Algorithms for VLSI Design Automation

Features:

- Full Color Book plus DVD
- Downloadable Workbook included in the DVD
- Over 100 hours Interactive E-lectures, Quiz and Videos in DVD
- DVD has many useful features for teachers to teach with digital resources in classroom

3G E-LEARNING

With DVD

ALGORITHMS FOR VLSI DESIGN AUTOMATION

3G E-LEARNING

ALGORITHMS FOR VLSI DESIGN AUTOMATION

Authored and Edited by 3G E-learning LLC, USA

3G E-LEARNING

3G E-learning LLC
www.3ge-learning.com
email: info@3ge-learning.com

ISBN: 978-1-68095-509-5

For more information visit **www.3ge-learning.com**

EDITORIAL BOARD

Aleksandar Mratinković is born on May 5, 1988. in Arandjelovac, Serbia. He has graduated on Economic high school (2007), The College of Tourism in Belgrade (2013), and also has a master degree of Psychology (Faculty of Philosophy, University of Novi Sad). He has been engaged in different fields of psychology (Developmental Psychology, Clinical Psychology, Educational Psychology and Industrial Psychology) and has published several scientific works

Dan Piestun (PhD) is currently a startup entrepreneur in Israel working on the interface of Agriculture and Biomedical Sciences and was formerly president-CEO of the National Institute of Agricultural Research (INIA) in Uruguay. Dan is a widely published scientist who has received many honours during his career including being a two-time recipient of the Amit Golda Meir Prize from the Hebrew University of Jerusalem, his areas of expertise includes stem cell molecular biology, plant and animal genetics and bioinformatics. Dan's passion for applied science and technological solutions did not stop him from pursuing a deep connection to the farmer, his family and nature. Among some of his interest and practices counts enjoying working as a beekeeper and onboard fishing.

Hazem Shawky Fouda has a PhD. In Agriculture Sciences, obtained his PhD. From the Faculty of Agriculture, Alexandria University in 2008, He is working in Cotton Arbitration & Testing General Organization (CATGO).

Felecia Killings is the Founder and CEO of LiyahAmore Publishing, a publishing company committed to providing technical and educational services and products to Christian Authors. She operates as the Senior Editor and Writer, the Senior Writing Coach, the Content Marketing Specialist, Editor-in-Chief to the company's quarterly magazine, the Executive and Host of an international virtual network, and the Executive Director of the company's online school for Authors. She is a former high-school English instructor and professional development professor. She possesses a Master of Arts degree in Education and a Bachelor's degree in English and African American studies.

Dr. Sandra El Hajj, Ph.D. in Health Sciences from Nova Southeastern University, Florida, USA is a health professional specialized in Preventive and Global Health. With her 12 years of education obtained from one of the most prominent universities in Beirut, in addition to two leading universities in the State of Florida (USA), Dr. Sandra made sure to incorporate interdisciplinary and multicultural approaches in her work. Her long years of studies helped her create her own miniature world of knowledge linking together the healthcare field with Medical Research, Statistics, Food Technology, Environmental & Occupational Health, Preventive Health and most noteworthy her precious last degree of Global Health. Till today, she is the first and only doctor specialized in Global Health in the Middle East area.

Fozia Parveen has a Dphil in Sustainable Water Engineering from the University of Oxford. Prior to this she has received MS in Environmental Sciences from National University of Science and Technology (NUST), Islamabad Pakistan and BS in Environmental Sciences from Fatima Jinnah Women University (FJWU), Rawalpindi.

Igor Krunic 2003-2007 in the School of Economics. After graduating in 2007, he went on to study at The College of Tourism, at the University of Belgrade where he got his bachelor degree in 2010. He was active as a third-year student representative in the student parliament.Then he went on the Faculty of science, at the University of Novi Sad where he successfully defended his master's thesis in 2013. The crown of his study was the work titled Opportunities for development of cultural tourism in Cacak". Later on, he became part of a multinational company where he got promoted to a deputy director of logistic. Nowadays he is a consultant and writer of academic subjects in the field of tourism.

Dr. Jovan Pehcevski obtained his PhD in Computer Science from RMIT University in Melbourne, Australia in 2007. His research interests include big data, business intelligence and predictive analytics, data and information science, information retrieval, XML, web services and service-oriented architectures, and relational and NoSQL database systems. He has published over 30 journal and conference papers and he also serves as a journal and conference reviewer. He is currently working as a Dean and Associate Professor at European University in Skopje, Macedonia.

Dr. Tanjina Nur finished her PhD in Civil and Environmental Engineering in 2014 from University of Technology Sydney (UTS). Now she is working as Post-Doctoral Researcher in the Centre for Technology in Water and Wastewater (CTWW) and published about eight International journal papers with 80 citations. Her research interest is wastewater treatment technology using adsorption process.

Stephen obtained his PhD from the University of North Carolina at Charlotte in 2013 where his graduate research focused on cancer immunology and the tumor microenvironment. He received postdoctoral training in regenerative and translational medicine, specifically gastrointestinal tissue engineering, at the Wake Forest Institute of Regenerative Medicine. Currently, Stephen is an instructor for anatomy and physiology and biology at Forsyth Technical Community College.

Michelle holds a Masters of Business Administration from the University of Phoenix, with a concentration in Human Resources Management. She is a professional author and has had numerous articles published in the Henry County Times and has written and revised several employee handbooks for various YMCA organizations throughout the United States.

TABLE OF CONTENTS

PREFACE

Gone are the days when huge computers made of vacuum tubes sat humming in entire dedicated rooms and could do about 360 multiplications of 10 digit numbers in a second. Though they were heralded as the fastest computing machines of that time, they surely don't stand a chance when compared to the modern day machines. Modern day computers are getting smaller, faster, and cheaper and more power efficient every progressing second. Very-large-scale integration (VLSI) is the process of creating an integrated circuit (IC) by combining thousands of transistors into a single chip. VLSI began in the 1970s when complex semiconductor and communication technologies were being developed. Electronic design automation (EDA), also referred to as electronic computer-aided design (ECAD), is a category of software tools for designing electronic systems such as integrated circuits and printed circuit boards. The tools work together in a design flow that chip designers use to design and analyze entire semiconductor chips. Since a modern semiconductor chip can have billions of components, EDA tools are essential for their design.

Content Coverage

Chapter One begins with VLSI system design process and tools. Furthermore, it focuses on data structures and algorithms. *Chapters Two* and *Three* present about fabrication of VLSI circuits and VLSI simulation, separately. *Chapter Four* is floor planning and pin assignment of identifying structures that should be placed close together, and allocating space for them in such a manner as to meet the sometimes conflicting goals of available space (cost of the chip), required performance, and the desire to have everything close to everything else. *Chapter Five* explores the overview of global routing as global routing in VLSI design is one of the most challenging discrete optimization problems in computational theory and practice. *Chapter Six* covers system-level synthesis and high-level synthesis form the front-end of a synthesis approach to digital system design and are together called system synthesis.

The following are the salient features of the book:

- This Text covers all stages of design from layout synthesis through logic synthesis to high-level synthesis.

- Topics are logically arranged according to the course outline of the subject.
- The use of simple language to facilitate a better understanding of each chapter.
- Illustrative examples of varying difficulties, wherever needed, are presented to enhance the interest of students.
- The provision of knowledge check test for each chapter is given at the end of chapter to strengthen the learning process of students.
- Case Study, Role Model, Keywords, and Examples are also provided to enrich the knowledge of students.

CHAPTER

1

INTRODUCTION TO VLSI DESIGN METHODOLOGIES

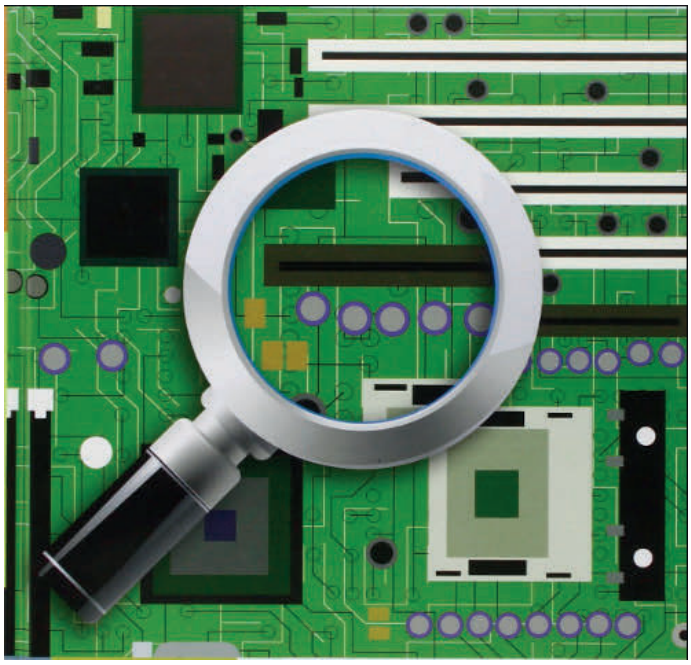
LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Discuss about VLSI system design process
2. Explain the VLSI design automation tools
3. Describe the tractable and intractable problems
4. Focus on data structures and algorithms

INTRODUCTION

As part of the larger design efforts, we also investigate concurrency theory and design methodologies for asynchronous systems. You have developed a significant tool suite for designing asynchronous chips, and the theory that supports this tool suite and a subset of some of the tools developed. Most of the theory we have developed is a direct result of problems encountered when using existing synthesis methods for asynchronous design. Therefore, our focus is on new theory and tools that enable large-scale design.



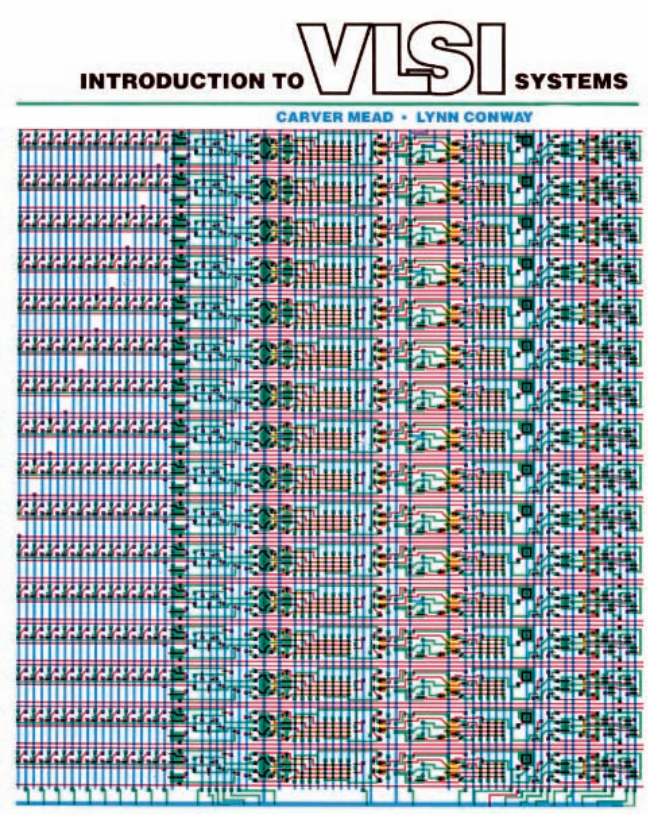
This text deals with the algorithms that are used inside VLSI design automation tools, also called computer-aided design (CAD) tools. It does not make sense to discuss internals of VLSI design automation tools without having a clear idea of how these tools are used. For this reason, a brief review of VLSI design methodologies is given. The term "design methodology" refers to the approach followed to solve the VLSI design problem. A review of VLSI design automation with concrete references to the sub-problems to be solved and the tools used. This knowledge for those that is unfamiliar with the topic. 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.

1.1 THE VLSI SYSTEM DESIGN PROCESS

A fundamental assumption about VLSI circuits is that they are designed by humans and built by machines. Thus all CAD systems act as translators between the two. On one end of a CAD system is the human interface that must be intelligent enough to communicate in a manner that is intuitive to the designer. On the other end is a generator of specifications that can be used to manufacture a circuit. In between are the many programming and design tools that are necessary in the production of a VLSI system.

The front end of a CAD system is the human interface and there are two basic ways that it can operate: graphically or textually. Graphic design allows the display

of a circuit to be manipulated interactively, usually with a pointing device. Textual design allows a textual description, written in a hardware-description language, to be manipulated with a keyboard and a text editor. *For example*, suppose a designer wants to specify the layout of a transistor that is coupled to a terminal. This can be done graphically by first pointing on the display to the desired location for the transistor and issuing a "create" command. A similar operation will create the terminal. Finally, the connecting wire can be placed by tracing its intended path on the display.



To do this same operation textually, the following might be typed:

- transistor at (53,100).
- terminal below transistor by 30, left by 3 or more.
- wire from transistor left then down to terminal.

Notice that the textual description need not be completely specific. This is one of the advantages of textual descriptions: the ability to underspecify and let the computer fill in the detail. In this example, other parts of the circuit and other spacing rules will help to determine the exact location of these components. Additional advantages of

text are the ease of verbal documentation, ease of parameterization, ease of moving the CAD system between computers, and a somewhat lower cost of a design workstation because of the reduced need for graphics display.

The disadvantage of text, however, is immediately clear: It is not as good a representation of the final circuit, because it does not visually capture the spatial organization. Text is one-dimensional and graphics is two-dimensional. Also, graphics provides faster and clearer feedback during design, so it is easier to learn and to use, which results in more productivity. Although graphics cannot handle verbal documentation as well, it does provide instant visual documentation, which can be more valuable. Even underspecified spacing can be achieved graphically by creating an abstract design that is subsequently fleshed out.

A number of design styles exist to bridge the gap between text and graphics. These attempts to be less demanding than are precise polygon drawing systems while still capturing the graphical flavor. In sticks design, the circuit is drawn on a display, but the components have no true dimensions and their spacing is similarly inaccurate. Virtual grid design also abstracts the graphics of a circuit, but it uses quasi-real component sizes to give something of the feel for the final layout. Closer to text is the SLIC design style, which uses special characters in a text file to specify layout. *For example*, an "X" indicates a transistor and is used for metal wires, so the adjacency of these two characters indicates a connection.

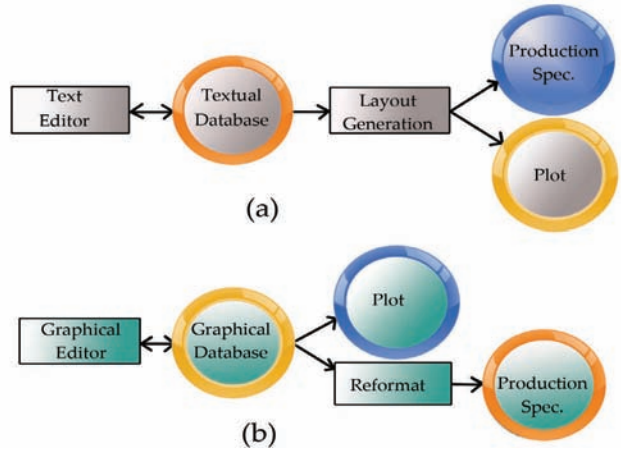


Figure 1.1 Simple CAD system structures: (a) Textual (b) Graphical.

At the back end of a design system is a facility for writing manufacturing specifications. Complex circuits cannot be built by hand, so these specifications are generally used as input to other programs in machines that control the fabrication process. There are many manufacturing devices (photo plotters, wafer etchers, and so on) and each has its own format.

The following Figure (1.1) shows the structure of a simple CAD system with its front end and back end in place

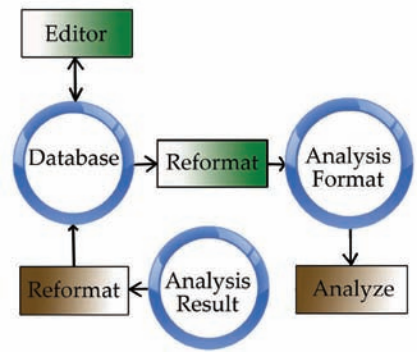


Figure 1.2: Steps typically required analyzing circuitry.

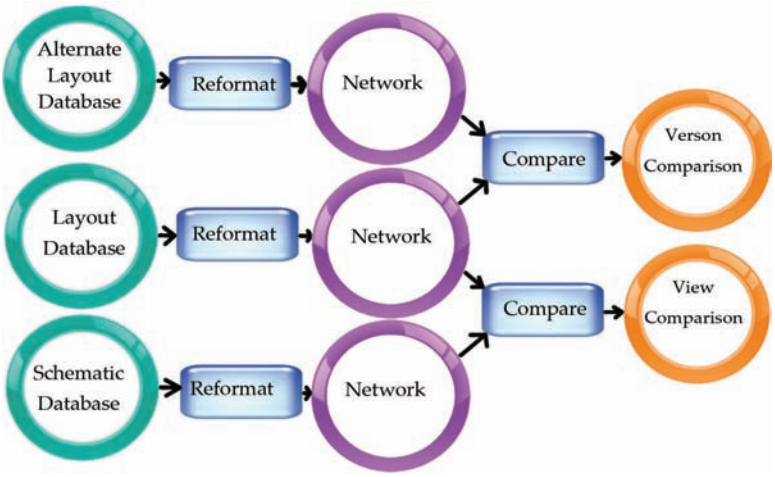


Figure 1.3: Comparing different versions of a circuit.

Between the front-end user interface and the back-end manufacturing specification are the analysis and synthesis tools that help reduce the tedium of creating correct layout. Analysis tools are able to detect local layout errors such as design-rule violations, and more global design errors such as logical failures, short-**circuits**, and power inadequacies. The following Figure (1.2) illustrates the sequence of analysis steps

Although standardization is constantly proposed there continue to be many output formats that a CAD system must provide.

in a typical design system. Analysis tools can also be used to compare different versions or different views of the same circuit that have been independently designed.

Synthesis tools help perform repetitious layout such as programmable logic array (PLA) generation, complex layout such as routing, and even layout manipulation such as compaction. Figure (1.4) illustrates the sequence of steps for a typical synthesis tool. As circuit design becomes more complex, these tools become more valuable and numerous.

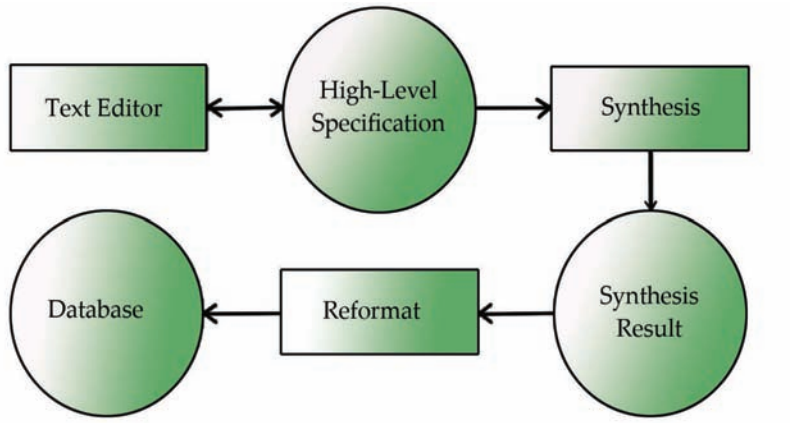


Figure 1.4 Steps typically required synthesizing circuitry.

Today's VLSI designers guide their circuit through the many different phases of the process outlined here. They must correctly control the initial creation of a design, the synthesis of additional detail, the analysis of the entire circuit, and the circuit's preparation for manufacturing. As **synthesis** tools become more reliable and complete, the need for analysis tools will lessen. Ultimately, the entire process will be automated, so that a system can translate directly from behavioral requirements to manufacturing specifications. This is the goal of silicon compilers, which can currently do such translation only in limited contexts by automatically invoking the necessary tools.

KEYWORD

Circuit is composed of individual electronic components, such as resistors, transistors, capacitors, inductors and diodes, connected by conductive wires or traces through which electric current can flow.

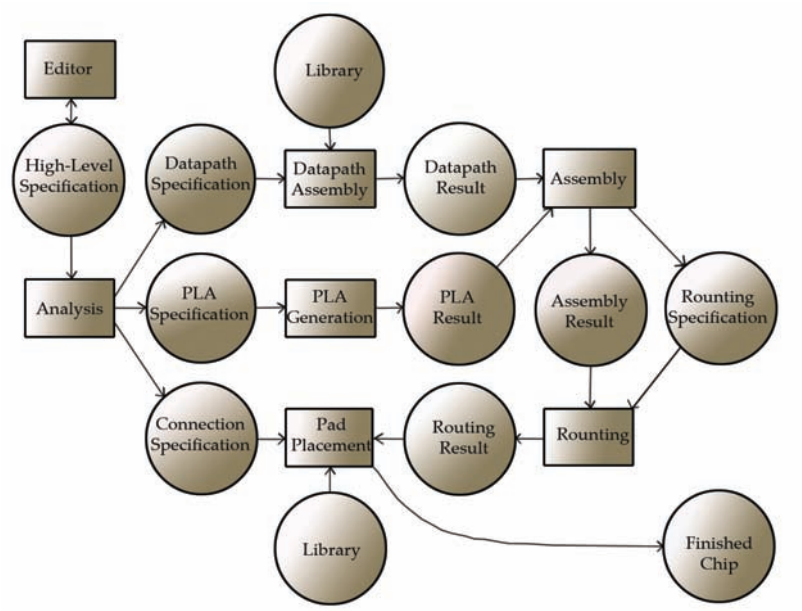


Figure 1.5 Structure of a typical silicon compiler.

The totally automated design has been sought for as long as there have been circuits to fabricate. Although today's systems can easily produce yesterday's circuits, new fabrication possibilities always seem to keep design capability behind production capability. Therefore it is unreasonable to expect complete automation, and a more realistic approach to CAD acknowledges the need for human guidance of the design process.

KEYWORD

Synthesis refers to a combination of two or more entities that together form something new; alternately, it refers to the creating of something by artificial means.

