youssef. *VLSI Physical Design Automation*. McGraw-Hill, London, 1995.
- [Sal94] R. Saleh, S.J. Jou, and A.R. Newton. *Mixed-Mode Simulation and Analog Multilevel Simulation*. Kluwer Academic Publishers, Boston, 1994.
- [Sap93] S.S. Sapatnekar and S.M. Kang. *Design Automation for Timing-Driven Layout Synthesis*. Kluwer Academic Publishers, Boston, 1993.

- [Sar96] M. Sarrafzadeh and C.K. Wong. *An Introduction to VLSI Physical Design*. McGraw-Hill, New York, 1996.
- [Sch72] D.G. Schweikert and B.W. Kernighan. A Proper Model for the Partitioning of Electrical Circuits. In *Proceedings of the ACM IEEE Design Automation Workshop*, pages 57–62, 1972.
- [Sch83a] W.L. Schiele. On a Longest Path Algorithm and Its Complexity If Applied to the Layout Compaction Problem. In *European Conference on Circuit Theory and Design*, pages 263–265, 1983.
- [Sch83b] M. Schlag, Y.Z. Liao, and C.K. Wong. An Algorithm for Optimal Two-Dimensional Compaction of VLSI Layouts. *Integration, The VLSI Journal*, 1:179–209, 1983.
- [Sco86] W.A. Scott and J.K. Ousterhout. Magic's Circuit Extractor. *IEEE Design and Test of Computers*, 3(1):24–34, February 1986.
- [Sec85] C. Sechen and A. Sangiovanni-Vincentelli. The TimberWolf Placement and Routing Package. *IEEE Journal of Solid-State Circuits*, SC-20(2):510–522, April 1985.
- [Sed88] R. Sedgewick. *Algorithms*. Addison-Wesley, Reading, Massachusetts, second edition, 1988.
- [Sed90] R. Sedgewick. *Algorithms in C*. Addison-Wesley, Reading, Massachusetts, 1990.
- [Sen96] E.M. Sentovich. A Brief Study of BDD Package Performance. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design*. Springer, Berlin, 1996. Lecture Notes in Computer Science, nr. 1166.
- [Séq83] C.H. Séquin. Managing VLSI Complexity: An Outlook. *Proceedings of the IEEE*, 71(1):149–166, January 1983.
- [Séq87] C.H. Séquin. Design and Layout Generation at the Symbolic Level. In W. Fichtner and M. Morf, editors, *VLSI CAD Tools and Applications*, pages 213–231. Kluwer Academic Publishers, Boston, 1987.
- [Sha90] K. Shahookar and P. Mazumder. A Genetic Approach to Standard Cell Placement Using Meta-Genetic Parameter Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(5):500–511, May 1990.
- [She93] N.A. Sherwani. *Algorithms for VLSI Physical Design Automation*. Kluwer Academic Publishers, Boston, 1993.
- [She95] N. Sherwani, S. Bhingarde, and A. Panyam. *Routing in the Third Dimension, From VLSI Chips to MCMS*. IEEE Press, New York, 1995.
- [Shi87] H. Shin and A. Sangiovanni-Vincentelli. A Detailed Router Based on Incremental Routing Modifications: Mighty. *IEEE Transactions on Computer Aided Design of Integrated Circuits*, CAD-6(6):942–955, November 1987.
- [Shi88] T. Shiple, P. Kollaritsch, D. Smith, and J. Allen. Area Evaluation Metrics for Transistor Placement. In *International Conference on Computer Design*, pages 428–433, 1988.
- [Shi95] W. Shi. An Optimal Algorithm for Area Minimization of Slicing Floorplans. In *International Conference on Computer-Aided Design*, pages 480–484, 1995.
- [Smi87] S.P. Smith, M.R. Mercer, and B. Brock. Demand Driven Simulation: BACKSIM. In *24th ACM/IEEE Design Automation Conference*, pages 181–187, 1987.
- [Smi88] S.G. Smith, M. Keightley, P.B. Denyer, and S. Nagara. SECOND: Synthesis of Elementary Circuits on Demand. *IEEE Journal of Solid-State Circuits*, 23(3):722–727, June 1988.
- [Smi97] M.J.S. Smith. *Application-Specific Integrated Circuits*. Addison-Wesley, Reading, Massachusetts, 1997.