1.2 REVIEW OF DATA STRUCTURES AND ALGORITHMS

VLSI layouts are represented as a collection of several layers of planar geometric elements. These elements are usually limited to Manhattan features (vertical and horizontal edges) and are not allowed to overlap within the same layer. The layouts have historically been manipulated by human layout designers to conform to the design rules and perform the specified functions. These manipulations were time consuming and error prone, even for small layouts. Rapid advances in

fabrication technology in recent years have dramatically increased the size and complexity of VLSI circuits. As a result, a single chip may include several million transistors. These technological advances have made it impossible for layout designers to manipulate the layout without the assistance of computers. Several Computer Aided Design (CAD) tools have been developed to assist the designers with the enormous amount of data required to represent and manipulate a layout. CAD tools require highly specialized algorithms and data structures to effectively manage layout information. A layout is usually stored in a high-level format, which specifies every geometric object in the layout with great precision.

This precision is necessary since this information has to be communicated to output devices such as plotters, video displays, and pattern-generating machines. Most importantly, the layout information must be specific enough so that it can be sent to the foundry for fabrication.

CAD tools fall in two categories. The first types of tools help a human de-signer to manipulate a layout, while the second type of tools are designed to perform some task on the layout automatically. Layout editor is a CAD tool of first type. It allows a human designer to manipulate a layout and may perform some functions such as routing and design rule checking automatically. The bulk of the research of physical design automation has focused on tools of the second type. The major accomplishment has been partitioning of the entire physical design problem into several smaller (and conceptually easier) problems. Unfortunately, even these problems are still computationally very hard.

As a result, the major focus has been on development on design and analysis of heuristic algorithms for partitioning, placement, routing and compaction. Many of these algorithms

are based on graph theory and computational geometry. As a result, it is important to have a basic understanding of these two fields. In addition, several special classes of graphs are used in physical design. It is important to understand properties and algorithms about these classes of graphs to develop effective algorithms in physical design. In the following, you will review basic graph theoretic and computation geometry algorithms which play a significant role in many different VLSI design algorithms.

1.2.1 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 105 transistors (in current-day technologies, circuits of 107 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).

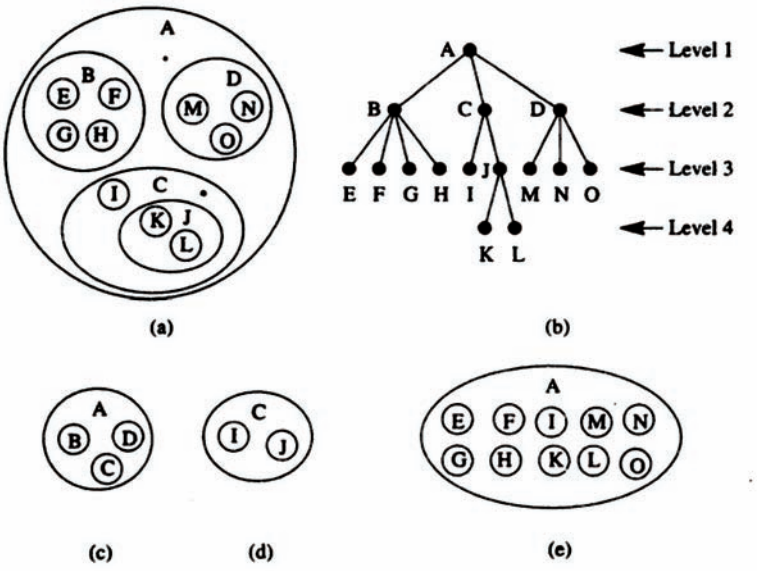
Designing such a circuit is a difficult task. A first requirement is, of course, that a given specification is realized. Besides this, there are different entities that one would like to optimize. These entities can often not be optimized simultaneously (one can only improve one entity at the expense 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 or cease working or will need extra (expensive) cooling. Besides, there is a special category of applications, viz. portable equipment powered by batteries, for which low power consumption is of primary importance. Here again there are trades-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 economic 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.
- *Testability*. As a significant percentage of the chips fabricated are 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 VLS/ 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. The Figure illustrates the concepts of hierarchy and abstraction.



1.2.2 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:

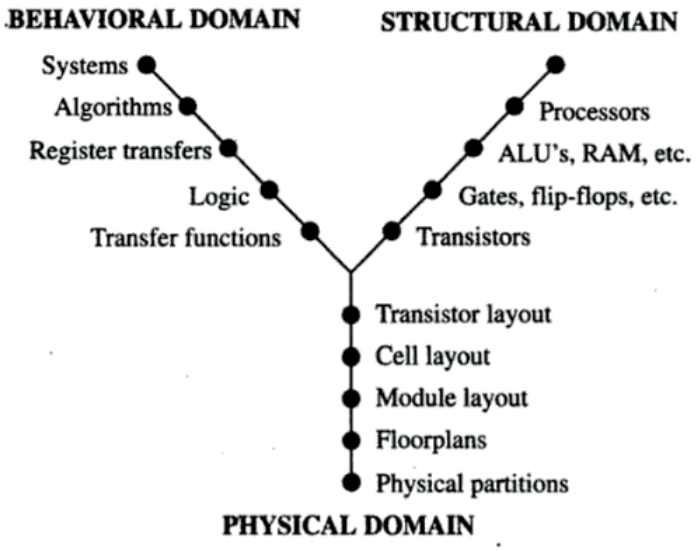
- The behavioral domain. In this domain, a part of the design (or the whole) is seen as a black box; the relations between outputs and inputs are given without a reference to the implementation of these relations. A behavioral description at the transistor level is e.g. an equation giving the channel 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 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 sub-circuits. A description in this domain gives information on the sub-circuits used and the way they are interconnected. Each of the sub-circuits 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.

Transistor is a semiconductor device used to amplify or switch electronic signals and electrical power.

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 tools are not necessarily discussed in the order that they should be invoked during a design.

The three domains and their hierarchies can be visualized on a so-called Y-chart as depicted in Figure. Each axis represents a design domain and the level of abstraction decreases from the outside to the center.



1.3 REVIEW OF VLSI DESIGN AUTOMATION TOOLS

The designing an integrated circuit is a sequence of many actions most of which can be done by computer tools. Only a few of these tools will receive detailed attention later. 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 as is shown in Figure. 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.

1.3.1 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.

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 re-search purposes. Currently, the languages VHDL and Verilog are the most widely used.

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 caned high-level synthesis. Yet another application of formal descriptions is in formal verification.

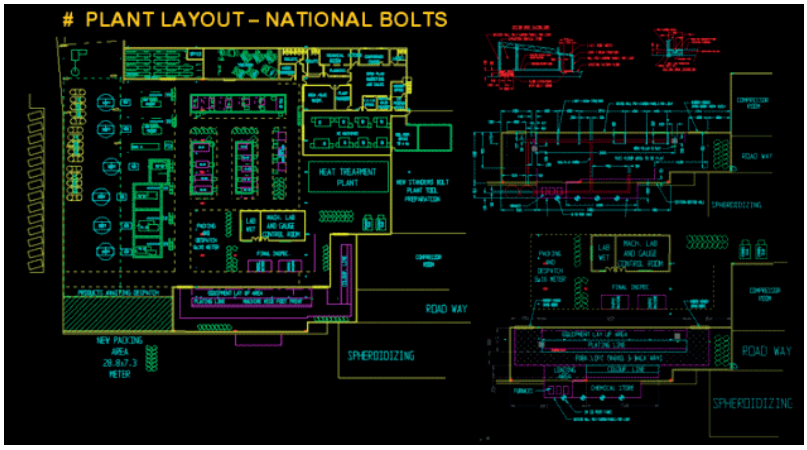
The term silicon compiler came already into existence in the early days of VLSI design when CAD tools were relatively primitive compared to what they are now. The goal was to

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.

KEYWORD

Silicon compiler is a software system that takes a user's specifications and automatically generates an integrated circuit (IC).

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.

1.3.2 Algorithmic Graph Theory and Computational Complexity

These notes cover graph algorithms, pure graph theory, and applications of graph theory to computer systems. The algorithms are presented in a clear algorithmic style, often with considerable attention to data representation, though no extensive background in either data structures or programming is needed. In addition to the classical graph algorithms, many new random and parallel graph algorithms are included. **Algorithm** design methods, such as divide and conquer, and search tree techniques, are emphasized. There is an extensive bibliography, and many exercises.

We very briefly discuss the theory of computability and the relationship between graphs and algorithms. We can only introduce the ideas very sketchily. For a more thorough introduction to complexity and NP-completeness; an excellent treatment of algorithmic graph theory is found. We first consider the question: How long does it take to carry out a computation? An answer in terms of seconds or minutes is of no use: not only does this depend on the computer used, but it does not take into account the difference between apparently similar instances of a problem.

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".

1.3.3 Computational Complexity Theory

Computational complexity theory is a branch of the theory of computation in theoretical computer science that focuses on classifying computational problems according to their inherent difficulty, and relating those classes to each other. A computational problem is understood to be a task that is in principle amenable to being solved by a computer, which is equivalent to stating that the problem may be solved by mechanical application of mathematical steps, such as an algorithm.

Algorithm is a self-contained step-by-step set of operations to be performed.

A problem is regarded as inherently difficult if its solution requires significant resources, whatever the algorithm used. The theory formalizes this intuition, by introducing mathematical models of computation to study these problems and quantifying the amount of resources needed to solve them, such as time and storage. Other complexity measures are also used, such as the amount of communication, the number of gates in a circuit (used in circuit complexity) and the number of processors (used in parallel computing). One of the roles of computational complexity theory is to determine the practical limits on what computers can and cannot do.

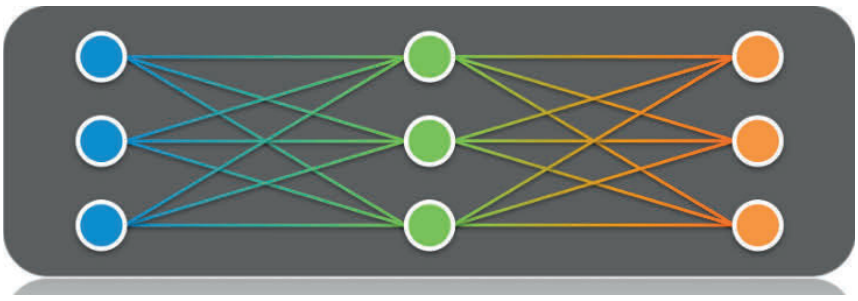
Closely related fields in theoretical computer science are analysis of algorithms and computability theory. A key distinction between analysis of algorithms and computational complexity theory is that the former is devoted to analyzing the amount of resources needed by a particular algorithm to solve a problem, whereas the latter asks a more general question about all possible algorithms that could be used to solve the same problem. More precisely, computational complexity theory tries to classify problems that can or cannot be solved with appropriately restricted resources. In turn, imposing restrictions on the available resources is what distinguishes computational complexity from computability theory: the latter theory asks what kind of problems can, in principle, be solved algorithmically.

1.4 TRACTABLE AND INTRACTABLE PROBLEMS

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 $0(n^k)$ 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". 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. 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. 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).

1.4.1 Combinatorial Optimization in VLSI Design

The ever increasing abundance, role and importance of computers in every aspect of our lives are clearly a proof of a tremendous scientific and cultural development if not revolution. When thinking about the conditions which made this development possible most people will probably first think mainly of technological aspects such as the invention and perfection of transistor technology, the possibility to fabricate smaller and smaller physical structures consisting of only a few atoms by now, and the extremely delicate, expensive yet profitable manufacturing processes delivering to the markets new generations of chips in huge quantities every couple of months. From this point of view the increase of complexity might be credited mainly to the skills of the involved engineering sciences and to the verve of the associated economic interests.

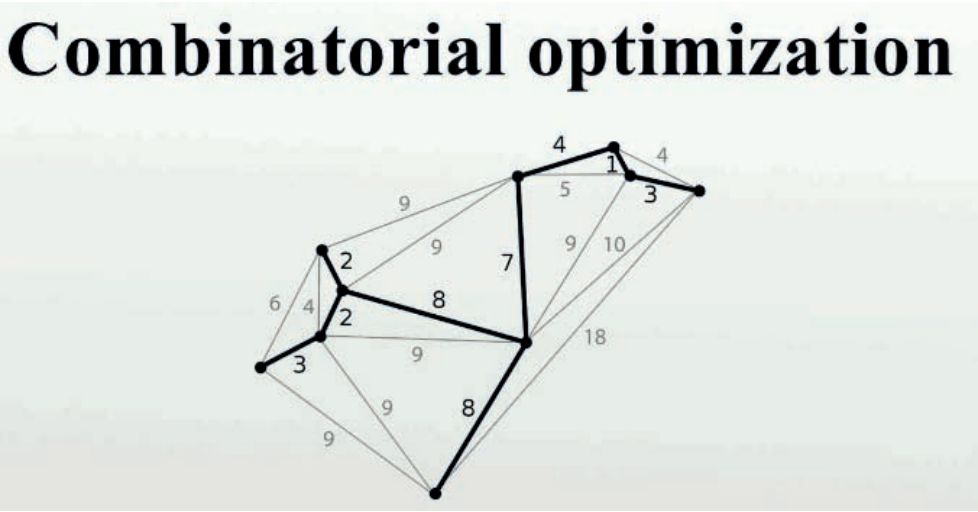


It is hardly conceived how important mathematics and especially mathematical optimization is for all parts of VLSI technology. Clearly, everybody will acknowledge that the physics of semiconductor material relies on mathematics and that, considered from a very abstract level; computer chips are nothing but intricate machines for the calculation of complex Boolean functions. Nevertheless, the role of mathematics is far from being fully described with these comments. Especially the steps of the design of a VLSI chip preceding its actual physical realization involve more and more mathematics. Many of the involved tasks which were done by the hands of experienced engineers until one or two decades ago have become so complicated and challenging that they can only be solved with highly sophisticated algorithms using specialized mathematics.

While the costs of these designs and planning issues are minor compared to the investments necessary to migrate to a new technology or even to build a single new chip factory, they offer large potentials for improvement and optimization. This and the fact that the arising optimization problems, their constraints and objectives, can be captured far more exactly in mathematical terms than many other problems arising in practical applications, make VLSI design one of the most appealing, fruitful and successful application areas of mathematics.

1.4.2 General Purpose Methods for Combinatorial Optimization

Combinatorial optimization is a topic that consists of finding an optimal object from a finite set of objects. In many such problems, exhaustive search is not feasible. It operates on the domain of those optimization problems, in which the set of feasible solutions is discrete or can be reduced to discrete, and in which the goal is to find the best solution. Some common problems involving combinatorial optimization are the traveling salesman problem (TSP) and the minimum spanning tree problem (MST).



Combinatorial optimization is a subset of mathematical optimization that is related to operations research, algorithm theory, and computational complexity theory. It has important applications in several fields, including artificial intelligence, machine learning, mathematics, auction theory, and software engineering.

Some research literature considers discrete optimization to consist of integer programming together with combinatorial optimization although all of these topics have closely intertwined research literature. It often involves determining the way to efficiently allocate resources used to find solutions to mathematical problems.

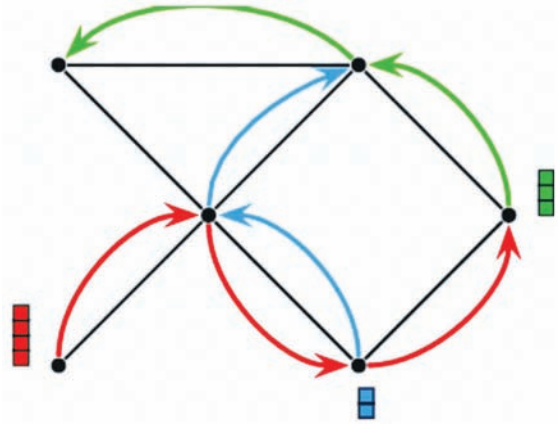
1.4.3 Method of Combinatorial Optimization

There is a large amount of literature on polynomial-time algorithms for certain special classes of discrete optimization, a considerable amount of it unified by the theory of linear programming. Some examples of combinatorial optimization problems that fall into this framework are shortest paths and shortest path trees, flows and circulations, spanning trees, matching, and matroid problems.

For NP-complete discrete optimization problems, current research literature includes the following topics:

- polynomial-time exactly solvable special cases of the problem at hand (e.g. see fixed-parameter tractable)
- algorithms that perform well on "random" instances (e.g. for TSP)
- approximation algorithms that run in polynomial time and find a solution that is "close" to optimal
- Solving real-world instances that arise in practice and do not necessarily exhibit the worst-case behavior inherent in NP-complete problems (e.g. TSP instances with tens of thousands of nodes.

Combinatorial optimization problems can be viewed as searching for the best element of some set of discrete items; therefore, in principle, any sort of search algorithm or meta-heuristic can be used to solve them. However, generic search algorithms are not guaranteed to find an optimal solution, nor are they guaranteed to run quickly (in polynomial time). Since some discrete optimization problems are NP-complete, such as the traveling salesman problem, this is expected unless P=NP.



1.4.4 Methods and Models for Combinatorial Optimization

Solution methods for Combinatorial Optimization Problems (COPs) fall into two classes: exact methods and heuristic methods. Exact methods are able, at least in theory, to provide an optimal solution, that is, a feasible solution that optimize (minimize or maximize) the value of the objective function, whereas heuristic methods provide a feasible solution with no optimality guarantee.

In some cases, we may be able to find "efficient" exact algorithms to solve a COP: for example, the problem of finding the shortest paths on a graph, under some reasonable assumptions often met in practice, can be solved by the Bellman-Ford algorithms, able to provide optimal solutions in running times according to a polynomial function (of small degree). For more complex problems, when no "efficient" algorithms are available, a possible approach may be formulating the COP as a Mixed Linear Programming (MILP) model and solving it by a MILP solver, which makes use of general purpose exact algorithms that guarantee, at least in theory, to find the optimal solution. These methods have an exponential computational complexity, so that the time to solve the problem may grow exponentially with its size.

It is not always possible or appropriate to apply exact solution methods, due to basically two concurrent issues: the inner complexity of a COP (e.g. an NP-Hard problem), and the time available to provide a solution, which may be limited. To this respect it is important to clarify that the use of a heuristic method instead of an exact one must be well motivated: the inner complexity of a problem does not justify in itself the use of heuristics, since literature may provide viable exact algorithms. The use of heuristic is thus motivated by the inner complexity of the COP together with consideration on the opportunity of implementing exact methods (which may require considerable implementation resources), the available computational time, and the size of the instances to be solved etc. *For example*, it is always advisable to make an

attempt to formulate a model of the COP (e.g. a MILP model): this effort is useful in the analysis phase, but also as an operational tool, since the growing efficiency of solvers may make the implementation of the model a viable approach to obtain an exact solution in reasonable running times. Nevertheless, the inner complexity of the problem may make a hard task to obtain an accurate enough formulation. Think about the problem of configuring a transportation network in order to minimize the congestion level, which depends also on the behaviour of the network users: even if several mathematical behavioral models are available in literature, often the only way for obtaining realistic results is simulation, which can be hardly embedded in a mathematical programming model (in general, an in a MILP model in particular). In other contexts, even if we have an accurate formulation, it may be the available time that inhibits the use of exact methods, since their computational complexity does not give any guarantee on the required running time: solving a problem in order of hours may be appropriate in some cases, non-acceptable in others.

The problem features and/or the solution context may make inappropriate the application of exact methods, while it is necessary to provides "good" feasible solutions in "reasonable" amount of time. It is worth nothing that, while in some cases the availability of a provably optimal solution is necessary, in many other cases, including perhaps the majority of real cases, a good approximate solution is enough, in particular for large size instances of a COP.

For example:

- for many parameters (data) determining a COP coming from a real application, just estimates are available, which may be also subject to error, and it may be not worth waiting a long time for a solution whose value (or even feasibility) cannot be ensured;
- A COP aims at providing one possible solution to a real problem aiming at a quick scenario evaluation (e.g., operational contexts, integration of optimization algorithms into interactive Decision Support Systems);
- A COP may be stated in a real time system, so that it is required that a "good" feasible solution is provided within a limited amount of time (e.g. fractions of seconds).

These examples attest for the extended use, in practice and in real applications, of methods that provide "good" solutions and guarantee acceptable computing times, even if they cannot guarantee optimality: they are called heuristic methods.

For many COPs it is possible to devise some specific heuristic that exploits some special feature of the COP itself and the human experience of who solves the problem in practice. In fact, very often, an "optimization" algorithm is just coding, when available, the rules applied to "manually" solve the problem. In this case, the quality of the obtained solution, that is, the effectiveness of the algorithm, depends on the rules themselves and, hence, on the amount of "good practice" that the algorithm

embeds: if the amount is high, we will obtain 'fairly "good" solutions (hardly better than the current ones, anyway); if the amount is little (or lacking, as it may happens if the developer has computer skills but no knowledge about the problem to be solved), the method is likely to be "the first reasonable algorithm come to our mind".

In the last decades, academic and practitioners' interest has been devoted to general heuristic approaches, able to outperform specific heuristics on the field. Related literature is vaster and has been fed by the ingenuity of researchers. So many techniques have been proposed that it is a very hard task to attempt a systematic and broadly accepted classification. A possible classification is the following:

- Constructive heuristics: they can be applied when the solution is given by selecting the "best" subset of a given set of elements. One starts from the empty set and iteratively adds one element to the solution, by applying some specific selection criterion. *For example*, if the selection criterion is some "local optimality" (e.g., the element that best improves the objective function), we obtain the so called greedy heuristics. The basic feature of such construction approaches is the fact that, in principle, the selection made at a certain step influences only the following steps, that is, no backtracking is applied;
- Meta-heuristics: they are general, multi-purpose methods or, better, algorithmic schemes which are devised independently from a specific COP. They define some components and their interactions, allowing them to provide a hopefully good solution. In order to devise a real algorithm for a specific COP, we need to devise and specialize each component. Among the well-known met-heuristics we cite: Local Search, Simulated Annealing, Tabu Search, Variable Neighborhood Search, Greedy Randomized Adaptive Search Techniques, Stochastic Local Search, Genetic Algorithms, Scatter Search, Ant Colony Optimization, Swarm Optimization, Neural Networks etc.
- approximation algorithms: they are a special class of heuristic methods able to provide a performance guarantee, that is, it is possible to formally prove that, for any instance of the COP, the obtained solution will never be worse than the optimal solution (which may be unknown) over a specified threshold (which is, often, rather large): for example, we will obtain a solution which is at most 30% far from the optimum.
- Iper-heuristics: we are here at the boundary between Operations Research and Artificial Intelligence, and the research is at its early stages. The aim is defining algorithm general algorithms able to automatically build good algorithms for a specific problem (for example, by trial and errors, they try to put together the right instructions, evaluating each attempt by applying each trial algorithm)

1.4.5 Algorithms Embedding Exact Solution Methods

The expansion criterion, that is choosing the best element to add, can be interpreted as an optimization (sub) problem, which is easier than the original COP. *For example*, the sub problem (or one approximation) may be modeled as a MILP and, if the MILP is still difficult to sole, the linear relaxation may be solved by standard solvers. The information provided by the solution may be used to define the scores. The sub-problem may be solved once before starting the heuristic procedure, or before each iteration, taking into account already fixed choices.

Normally, the time needed to run these algorithms is longer than the one required by greedy procedures, but the provided solutions are generally better. In fact, using an optimization model involves all (still-to-be-fixed) decision variables, so that the choices made at each step take to some extent into account a global optimality criterion.

Simplifying exact procedures some heuristic approaches simplifies exact procedures in order to make their complexity polynomial in time, at the cost of losing optimality guarantee. For example, an exact algorithm may be based on implicit enumeration (e.g., branch-and-bound) that gives rise to an exponential number of alternatives to be evaluated: a heuristic approach is to use some greedy criteria to select only a subset of alternatives. A simpler alternative is to stop an enumeration scheme, visited in depth, after a given number of alternatives have been explored, or after a fixed time-limit, and take the best solution generated so far. In practice, we may implement a MILP model using a standard solver and run it for a fixed amount of time, getting the best incumbent solution (if any).

A more sophisticated variant to an enumeration scheme is the beam search. It considers a partial breath-first visit of an enumeration tree: at each node, all the b alternatives (child nodes) are generated and, at each level of the tree, only k nodes are branched, k being a parameter to be calibrated according to the available computational time. The k nodes are chosen by associating to each node an evaluation that should be related to the likelihood a node is on the way to an optimal solution: a rapid evaluation (e.g., by greedy completing the partial solution at the node) of a possible solution in the sub-tree rooted at the node itself, a bound on the value of the optimal solution in the sub-tree, a weighted sum of these two values or any other evaluation. Then the k most promising nodes are chosen at each level, whereas the others are discarded: in such a way, we do not have the combinatorial explosion of the size of the enumeration tree, since at each level (at most) k nodes are taken and the enumeration trees reduces to a beam of $n \cdot k$ nodes (n being the number of levels of the enumeration tree). Hence, the overall complexity is polynomial, if polynomial is the node evaluation procedure. More in details, denoting by n the size of the problem in terms of variables to be fixed (number of tree's levels), by b the number of alternatives for each decision variable, that is, the number of children of each node, and by k the beam size, we will evaluate $O(n \cdot k \cdot b)$ nodes. The bottom level is made of k leaves, corresponding to k alternative

solutions among which the best is chosen. Notice that, starting from the time needed to evaluate a single node, one advantage of this technique is the possibility of a good estimation of the time needed to run it, once k is fixed; or, it is possible to estimate the parameter k based on the maximum available time.

SUMMARY

- A fundamental assumption about VLSI circuits is that they are designed by humans and built by machines.
- The front end of a CAD system is the human interface and there are two basic ways that it can operate: graphically or textually.
- A number of design styles exist to bridge the gap between text and graphics. These attempts to be less demanding than are precise polygon drawing systems while still capturing the graphical flavor.
- The designing an integrated circuit is a sequence of many actions most of which can be done by computer tools.
- At the earliest stage of the design, there is a necessity to experiment with specifications, to try to formalize them, etc.
- The algorithms are presented in a clear algorithmic style, often with considerable attention to data representation, though no extensive background in either data structures or programming is needed
- Computational complexity theory is a branch of the theory of computation in theoretical computer science that focuses on classifying computational problems according to their inherent difficulty, and relating those classes to each other.
- The goal is that the reader becomes aware that some problems in VLSI design automation are inherently difficult.
- It is hardly conceived how important mathematics and especially mathematical optimization is for all parts of VLSI technology.
- Combinatorial optimization is a subset of mathematical optimization that is related to operations research, algorithm theory, and computational complexity theory.

KNOWLEDGE CHECK

1. The utilization of CAD tools for drawing timing waveform diagram and transforming it into a network of logic gates is known as .....

a. Waveform Editor

b. Waveform Estimator

c. Waveform Simulator

d. Waveform Evaluator

2. **Which among the following is a process of transforming design entry information of the circuit into a set of logic equations?**
   a. Simulation
   b. Optimization
   c. Synthesis
   d. Verification
3. **Which level of system implementation includes the specific function oriented registers, counters & multiplexers?**
   a. Module level
   b. Logical level
   c. Physical level
   d. All of the above
4. **Graphic design allows the display of a circuit to be manipulated interactively, usually with a pointing device.**
   a. True
   b. False
5. **A number of design styles exist to bridge the gap between text and graphics.**
   a. True
   b. False

REVIEW QUESTIONS

1. How to solve VLSI design problem?
2. Discuss about algorithm and system design.
3. Explain the algorithmic graph theory and computational complexity.
4. What do you understand by combinatorial optimization in VLSI design?
5. Define the method of combinatorial optimization.

CHECK YOUR RESULT

1. (a) 2. (c) 3. (a) 4. (a) 5. (a)

REFERENCES

1. http://emicroelectronics.free.fr/onlineCourses/VLSI/ch01.html
2. http://www.or.uni-bonn.de/research/montreal.pdf
3. http://www.math.unipd.it/~luigi/courses/metmodoc/m02.meta.en.partial01.pdf
4. http://thesis.library.caltech.edu/1871/6/5chapter_5.pdf

CHAPTER

2

DESIGN AND FABRICATION OF VLSI DEVICES

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Define the fabrication materials
2. Explain the transistor fundamentals
3. Describe the fabrication of VLSI circuits
4. Discuss about design rules and layout of basic devices

INTRODUCTION

VLSI chips are manufactured in a fabrication facility usually referred to as a "fab". A fab is a collection of manufacturing facilities and "clean rooms", where wafers are processed through a variety of cutting, sizing, polishing, and deposition, etching and cleaning operations. Clean room is a term used to describe a closed environment where air quality must be strictly regulated. The number and size of dust particles allowed per unit volume is specified by the classification standard of the clean room. Usually space-suit like overalls and other dress gear is required for humans, so they do not contaminate the clean room. The cleanliness of air in a fab is a critical factor, since dust particles cause major damage to chips, and thereby affect the overall yield of the

fabrication process. The key factor which describes the fab in terms of technology is the minimum feature size it is capable of manufacturing. For example, a fab which runs a0.25 micron fabrication process is simply referred to as a 0.25 micron fab.

A chip consists of several layers of different materials on a silicon wafer. The shape, size and location of material in each layer must be accurately specified for proper fabrication. A mask is a specification of geometric shapes that need to be created on a certain layer. Several masks must be created, one for each layer. The actual fabrication process starts with the creation of a silicon wafer by crystal growth. The wafer is then processed for size and shape with proper tolerance. The wafer's size is typically large enough to fabricate several dozen identical/different chips. Masks are used to create specific patterns of each material in a sequential manner, and create a complex pattern of several layers. The order in which each layer is defined, or 'patterned' is very important. Devices are formed by overlapping a material of certain shape in one layer by another material and shape in another layer. After patterning all the layers, the wafer is cut into individual chips and packaged. Thus, the VLSI physical design is a process of creating all the necessary masks that define the sizes and location of the various devices and the interconnections between them.

The complex process of creating the masks requires a good understanding of the functionality of the devices to be formed, and the rigid rules imposed by the fabrication process. The manufacturing tolerances in the VLSI fabrication process are so tight that misalignment of a shape in a layer by a few microns can render the entire chip useless. Therefore, shapes and sizes of all the materials on all the layers of a wafer must conform to strict design rules to ensure proper fabrication. These rules play a key role in defining the physical design problems, and they depend rather heavily on the materials, equipment used and maturity of the fabrication process. The understanding of limitations imposed by the fabrication process is very important in the development of efficient algorithms for VLSI physical design.

2.1 FABRICATION MATERIALS

The electrical characteristics of a material depend on the number of 'avail-able' electrons in its atoms. Within each atom electrons are organized in con-centric shells, each capable of holding a certain number of electrons. In order to balance the nuclear charge, the inner shells are first filled by electrons and these electrons may become inaccessible. However, the outermost shell may or may not be complete, depending on the number of electrons available. Atoms organize themselves into molecules, crystals, or form other solids to completely fill their outermost shells by sharing electrons. When two or more atoms having incomplete outer shells approach close enough, their accessible outer most or valence electrons can be shared to complete all shells. This process leads to the formation of covalent bonds between atoms. Full removal of electrons from an atom leaves the atom with a net positive charge, of course, while the addition of electrons leaves it with a net negative charge. Such electrically unbalanced atoms are called ions.

The current carrying capacity of a material depends on the distribution of electrons within the material. In order to carry electrical current, some 'free' electrons must be available. The resistance to the flow of electricity ismeasured in terms of the amount of resistance in ohms (Ω) per unit length or resistivity. On the basis of resistivity, there are three types of materials, as described below:

Insulators

Materials which have high electrical resistance are called insulators. The high electric resistance is due to strong covalent bonds which do not permit free movement of electrons. The electrons can be set free only by large forces and generally only from the surface of the solid. Electrons within the solids cannot move and the surface of the stripped insulator remains charged until new elections are reintroduced.Insulators have electrical resistivity greater than millions of $G\Omega - cm$. The principle insulator used in VLSI fabrication is silicon dioxide. It is used to electrically isolate different devices, and different parts of a single device to satisfy design requirements.

Conductors

Materials with low electrical resistance are referred to as conductors. Low resistance in conductors is due to the existence of valence electrons. These electrons can be easily separated from their atoms. If electrons are separated from their atoms, they move freely at high speeds in all directions in the conductor, and frequently collide with each other. If some extra electrons are introduced into this conductor, they quickly disperse themselves throughout the material. If an escape path is provided by an electrical circuit, then electrons will move in the direction of the flow of electricity. The movement of electrons, in terms of the number of electrons pushed along per second, depends on how hard they are being pushed, the cross-sectional area of the

conductive corridor, and finally the electron mobility factor of the conductor. Conductors can have resistivity as low as $1\ \mu\Omega$-cm and are used to make connections between different devices on a chip. Examples of conductors used in VLSI fabrication include aluminum and gold. A material that has almost no resistance, i.e., close to zero resistance, is called a superconductor. Several materials have been shown to act as superconductors and promise faster VLSI chips. Unfortunately, all existing superconductors work at very low temperatures, and therefore cannot be used for VLSI chips without specialized refrigeration equipment.

Semiconductors

Materials with electrical resistivity at room temperature ranging from $10\ m\Omega$-cm to $1\ G\Omega$-cm are called semiconductors. The most important property of a semiconductor is its mode of carrying electric current. Current conduction in semiconductors occurs due to two types of carriers, namely, holes and free electrons.

Let us explain these concepts by using the example of semiconductor silicon, which is widely used in VLSI fabrication. A silicon atom has four valence electrons which can be readily bonded with four neighboring atoms. At room temperatures the bonds in silicon atoms break randomly and release electrons, which are called free electrons. These electrons make bonds with bond deficient ionized sites. These bond deficiencies are known as holes. Since the breaking of any bond releases exactly one hole and one free electron, while the opposite process involves the capture of one free electron by one hole, the number of holes is always equal to number of free electrons in pure silicon crystals (see Figure 2.1). Holes move about and repel one another, just as electrons do, and each moving hole momentarily defines a positive ion which inhibits the intrusion of other holes into its vicinity. In silicon crystals, the mobility of such holes is about one third that of free electrons, but charge can be 'carried' by either or both. Since these charge carriers are very few in number, 'pure' silicon crystal will conduct weakly. Although there is no such thing as completely pure crystalline silicon, it appears that, as pure crystals, semiconductors seem to have no electrical properties of great utility.

DID YOU KNOW?

Structured VLSI design had been popular in the early 1980s, but lost its popularity later because of the advent of placement and routing tools wasting a lot of area by routing, which is tolerated because of the progress of Moore's Law.

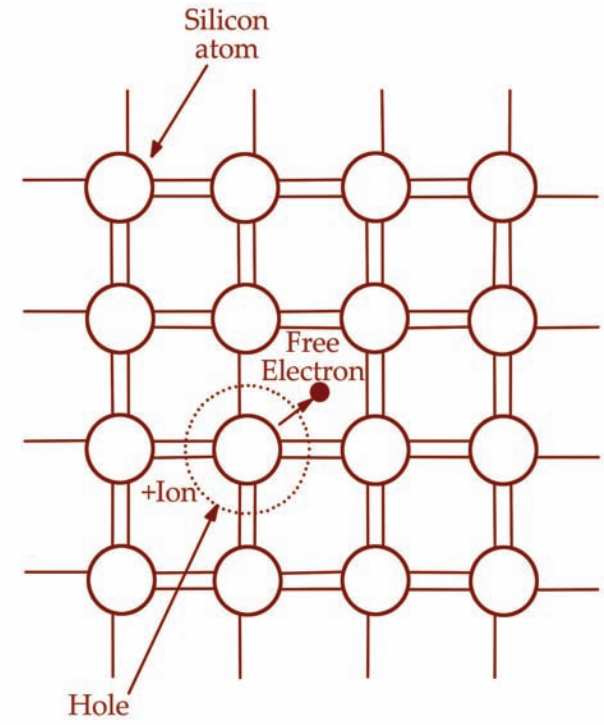


Figure2.1: Electrons and Holes.

KEYWORD

Capability is the ability to perform or achieve certain actions or outcomes.

Semiconductor crystals can be enriched either in holes or electrons by embedding some atoms of certain other elements. This fact makes it possible to build useful devices. An atom of phosphorus has five valence electrons, whereas an atom of boron has three valence electrons. Atoms of either kind can be locked into the silicon lattice, and even a few atoms can make a dominant contribution. Once placed into a silicon lattice, the fifth valence electron of each phosphorus atom is promptly freed. On the other hand, if a boron atom is placed into a silicon lattice, the covalence deficit of a boron atom is no less promptly covered by a neighborly silicon atom, which takes a hole in exchange and passes it on. With enough phosphorus (or boron) atoms per crystal, the number of free electrons or holes in general circulation can be increased a million fold.

This process of substituting other atoms for some of the semiconductor atoms is called *doping*. Semiconductors doped with electron donors such as phosphorus are said to be of the *n-type*, while boron doping, which results in extra holes,

produces *p-type* semiconductors. Though the doping elements give semiconductors desirable characteristics, they are referred to as impurities. Doping of silicon is easily accomplished by adding just the right amount of the doping element to molten silicon and allowing the result to cool and crystallize. Silicon is also doped by diffusing the dopant as a vapor through the surface of the crystalline solid at high temperature.

At such temperatures all atoms are vibrating significantly in all directions. As a result, the dopant atoms can find accommodations in minor lattice defects without greatly upsetting the overall structure. The conductivity is directly related to the level of doping. The heavilydoped material is referred to as n^+ or p^+ . Heavier doping leads to higher conductivity of the semiconductor.

A large chip may contain millions of vias, all of which must be opened properly for the chip to work.

In VLSI fabrication, both silicon and germanium are used as semiconductors. However, silicon is the dominant semiconductor due to its ease of handling and large availability. A significant processing advantage of silicon lies in its **capability** of forming a thermally grown silicon dioxide layer which is used to isolate devices from metal layers.

2.2 TRANSISTOR FUNDAMENTALS

In digital circuits, a 'transistor' primarily means a 'switch'- a device that either conducts charge to the best of its ability or does not conduct any charge at all, depending on whether it is 'on' or 'off'. Transistors can be built in a variety of ways, exploiting different phenomenon. Each transistor type gives rise to a circuit family. There are many different circuit families. A partiallist would include TTL (Transistor-Transistor Logic), MOS (Metal-Oxide-Semiconductor), and CMOS (Complimentary MOS) families, as well as theCCD (Charge-Coupled Device), ECL (Emitter-Coupled Logic), and I^2L (Integrated Injection Logic) families. Some of these families come in either p or n flavor (in CMOS both at once), and some in both high-power and low-power versions. In addition, some families are also available in both high and low speed versions. We restrict our discussion to TTL and MOS (and CMOS), and start with basic device structures for these types of transistors.

2.2.1 Basic Semiconductor Junction

If two blocks, one of n-type and another of p-type semiconductor are joined together to form a semiconductor junction, electrons and holes immediately start moving across the interface. Electrons from the n-region leave behind a region which is rich in positively charged immobile phosphorus ions. On the other hand, holes entering the interface from the p-region leave behind a region with a high concentration of uncompensated negative boron ions. Thus we have three different regions as shown in Figure 2.2. These regions establish a device with a remarkable one-way-flow property. Electrons cannot be introduced in the p-region, due to its strong repulsion by the negatively charged ions. Similarly, holes cannot be introduced in the n-region. Thus, no flow of electrons is possible in the p-to-n direction. On the other hand, if electrons are introduced in the n-region, they are passed along towards the middle region. Similarly, holes introduced from the other side flow towards the middle, thus establishing a p-to-p flow of holes.

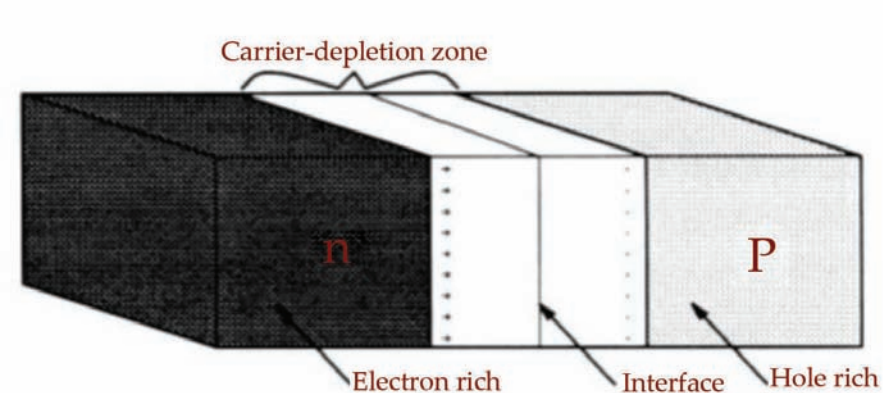
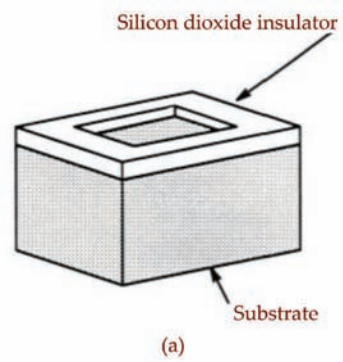


Figure2.2: The three regions in a n-p junction.



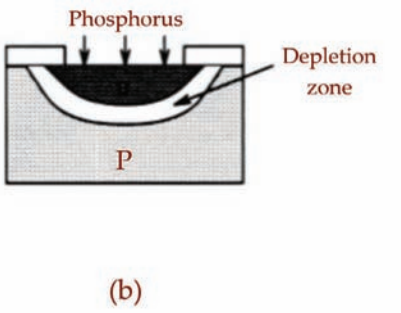
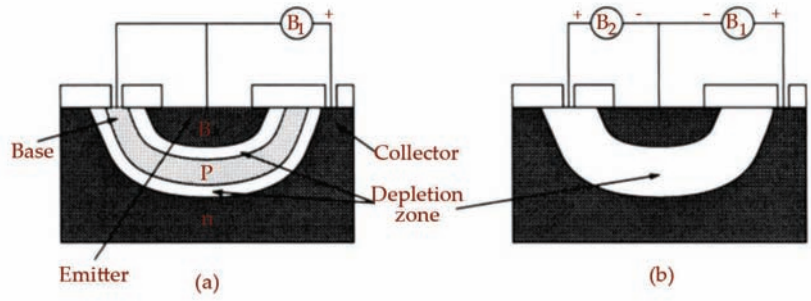


Figure2.3: Formation of a diffused junction.

The one-way-flow property of a semiconductor junction is the principle of the diode, and it can be used to develop two types of devices: unipolar and bipolar. Unipolar devices are created by using the semiconductor junction, under suitable external conditions, to modulate the flow of charge between two regions of opposite polarity. On the other hand, bipolar devices are created, under suitable external conditions, by isolating one semiconductor region from another of opposite majority-carrier polarity, thus permitting a charge to flow within the one without escaping into the other.

Great numbers of both types of devices can be made rather easily by doping a silicon wafer with diffusion of either phosphorus or boron. The silicon wafer is pre-doped with boron and covered with silicon dioxide. Diffused regions are created near the surface by cutting windows into a covering layer of silicon dioxide to permit entry of the vapor, as shown in Figure 2.3. Phosphorus vapor, for example, will then form a bounded n-region in a boron doped substrate wafer if introduced in sufficient quantity to overwhelm the contribution of the boron ions in silicon. All types of regions, with differing polarities, can be formed by changing the diffusion times and diffusing vapor compositions, therefore creating more complex layered structures. The exact location of regions is determined by the mask that is used to cut windows in the oxide layer.



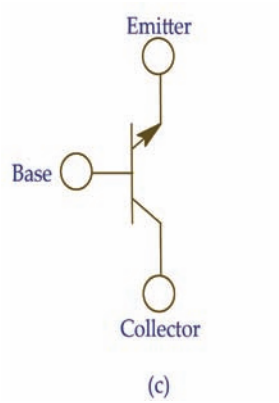


Figure2.4: TTL transistor.