- [Spa85] L. Spaanenburg, M. Beunder, F.A. Beune, S.H. Gerez, B. Holstein, R.C.C. Luchtmeijer, J. Smit, A. van der Werf, and H. Willems. MOD/R: A Knowledge Assisted Approach towards Top-Down only CMOS VLSI Design. *Microprocessing and Microprogramming, The Euromicro Journal*, 16(2 & 3):83–88, 1985.
- [Spr94] D.L. Springer and D.E. Thomas. Exploiting the Special Structure of Conflict and Compatibility Graphs in High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(7):843–856, July 1994.
- [Sta98] J. Staunstrup and W. Wolf. *Hardware/Software Co-Design: Principles and Practice*. Kluwer Academic Publishers, Boston, 1998.
- [Sto83] L. Stockmeyer. Optimal Orientations of Cells in Slicing Floorplan Designs. *Information and Control*, 57:91–101, 1983.
- [Sto92] L. Stok and J.A.G. Jess. Foreground Memory Management in Data Path Synthesis. *International Journal of Circuit Theory and Applications*, 20:235–255, 1992.
- [Sto94] L. Stok. Data Path Synthesis. *Integration, The VLSI Journal*, 18:1–71, 1994.
- [Sun95] W.J. Sun and C. Sechen. Efficient and Effective Placement for Very Large Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(3):349–359, March 1995.
- [Suz86] K. Suzuki, Y. Matsunaga, M. Tachibana, and T. Ohtsuki. A Hardware Maze Router with Application to Interactive Rip-Up and Reroute. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, CAD-5(4):466–476, October 1986.
- [Szy85] T.G. Szymanski. Dogleg Channel Routing is NP-Complete. *IEEE Transactions on Computer-Aided Design*, CAD-4(1):31–41, January 1985.
- [Szy88] T.G. Szymanski and C.J. Van Wyk. Layout Analysis and Verification. In B.T. Preas and M.J. Lorenzetti, editors, *Physical Design Automation of VLSI Systems*, pages 347–407. Benjamin Cummings, Menlo Park, California, 1988.
- [Ter85] C.J. Terman. Timing Simulation for Large Digital MOS Circuits. In A. Sangiovanni-Vincentelli, editor, *Advances in Computer-Aided Engineering Design*, volume 1, pages 1–92. JAI Press, Greenwich, Connecticut, 1985.
- [Tho91] D.E. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, Boston, 1991.
- [Tri81] S. Trimberger, J.A. Rowson, C.L. Lang, and J.P. Gray. A Structured Design Methodology and Associated Software Tools. *IEEE Transactions on Circuits and Systems*, CAS-28(7):618–634, July 1981.
- [Tse86] C.J. Tseng and D.P. Siewiorek. Automated Synthesis of Data Paths in Digital Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-5(3):379–395, July 1986.
- [Ull84] J.D. Ullman. *Computational Aspects of VLSI*. Computer Science Press, Rockville, Maryland, 1984.
- [Ulr69] E.G. Ulrich. Exclusive Simulation of Activity in Digital Networks. *Communications of the ACM*, 12(2):102–110, February 1969.
- [Uye95] J.P. Uyemura. *Physical Design of CMOS Integrated Circuits Using L-EDIT*. PWS Publishing Company, Boston, 1995.
- [Van93] J. Vanhoof, K. Van Rompaey, I. Bolsens, G. Goossens, and H. De Man. *High-Level Synthesis for Real-Time Digital Signal Processing*. Kluwer Academic Publishers, Dordrecht, 1993.
- [Ver91] W.F.J. Verhaegh, E.H.L. Aarts, J.H.M. Korst, and P.E.R. Lippens. Improved Force-Directed Scheduling. In *European Design Automation Conference*, pages 430–435, 1991.
- [Ver92] W.F.J. Verhaegh, P.E.R. Lippens, E.H.L. Aarts, J.H.M. Korst, A. van der Werf, and J.L. van Meerbergen. Efficiency Improvements for Force-Directed Scheduling. In *International Conference on Computer-Aided Design*, pages 286–291, 1992.

- [Ver97] W.F.J. Verhaegh, P.E.R. Lippens, E.H.L. Aarts, and J.L. van Meerbergen. Multidimensional Periodic Scheduling: A Solution Approach. In *European Design and Test Conference, ED&TC '97*, pages 468–474, 1997.
- [Vil97] T. Villa, T. Kam, R.K. Brayton, and A. Sangiovanni-Vincentelli. *Synthesis of Finite State Machines: Logic Optimization*. Kluwer Academic Publishers, Boston, 1997.
- [Vla83] J. Vlach and K. Singhal. *Computer Methods for Circuit Analysis and Design*. Van Nostrand Reinhold, New York, 1983.
- [Wal85] R.A. Walker and D.E. Thomas. A Model of Design Representation and Synthesis. In *22nd Design Automation Conference*, pages 453–459, 1985.
- [Wal91] R.A. Walker and R. Camposano. *A Survey of High-Level Synthesis Systems*. Kluwer Academic Publishers, Boston, 1991.
- [Wal95] R.A. Walker and S. Chaudhuri. Introduction to the Scheduling Problem. *IEEE Design and Test of Computers*, pages 60–69, Summer 1995.
- [Wan90] Z. Wang and P.M. Maurer. LECSIM: A Levelized Event Driven Compiled Logic Simulator. In *27th ACM/IEEE Design Automation Conference*, pages 491–496, 1990.
- [Wan95] C.Y. Wang and K.K. Parhi. High-Level DSP Synthesis Using Concurrent Transformations, Scheduling and Allocation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(3):274–295, March 1995.
- [War90] S.A. Ward and R.H. Halstead. *Computation Structures*. The MIT Press, Cambridge, Massachusetts, 1990.
- [Wes81] N. Weste. Virtual Grid Symbolic Layout. In *18th Design Automation Conference*, pages 225–233, 1981.
- [Wes93] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design, A Systems Perspective*. Addison-Wesley, Reading, Massachusetts, second edition, 1993.
- [Wil86] H.S. Wilf. *Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
- [Wim89] S. Wimer, I. Koren, and I. Cederbaum. Optimal Aspect Ratios of Building Blocks in VLSI. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(2):139–145, February 1989.
- [Wol88] W.H. Wolf and A.E. Dunlop. Symbolic Layout and Compaction. In B.T. Preas and M.J. Lorenzetti, editors, *Physical Design Automation of VLSI Systems*, pages 211–281. Benjamin Cummings, Menlo Park, California, 1988.
- [Wol92] W. Wolf, A. Takach, C.Y. Huang, R. Mannu, and E. Wu. The Princeton University Behavioral Synthesis System. In *29th Design Automation Conference*, pages 182–187, 1992.
- [Wol94a] P. van der Wolf. *CAD Frameworks, Principles and Architecture*. Kluwer Academic Publishers, Boston, 1994.
- [Wol94b] W. Wolf. *Modern VLSI Design, A Systems Approach*. PTR Prentice Hall, Englewood Cliffs, New Jersey, 1994.
- [Wol94c] W.H. Wolf. Hardware-Software Co-Design of Embedded Systems. *Proceedings of the IEEE*, 82(7):967–989, July 1994.
- [Won88] D.F. Wong, H.W. Leong, and C.L. Liu. *Simulated Annealing for VLSI Design*. Kluwer Academic Publishers, Boston, 1988.
- [Yoc90] M. Yoeli, editor. *Formal Verification of Hardware Design*. IEEE Computer Society Press, Los Alamitos, California, 1990.
- [Yoc91] U. Yoeli. A Robust Channel Router. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(2):212–219, February 1991.
- [Yos86] K. Yoshida. Layout Verification. In T. Ohtsuki, editor, *Advances in CAD for VLSI, Vol. 4: Layout Design and Verification*, pages 237–265. North-Holland, Amsterdam, 1986.