In the following, we will discuss how the semi-conductor junction is used in the formation of both TTL and MOS transistors. TTL is discussed rather briefly to enable more detailed discussion of the simpler, unipolar MOS technology.

2.2.2 TTL Transistors

A TTL transistor is an n-p-n device embedded in the surface of p-type semi-conductor substrate (see Figure 2.4(a), the p-substrate is not shown). There are three regions in a TTL transistor, namely the emitter, the base, and the collector. The main idea is to control the flow of current between collector (n-region) and emitter $(n^+\text{-region})$ by using the base (p-region). The basic construction of a TTL transistor, both in its 'on' state and 'off' state, along with its symbol is shown in Figure 2.4. In order to understand the operation of a TTL transistor, consider what happens if the regions of the transistor are connected to a battery B_1 as shown in Figure 2.4(a). A few charge carriers are removed from both base and collector; however, the depletion zones at the emitter-base and base-collector interfaces prevent the flow of currents of significant size along any pathway. Now if another battery with a small voltage B_2 is connected as shown in Figure 2.4(b), then two different currents begin to flow. Holes are introduced into the base by B_2, while electrons are sent into the emitter by both B_1 and B_2. The electrons in the emitter cross over into the base region. Some of these electrons are neutralized by some holes, and since the base region is rather thin, most of the electrons pass through the base and move into the collector. Thus a flow of current is established from emitter to collector. If B_2 is disconnected from the circuit, holes in the base cause theflow to stop. Thus the flow of a very small current in the B_2-loop modulates the flow of a current many times its size in the B_1-loop.

2.2.3 MOS Transistors

MOS transistors were invented before bipolar transistors. The basic principle of a MOS transistor was discovered by J. Lilienfeld in 1925, and O. Heil proposed a structure closely resembling the modern MOS transistor. However, material problems failed these early attempts. These attempts actually led to the development of the bipolar transistor. Since the bipolar transistor was quite successful, interest in MOS transistors declined. It was not until1967 that the fabrication and material problems were solved, and MOS gained some commercial success. In 1971, nMOS technology was developed and MOS started getting wider attention.

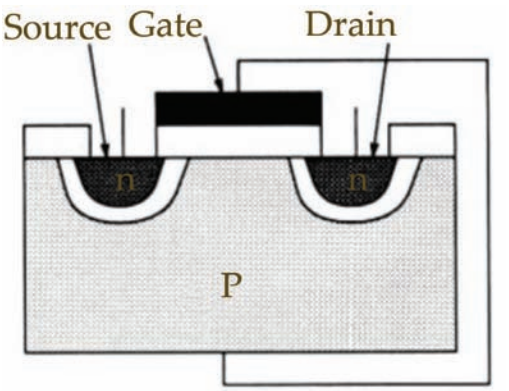
Digital circuit designs scale because the capacitive loads that must be driven by logic gates shrink faster than the currents supplied by the transistors in the circuit.

MOS transistors are unipolar and simple. The field-induced junction pro-vides the basic unipolar control mechanism of all MOS integrated circuits. Let us consider the n-channel MOS transistor shown in Figure 2.5(a). A p-type semiconductor substrate is covered with an insulating layer of silicon dioxide or simply oxide. Windows are cut into oxide to allow diffusion. Two separate n-regions, the source and the drain, are diffused into the surface of a p substrate through windows in the oxide. Notice that source and drain are insulated from each other by a p-type region of the substrate. A conductive material (polysilicon or simply poly) is laid on top of the gate.

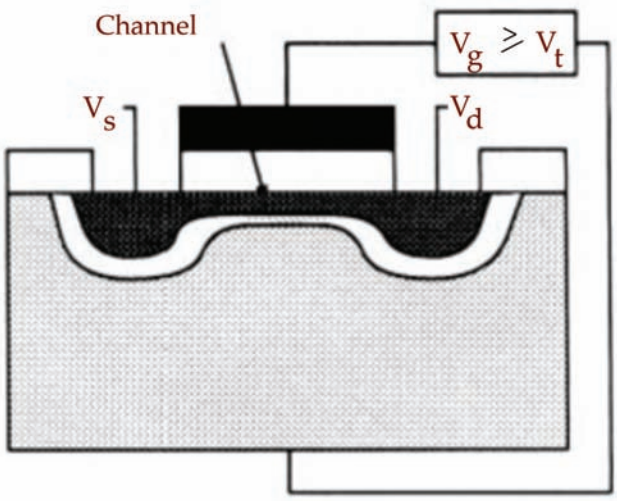
If a battery is connected to this transistor as shown in Figure 2.5(b), the poly acquires a net positive charge, as some of its free electrons are conducted away to the battery. Due to this positive charge, the holes in the substrate beneath the oxide are forced to move away from the oxide. As a result, electrons begin to accumulate beneath the oxide and form an n-type channel if the battery pressure, or more precisely the gate voltage V_g, is increased beyond a threshold value V_t. As shown in Figure 2.5(b), this channel provides a pathway for theflow of electrons from source to drain. The actual direction of flow depends on the source voltage (V_s) and the drain voltage (V_d) . If the battery is now disconnected, the charge on the poly disappears. As a result, the channel disappears and the flow stops. Thus a small voltage on the gate can be used to control the flow of current from source to drain. The symbols of an n-channel MOS gate are shown in Figure 2.5(c). A p-channel MOS transistor is a device complementary to the n-channel

transistor, and can be formed by using an n-type substrate and forming two p-type regions.

Integrated systems in metal-oxide semiconductor (MOS) actually contain three or more layers of conducting materials, separated by intervening layers of insulating material. As many as four (or more) additional layers of metal are used for interconnection and are called metal1, metal2, metal3 and so on.



(a)



(b)

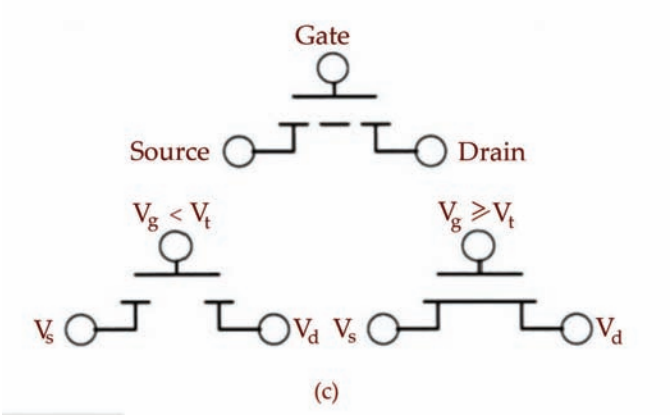


Figure2.5: A nMOS transistor.

Different patterns for paths on different levels, and the locations for contact cuts through the insulating material to connect certain points between levels, are transferred into the levels during the fabrication process from masks. Paths on the metal level can cross poly or diffusion levels in the absence of contact cuts with no functional effects other than a parasitic capacitance. However, when a path on the poly level crosses a path on the diffusion level, a transistor is formed.

The nMOS transistor is currently the preferred form of unipolar integration technology. The name MOS survives the earlier period in which gates were made of metal (instead of poly). Aluminum is the metal of choice for all conductivity pathways, although unlike the aluminum/oxide/semiconductor sandwich that provides only two topological levels on which to make interconnections, the basic silicon-gate structures provide three levels, and are therefore more compact and correspondingly faster. Recent advances in fabrication have al-lowed the use of up to four (or more) layers of metal. However, that process is expensive and is only used for special chips, such as microprocessors. Two or three metal technology is more commonly used for general purpose chips.

The transistors that are non-conducting with zero gate bias (gate to source voltage) are called enhancement mode transistors. Most MOS integrated circuits use transistors of the enhancement type. The transistors that conduct with zero gate bias are called depletion mode transistors. For a depletion modetransistor to turn off, its gate voltage V_g must be more negative than its thresh-old voltage (see Figure 2.6). The channel is enriched in electrons by an implant step; and thus an n-channel is created between the source and the drain. This channel allows the flow of electrons, hence the transistor is normally in its 'on' state. This type of transistor is used in nMOS as a resistor due to poor conductivity of the channel as shown in Figure 2.6(d). The MOS circuits dissipate DC power i.e., they dissipate power even when the output is low. The heat generated

is hard to remove and impedes the performance of these circuits. For nMOS transistors, as the voltage at the gate increases, the conductivity of the transistor increases. For pMOS transistors, the p-channel works in the reverse, i.e., as the voltage on the gate increases, the conductivity of the transistor decreases. The combination of pMOS and nMOS transistors can be used in building structures which dissipate power only while switching. This type of structure is called CMOS (Complementary Metal-Oxide Semiconductor).

CMOS technology was invented in the mid 1960's. In 1962, P. K. Weimer discovered the basic elements of CMOS flip-flops and independently in 1963, F. Wanlass discovered the CMOS concept and presented three basic gate structures. CMOS technology is widely used in current VLSI systems. CMOS is an inherently low power circuit technology, with the capability of providing a lower power-delay product comparable in design rules to nMOS and pMOS technologies. For all inputs, there is always a path from '1' or '0' to the output and the full supply voltage appears at the output. This 'fully restored' condition simplifies circuit design considerably. Hence the transistors in the CMOS gate do not have to be 'ratioed', unlike the MOS gate where the lengths of load and driver transistors have to be adjusted to provide proper voltage at the output. Another advantage of CMOS is that there is no direct path between VDD and GND for any combination of inputs. This is the basis for the low static power dissipation in CMOS. Table 2.1 illustrates the main differences between nMOS and CMOS technology. As shown in the table, the major drawback of CMOS circuits is that they require more transistors than nMOS circuits. In addition, the CMOS process is more complicated and expensive. On the other hand, power consumption is critical in nMOS and bipolar circuits, while it is less of a concern in CMOS circuits. Driver sizes can be increased in order to reduce net delay in CMOS circuits without any major concern of power. This difference in power consumption makes CMOS technology superior to nMOS and bipolar technologies in VLSI design.

2.3 FABRICATION OF VLSI CIRCUITS

Design and Layout of VLSI circuits is greatly influenced by the fabrication process; hence a good understanding of the fabrication cycle helps in designing efficient layouts. In this section, we review the details of fabrication.

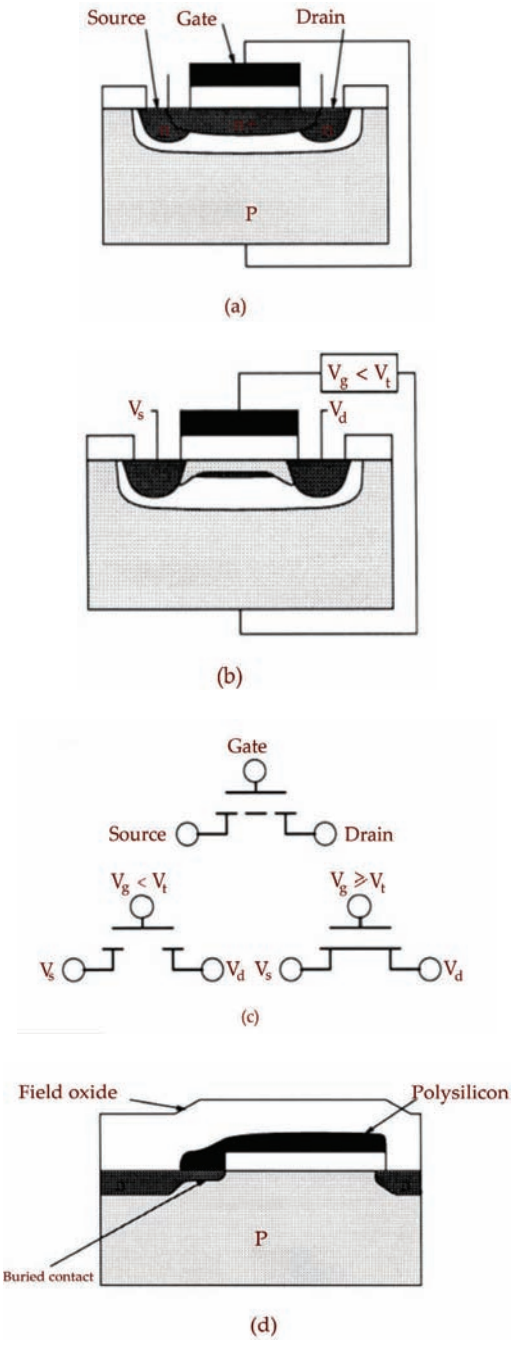


Figure 2.6: A depletion mode transistor.

Table 2.1: Comparison of CMOS and MOS characteristics

CMOS	MOS
Zero static power dissipation	Power is dissipated in the circuit with output of gate at '0'
Power dissipated during logic transition	Power dissipated during logic transition
Requires 2N devices for N inputs for complementary static gates	Requires (N+1) devices for N inputs
CMOS encourages regular layout styles	Depletion, load and different driver transistors create irregularity in layout

Fabrication of a VLSI chip starts by growing a large silicon crystal ingot about 20 centimeters in diameter. The ingot is sliced into several wafers, each about a third of a millimeter thick. Under various atmospheric conditions, phosphorus is diffused, oxide is grown, and polysilicon and aluminum are each deposited in different steps of the process. A complex VLSI circuit is defined by 6 to 12 separate layer patterns. Each layer pattern is defined by a mask. The complete fabrication process, which is a repetition of the basic three-step process (shown in Figure 2.7), may involve up to 200 steps.

- *Create*: This step creates material on or in the surface of the silicon wafer using a variety of methods. Deposition and thermal growth are used to create materials on the wafer, while ion implantation and diffusion are used to create material (actually they alter the characteristics of existing material) in the wafer.
- *Define*: In this step, the entire surface is coated with a thin layer of light sensitive material called photoresist. Photoresist has a very useful property. The ultraviolet light causes molecular breakdown of the photoresist in the area where the photoresist is exposed. A chemical agent is used to remove the dis-integrated photoresist. This process leaves some regions of the wafer covered with photoresist. Since exposure of the photoresist occurred while using the mask, the pattern of exposed parts on the wafer is exactly the same as in the mask. This process of transferring a pattern from a mask onto a wafer is called photolithography and it is illustrated in Figure 2.8.
- *Etch*: Wafers are immersed in acid or some other strong chemical agent to etch away either the exposed or the unexposed part of the pattern, depending on whether positive or negative photoresist has been used. The photoresist is then removed to complete the pattern transfer process.

This three step process is repeated for all the masks. The number of masks and actual details of each step depend on the manufacturer as well as the technology. In

the following analysis, we will briefly review the basic steps in nMOS and CMOS fabrication processes.

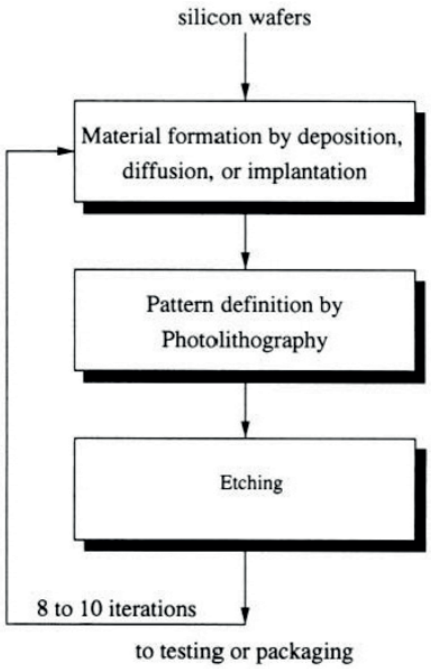


Figure2.7: Basic steps in MOS fabrication process.

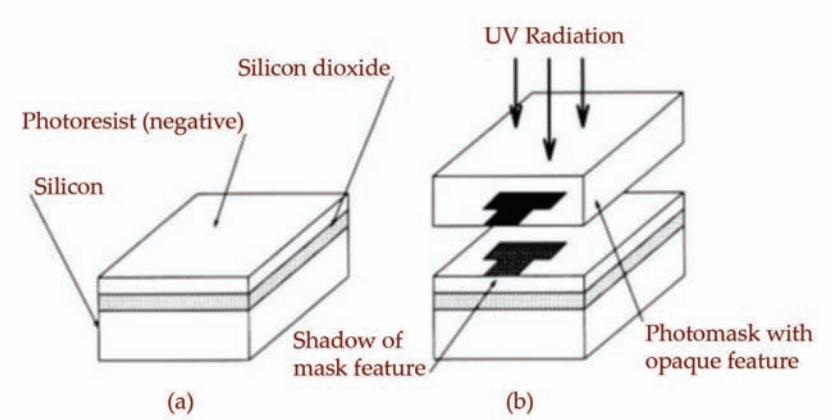
KEYWORD

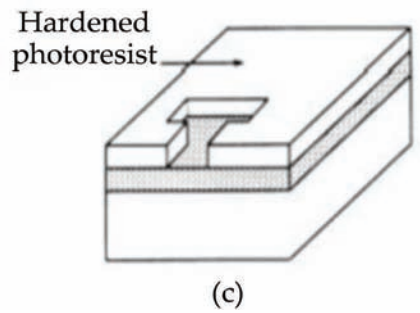
Diffusion refers to the process by which molecules intermingle as a result of their kinetic energy of random motion.

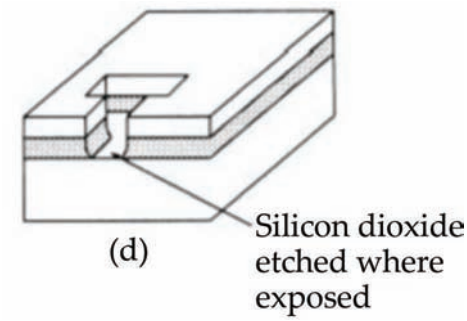
2.3.1 nMOS Fabrication Process

The first step in the n-channel process is to grow an oxide layer on lightly doped p-type substrate (wafer). The oxide is etched away (using the diffusion mask) to expose the active regions, such as the sources and drains of all transistors. The entire surface is covered with poly. The etching process using the poly mask removes all the poly except where it is necessary to define gates. Phosphorus is then diffused into all uncovered silicon, which leads to the formation of source and drain regions. Poly (and the oxide underneath it) stops diffusion into any other part of the substrate except in source and drain areas. The wafer is then heated, to cover the entire surface with a thin layer of oxide. This layer insulates the bare semiconductor areas from the pathways to be formed on top.

Oxide is patterned to provide access to the gate, source, and drain regions as required. It should be noted that the task of aligning the poly and **diffusion** masks is rather easy, because it is only their intersections that define transistor boundaries. This self-alignment feature is largely responsible for the success of silicon-gate technology. The formation of a depletion mode transistor requires an additional step of ion implantation.







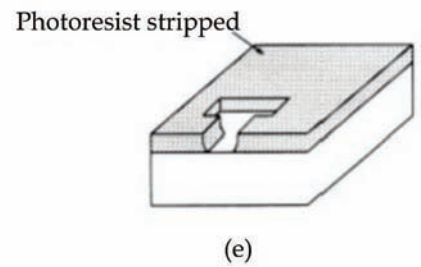


Figure2.8: Photolithographic process.

A thin covering of aluminum is deposited over the surface of the oxide, which now has 'hills' and 'valleys' in its structure. Etching then removes all but the requisite wires and contacts. Additional metal layers may be laid on top if necessary. It is quite common to use two layers of metal. At places where connections are to be made, areas are enlarged somewhat to assure good inter level contact even when masks are not in perfect alignment. All pathways are otherwise made as small as possible, in order to conserve area. In addition to normal contacts, an additional contact is needed in nMOS devices. The gate of a depletion mode transistor needs to be connected to its source. This is accomplished by using a buried contact, which is a contact between diffusion and poly.

The final steps involve covering the surface with oxide to provide mechanical and chemical protection for the circuit. This oxide is patterned to form windows which allow access to the aluminum bonding pads, to which gold wires will be attached for connection to the chip carrier. The windows and pads are very large as compared to the devices.

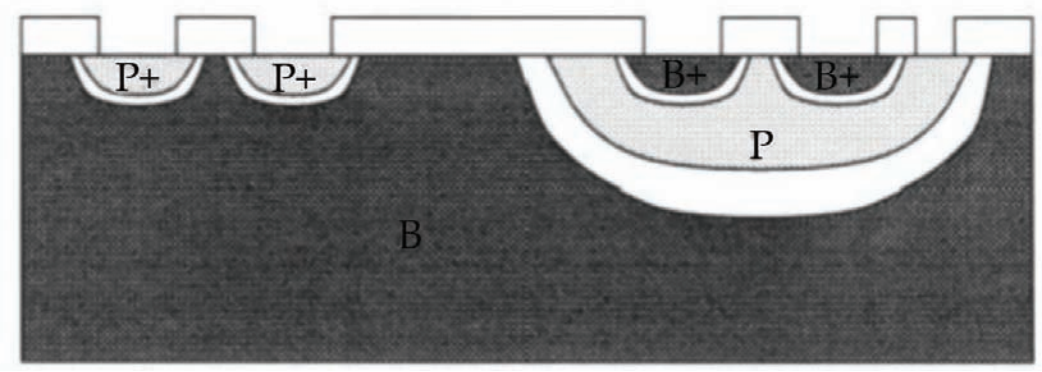


Figure 2.9: A p-well CMOS transistor.

2.3.2 CMOS Fabrication Process

CMOS transistors require both a p-channel and an n-channel device. How-ever, these two types of devices require two different substrates. nMOS transistors require a p-type substrate, while pMOS transistors require a n-type substrate. CMOS transistors are created by diffusing or implanting an n-type well in the original p-substrate. The well is also called a tub or an island. The p-channel devices are placed in the n-well. This is called the n-well CMOS process. A complementary process of p-well CMOS starts with an n-type substrate for pMOS devices, and creates a p-well for nMOS devices. The structure of a CMOS transistor is shown in Figure 2.9 (p-substrate is not shown). A twin-tub CMOS process starts with a lightly doped substrate and creates both a n-well and a p-well.

As compared to the nMOS process, the CMOS process requires a additional mask for ion implanting to create the deep p-wells, n-wells, or both.

2.3.3 Details of Fabrication Processes

The complete fabrication cycle consists of the following steps: crystal growth and wafer preparation, epitaxy, dielectric and polysilicon film deposition, oxidation, diffusion, ion implantation, lithography and dry etching. These steps are used in a generic silicon process, however they are similar to those of other technologies. Below, we discuss details of specific fabrication processes.

Crystal Growth and Wafer Preparation:

Growing crystals essentially involves a phase change from solid, liquid, or gas phases to a crystalline solid phase. The predominant method of crystal growth is Czochralski (CZ) growth, which consists of crystalline solidification of atoms from the liquid phase. In this method, single-crystal ingots are pulled from molten silicon contained in a fused silica crucible.

Electronic-grade silicon, which is a polycrystalline material of high purity, is used as the raw material for the preparation of a single crystal. The conditions and the parameters during the crystal-pulling operation de-fine many properties of the wafer, such as dopant uniformity and oxygen concentration. The ingots are ground to a cylindrical shape of precisely controlled diameter and one or more flats are ground along its length. The silicon slices are sawed from the ingot with an exact crystallographic orientation. Crystal defects adversely affect the performance of the device. These defects may be present originally in the substrate or may because by subsequent process steps. The harmful impurities and defects are removed by careful management of the thermal processes.

Epitaxy

This is the process of depositing a thin single-crystal layer on the surface of a single-crystal substrate. The word epitaxy is derived from two Greek words: epi, meaning 'upon', and taxis, meaning 'ordered'. Epitaxy is a Chemical Vapor Deposition (CVD) process in which a batch of wafers is placed in a heated chamber. At high temperatures (900° to 1250°C), deposition takes place when process gases react at the wafersurface. A typical film growth rate is about $1\ \mu m/min$. The thickness and doping concentration of the epitaxial layer is accurately controlled and, unlike the underlying substrate, the layer can be made oxygen- and carbon-free. A limitation of epitaxy is that the degree of crystal perfection of the deposited layer cannot be any better than that of the substrate. Other process-related defects, such as slip or impurity precipitates from contamination can be minimized.

In bipolar device technology, an epi-layer is commonly used to provide a high-resistivity region above a low-resistivity buried layer, which has been formed by a previous diffusion or 'implant and drive-in' process. The heavily doped buried layer serves as a low-resistance collector contact, but an additional complication arises when epitaxial layers are grown over patterned buried layer regions. To align the subsequent layers in relation to the pattern of the buried layer, a step is produced in the pre-epitaxial processing.

Dielectric and Polysilicon film deposition

The choice of a particular reaction is often determined by the deposition temperature (which must be compatible with the device materials), the properties, and certain engineering aspects of deposition (wafer throughput, safety, and reactor maintenance).

The most common reactions for depositing silicon dioxide for VLSI circuits are:

- Oxidizing silane (silicon hydrate) with oxygen at 400°-450°C.
- Decomposing tetra-ethoxysilane at 650° to 750°C, and reacting dichlorosilane with nitrous oxide at 850° to 900°C.

Doped oxides are prepared by adding a dopant to the deposition reaction. The hydrides arsine, phosphine, or diborane are often used because they are readily available gases. However, halides and organic compounds can also be used. Polysilicon is prepared by paralyzing silane at 600° to650°C.

Oxidation

The process of oxidizing silicon is carried out during the entire process of fabricating integrated circuits. The production of high-quality IC's requires not only an understanding of the basic oxidization mechanism, but also the electrical properties of the oxide. Silicon dioxide has several properties:

- Serves as a mask against implant or diffusion of dopant into silicon.
- Provides surface passivation.
- Isolates one device from another.
- Acts as a component in MOS structures.
- Provides electrical isolation of multilevel metallization systems.

Several techniques such as thermal oxidation, wet iodization, CVD etc. are used for forming the oxide layers.

When a low charge density level is required between the oxide and the silicon, Thermal oxidation is preferred over other techniques. In the thermal oxidation process, the surface of the wafer is exposed to an oxidizing ambient of O_2 or H_2O at elevated temperatures, usually at an ambient pressure of one atmosphere.

Diffusion

The process in which impurity atoms move into the crystal lattice in the presence of a chemical gradient is called diffusion. Various techniques to introduce dopants into silicon by diffusion have been studied with the goals of controlling the dopant concentration, uniformity, and reproducibility, and of processing a large number of device wafers in a batch to reduce the manufacturing costs. Diffusion is used to form bases, emitters, and resistors in bipolar device technology, source and drain regions, and to dope polysilicon in MOS device technology. Dopant atoms which span a wide range of concentrations can be introduced into silicon wafers in the following ways:

- Diffusion from a chemical source in vapor form at high temperatures.
- Diffusion from doped oxide source.
- Diffusion and annealing from an ion implanted layer.

Ion Implantation

Ion implantation is the introduction of ionized projectile atoms into targets with enough energy to penetrate beyond surface regions. The most common application is the doping of silicon during device fabrication. The use of 3-keV to 500-keV energy for doping of boron, phosphorus, or arsenic dopant ions is sufficient to implant the ions from about 100 to 10,000A° below the silicon surface. These depths place the atoms beyond any surface layers of 30A° native SiO_2 and there-fore any barrier effect of the surface oxides during impurity introduction is avoided. The depth of implantation, which is nearly proportional to the ion energy, can be selected to meet a particular application.

With ion implantation technology it is possible to precisely control the number of implanted dopants. This method is a low-temperature process and is compatible with other processes, such as photoresist masking.

Lithography

As explained earlier, lithography is the process delineating the patterns on the wafers to fabricate the circuit elements and provide for component interconnections. Because the polymeric materials resist the etching process they are called resists and, since light is used to expose the IC pattern, they are called photoresists.

The wafer is first spin-coated with a photoresist. The material proper-ties of the resist include (1) mechanical and chemical properties such as flow characteristics, thickness, adhesion and thermal stability, (2) optical characteristics such as photosensitivity, contrast and resolution and (3) processing properties such as metal content and safety considerations. Different applications require more emphasis on some properties than on others. The mask is then placed very close to the wafer surface so that it faces the wafer. With the proper geometrical patterns, the silicon wafer is then exposed to ultraviolet (UV) light or radiation, through a photo mask. The radiation breaks down the molecular structure of areas of exposed photoresist into smaller molecules. The photoresist from these areas is then removed using a solvent in which the molecules of the photoresist dissolve so that the pattern on the mask now exists on the wafer in the form of the photoresist. After exposure, the wafer is soaked in a solution that develops the images in the photosensitive material. Depending on the type of polymer used, either exposed or non-exposed areas of film are removed in the developing process. The wafer is then placed in an ambient that etches surface areas not protected by polymer patterns. Resists are made of materials that are sensitive to UV light, electron beams, X-rays, or ion beams. The type of resist used in VLSI lithography depends on the type of exposure tool used to expose the silicon wafer.

The circuit designer must have a clear understanding of the roles of various masks used in the fabrication process, and how the masks are used to define various features of the devices on-chip.

Metallization

Metal is deposited on the wafer with a mechanism similar to spray painting. Motel metal is sprayed via a nozzle. Like spray painting, the process aims for an even application of metal. Unlike spray painting, process aims to control the thickness within few nanometers. An uneven metal application may require more CMP. Higher metal layers which are thick may require several application of the process get the desired height.

Copper, which has better interconnect properties is increasing becoming popular as the choice material for interconnect. Cop-per does require special handling since a liner material must be provide between copper and other layers, since copper atoms may migrate into other layers due to electr-migration and cause faults.

Etching

Etching is the process of transferring patterns by selectively removing unmasked portions of a layer. Dry etching techniques have become the method of choice because of their superior ability to control critical dimensions reproducibly. Wet etching techniques are generally not suitable since the etching is isotropic, i.e., the etching proceeds at the same rate in all directions, and the pattern in the photoresist is undercut.

Dry etching is synonymous with plasma-assisted etching, which denotes several techniques that use plasmas in the form of low pressure gaseous discharges. The dominant systems used for plasma-assisted etching are constructed in either of two configurations: parallel electrode (planar) reactors or cylindrical batch (hexode) reactors. Components common to both of these include electrodes arranged inside a chamber maintained at low pressures, pumping systems for maintaining proper vacuum levels, power supplies to maintain a plasma, and systems for controlling and monitoring process parameters, such as operating pressure and gas flow.

Planarization

The Chemical Mechanical Planarization (CMP) of silicon wafers is an important development in IC manufacturing. Before the advent of CMP, each layer on the wafer was more un-even then the lower layer, as a result, it was not possible to increase the number of metal layers. CMP provides a smooth surface after each metallization step. CMP has allowed essentially unlimited number of layers of interconnect. The CMP process is like “Wet Sanding” down the surface until it is even. Contact and via layers are filled with tungsten plugs and planarized by CMP. ILD layers are also planarized by CMP.

Packaging

VLSI fabrication is a very complicated and error prone process. As a result, finished wafers are never perfect and contain many ‘bad’ chips. Flawed chips are visually identified and marked on the wafers. Wafers are then diced or cut into chips and the marked chips are dis-carded. ‘Good’ chips are packaged by mounting each chip in a small plastic or ceramic case. Pads on the chip are connected to the legs on the case by tiny gold wires with the help of a microscope, and the case is sealed to protect it from the environment. The finished package is tested and the error prone packages are discarded. Chips which are to be used in an MCM are not packaged, since MCM uses

unpackaged chips. The VLSI fabrication process is an enormous scientific and engineering achievement. The manufacturing tolerances maintained throughout the process are phenomenal. Mask alignment is routinely held to 1 micron in 10 centimeters, an accuracy of one part in 10^5 which is without precedent in industrial practice. For comparison, note that a human hair is 75 microns in diameter.

2.3.4 The Difference between CMOS technology and NMOS technology

KEYWORD

Noise immunity is the ability of a system to perform even when there is noise present.

The difference between CMOS technology and NMOS technology can be easily differentiated with their working principles, advantages and disadvantages as discussed.

CMOS Technology

Complementary metal oxide semiconductor (CMOS technology) is used to construct ICs and this technology is used in digital logic circuits, microprocessors, microcontrollers and static RAM. CMOS technology is also used in several analog circuits like data converters, image sensors and in highly integrated transceivers. The main features of CMOS technology are low static power consumption and high **noise immunity**.

When the couple of transistors are in OFF condition, the combination of series draws significant power only during switching between ON & OFF states. So, MOS devices do not generate as much waste heat as other forms of logic. For example, TTL (Transistor-Transistor Logic) or MOS logic, which normally have some standing current even when not changing

state. This allows a high density of logic functions on a chip. Due to this reason, this technology most widely used and is implemented in VLSI chips.

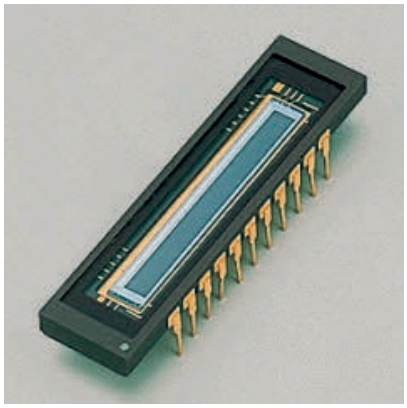
Advantages of CMOS Technology

These devices are used in a range of applications with analog circuits like, image sensors, data converters, etc. The advantages of CMOS technology over NMOS are as follows.

- Very low static power consumption
- Reduce the complexity of the circuit
- High density of logic functions on a chip
- Low static power consumption
- High noise immunity

NMOS Technology

NMOS is nothing but negative channel metal oxide semiconductor; it is pronounced as en-moss. It is a type of semiconductor that charges negatively. So that transistors are turned ON/OFF by the movement of electrons. In contrast, Positive channel MOS -PMOS works by moving electron vacancies. NMOS is faster than PMOS.



2.4 DESIGN RULES

The constraints imposed on the geometry of an integrated circuit layout, in order to guarantee that the circuit can be fabricated with an acceptable yield, are called design rules. The purpose of design rules is to prevent unreliable, or hard-to-fabricate (or unworkable) layouts. More specifically, layout rules are introduced to preserve the integrity of topological features on the chip and to prevent separate, isolated features

from accidentally short circuiting with each other. Design rules must also ensure thin features from breaking, and contact cuts from slipping outside the area to be contacted. Usually, design rules need to be re-established when a new process is being created, or when a process is upgraded from one generation to the next. The establishment of new design rules is normally a compromise between circuit design engineers and process engineers. Circuit designers want smaller and tighter design rules to improve performance and decrease chip area, while process engineers want design rules that lead to controllable and reproducible fabrication. The result is a set of design rules that yields a competitive circuit designed and fabricated in a cost-effective manner.

Design rules must be simple, constant in time, applicable in many processes and standardized among many fabrication facilities. Design rules are formulated by observing the interactions between features in different layers and limitations in the design process. For example, consider a contact window between a metal wire and a polysilicon wire. If the window misses the polysilicon wire, it might etch some lower level or the circuit substrate, creating a fatal fabrication defect. One should, undoubtedly, take care of basic width, spacing, enclosure, and extension rules. These basic rules are necessary parts of every set of design rules. Some conditional rules depend on electrical connectivity information. If, for instance, two metal wires are part of the same electrical node, then a short between them would not affect the operation of circuit. Therefore, the spacing requirement between electrically connected wires can be smaller than that between disconnected wires.

REMEMBER

A design rule checker must identify transistors and vias to ensure proper checks—otherwise, it might highlight a transistor as a poly-diffusion spacing error.

The design rules are specified in terms of microns. However, there is a key disadvantage of expressing design rules in microns. A chip is likely to remain in production for several years; however newer processes may be developed. It is important to produce the chip on the newer processes to improve yield and profits. This requires modifying or shrinking the layout to obey the newer design rules. This leads to smaller die sizes and the operation is called process shifting. If the layout is specified in microns, it may be necessary to rework the entire layout for process shifting. To overcome this scaling problem, Mead and Conway suggested the use of a single parameter to design the entire layout. The basic

idea is to characterize the process with a single scalable parameter called lambda (λ) defined as the maximum distance by which a geometrical feature on any one layer can stray from another feature, due to over-etching, misalignment, distortion, over or underexposure, etc., with a suitable safety factor included. λ is thus equal to the maximum misalignment of a feature from its intended position in the wafer. One can think of λ as either some multiple of the standard deviation of the process or as the resolution of the process. Currently, λ is approximately 0.25×10^{-6} m (0.25 μm). In order to simplify our presentation, we will use lambda.

Table 2.2: Basic nMOS design rules

Diffusion Region Width	2λ
Polysilicon Region Width	2λ
Diffusion-Diffusion Spacing	3λ
Poly-Poly Spacing	2λ
Polysilicon Gate Extension	2λ
Contact Extension	λ
Metal Width	3λ

Design rules used by two different fabrication facilities may be different due to the availability of different equipment. Some facilities may not allow use of a fourth or a fifth metal layer, or they may not allow a 'stacked via'. Usually, design rules are very conservative (devices take larger areas) when a fabrication process is new and tend to become tighter when the process matures. Design rules are also conservative for topmost layers (metal4 and metal5 layers) since they run over the roughest terrain.

The actual list of design rules for any particular process may be very long. Our purpose is to present basic ideas behind design rules, therefore, we will analyze simplified nMOS design rules. Table 2.2 lists basic nMOS design rules. We have omitted several design rules dealing with buried contact, implant, and others to simply our discussion.

As stated earlier, design rules are specified in fractions of microns. For example, separation for five metal layers may be 0.35 μm, 0.65 μm, 0.65 μm, 1.25 μm, and 1.85 μm respectively. Similar numbers are specified for each rule. Such rules do make presentation rather difficult, explaining our motivation to use the simpler lambda system. Although the lambda system is simple, sometimes it can become over-simplifying or even misleading. At such places we will indicate the problems caused by our simplified design rules.

In order to analyze design rules it is helpful to envision the design rules as a set of constraints imposed on the geometry of the circuit layout. We classify the rules in three types.

Size Rules

The minimum feature size of a device or an interconnect line is determined by the line patterning capability of lithographic equipment used in the IC fabrication. In 1998, the minimum feature size is $0.25\ \mu m$. Interconnect lines usually run over a rough surface, unlike the smooth surface over which active devices are patterned. Consequently, the minimum feature size used for interconnects is somewhat larger than the one used for active devices, based on pattern ability considerations. However, due to advances in planarization techniques, roughness problem of higher layers is essentially a solved problem.

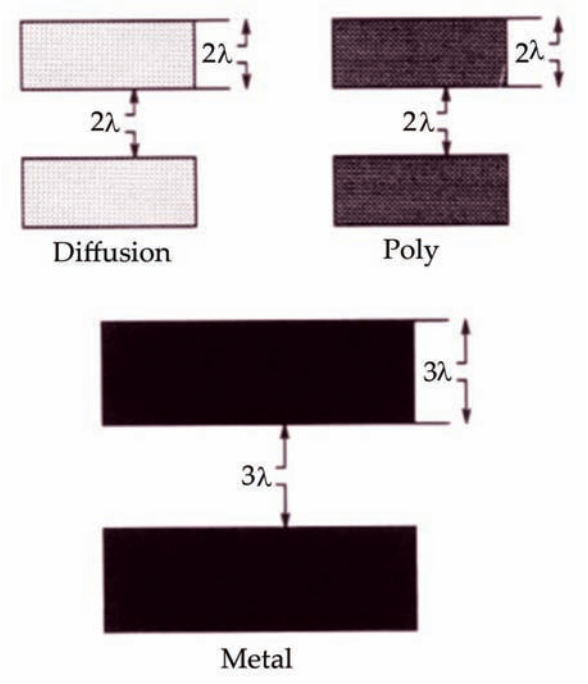


Figure2.10: Size and separation rules.

The design rule must specify the minimum feature sizes on different layers to ensure a valid design of a circuit. Figure 2.10 shows different size rules for feature sizes in different layers.

Separation Rules

Different features on the same layer or in different layers must have some separation from each other. In ICs, the interconnect line separation is similar to the size rule. The primary motivation is to maintain good interconnect density. Most IC processes have a spacing rule for each layer and overlap rules for vias and contacts. Figure 2.10 also shows the different separation rules in terms of λ.

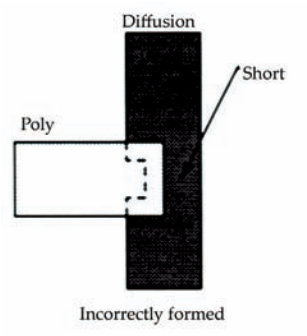
Overlap Rules

Design rules must protect against fatal errors such as a short-circuited channel caused by the mismigration of poly and diffusion, or the formation of an enhancement-mode FET in parallel with a depletion-mode device, due to the misregistration of the ion-implant area and the source/drain diffusion as shown in Figure 2.11. The overlap rules are very important for the formation of transistors and contact cuts or vias.

Figure 2.12 shows the overlap design rules involved in the formation of a contact cut.

In addition to the rules discussed above, there are other rules which do not scale. Therefore they cannot be reported in terms of lambda and are reported in terms of microns. Such rules include:

- The size of bonding pads, determined by the diameter of bonding wire and accuracy of the bonding machine.
- The size of cut in over glass (final oxide covering) for contacts with pads.
- The scribe line width (The line between two chips, which is cut by a diamond knife).
- The feature distance from the scribe line to avoid damage during scribing.
- The feature distance from the bonding pad, to avoid damage to the devices during bonding.
- The bonding pitch, determined by the accuracy of bonding machine.



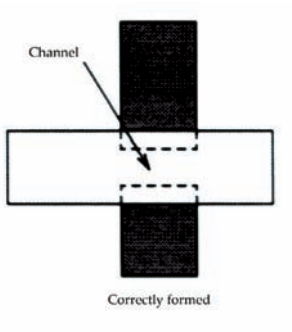


Figure2.11: (a) Incorrectly formed channel; (b) Correctly formed channel.

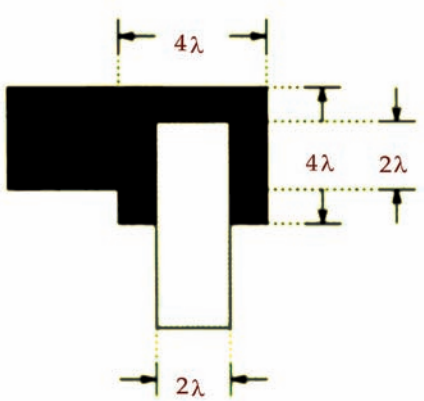


Figure2.12: Overlap rules for contact cuts.

We have presented a simple overview of design rules. One must study actual design rules provided by the fabrication facility rather carefully before one starts the layout. CMOS designs rules are more complicated than nMOS design rules, since additional rules are needed for tubs and pMOS devices.

The entire layout of a chip is rarely created by minimum design rules as discussed above. For performance and/or reliability reasons devices are de-signed with wider poly, diffusion or metal lines.

Long metal lines are sometimes drawn using two or even three times the minimum design rule widths. Some metal lines are even tapered for performance reasons. The purpose of these examples is to illustrate the fact that layout in reality is much more complex. Although we will maintain the simple rules for clarity of presentation, we will indicate the implications of complexity of layout as and when appropriate.