- [Zim86] G. Zimmermann. Top-Down Design of Digital Systems. In E. Hoerbst, editor, *Advances in CAD for VLSI, Vol 2: Logic Design and Simulation*. North-Holland, Amsterdam, 1986.
- [Zob93] G.W. Zobrist, editor. *VLSI Fault Modeling and Testing Techniques*. Ablex Publishing Corporation, Norwood, NJ, 1993.

Index

- abstract
 - data type, 177, 206
 - routing, 125
- abstraction, 4, 104, 155, 167
 - level, 167
- abutment, 106, 123
- acyclic data-flow graph, 248
- adder, 14, 236
- adjacency
 - list, 25, 187
 - matrix, 24
- adjacent, 22, 199
- agent, 262
- ALAP scheduling, 254
- algorithmic
 - graph theory, 21
 - transformation, 269
- allocation, 248
- analog, 167, 277
- analysis tool, 8
- application-specific integrated circuit, 3
- approximation algorithm, 49, 53
- arc, 22
- area, 4, 125, 251
 - routing, 134, 156
- arithmetic logic unit, 236, 238, 260
- ASAP scheduling, 253, 254
- assignment, 247, 261, 267
 - register, *see* register assignment
- associativity, 267
- asynchronous circuit, 236
- atomic node, 240
- automatic test-pattern generation, 14
- average-case time complexity, 28, 67
- backtracking, 56
- backward edge, 93
- BDD package, 206, 214, 216
- behavior, 168, 196, 217
- behavioral domain, 5, 119
- Bellman-Ford algorithm, 95, 96
- bidirectional signal flow, 180, 279
- big
 - $O(\mathcal{O})$, 28
 - $\Omega(\Omega)$, 28
 - $\Theta(\Theta)$, 28
- binary
 - decision diagram, 195
 - ordered, *see* ordered *etc.*
 - reduced ordered, *see* reduced *etc.*
 - signal, 170
 - binate covering, 222
- binding, 248
- bipartite graph, 22, 104
- bipartitioning, 108
- Blake's canonical form, 198
- blocking (in switch-level simulation), 184
- Boltzmann distribution, 72
- Boole, 202
- Boolean
 - algebra, 5, 201, 281
 - function, 196, 201
 - signal, 170, 174
 - value, 196, 277
 - variable, 196
- bottom-up design, 7, 119
- branch, 22
- branch-and-bound, 59, 64, 225
- breadboarding, 17
- breadth-first search, 32, 62, 91
- break point, 127
- buffer, 240, 244
- building block
 - layout, 107, 138, 153
 - placement, 106, 111

- bus, 237, 262
- C (programming language), *see* pseudo-code
- canonical form, 66, 197, 198, 202
strong, *see* strong canonicity
- capacitance, 15, 152, 168, 172, 181, 191
parasitic, *see* parasitic capacitance
- capacitive coupling, 139
- capacitor, 168
- cell, 6, 54, 98, 102, 119, 121, 150
compiler, 16, 125
generation, 125
- center of gravity, 111
- chaining, 238
- channel, 106, 123, 133, 150
definition, 153
density, *see* density
height, 138
of CMOS transistor, 283
ordering, 153
routing, 134, 138
model, 139
- channel-connected component, 184
- characterization, 15, 125
- charge sharing, 183
- charged net, 186
- checking
solution, *see* solution checking
- child, 58, 75, 121, 203
- chromosome, 75, 78
- circuit, *see* cycle
extraction, 15, 17
representation, 101, 173, 246
- circuit-level simulation, 15, 168, 181
- circular-arc graph, 264
- clique, 22, 104, 106, 108, 112
partitioning, 265
- clock, 236, 238
phase, 238
two-phase, *see* two-phase clock
- clustering, 108, 109
- CMOS
complex gate, 279
domino logic, *see* domino logic
inverter, 279
static, *see* static CMOS
technology, 9, 183, 222, 277
- co-design, *see* hardware-software co-design
- co-NP-complete, 215
- co-simulation, *see* hardware-software co-simulation
- code generation, 13
- cofactor, 202, 213
- coloring, *see* graph coloring
- column dominance, 224
- combinational logic, 168, 173, 195, 213, 217, 235
- combinatorial optimization problem, 42, 53
- decision version, 44
- evaluation version, 43
- optimization version, 43
- commutativity, 267
- compaction, 17, 83, 86
one-dimensional, *see* one-dimensional compaction
two-dimensional, *see* two-dimensional compaction
- compatibility graph, 263
- compiler-driven simulation, 173, 180
- complement, 196
graph, 263
- complementary network, 182, 279
- complete graph, 22, 49, 263
- complex gate, *see* CMOS complex gate
- complexity
class, 45
NP, *see* NP
NPC, *see* NPC
P, *see* P
- computational, *see* computational complexity
- space, *see* space complexity
- time, *see* time complexity
- component
channel-connected, *see* channel-connected component
connected, *see* connected component
strongly connected, *see* strongly etc.
weakly connected, *see* weakly etc.
- composite
cell, 121, 127
node, 240
- composition problem, 213
- computational complexity, 21, 26
- geometry, 17
- node, 239
step, 28
- conditional node, 240, 241

- conductance, 181
conflict graph, 263, 264
connected
 channel, *see* channel-connected component
 component, 23
 graph, 23
 strongly, *see* strongly connected *etc.*
vertices, 23
 weakly, *see* weakly connected
connectivity modeling, 172, 181
constant time, 29
constraint
 explicit, *see* explicit constraint generation, 98
 graph, 89, 93, 97, 98
 vertical, *see* vertical *etc.*
 implicit, *see* implicit constraint
constructive placement, 108
contact cut, 86, 133, 283
continuous optimization problem, 42
control
 circuitry, 238
 signal, 238
 step, 238, 252
control-dominated application, 13, 247
control-flow graph, 239
controller, 238
 synthesis, 238
Cook's theorem, 48
cooling schedule, 73
correct by construction, 8, 18
cost, 251
 function, 42, 56, 71
 for VLSI, *see* VLSI cost function
cover
 irredundant prime, *see* irredundant *etc.*
covering
 binate, *see* unate covering
 matrix, 221
 set, *see* set covering
 unate, *see* unate covering
critical
 path, 97, 152, 260, 268, 270
 list scheduling, 260
 sink, 152
crossover, 75, 78
 order, *see* order crossover
crosstalk, 139
cube, 197, 222
current source, 168
cycle, 23, 24, 238
 directed, *see* directed cycle
 Hamiltonian, *see* Hamiltonian cycle
 negative, *see* negative cycle
 positive, *see* positive cycle
 simple, *see* simple cycle
cyclic core, 225
data
 dependency, 240
 management, 18
 model, 102
 path, 238
 synthesis, 238
 signal, 238
 structure (for graphs), 24
data-dominated application, 247, 267
data-flow graph, 239, 266
 acyclic, *see* acyclic *etc.*
dc-set, 196, 217, 223, 226
De Morgan's Rule, 201, 222, 281
deadlock, 244
decision
 problem, 44, 48, 138
 version, 44
decomposition tree, 5
degree, 22, 158
delay, 238
 modeling, 152, 171
 node, 245, 270
demand-driven simulation, 180
density, 143, 152, 153, 155
 local, *see* local density
depletion nMOS, 183
depth-first search, 30, 62, 153, 162, 184, 188
description
 structural, *see* structural description
design
 action, 7
 constraint, 4
 domain, 5, 119
 management tool, 8, 18
 methodology, 3, 119, 124
 floorplan-based, *see* floorplan-based *etc.*
rule, 17, 83, 125, 135
 checker, 17, 85
time, 4

verification, *see* verification
 detailed routing, *see* local routing
 device-level simulation, 167
 diffusion, 282
 digital
 abstraction, 277
 domain, 167
 signal
 processing, 239, 244, 269
 processor, 3
 digraph, 23
 Dijkstra's algorithm, 34, 49, 63, 97
 directed
 acyclic graph, 89, 96
 cycle, 24, 90
 graph, 23
 path, 24
 disconnected
 graph, 23
 discrete optimization problem, 42
 distance
 Euclidean, *see* Euclidean distance
 maximum, *see* maximum distance
 minimum, *see* minimum distance
 rectilinear, *see* rectilinear distance
 distributivity, 201, 267, 269
 distributor node, 241
 divide-and-conquer algorithm, 156
 dogleg, 140, 150
 domino logic, 183
 don't care, 196
 dopant, 282
 drain, 181, 278
 driven net, 186
 dual network, 282
 dynamic
 partitioning, 184
 programming, 62, 148
 search tree, 62
 switch-level simulation, 191
 edge, 22
 parallel, *see* parallel edge
 edge-weighted graph, 24, 34, 220
 EDIF, 19, 117
 ellipsoid algorithm, 67
 Elmore delay, 152
 endpoint, 22
 energy, 71
 enhancement nMOS, 183

equivalent terminal, 134
 essential prime implicant, 224
 estimation
 wire-length, *see* wire-length estimation
 Euclidean
 distance, 43, 105
 squared, *see* squared Euclidean
 distance
 traveling salesman, 43, 76
 evaluation version, 43
 event, 176
 queue (or event list), 176
 event-driven simulation, 173, 176, 180
 EXCLUSIVE-OR, 205
 exhaustive search, 53, 56
 explicit constraint, 56
 exponential order, 29
 fanout, 172
 fault simulation, 14
 feasible solution, 42, 56, 69, 71, 75, 78
 feedthrough
 cell, 151
 wire, 121, 123, 152
 field-programmable gate array, 9, 17
 FIFO queue, 32, 73
 firing, 240
 first improvement, 70
 first-in-first-out order, 62
 fixed terminal, 134, 138, 154
 flexible cell, 123, 125, 153
 floating terminal, 134, 138, 154
 floorplan
 of order 2, 122
 of order 5, 122, 129
 sizing, *see* sizing
 tree, 122, 130
 floorplan-based design methodology, 119, 156
 floorplanning, 16, 120
 force computation, 256
 force-directed
 placement, 111
 scheduling, 256
 formal
 description, 12
 verification, 12, 18
 forward edge, 93
 framework, 19
 freedom, 97, 272