2.5 LAYOUT OF BASIC DEVICES

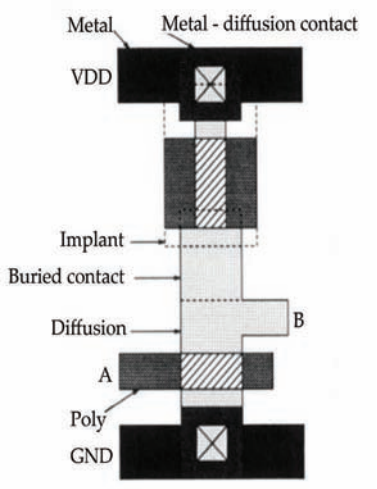
Layout is a process of translating schematic symbols into their physical representations. The first step is to create a plan for the chip by understanding the relationships between the large blocks of the architecture. The layout designer partitions the chip into relatively smaller sub circuits (blocks) based on some criteria. The relative sizes of blocks and wiring between the blocks are both estimated and blocks are arranged to minimize area and maximize performance. The layout designer estimates the size of the blocks by computing the number of transistors times the area per transistor. After the top level 'floor plan' has been decided, each block is individually designed. In very simple terms, layout of a circuit is a matter of picking the layout of each sub circuit and arranging it on a plane. In order to design a large circuit, it is necessary to understand the layout of simple gates, which are the basic building blocks of any circuit. In this section, we will discuss the structure of various VLSI devices such as the Inverter, NAND and NOR gates in both MOS and CMOS technologies.

2.5.1 Inverters

The basic function of an inverter is to produce an output that is complement of its input. The logic table and logic symbol of a basic inverter are shown in Figure 2.13(a) and (d) respectively. If the inverter input voltage A is less than the transistor threshold voltage V_t then the transistor is switched off and the output is pulled up to the positive supply voltage VDD. In this case the output is the complement of the input. If A is greater than V_t, the transistor is switched on and current flows from the supply voltage through the resistor R to GND. If R is large, V_{out} could be pulled down well below V_t thus again complementing the input. The main problem in the design of an inverter layout is the creation of the resistor. Using a sufficiently large resistor R would require a very large area compared to the area occupied by the transistor. This problem of large resistor can be solved by using a depletion mode transistor. The depletion mode transistor has a threshold voltage which is less than zero. Negative voltage is required to turn off a depletion mode transistor. Otherwise the gate is always turned on.

A	B
1	0
0	1

(a)



(b)

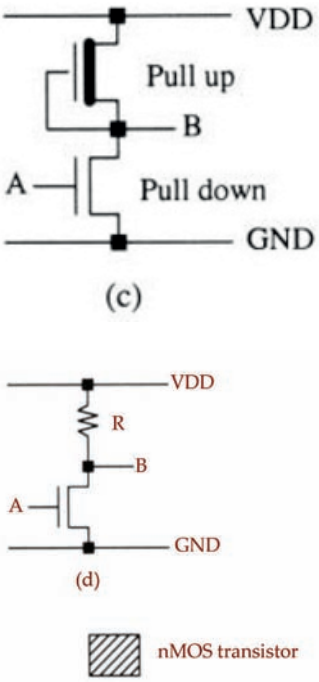


Figure2.13: An nMOS inverter.

The circuit diagram of an inverter is shown in Figure 2.13(c). The basic inverter layout on the silicon surface in MOS is given in Figure 2.13(b).It consists of two polysilicon (poly) regions overhanging a path in the diffusion level that runs between VDD and GND. This forms the two MOS transistors of the inverter. The transistors are shown by hatched regions in Figure 2.13(b).The upper transistor is called pull-up transistor, as it pulls up the output to 1.Similarly, the lower transistor is called the pull-down transistor as it is used to pull-down the output to zero. The inverter input A is connected to the poly that forms the gate of the lower of the two transistors. The pull-up is formed by connecting the gate of the upper transistor to its drain using a buried contact. The output of the inverter is on the diffusion level, between the drain of the pull-down and the source of the pull-up. The pull-up is the depletion mode transistor, and it is usually several times longer than the pull-down in order to achieve the proper inverter logic threshold. VDD and GND are laid out inmetal1 and contact cuts or vias are used to connect metal1 and diffusion.

If circuits became slower with smaller transistors, then circuits and layouts would have to be redesigned for each process.

The CMOS inverter is conceptually very simple. It can be created by connecting a p-channel and a n-channel transistor. The n-channel transistor acts as a pull-down transistor and the p-channel acts as a pull-up transistor. Figure 2.14 shows the layout of a CMOS inverter. Depending on the input, only one of two transistors conduct. When the input is low, the p-channel transistor between VDD and output is in its "on" state and output is pulled-up to VDD, thus inverting the input. During this state, the n-channel transistor does not conduct. When input is high, the output goes low. This happens due to the "on" state of the n-channel transistor between GND and output. This pulls the output down to GND, thus inverting the input.

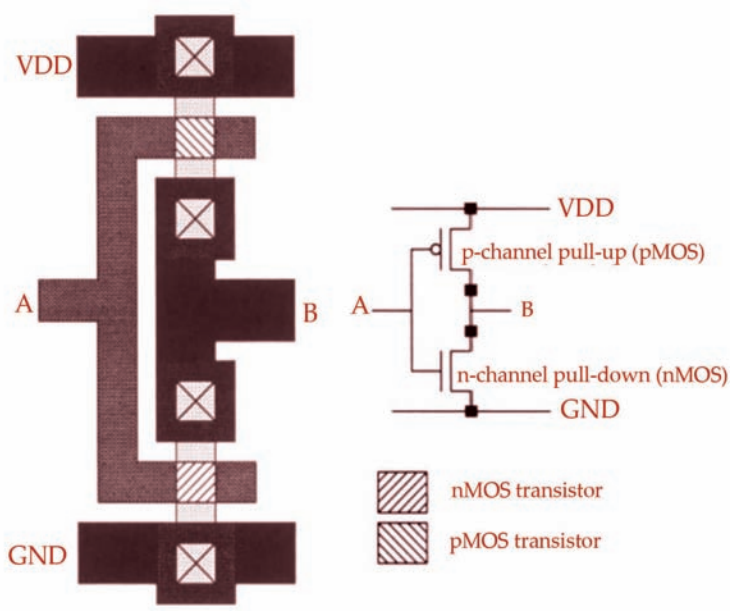


Figure2.14: A CMOS inverter.

During this state, p-channel transistor does not conduct. The design rules for CMOS are essentially same as far as poly, diffusion, and metal layers are concerned. Additional CMOS rules deal with tub formation.

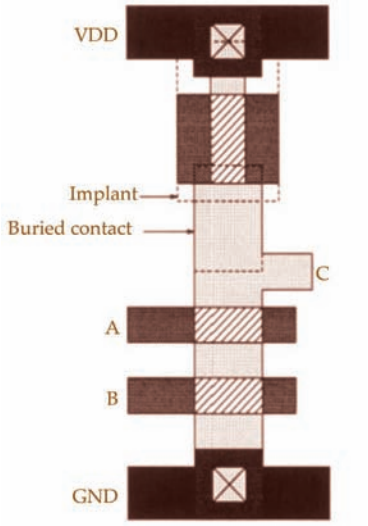
2.5.2 NAND and NOR Gates

NAND and NOR logic circuits may be constructed in MOS systems as a simple extension of the basic inverter circuit. The circuit layout in nMOS, truth tables, and logic symbols of a two-input NAND gate are shown in Figure 2.15 and NOR gate is shown in Figure 2.16.

In the NAND circuit, the output will be high only when both of the inputs A and B are high. The NAND gate simply consists of a basic inverter with an additional enhancement mode transistor in series with the pull-down transistor (see Figure 2.15). NAND gates with more inputs may be constructed by adding more transistors in series with the pull-down path. In the NOR circuit, the output is low if either of the inputs, A and B is high or both are high. The layout (Figure 2.16) of a two-input NOR gate shows a basic inverter with an additional enhancement mode transistor in parallel with the pull-down transistor. To construct additional inputs, more transistors can be placed in parallel on the pull-down path. The logic threshold voltage of an n-input NOR circuit decreases as a function of the number of active inputs (inputs moving together from logic-0 to logic-1). The delay time of the NOR gate with one input active is the same as that of an inverter of equal transistor geometries, except for added stray capacitance. In designing such simple combined circuits, a single pull-up resistor must be fixed above the point of output.

A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

(a)



(b)

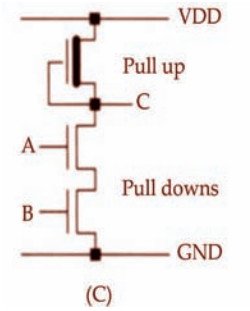
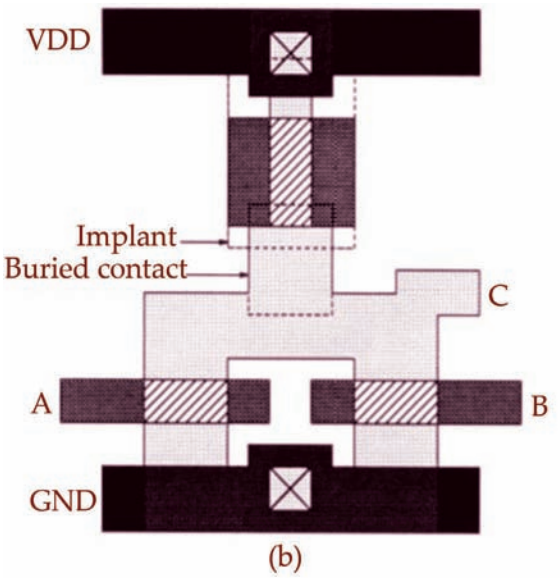


Figure2.15: A nMOS NAND gate.

A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

(a)



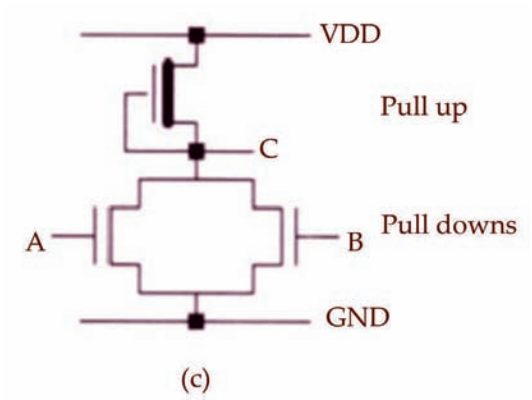


Figure2.16: A nMOS NOR gate.

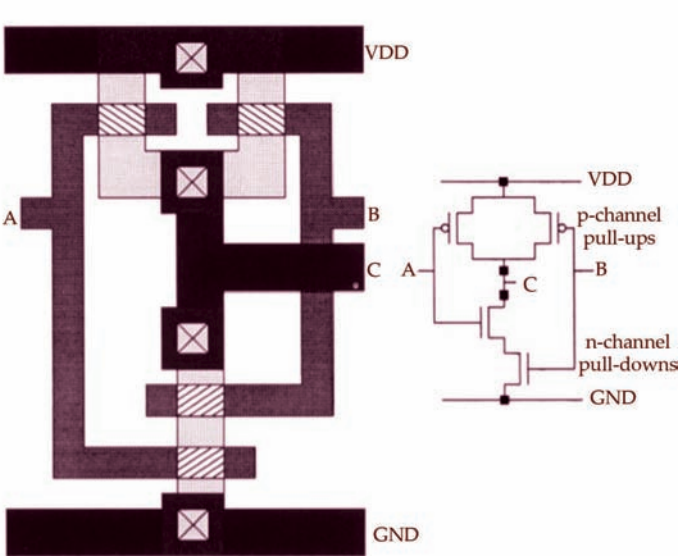


Figure2.17: A CMOS NAND gate.

The layouts of CMOS NAND and NOR gates are shown in Figure 2.17 and Figure 2.18 respectively. It is clear from Figure 2.17, that both inputs must be high in order for the output to be pulled down. In all other cases, the output will be high and therefore the gate will function as a NAND. The CMOS NOR gate can be pulled up only if both of the inputs are low. In all other cases, the output is pulled down by the two n-channel transistors, thus enabling this device to work as a NOR gate.

2.5.3 Memory Cells

Electronic memory in digital systems ranges from fewer than 100 bits from a simple four-function pocket calculator, to $10^5 - 10^7$ bits for a personal computer. Circuit designers usually speak of memory capacities in terms of bits since a unit circuit (for example a filp-flop) is used to store each bit. System designers in the other hand state memory capacities in the terms of bytes (typically 8-9bits) or words representing alpha-numeric characters, or scientific numbers. A key characteristic of memory systems is that only a single byte or word is stored or retrieved during each cycle of memory operation. Memories which can be accessed to store or retrieve data at a fixed rate, independent of the physical location of the memory address are called Random Access Memories or RAMs. A typical RAM cell is shown in Figure 2.19.

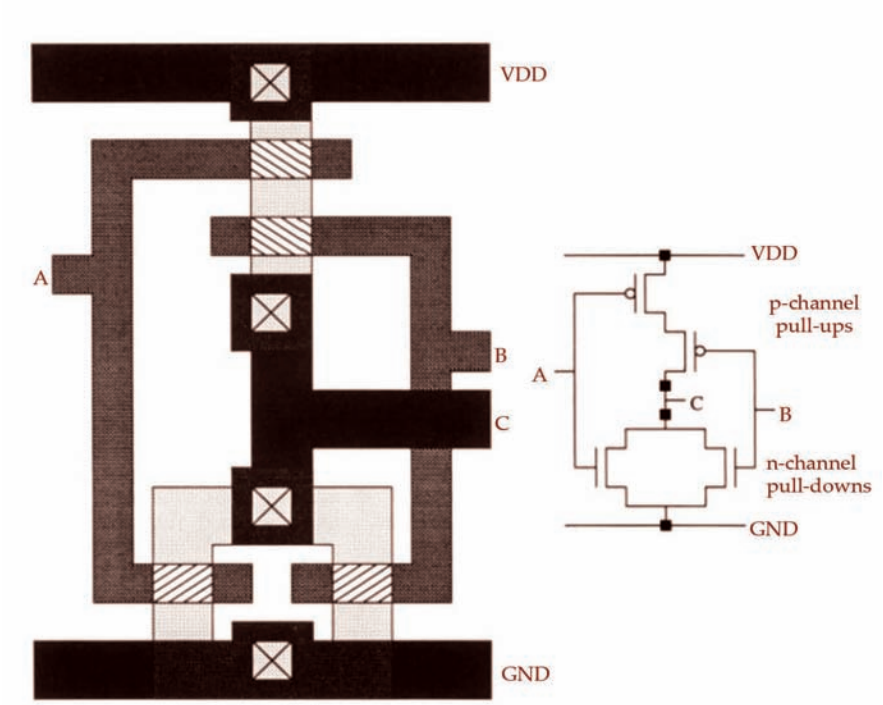


Figure2.18: A CMOS NOR gate.

Static Random Access Memory (SRAM)

SRAMs use static CMOS circuits to store data. A common CMOS SRAM is built using cross-coupled inverters to form a bi-stable latch as shown in Figure 2.20. The memory

cell can be accessed using the pass transistors P1 andP2 which are connected to the BIT and the BIT′ lines respectively. Consider the read operation. The n-MOS transistors are poor at passing a one and the p-transistors are generally quite small (large load resistors). To overcome this problem, the BIT and the BIT′ lines are pre-charged to a n-threshold below VDD before the pass transistors are switched on. When the SELECT lines (word line) are asserted, the RAM cell will try to pull down either the BIT or the BIT′ depending on the data stored. In the write operation, data and its complement are fed to the BIT and the BIT′ lines respectively. The word line is then asserted and the RAM cell is charged to store the data. A low on the SELECT lines, decouples the cell from the data lines and corresponds to a hold state.

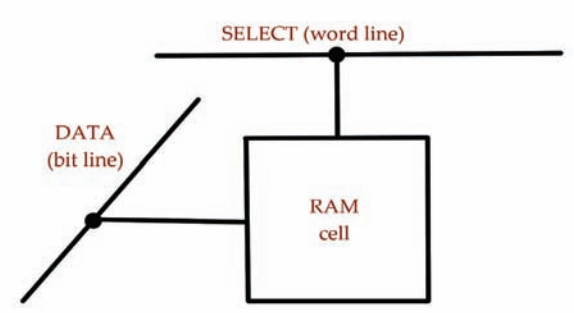


Figure2.19: Block diagram of a generic RAM cell.

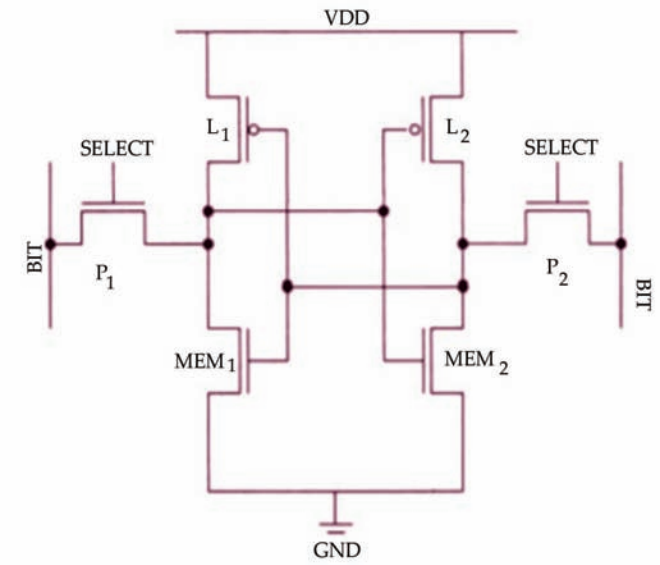


Figure2.20: A CMOS Static RAM cell.

The key aspect of the pre-charged RAM read cycle is the timing relationship between the RAM addresses, the pre-charge pulse and the row decoder (SELECT line). A word line assertion preceding the pre-charge cycle may cause the RAM cell to flip state. On the other hand, if the address changes after the pre-charge cycle has finished, more than one RAM cell will be accessed at the same time, leading to erroneous read data.

The electrical considerations in such a RAM are simple to understand as they directly follow the CMOS Inverter characteristics. The switching time of the cell is determined by the output capacitance and the feedback network. The time constants which control the charging and discharging are

$$\tau_{ch} = \frac{C_L}{\beta_p(VDD - |V_{Tp}|)}$$

$$\tau_{dis} = \frac{C_L}{\beta_n(VDD - V_{Tn})}$$

where C_L is the total load capacitance on the output nodes and β_n and β_p are the trans-conductance parameters for the n and p transistors respectively.

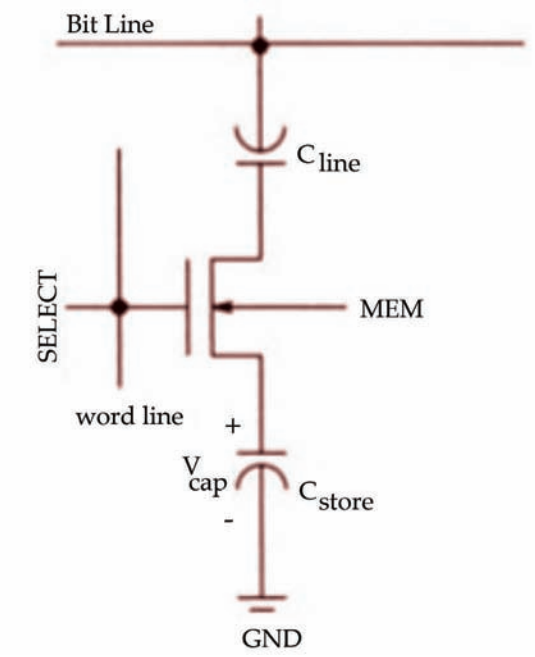


Figure2.21: A CMOS Dynamic RAM cell.

Minimizing the time constants within the constraints of the static noise margin requirements gives a reasonable criterion for the initial design of such cells.

Dynamic Random Access Memory (DRAM)

A DRAM cell uses a dynamic charge storage node to hold data. Figure 2.21shows a basic 1-Transistor cell consisting of an access nMOS MEM, a storage capacitor C_{store} and the input capacitance at the *bit line* C_{line}.

When the *Select* is set to high, C_{store} gets charged up to the bit line voltage according to the formula,

$$V_{cap}(t) = V_{max}[\frac{t/\tau_{ch}}{1 + t/\tau_{ch}}]$$

where $\tau_{ch} = 2C_{store}/\beta_n V_{max}$ is the charging time constant. The 90% voltage point $(0.9V_{max})$ is reached in a low-high time of $t_{LH} = 9\tau_{ch}$ and is the minimum logic 1 loading interval. Thus a logic one is stored into the capacitor. When a logic zero needs to be stored, the *Select* is again set to high and the charge on C_{store} decays according to the formula,

$$V_{cap}(t) = V_{max}[\frac{2e^{-t/\tau_{dis}}}{1 + e^{-t/\tau_{dis}}}]$$

Where $\tau_{dis} = C_{store}/\beta_n V_{max}$ is discharge time constant? The 10% voltage point $(0.1V_{max})$ requires a high-low time of $t_{HL} = 2.94\tau_{dis}$. Thus it takes longer to load a logic 1 $(t_{LH} = 6.11t_{HL})$ than to load a logic 0. This is due to the fact that the gate-source potential difference decreases during a logic 1 transfer.

The read operation for a dynamic RAM cell corresponds to a charge sharing event. The charge on C_{store} is partly transferred onto C_{line}. Suppose C_{store} has an initial voltage of V_c. The bit line capacitance C_{line} is initially charged to a voltage V_{pre} (typically 3V). The total system charge is thus given by $Q_T = V_C C_{store} + V_{pre} C_{line}$. When the SELECT is set to a high voltage, M becomes active and conducts current.

REMEMBER

Designers now must simulate across multiple fabrication process corners before a chip is certified ready for production, or use system-level techniques for dealing with effects of variation.