- full-custom design, 8, 15, 16, 125
- functional unit, 236, 262
- garbage collection, 214, 216
- gate, 181
 - array, 9
 - complex, *see* CMOS complex gate
 - logic, *see* logic gate
 - modeling, 170
 - NAND, *see* NAND gate
 - NOR, *see* NOR gate
 - of transistor, 278, 282
- gate-level simulation, 168, 169, 173, 181
- general-cell layout, 107
- general-purpose
 - integrated circuit, 3
 - optimization method, 53, 220
- genetic algorithms, 75, 78, 129
- genotype, 75
- geometric layout, 85
- geometry, 85
- glitch, 174
- global routing, 133, 150
- graceful degradation, 213
- graph, 21
 - algorithms, 29
 - coloring, 145, 263, 264
 - data structure, 24
 - model, 103
 - theory, 21
- greedy algorithm, 143, 230
- gridded routing, 134
- gridless routing, 134, 139
- half-perimeter metric, 105
- halting problem, 45
- Hamiltonian cycle, 47
- Hanan point, 159
- hardware
 - description language, 11, 168, 169, 239
 - model, 236
- hardware-software
 - co-design, 13
 - co-simulation, 13, 169
- hash table, 206, 208
- heuristics, 50, 53, 78, 146, 225
- hierarchical routing, 156
- hierarchy, 4, 98, 104, 121, 122, 124, 246
- high-level
 - synthesis, 12, 102, 235
- transformation, 267
- horizontal
 - composition, 127
 - constraint, 143, 148
- hybrid algorithm, 78
- hyperedge, 104
- hypergraph, 104
- if-then-else operator, 210
- implementation, 217
- implicant, 198
 - prime, *see* prime implicant
- implicit constraint, 56
- in-degree, 23
- incident
 - from, 23
 - to, 23
 - with, 22
- independent set, 148
- inertial delay, 172
- initial placement, 108
- input
 - cell, 103
 - net, 181, 184
 - node, 239
 - size, 26, 65
 - terminal, *see* terminal
- inset cell, 126
- instance, 41, 42, 102
- integer linear programming, 65, 67
- inter-iteration parallelism, 249
- interacting layer, *see* layer interaction
- interactive tool, 8
- interval graph, 144, 148, 264
- intra-iteration parallelism, 248
- intractable problem, 41, 53
- inverter, *see* CMOS inverter
- irredundant prime cover, 198, 223
- iteration period, 245, 251, 270
- iterative
 - data flow, 243
 - improvement, 108, 109, 138
 - placement, 108
- jog insertion, 97
- Kernighan-Lin partitioning algorithm, 73, 112
- kill, 59
- lambda (λ), 84

- last-in-first-out order, 62
- layer, *see* wiring layer
 - interaction, 84
- layout, 15, 119
 - compaction, *see* compaction
 - domain, *see* physical domain
 - editor, 16
 - language, 85
- leaf
 - cell, 121
 - node, 58
 - vertex, 203
- least-cost order, 62
- Lec's algorithm, 134, 156, 163
- left-edge algorithm, 143, 264
- length
 - of path, *see* path length
 - of transistor, *see* transistor length
 - of tree, *see* tree length
- level, 181
- leveling, 173
- library, 9, 14, 15, 102, 106, 119, 238, 252
- life time, 262
- LIFO queue, 32
- linear
 - order, 29
 - programming, 65, 220
- list
 - adjacency, *see* adjacency list
 - scheduling, 259
- literal, 196, 222
- local
 - density, 143
 - minimum, 71
 - routing, 133, 150
 - search, 69, 78, 111, 117, 163
 - transformation, 70, 163
- logic
 - combinational, *see* combinational logic
 - gate, 279
 - multiple-valued, *see* multiple-valued logic
 - signal, 106
 - synthesis, 14, 102, 195
 - multilevel, *see* multilevel etc.
 - two-level, *see* two-level etc.
 - verification, 195, 217
- logic-level simulation, *see* gate-level simulation
- logistic signal, 106
- longest path, 49, 96, 141, 253, 255
 - in a DAG, 89, 91, 93, 96, 129, 173
- loop, *see* cycle
 - folding, 249
- loose routing, *see* global routing
- low power, 4, 10, 277
- macro, 174
 - cell, 106, 125
- macro-level simulation, 168
- Manhattan distance, *see* rectilinear distance
- mask
 - layout, *see* geometric layout
 - programmable, 9
- mask-to-symbolic extraction, 86
- master, 102, 105
- matrix
 - adjacency, *see* adjacency matrix
 - covering, *see* covering matrix
- maximum distance, 90, 93
- maze routing, 134, 137, 149, 150, 163
- memory, 3
 - element, 6, 236
 - multiport, *see* multiport memory
 - random-access, *see* RAM
 - read-only, *see* ROM
- metal layer, 283
- methodology management, 19
- metric
 - wire-length, *see* wire-length metric
- microcell, 16
- microprocessor, 3
- min-cut
 - partitioning, 108, 125
 - placement, 112, 156
- minimum
 - distance, 84, 93
 - feature size, 17, 84
 - spanning tree, 37, 60
 - rectilinear, 49, 105
- Steiner tree
 - rectilinear, 49, 105, 137, 151
- minimum-cost satisfiability, 219
- minterm, 196, 223
- mixed integer linear programming, 69
- mixed-level simulation, 169
- mixed-mode simulation, 169
- mobility, 97
- mobility-based scheduling, 253
- module, 9