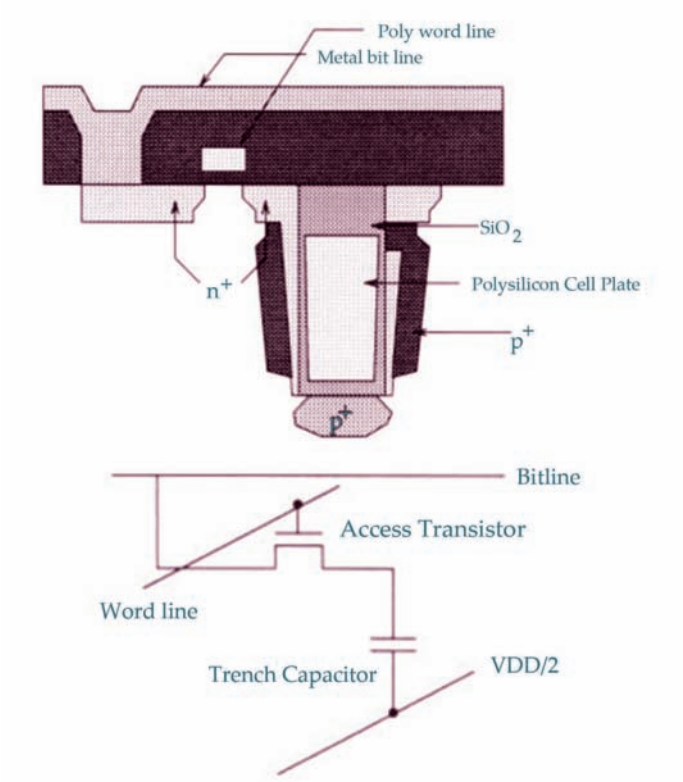


Figure2.22: A deep trench CMOS Dynamic RAM cell.

After the transients have decayed, the capacitors are in parallel and equilibrate to the same final voltage V_f such that

$$V_f = \frac{V_C C_{store} + V_{pre} C_{line}}{C_{store} + C_{line}}.$$

Defining the capacitance ratio $r = C_{line}/C_{store}$ yields the final voltage V_f as

$$V_f = \frac{V_C + rV_{pre}}{1 + r}.$$

If a logic 1, is initially stored in the cell, then $V_C = V_{max}$ and

$$V_1 = \frac{V_{max} + rV_{pre}}{1 + r}.$$

Similarly for $V_C = 0$ volts,

$$V_0 = \frac{rV_{pre}}{1 + r}.$$

Thus the difference between a logic 1 and a logic 0 is

$$\Delta V = \frac{V_{max}}{1 + r}.$$

The above equation clearly shows that a small r is desirable. In typical 16 Mb designs, $C_{store} \approx 30fF$ and $C_{line} \approx 250fF$ giving $r \approx 8$.

Dynamic RAM cells are subject to charge leakage, and to ensure data integrity, the capacitor must be refreshed periodically. Typically a dynamic re-fresh operation takes place at the interval of a few milliseconds where the peripheral logic circuit reads the cell and re-writes the bit to ensure integrity of the stored data.

High-value/area capacitors are required for dynamic memory cells. Recent processes use three dimensions to increase the capacitance/area. One such structure is the trench capacitor shown in Figure 2.22.

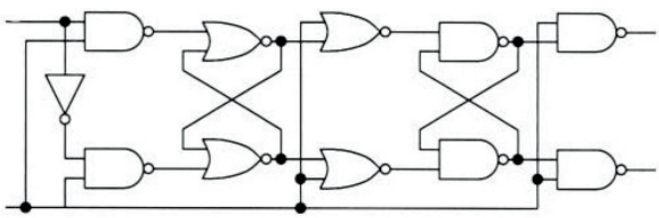


Figure2.23: Circuit 1.

The sides of the trench are doped n^+ and coated with a thin 10 nm oxide. Sometimes a thin oxynitride is used because its high dielectric constant increases the capacitance. The cell is filled with a polysilicon plug which forms the bottom plate of the cell storage capacitor. This is held at VDD/2 via a metal connection at the edge of thearray. The sidewall n^+ forms the other side of the capacitor and one side of the pass transistor that is used to enable data onto the bit lines. The bottom of the trench has a p^+ plug that forms a channel-stop region to isolate adjacent capacitors.

ROLE MODEL

LYNN CONWAY: AN AMERICAN COMPUTER SCIENTIST, ELECTRICAL ENGINEER, INVENTOR, AND TRANSGENDER ACTIVIST

Lynn Ann Conway (born January 2, 1938) is an American computer scientist, electrical engineer, inventor, and transgender activist.

Conway is notable for a number of pioneering achievements, including the Mead & Conway revolution in VLSI design, which incubated an emerging electronic design automation industry. She worked at IBM in the 1960s and is credited with the invention of generalized dynamic instruction handling, a key advance used in out-of-order execution, used by most modern computer processors to improve performance.

Lynn Conway was born in Mount Vernon, N.Y., in 1938. After studying physics at M.I.T., and earning a B.S. (1962) and M.S.E.E. (1963) at Columbia University, Lynn joined IBM Research in Yorktown Heights, N.Y. Working on IBM's Advanced Computing Systems project, she made foundational contributions to computer architecture including invention of multiple-out-of-order dynamic instruction scheduling.

Sadly, IBM fired Lynn as she underwent gender transition in 1968. Starting all over again in a covert new identity, Lynn advanced rapidly to become a computer architect at Memorex, but also began decades living in fear of being 'outed' and again losing her career.

Recruited by Xerox PARC in 1973, Lynn invented scalable design rules for VLSI chip design, became principal author of the famous Mead-Conway text Introduction to VLSI systems, and in 1978, while serving as a Visiting Associate Professor of EECS at M.I.T., pioneered the teaching of the new digital system design methods - thereby launching a revolution in microchip design in the 1980's.

While at PARC Lynn also invented and demonstrated an internet e-commerce infrastructure for rapid chip prototyping, spawning the "fabless-design + silicon-foundry" paradigm of semiconductor design and manufacturing. Institutionalized by DARPA at USC-ISI, the resulting "MOSIS" system enabled the rapid development of thousands of chip designs, leading to many major startups in the 80's and beyond.

As Assistant Director for Strategic Computing at DARPA, Lynn next crafted the meta-architecture and led the planning of the Strategic Computing Initiative, a major

1980's effort to expand the technology-base for modern intelligent-weapons systems. Lynn joined the University of Michigan in 1985 as Professor of EECS and Associate Dean of Engineering, where she continued her distinguished career. Now retired, she lives with her engineer husband Charlie on their 23 acre homestead in rural Michigan.

In 2012, the IEEE published Lynn's "VLSI Reminiscences" in a special issue of Solid-State Circuits Magazine. In that memoir Lynn finally began revealing how - closeted and hidden behind the scenes - she conceived the ideas and orchestrated the events that changed an industry.

A Fellow of the IEEE, Lynn received the Computer Pioneer Award of the IEEE Computer Society, holds an honorary doctorate from Trinity College, and is a Member of the National Academy of Engineering.

SUMMARY

- VLSI chips are manufactured in a fabrication facility usually referred to as a "fab".
- A chip consists of several layers of different materials on a silicon wafer. The shape, size and location of material in each layer must be accurately specified for proper fabrication.
- Masks are used to create specific patterns of each material in a sequential manner, and create a complex pattern of several layers.
- The manufacturing tolerances in the VLSI fabrication process are so tight that misalignment of a shape in a layer by a few microns can render the entire chip useless.
- The principle insulator used in VLSI fabrication is silicon dioxide. It is used to electrically isolate different devices, and different parts of a single device to satisfy design requirements.
- In VLSI fabrication, both silicon and germanium are used as semiconductors.
- In digital circuits, a 'transistor' primarily means a 'switch'- a device that either conducts charge to the best of its ability or does not conduct any charge at all, depending on whether it is 'on' or 'off'.

KNOWLEDGE CHECK

1. Chip utilization depends on ___.

a. Only on standard cells
b. Standard cells and macros
c. Only on macros
d. Standard cells macros and IO pads

2. What are pre-routes in your design?

a. Power routing
b. Signal routing
c. Power and Signal routing
d. None of the above.

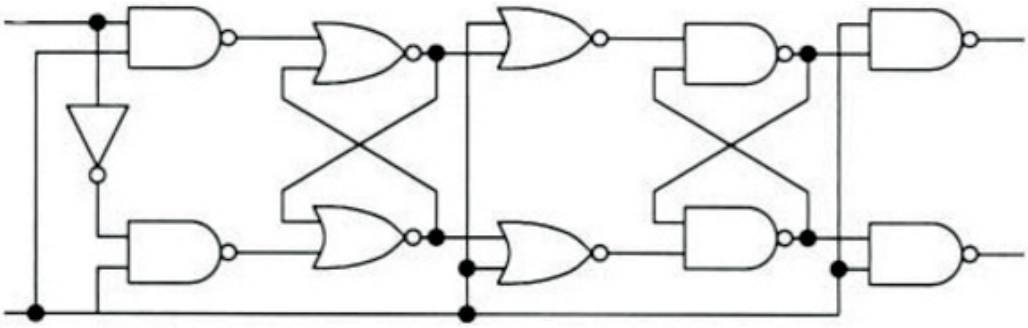
3. Utilization of the chip after placement optimization will be ___.

a. Constant
b. Decrease
c. Increase
d. None of the above

4. **ICs are used in .....**
   a. Linear devices only
   b. Digital devices only
   c. Both linear and digital devices
   d. None of the above
5. **Which of the following statement is true?**
   a. Fabrication of p-MOS transistor require few additional steps compared to n-MOS transistor
   b. Fabrication of n-MOS transistor require few additional steps compared to p-MOS transistor
   c. Fabrication on n-MOS is same as that of p-MOS transistor
   d. Fabrication on n-MOS is different from that of p-MOS transistor

REVIEW QUESTIONS

1. Explain the fabrication of VLSI circuits.
2. Differentiate between static random access memory and dynamic random access memory.
3. Explain the different types of design rules.
4. What do you mean by fabrication processes?
5. Compute the change in area if CMOS gates are used in a minimum area layout of the circuit in below figure.



Check Your Result

1. (b) 2. (a) 3. (c) 4. (a) 5. (a)

REFERENCES

1. Readcube, http://www.readcube.com/articles/10.1007/0-306-47509-X_2
2. Scribd, https://www.scribd.com/doc/51093768/Algorithms-for-VLSI-Physical-Design-Automation-Third-Edition
3. Elprocus, https://www.elprocus.com/difference-between-nmos-cmos-technology/
4. Scribd, https://www.scribd.com/doc/154484115/CAD-for-VLSI-Algorithms-for-VLSI-Physical-Design-Automation-by-Sherwani
5. http://bwrcs.eecs.berkeley.edu/Classes/icdesign/ee141_f01/Notes/chapter2.pdf

CHAPTER

3

VLSI SIMULATION

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Explain the advances of VLSI simulation
2. Define gate-level logic and fault simulation
3. Explain parallel switch-level simulation
4. Discuss on multilevel simulation: preprocessing and simulation

INTRODUCTION

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

Simulation is more than a mere convenience it allows a designer to explore his circuit in ways which may be otherwise impractical or impossible. The effects of

manufacturing and environmental parameters can be investigated without actually having to create the required conditions; the ability to detect manufacturing errors can be evaluated beforehand; voltages and currents can be determined without the difficulties associated with attaching a probe to a wire 500 times smaller than the period at the end of this sentence; and so on. To paraphrase a popular corporate slogan: without simulation, VLSI itself would be impossible! To use a simulator, the designer enters a design into the computer, typically in the form of a list of circuit components where each component connects to one or more nodes.

A node serves as a wire, transmitting the output of one circuit component to other components connected to the same node. The designer then specifies the voltages or logic levels of particular nodes, and calls upon the simulator to predict the voltages or logic levels of other nodes in the circuit. The simulator bases its predictions on models that describe the operation of the components. To be successful, a simulator requires the following characteristics of its models:

- The underlying model must not be too computationally expensive since the empirical nature of the verification provided by simulation suggests that it must be applied extensively if the results are to be useful.
- Component-level simulation is necessary to accurately model the circuit structures found in MOS designs. This allows the designer to simulate what was designed an advantage, since requiring separate specification of a design for simulation purposes only introduces another opportunity for error.
- The results must be correct, or at least conservative; a misleading simulation that results in unfounded confidence in a design is probably worse than no simulation at all. Here, we must trade off the conflicting desires of accuracy and efficiency.

Three of the more popular approaches to modeling are:

- Component models based on the actual physics of the component; for example, a transistor model that relates current flow through the transistor to the terminal voltages, device topology, and manufacturing parameters of the actual device.
- Component models based on a description of the logic operation performed by the component, e.g., NAND and NOR gates.
- Component models based on hybrid approaches which aim to approximate the predictions made by physical models, at a computational cost equal to that of gate-level models.

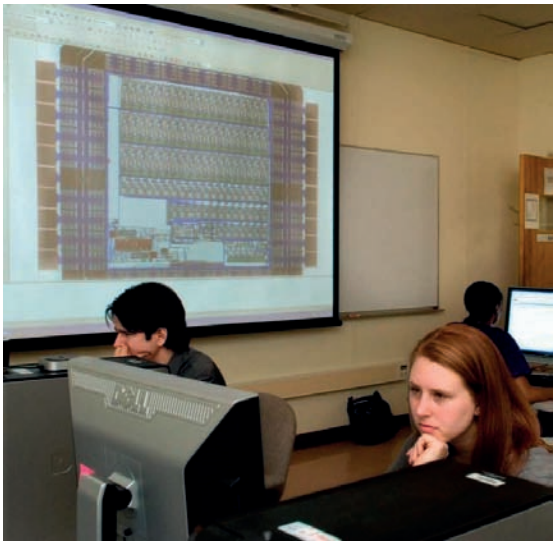
3.1 ADVANCES OF VLSI SIMULATION

Simulation has become indispensable in the process of designing, verifying, and testing complex digital systems. It is employed at different stages in the design process and at different levels of abstraction: a system is simulated at a high level before any logic design has taken place to answer architectural "what-if' questions; it is also simulated at the circuit level to determine the electrical behavior of the actual implementation. Simulation has largely replaced hardware prototyping for a variety of reasons: detailed software models for a digital system can be built faster and at lower cost; they provide a detailed picture of the system's operation and are easy to modify.

Simulation needs and requirements are clearly related to the rapid advances of Very Large Scale Integration (VLSI) technology. Individual chips have become larger and more complex, leading in turn to more complex systems. As a consequence, the demands on simulation have grown continuously, both in terms of computation time and memory requirements. Designers wish to simulate systems with a high degree of detail and with a large number of different test cases to shorten the design cycle and eliminate design errors early. Thus, simulation has become a potential bottleneck in design projects. To alleviate the problem one can pursue the following strategies either alone or in combination: take simulation "short-cuts" by simulating systems (at least partly) at a higher

Multiprocessor is a computer system having two or more processing units (multiple processors) each sharing main memory and peripherals, in order to simultaneously process programs.

level, possibly sacrificing some detail for increased performance; build dedicated hardware simulators; map simulation tools onto general purpose **multiprocessors**.



Clearly, system design has benefited a great deal from the progress of VLSI technology. Highly integrated processor chips have been designed into computer systems that offer an unprecedented cost-performance ratio. Increasingly, provisions are made to link several stand-alone systems using a high bandwidth interconnection network to allow the sharing of processing power and memory. Furthermore, tightly coupled multiprocessor systems are becoming available and offer potential supercomputing performance at much lower cost.

While multitasking applications can be mapped to such parallel machines in a straightforward manner by assigning tasks to processors, speeding up a single job by distributing it across processors is harder. It usually requires a substantial amount of code rewriting and also relies on efficient operating systems support. In fact, lack of software support and applications definitely is a roadblock for tightly coupled multiprocessors to become widely used. For their ultimate success in the marketplace it is crucial that a wide variety of applications runs on them efficiently.

The work has two aspects: on the one hand, we are interested in developing and implementing parallel algorithms for different types of simulation to provide cost effective simulation tools; on the other hand, we address general parallel processing issues that are relevant for other applications as well, such as portability, scalability, and strategies for mapping a problem onto a parallel framework. In particular, we are defining a general framework for the parallel simulation of digital systems to fulfill two requirements: one should be able to run simulation efficiently on machines of different underlying architectures (in particular shared versus distributed memory);

secondly, it should accommodate different types of simulation, for example both gate-level logic and fault simulation.

3.1.1 Logic and Fault Simulation

The simulation of VLSI systems is performed at different levels of abstraction, ranging from architectural to gate- and switch-level to circuit and device simulation. In the case of high level simulation one abstracts largely from the actual implementation and models the system as an interconnection of functional blocks specified as subroutines in a high-level programming language. Corresponding to the level of abstraction, the time granularity in the different types of simulation varies: architectural simulation typically has a time resolution based on complete bus or machine cycles, gate- and switch-level simulation is based on gate delays and clock cycles; circuit simulators in contrast compute complete continuous waveforms. The time granularity in the simulation is a significant parameter for the performance of parallel simulation schemes.

Logic simulators are in widespread use as tools to analyze the behavior of digital circuits. Logic simulators rely on abstract models of the functioning of a digital system. They yield discrete value outputs and crude timing information. For gate-level simulation, circuits are described in terms of 'primitive1 logic gates. They are typically evaluated through table look-up or by calling a software function. Gate-level simulators are popular, because they can be implemented efficiently and because the gate model is in direct correspondence to the Boolean equations representation of digital designs. In contrast, a switch-level simulator operates directly on the transistor-level description of a circuit.

The transistors are modeled to behave like switches and the steady state of the transistor network is computed iteratively. Switch-level simulators can handle a larger class of circuits and design styles since phenomena such as bidirectional signal flow and charge sharing are modeled. Also, the simulation model is in close correspondence to the actual physical implementation; for example, a circuit description can be extracted from the layout and simulated directly. In fault simulation, one is interested in the behavior of a digital design in the presence of certain predefined faults. A widely used assumption is that physical failures can be abstracted in the stuck-at fault model, where individual signal lines have permanent values '0' and '1' respectively under fault. Fault simulators are mainly used for the following purposes: (1) fault grading; the number of faults actually detected by a given set of test vectors is determined to compute the coverage of the test. (2) (Automatic) test pattern generation; the circuit is simulated with test patterns that have already been generated for specific faults. Thus, one finds out which additional faults are detected and can therefore be dropped from further consideration in order to save test generation time.

Clearly, both time and space requirements of fault simulation exceed those of logic simulation: large portions of the circuit need to be re-evaluated to determine the effect

In gate-level and switch-level simulation, circuits are modeled as the interconnection of Boolean gates and idealized transistors respectively, while circuit simulators are based on more detailed physical device models.

of a fault; additional memory is required to store the values of signals under faults.

An event is a change in the value of a signal and an element of the model to be simulated is recomputed only if one of its inputs changes. This is in contrast to all-event simulation (employed for example in the EVE hardware accelerator), where the entire model is updated for each time step. The main steps of event-driven simulation are summarized in Figure 3.1.

3.1.2 Parallelism in Simulation

We find the following types of parallelism in logic and fault simulation:

- **Algorithm parallelism:** during simulation a number of different tasks need to be executed, such as computing new output values, scheduling fan-out elements, and processing primary inputs and outputs. Some of these operations can be performed in a pipelined fashion by letting dedicated processors take care of specific tasks. Thus, the simulation algorithm is partitioned functionally.

```
simulate_event_driven
for all input vectors to be simulated {
  (1) process new inputs: update input nodes and schedule
     connected elements.
  (2) while ( elements left for evaluation ) {
        evaluate element;
        if ( change on output ){
          update all fanout nodes and schedule
          connected elements;
        }
     } /* while */
} /* for */
```

Figure 3.1. Event driven simulation algorithm.

- **Data parallelism:** if a design is to be simulated with different sets of input vectors, each of these test cases can run independently on one of the processors in

a multiprocessor system. Furthermore, in fault simulation, the set of faults to be considered can be split up; again, each subset can be processed on one of the processors. In both cases, a copy of the whole design is simulated on each processor. Data parallelism is particularly attractive for vector processors where the main objective is to create long sequences of identical operations. Thus the sequence of input patterns for a circuit can be viewed as a vector; similarly, the signal values for a node in a circuit for the fault free case and under a number of single faults, respectively, form a vector.

- **Model parallelism:** since the electrical signals in a circuit propagate concurrently along many paths, different elements are active at the same time and can be computed by several processors in parallel. In this framework, the model is partitioned and each processor 'owns' a sub-circuit.

While algorithm parallelism seems to be exploited best by special purpose hardware, data and model parallel approaches lend themselves to an implementation in a loosely coupled distributed or a tightly coupled parallel environment. Typically, data parallel approaches take the least amount of recoding effort and can yield perfect linear speedup; however, their applicability is limited due to space and algorithm considerations. Model parallel approaches are geared to handle large designs that need to be partitioned; they involve complex preprocessing and usually require complete redesign of the simulation system.

3.1.3 Hardware Accelerators, Vector Machines, and Data Parallelism

Several research projects have investigated the use of dedicated hardware accelerators for VLSI simulation. Examples for accelerators that use model parallelism are the Yorktown Simulation Engine (YSE), EVE. The EVE accelerator is in production use at IBM. It uses more than 200 special purpose processors. They are either logic processors that evaluate arbitrary 4-input 1-output logic functions or so-called array processors that simulate memory elements (the so-called arrays). The processors are connected by a full 256-to-256 cross-point switch that is scheduled statically for inter-processor communication. The design to be simulated is partitioned automatically among the processors based on the rank ordering of the logic. EVE can simulate up to L8 million gates and 50 Mbytes of storage. Its peak performance is rated at 2.2 billion gate evaluations per second; typical real performance lies at about 80% of the peak performance.

A widely used commercial accelerator for gate-level simulation is offered by Zycad; it is a multiprocessor with up to sixteen processors that implement scheduling and evaluation in hardware. It exhibits a peak performance of 16 million gate evaluations per second and can handle up to 1.1 million gates.

The approach in suggests a pipelined design where each functional unit performs specific subtask. A similar approach has been followed in the design and implementation

of the MARS hardware accelerator. The hardware consists of 15 processing elements connected to a crossbar switch. The processors are micro-programmed engines with special message passing and table manipulation capabilities. The simulation algorithm is partitioned among the processors. In contrast to other accelerators MARS has general purpose features such as multiple delay simulation and spike and race analysis. The application of the accelerators to fault simulation has been limited. Running switch-level simulation is not straightforward due to the iterative nature of the computation.

Algorithms and implementations of logic and fault simulation on vector machines and multiprocessors with vector processing. For highest efficiency, the algorithms here are mostly restricted to handle only combinational gate-level circuits.