- generation, 9, 16
selection, 248
MOS, 282
transistor, 181, 278, 282
move, 70, 71, 78, 107
uphill, *see* uphill moves
multicycle operation, 238
multigraph, 22, 185
multilevel logic synthesis, 195, 200, 222
multiple layers, 137
multiple-valued logic, 170, 174
multiplexer, 237, 262
multiplier, 14, 236
multiport memory, 262
multiprocessor scheduling, 80
mutation, 77
- n-channel transistor, 183, 278
NAND gate, 6, 102, 172, 195, 279
negation, 196
negative cycle, 92
neighbor, 69
neighborhood, 69, 73
net, 54, 102, 121, 173, 181
netlist, 54, 125, 133, 151
neural network, 78
nMOS technology, 183, 222
nMOST, 183, 278
node, 22, 181, 239
nondeterministic polynomial time
complexity, 45
nonoverlapped scheduling, 264, 270
nonplanar graph, 23
nonreserved-layer model, 139
NOR gate, 195, 279
NP, 45, 46, 48
NP-complete, 45
problem, 41, 45, 46
proof, see proving NP-completeness
NP-hard, 49
NPC, 45, 46, 48
- O
big (\mathcal{O}), see big O
object-oriented programming, 105, 206
obstacle, 134, 135
off-set, 196
Omega
big (Ω), see big Omega
on-set, 196, 218, 223, 225
- one-dimensional compaction, 86, 97
operation chaining, *see* chaining
optimization
combinatorial, *see* combinatorial etc.
continuous, *see* continuous etc.
discrete, *see* discrete etc.
problem, 42
tool, 8
version, 43
- order, 28, 29
crossover, 76
of floorplan, *see* floorplan, etc.
of variables, *see* variable ordering
ordered binary-decision diagram, 203
out-degree, 23
output
cell, 103
node, 239
terminal, *see* terminal
overconstrained layout, 93
overflow list, 178
overlapped scheduling, 249, 264, 270
- P, 45, 48
p-channel transistor, 278
parallel
composition, 281
edge, 22
parallelism, 248
parasitic capacitance, 15, 84, 119, 125, 170,
172, 283, 285
parent, 58, 75, 121
parity function, 198
partial
schedule, 255
solution, 56
partitioning, 16, 73, 101, 108, 112
dynamic, see dynamic partitioning
static, see static partitioning
pass, 113
path, 23, 24, 56
connection algorithm, 134
directed, *see* directed path
length, 23, 34
longest, *see* longest path
shortest, *see* shortest path
simple, *see* simple path
performance-driven layout, *see*
timing-driven layout
permutable terminals, 134

- phenotype, 75
- physical domain, 6, 98, 119
- pipelining, 238
- placement, 15, 16, 54, 73, 78, 101, 102, 119, 125, 133, 150
 - building-block*, *see building-block placement*
 - standard-cell*, *see standard-cell placement*
- planar graph, 23, 49
- pMOS, 278
- point
 - Steiner, *see Steiner point*
- polar graph, 122, 129
- polynomial
 - order, 29
 - reduction, 46
 - time complexity, 45
- polysilicon, 282
- population, 75, 78
- port, 102, 121, 246, 262
- positive cycle, 93
- power, 4, 119, 251, 270
 - low, *see low power*
 - set, 69
- precedence relation, 249
- precharging, 183
- predecessor force, 259
- Prim's algorithm, 37, 49, 157
- prime implicant, 198, 222, 225
- principle of optimality, 62
- printed circuit board, 138
- priority queue, 178
- problem, 41
 - decision, *see decision problem*
 - instance, *see instance*
 - intractable, *see intractable problem*
 - NP-complete, *see NP-complete problem*
 - optimization, *see optimization problem*
 - size, *see input size*
 - tractable, *see tractable problem*
 - undecidable, *see undecidable problem*
- product of sums, 201
- programmable logic array, 14, 222
- programming, 53
- propagation delay, 125, 171
- prototyping, 17
- proving NP-completeness, 46
- pruning, 59, 225
- pseudo-cell, 103
- pseudo-code, 287
- array, 288
- assignment statement, 290
- block, 290
- compound statement, 290
- conditional statement, 290
- data type
 - Boolean, 288
 - primitive (int/float/char), 288
 - set, 293
 - unspecified, 292
- do statement, 291
- double quotes, 292
- expression, 289
- for each statement, 293
- for statement, 291
- function
 - call, 291
 - definition, 291
 - unspecified, 292
- if statement, 290
- iteration statements, 290
- main, 292
- pointer, 288, 289
- return statement, 292
- string, 288
- structure (*struct*), 288
- typography, 287
 - mathematical, 293
- while statement, 290
- quadratic order, 29
- queue, 62, 240
 - FIFO, *see FIFO queue*
 - LIFO, *see LIFO queue*
- Quine-McCluskey algorithm, 223
- radix sort, 165
- RAM, 237
- random
 - logic, 14
 - number, 72, 75
 - perturbation, 72, 111
- rapid system prototyping, 17
- ready list, 260
- real time, 17
- reasonable encoding, 27
- rectilinear
 - distance, 49, 105, 156
 - polygon, 83
 - segment, 49

- spanning tree, 49
 - minimum, *see* minimum spanning tree
- Steiner tree, 49, 157
 - minimum, *see* minimum Steiner tree
- reduced ordered binary-decision diagram, 201, 203, 223, 225
- reduction, *see also* polynomial reduction, 65
- redundant
 - floorplan, 127
 - solution, 221
- register, 173, 236
 - assignment, 250
 - file, 236
- register-transfer level, 6, 120
 - simulation, 168
- requirement function, 252
- reserved-layer model, 134, 139
- resistance, 86, 152, 168, 181, 191, 284
- resistor, 15, 168
- resource
 - allocation, *see* allocation
 - requirement distribution function, 256, 271
 - type, 252
- resource-constrained synthesis, 252, 259
- restriction, 201, 208, 210, 213, 228
- retiming, 270
- rigid
 - cell, 126
 - rectangle, 86, 97
- rip up and reroute, 149, 150, 163
- rise/fall delay model, 172
- robust channel router, 146
- ROM, 237
- root, 58, 203
- routing, 15, 49, 55, 101, 119, 133
 - channel, *see* channel
 - track, *see* track
- row dominance, 224
- RS-latch, 102
- satisfiability, 42, 48, 78, 219
 - minimum cost, *see* minimum-cost satisfiability
- schedule
 - cooling, *see* cooling schedule
 - partial, *see* partial schedule
- scheduling, 247, 261, 267
 - overlapped, *see* overlapped scheduling
- range, 254
- schematic
 - editor, 13
 - entry, 169
- sea of gates, 9, 154
- search
 - breadth-first, *see* breadth-first search
 - depth-first, *see* depth-first search
 - space, 42, 56
 - tree, 58, 59, 62
- selector node, 241
- self-force, 259
- selfloop, 22
- semicustom design, 8, 15
 - synthesis, 195
- series composition, 281
- series-parallel network, 281
- set, 219
 - covering, 220, 223
- Shannon expansion, 202, 213
- shape function, 125
- shortest path, 34, 42, 49, 63, 96
 - Dijkstra's algorithm, *see* Dijkstra's algorithm
 - in a DAG, 96, 220
- signal, 167, 181
 - level, 170
 - modeling, 170, 181
 - strength, 170
 - transition, 170
- signal-flow graph, 239
- Silage, 239
- silicon
 - compiler, 12
 - oxide, 282, 283
 - polycrystalline, *see* polysilicon
 - substrate, 282
- simple
 - cycle, 23, 43, 47
 - graph, 22
 - path, 23
- simplex algorithm, 67
- simulated
 - annealing, 71, 78, 99, 111, 163, 271
 - evolution, 78
- simulation, 12–14, 17, 102, 167
 - fault, *see* fault simulation
- simulator
 - kernel, 169, 173

- module, 169
- sink terminal, 152
- sizing, 125, 129
- slack variable, 67
- slicing
 - floorplan, 121, 153
 - tree, 111, 121
- software pipelining, 249
- solution
 - checking, 44–46
 - feasible, *see* feasible solution
 - space, *see* search space
- source, 181, 278
 - terminal, 152
 - vertex, 89, 97
- space complexity, 27
- spanning tree, 37
 - minimum, *see* minimum spanning tree
 - rectilinear, *see* rectilinear spanning tree
- specification, 217
- speed, 4, 251
- spiral floorplan, *see* wheel floorplan
- squared Euclidean distance, 106
- standard
 - cell, 9, 12, 14, 15, 102, 106, 125, 222
 - layout, 138, 151
 - placement, 106, 111
 - form, 66, 220
- state, 14, 173, 235
- static
 - CMOS, 182
 - partitioning, 184
 - search tree, 62
 - switch-level simulation, 190
- statistical cooling, *see* simulated annealing
- steepest descent, 70
- Steiner
 - point, 49, 154
 - tree
 - graph version, 154
 - rectilinear, *see* rectilinear Steiner tree
- step
 - computational, *see* computational step
- stimuli, 169, 174, 179
- storage
 - net, 181
 - value, 248
- strength, *see also* signal strength, 181
- stretchable rectangle, 86, 97
- strong canonicity, 217
- strongly connected
 - component, 24
 - vertices, 24
- structural
 - description, 101, 102
 - domain, 6, 119, 124
- subgraph, 22, 69
- sublinear order, 29
- subroutine, 171
- substrate, *see* silicon substrate
- subtree, 58
- successor force, 259
- sum of minterms, 197, 202
- supervertex, 265
- switch-level
 - simulation, 15, 168, 180
 - dynamic, *see* dynamic etc.
 - static, *see* static etc.
 - timing simulation, 191
- switchbox routing, 138, 154
- symbolic
 - computation, 207, 229
 - layout, 17, 85
 - editor, 85
- synchronous
 - circuit, 168, 173, 236
 - data flow, 239
- synthesis, 12
 - high-level, *see* high-level synthesis
 - logic, *see* logic synthesis
 - tool, 8
- system-level simulation, 168
- tabu
 - list, 73
 - search, 73, 111
- task, 262
- tautology, 218
- technology, 9, 133
 - CMOS, *see* CMOS technology
 - file, 85
 - mapping, 14
 - temperature, 71
 - terminal, 102, 103, 121, 133, 151
 - test pattern, 14
 - testability, 4
- Theta
 - big (Θ), *see* big Theta
- three-sided channel, 154
- three-state driver, 237, 262

- time
complexity, 27
average case, *see* average-case *etc.*
nondeterministic polynomial, *see*
nondeterministic *etc.*
polynomial, *see* polynomial *etc.*
worst case, *see* worst-case *etc.*
- frame, 254
wheel, 178
- time-constrained synthesis, 252, 259
- timing, 119
analysis, 14
- timing-driven layout, 16, 152
- timing-level simulation, 15, 168
- token, 240
firing, *see* firing
- tool integration, 19
- top-down design, 7, 16, 119
- topological layout, *see* symbolic layout
- topology, 17, 85
tour, 47
- track, 55
- tractable problem, 41
- transfer, 262
- transformation
algorithmic, *see* algorithmic
transformation
high-level, *see* high-level transformation
- transistor, 15, 86, 125, 168
length, 284
MOS, *see* MOS transistor
width, 284
- transition, *see* signal transition
- transmission line, 152
- traveling salesman, 43, 76
Euclidean, *see* Euclidean traveling
salesman
in graphs, 43, 46, 57, 60, 63, 67
- tree, 37, 49, 58, 157
decomposition, *see* decomposition tree
length, 37, 49
search, *see* search tree
spanning, *see* spanning tree
Steiner, *see* Steiner tree
- tri-state driver, *see* three-state driver
- tripartite graph, 103
- truth table, 6, 171, 197, 201, 202, 235
- Turing machine, 48
- two-dimensional compaction, 87, 97
- two-level logic synthesis, 195, 222
- two-phase clock, 238
- unate covering, 222
- undecidable problem, 45
- undirected graph, 23
- unidirectional signal flow, 152, 168, 180, 183, 279
- uninitialized (signal value), 170
- unique table, 206
- unit-delay model, 172, 174
- unit-size placement, 54, 58, 61, 65, 69, 70, 74, 75, 101, 107, 111
- unknown (signal value), 170, 181
- uphill move, 71, 73
- user interface, 19
- value grouping, 262
- variable ordering, 203, 210, 215, 230
- V_{dd} , 279
- verification, 17, 167, 197
logic, *see* logic verification
tool, 8
- Verilog, 11, 239
- version management, 18
- vertex, 22
connected, *see* connected vertices
- vertex-weighted graph, 24
- vertical
composition, 127
constraint, 140, 143, 146
graph, 139, 140
- VHDL, 11, 168, 170, 199, 239
- via, 133, 137, 138
- virtual-grid compaction, 99
- VLSI, 3
cost function, 4
- voltage, 167
source, 168
- V_{ss} , 279
- weakly connected
component, 24
vertices, 24
- weighted graph, 24
- well, 283
- wheel
floorplan, 121
time, *see* time wheel
- width of transistor, *see* transistor width
- wire, 86, 237
feedthrough, *see* feedthrough wire

- length, 138
 - estimation, 105, 119
 - metric, 105
 - wiring, *see* routing
 - layer, 133
 - word length, 9, 236, 269
 - worst-case time complexity, 28
- Y-chart, 6, 11, 119, 235
 - yield, 4
- zero-delay model, 172, 173
 - zero-one integer linear programming, 67, 220

Algorithms for VLSI Design Automation

Sabih H. Gerez

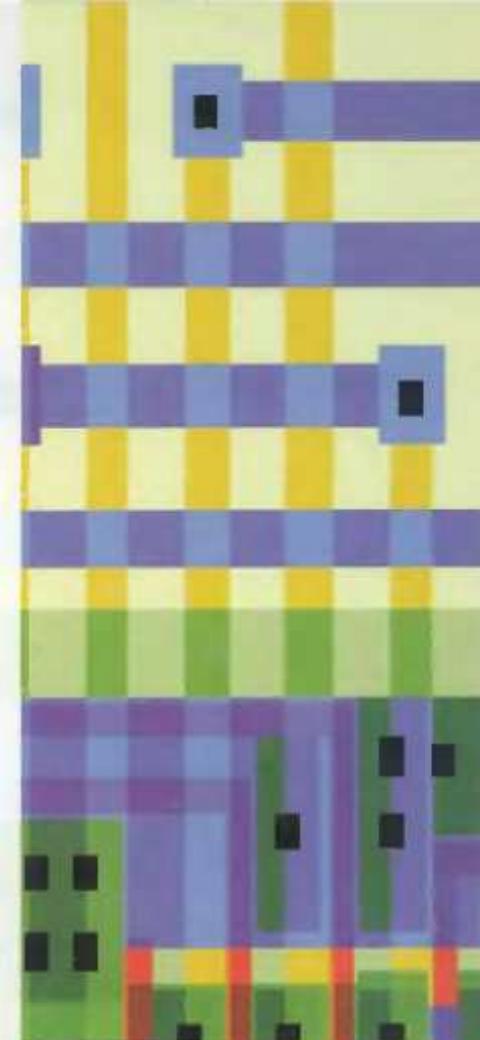
Very large scale integrated (VLSI) circuits nowadays contain many millions of components, and cannot be designed without the aid of design automation tools. This book provides an insight into the algorithms used inside these computer-aided design (CAD) tools, and will be a good starting point for designers who want to specialize in building CAD tools themselves.

Highlights of the book include:

- Special attention to background knowledge from mathematics and computer science: graph theory, complexity of algorithms, and general-purpose methods for combinatorial optimization
- About 50 algorithms (from graph theory, layout design, simulation, logic synthesis and high-level synthesis) presented in depth by means of pseudo-code and step-by-step examples

It will be an ideal text for students in Computer Science or Electronic Engineering taking VLSI design automation courses, and for chip designers or programmers in industry developing CAD tools.

The author, Sabih Gerez, has based the book on a course given to his students at the University of Twente, Enschede, in the Netherlands. As an assistant professor at the Department of Electrical Engineering, he teaches courses on circuit theory and VLSI design. His research focuses on VLSI design automation, especially high-level synthesis. Dr Gerez holds an M.Sc. degree (with honors) in Electrical Engineering and a Ph.D. degree in Applied Sciences, both from the University of Twente.



www.wiley.com/college/wave
Web Added Value Education

Check out the author's Web site for additional resources
for students and instructors:
<http://utelint.el.utwente.nl/links/gerez/cadvlsi/book.html>

ISBN 0-471-98489-2



9 780471 984894

JOHN WILEY & SONS

Chichester·New York·Weinheim·Brisbane·Singapore·Toronto

www.wiley.com