The parallelism in the simulation model is increased in various preprocessing steps, for example by inserting fan-out trees to limit the amount of fan-out per processor.

For switch-level simulation, several special purpose hardware engines have been proposed and built. Recently, the compiled switch-level simulator COSMOS has been mapped onto a massively parallel framework. COSMOS preprocesses a transistor net list into a set of Boolean formulas that capture the switch-level behavior. These formulas are mapped directly onto the processing elements of a massively parallel computer (in this study the Thinking Machines connection machine). As in the case of the EVE hardware accelerator, the evaluation processed in an oblivious manner that is all elements (in this case Boolean expressions) are evaluated for each time step to avoid the overhead of scheduling for event-driven evaluation. Although the feasibility of mapping switch-level simulation onto a massively parallel framework was demonstrated, the cost effectiveness of the approach could not be shown: while one benchmark circuit gave a speedup of two over a VAX 8800 implementation.

Some limitation of the various approaches are: (1) the use of dedicated accelerators and supercomputers will not always be cost-effective due to the high investment in hardware necessary; (2) accelerators and vector machines are not suited for a wide range of simulation tasks; (3) running copies of a design with different input sets or disjoint fault sets on individual processors of a multiprocessor is possible only for moderate circuit sizes.

3.2 GATE-LEVEL LOGIC AND FAULT SIMULATION

The simulation algorithms for parallel logic and fault simulation are presented the complexity of the synchronization scheme is greatly reduced due to the partitioning strategy, thus underlying the need to always consider partitioning and actual simulation as one problem rather than separate issues.

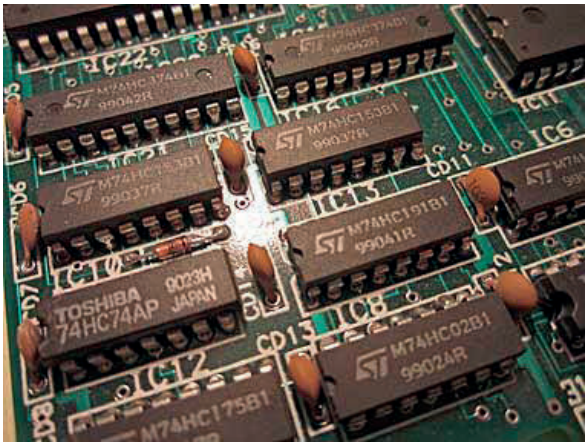
3.2.1 Circuit Model

Apart from the usefulness of gate-level simulation for design verification and fault simulation, the example has the advantage of having a 'homogeneous domain of decomposition', because the model to be simulated is a collection of simple building blocks which can be evaluated at uniform cost. Furthermore, the example is an extreme case due to its low computational requirements and can be useful for comparison with computationally more demanding types of simulation.

Logic gate is an idealized or physical device implementing a Boolean function; that is, it performs a logical operation on one or more binary inputs, and produces a single binary output.

The circuits considered here contain two types of elements: zero-delay **logic gates** and unit delay elements. The unit-delay elements will often be realized as clocked latches. All feedback loops are required to contain at least one delay element. Thus, the circuit can be uniquely levelized by assigning level zero to all primary inputs and latches. Computing the steady state of the circuit reduces to first evaluating the zero-delay combinational logic in one pass and the updating all latches for the next clock phases. All gates are evaluated only when all inputs fanning into it are available. This way, no unnecessary evaluation take place and fictitious glitches are eliminated.

In a model parallel framework, the inputs of some element in the sub-circuit of one processor may originate on a different processor. Thus, in a partitioning scheme that potentially cuts arbitrary signal lines, the processors need to be synchronized level by level. A corresponding simulation algorithm is outlined in Figure 3.2.



3.2.2 Partitioning Approach

Let the zero-delay fan-in cone of a latch be defined recursively as all zero-delay elements fanning into the latch and their zero-delay fan-in cones. Then the goal of the partitioning is to assign the latches of the circuit to different processors and to ensure that each processor also owns the complete zero-delay fan-in cone of that latch. Thus, all processors can compute a complete zero-delay cycle independently and need to synchronize on the clock boundary only. Note that the communication is synchronous; however, the time granularity has been increased.

```
Simulate_Level {
  for all input vectors to be simulated {
    for all levels l = 1 to l = maximum level {
      receive all needed signals (at rank l-1) from other
      processors;
      schedule affected gates;
      evaluate gates at rank l and schedule fanout elements;
      send required signals for rank l+1 to other processors;
    } /* for all levels */
  } /* for all input vectors */
} /* Simulate_Level */
```

Figure 3.2. Simulation algorithm, synchronization by level.

The following requirements for partitioning a given circuit into a number of sub-circuits:

(1) The partitioner should build connected circuit blocks of substantial size;

(2) It should cluster together events as much as possible and reduce the number of synchronizations needed;

(3) It should be efficient (preferably linear-time) so that very large designs can be handled. This is also motivated by the conjecture that a kind of 'economics of scale' applies to the partitioning problem: if the total problem size and consequently the grain size (i.e. the problem size per processor) grows, the sub-optimality of a scheme becomes less significant.

3.2.3 Logic Simulation Algorithm

The proposed partitioning approach leads to a straightforward cycle by cycle simulation scheme. Synchronization is performed on the clock boundary only, instead of level by level.

```
cones () {
   currentProcessor  = 0 ;
   BlockSize = total number of latches/ number of processors ;

   while ( unassigned latches left ) {
     pick unassigned latch ;
     assign latch to currentProcessor ;
     if ( number of latches in currentProcessor == BlockSize )
       currentProcessor += 1 ;
     for all elements elem fanning into latch
       traceBack ( elem ) ;
   } /* while */
} /* cones */

traceBack ( fanin ) {
  if ( fanin is latch ) {
    if ( unassigned ) {
      assign latch to currentProcessor ;
      if ( number of latches in currentProcessor == BlockSize )
        currentProcessor += 1 ;
      for all elements elem fanning into latch
        traceBack ( elem ) ;
    }
  } else {
      if ( fanin marked as visited ) return ;
      else {
         mark fanin as visited ;
         for all elements elem fanning into fanin
           traceBack ( elem ) ;
      }
  }
} /* traceBack */
```

Figure 3.3. Partitioning routines.

Assuming the typical maximum number of levels in a circuit to be on the order of 10 to 30, a considerable reduction of the number of synchronizations needed between processors is achieved. Furthermore that due to our partitioning approach all signals transmitted between processors are the outputs of latches. Hence, they need to be sent only when the latches are enabled. Then all signals 'owned' by a particular clock can be grouped together and are sent only when the clock is active. For the special case of one system clock, the number of synchronizations is reduced by a factor two (if the clock is inactive half the time).

The basic features of the sequential evaluation routine are similar to those in the simulator ECS, which has an external interface similar to that used in the EVE hardware accelerator. All primitive elements are four-input/one-output gates. Using four-valued logic with values {0,1,X,H (for high impedance)}, the elements are evaluated by table lookup with 256-byte zoom tables: the current input state is held in a one byte character which is concatenated with the function index for the respective gate to yield the offset in to the function table. In case the output changes, it is propagated to all fan-out elements in a spread step: again using table lookup, the output is shifted to the correct input pin position of the receiving gate and - if desired - at the same time inverted.

```
Simulate_Cycle {
  for all input vectors to be simulated {
    receive all needed signals from other processors;
    schedule affected gates;
    for all levels l = 1 to l = maximum level {
      evaluate gates at rank l and schedule fanout elements;
    } /* for all levels */
    send required signals for next cycle to other processors;
  } /* for all input vectors */
} /* Simulate_Cycle */
```

Figure 3.4. Simulation algorithm, synchronization by cycle.

To process the activity of receiving signals from the input buffers and writing out values to the output buffers, two new gate types, the receive gate and the send gate, are introduced in the following way. When a new vector of signals from another processor is received it is immediately spread to the receive gates to clear the buffer. The receive gates then propagate the changes, if any, to the circuit. Thus, if a signal from processor A is connected to several gates of B, it will be sent only once and a receive gate in B will take care of fanning it out. Similarly, a send gate writes new signal values, if any, directly into the buffers that will be transmitted to other processors. Send and receive gates thus represent 'pins' on the boundary of a sub-circuit and ensure the event-driven updating of the associated signal values. This is illustrated in Figure 3.5.

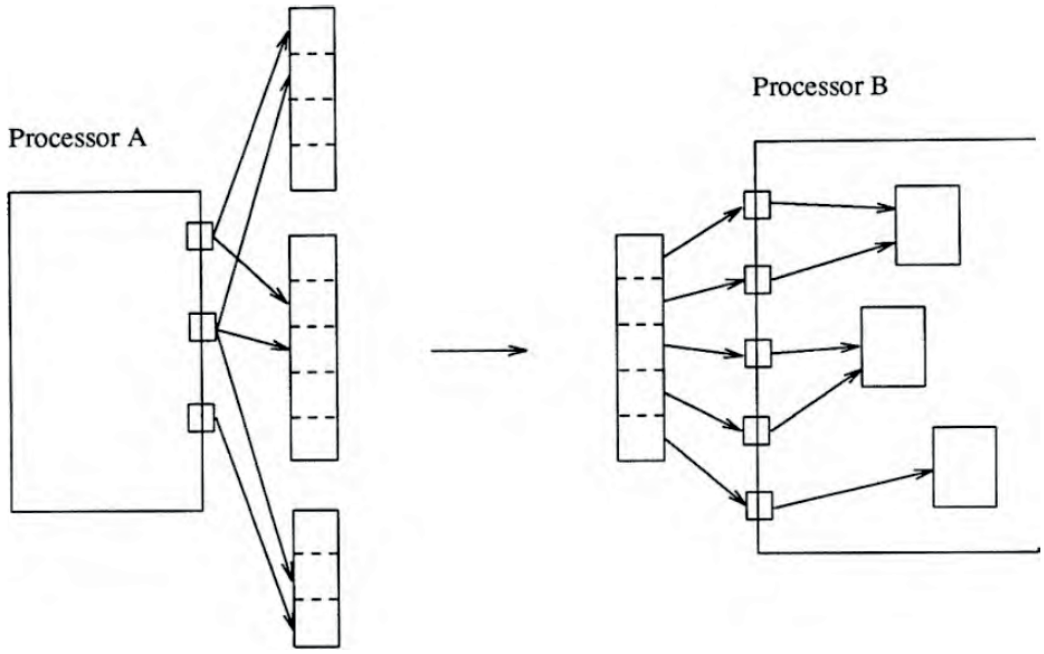


Figure 3.5. Buffered data transfer between processors.

Care must be taken to avoid I/O bottlenecks. It is unacceptable to let individual processors perform file access to read and write inputs and outputs respectively or to have a master process distribute and collect input/output data every clock cycle. Instead, one installs a test-case driver data structure which is partitioned according to the circuit partitioning and resides on each processor. A test case is given as a list of commands (typically stored in a separate file for batch mode operation) using the following primitive instructions: put-net name value (for setting the electrical net name to value), get-net name (for retrieving and later printing the electrical net name), run number of cycles (to execute the specified number of cycles),, and end (to terminate the test-case. For example, if the circuit to be simulated is a counter, the following test case lets the counter count up to 10 and checks the outputs (all internal nets are initialized to zero):

```
put-net carry_in 0;
run 10 ;
get-net count0 ;
get-net count1 ;
get-net count2 ;
get-net count3 ;
end ;
```

In a preprocessing step the test case is translated into a linked list data structure; each list element has a time stamp and specifies the set of put-net and get-net commands to be performed. Since only input changes are stored in the first, the stimuli to the circuit can be represented in a compact form. During simulation, a pointer to the current list element is kept. The current time is checked against the element's time stamp; in case they are equal, the instructions stored with the element are performed, else simulation cycles are run until the time has advanced to the time stamp or the end is reached. To avoid a bottleneck, each processor in a parallel machine owns its own linked list structure, which is constructed in a preprocessing step based on the circuit partitioning.

3.2.4 Fault Simulation

The parallel fault simulation differs from the straightforward data parallel approach where the faults are partitioned and each processor simulates a copy of the design for the same input data. We use a standard concurrent algorithm and the single stuck-at fault model. Lists of fault effects are maintained for each primitive in the circuit. The send gates introduced above collect all the faults visible on the boundary of a sub-circuit. If the send gate corresponds to a primary output, the faults are detected; otherwise, the faults need to be sent to the connected processors. The array of signal values exchanged for good machine evaluation is replaced by an array of pairs containing both the signal values and the number of faults visible at a particular pin. To avoid allocating worst case buffer space for each pin, one large buffer for all the output channels of a processor is provided and partitioned dynamically.

Then, for transmitting fault information three steps are performed:

- Count faults associated with each send gate;
- Partition buffer for fault information;

- Pack fault information into buffer.

Correspondingly, on the receiver side, fault lists at the receive gates are built, thereby clearing the receiving buffers, and the faults are propagated. If we assume that no faults are present on the clock lines, fault simulation can be simplified in the following way: based on our partitioning strategy one groups together latches. Both the good and faulty machine output of the latch remain unchanged when the latch is disabled. Hence the fault information at some send gate needs to be processed only when the latch fanning into it is active.

For large circuits, each processor will own a sub-circuit of substantial size (on the order of several thousand gates). Due to the memory requirements of concurrent fault simulation, it will usually not be possible to simulate all faults in one run. Therefore, one fault simulates the circuit in several passes, each time injecting only a subset of the faults. Clearly, the choice of fault sets to be simulated in each pass will affect the load balance between the processors and determine the total throughput.

One notes that the event rate and hence the work load for a given set of faults is strongly data dependent. Thus, the problem of assigning fault sets to processors in an optimal fashion is difficult to formulate deterministically. A simple heuristic is to initially inject an equal number of faults into each sub-circuit and to choose the fault sites such that they are evenly distributed across the sub-circuit. Then, during simulation, the total number of events and the maximum number of fault list entries for the pass is recorded. For the next pass, the number of faults to be injected is adjusted according to the derivation in the number of events from the average number of events and according to the number of list entries used.

A fault array needs to be maintained to keep track of which faults have been detected and can therefore be dropped. It is the union (computed as the bitwise logic OR) of all fault arrays private to the processors; hence, it can be updated by a pairwise union operation.

3.2.5 Circuit Model Extension

An important extension to the circuit model is to include descriptors for memory blocks (so-called arrays). An array is defined by specifying its number of rows (the number of data words to be stored), its number of columns (the number of bits in each data word), and a number of access methods, called ports. A port can be of type write or read and is essentially

described by defining an enable line, a number of address lines, a number of data lines, and a priority value that determines the order of write and read operations. The array construct is very useful since it allows the space efficient simulation of memories that would be too large to be described in a "flat" manner as the interconnection of flip-flops.

In the case of designs containing arrays the duplication of arrays during partitioning should be avoided, both in order not to waste memory space and not to duplicate the potentially large amount of logic controlling the array. We define an array cluster to be the union of an array and the so-called fan-out set of all its output data lines. The fan-out set of an output data line is found by tracing all paths starting at the data line forward until a latch is encountered; the fan-out set then consists of all zero delay elements on the paths and the terminating latches. Often some of the arrays in a design far exceed all other arrays in size, thus leading to highly unbalanced clusters. This problem can be alleviated by dividing the array into smaller sub arrays, each containing a subset of the data bits.

Let a cone cluster be a set of latches and their zero delay fan-in cones. If all latches of the circuit lie in one type of cluster, we can describe the circuit as a cluster graph that is defined in analogy to the cone graph: the clusters are the vertices and there is an edge between vertices if there is a connection in the circuit between elements in the clusters.

Then the partitioning takes place in two steps:

- Clustering step: for all arrays in the circuit compute the array clusters; for all latches that do not lie in any array cluster compute the fan-in cone; collect a chosen number of latches with their fan-in cones into a cone cluster.
- Assignment step: assign the clusters to processors.

For efficiency we again use the greedy heuristic described above for the assignment. Note that the partitioning is again performed not on the actual circuit graph but on the derived cluster graph. This graph will typically be an order of magnitude smaller; furthermore we can control its size by choosing the number of cones per cone cluster and by merging connected clusters.

3.3 PARALLEL SWITCH-LEVEL SIMULATION

Switch-level simulation is widely used in the design verification process of Very Large Scale Integrated (VLSI) MOS circuits. Our target machines are medium-grain multiprocessors (shared memory or message passing machines) and we only consider model parallel computation, where the model of the design to be simulated is partitioned among processors. We introduce efficient strategies for circuit partitioning as well as the corresponding simulation algorithms. In our approach, we try to minimize the total number of synchronizations between processors, as well as ensure portability and

scalability. A preprocessor and simulator were implemented and good performance was obtained for a set of benchmarks. The problem of tight coupling between processors that evaluate a strongly connected component in the circuit in a distributed fashion is highlighted.

3.3.1 Mapping Switch-level Simulation to a Parallel Framework

Preprocessing

In the preprocessing steps, a three-level hierarchy is found in the original "flat" transistor net list and taken as the starting point for the assignment to processors. First find the dc- connected components (DCC) (also called groups) in the circuit. On the groups, we define a new graph, the so-called group graph that has the groups as its vertices and an edge between two vertices if the corresponding groups are connected in the circuit. In the group graph, we find the strongly connected components (SCC) using a standard procedure. On the SCO's, we again define a new graph, the SCC-graph, by taking the SCC's as vertices and connecting two vertices with an edge if they contain groups that are connected in the group graph. By definition, the SCC-graph is acyclic and we can therefore levelize it uniquely using topological sort. The main objective of the preprocessing is to extract structural information from the circuit description that is both useful for general simulation efficiency and for parallel execution.

Identifying Parallelism

To identify potential parallelism we view the SCC-graph as an acyclic task graph: each SCC constitutes a task and all the tasks form a "computation rectangle" as illustrated in Figure 3.6. The width of the rectangle represents one simulation cycle that is made up of a number of sub-cycles, given by the levels (also called ranks) in the SCC-graph. The height-to width ratio of the rectangle depends not only on the topology of the circuit but also on the delay model used: using a zero-delay model (the outputs of the DCC's immediately reflect a change on the inputs) one will obtain a wide rectangle with many ranks. In the other extreme case of assigning a unit delay to each DCC (that is, the outputs of a DCC are updated to their new value only after a full simulation cycle) the rectangle is tall and only one rank wide. We note furthermore that the number of SCC's per rank can vary by a large amount and that the SCC's themselves can be of very different size.

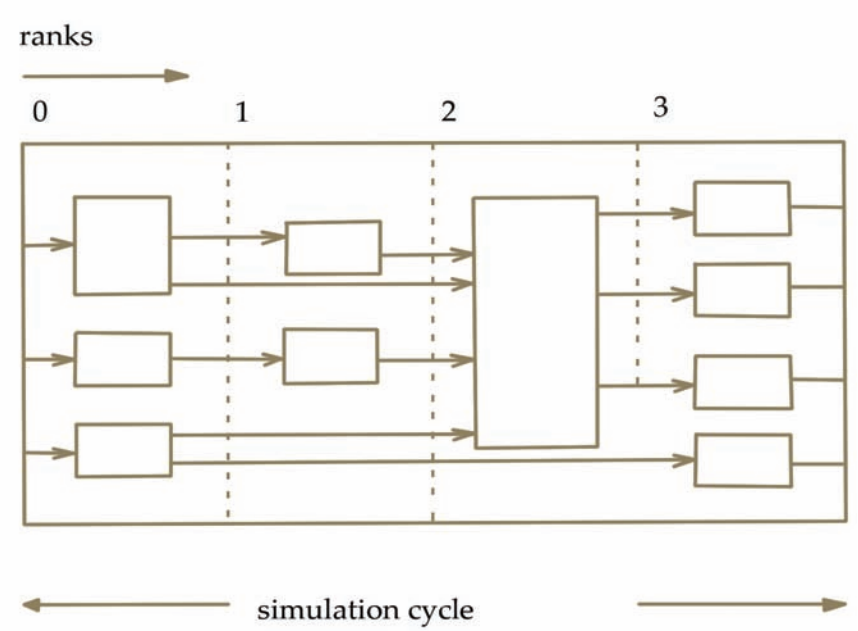


Figure 3.6. Computation rectangle.

The problem exhibits two kinds of parallelism: (i) SCC's in the same rank can be processed in parallel since there is no data dependence between them, (ii) SCC's in different ranks can be processed in parallel for different time frames: an SCC at rank n at time 11 can be computed in parallel with an SCC at rank r_2 at time t_2 if r\ < r_2 and $t_1 > f_2$. In other words, the computation can be organized in a pipeline fashion.

A Simple Mapping

To illustrate how the choice of a partitioning scheme influences the way synchronization is organized in the simulation algorithm, we first look at a very simple approach (outlined in Figure 3.7) in which complete SCC's are assigned to processors by ranks.

The assignment procedure in Figure 3.7 essentially produces a **pipeline** arrangement where several processors can be assigned to the same rank. An approximate static load balance is achieved by assigning roughly the same number of primitives to each processor (the size function counts the number of primitives in an SCC or processor respectively); in Figure 3.7, MIN_SIZE and MAX_SIZE can be chosen in relation to the average number of primitives per part (average_size), for example as 0.8*average_size and 1.2*average_size respectively for a 20% deviation. Note that we also allow the actual number of processors used to deviate from the number of parts given initially.

```
assign_by_rank {

    average_size = total number of primitives/number of parts
    proc = 0 ;

    for all ranks r {
                        for all SCC's s in r {

                            if ( size ( proc ) > MIN_SIZE  &&
                               (size ( proc ) + size ( s )) > MAX_SIZE ) ++proc ;
                            assign s to proc ;
                            size ( proc ) += size ( s );
                            if ( size ( proc ) > average ) ++proc ;
                        }
    }
}
```

Figure 3.7. Assignment by Rank.

The assignment has the following important property: define a directed processor graph with the processors as vertices and an edge from processor A to B if B receives signals from A. Then the acyclic task graph is mapped onto an acyclic processor graph; this leads to a simulation algorithm with a simple synchronization scheme, as shown in Figure 3.8.

All the signal values that need to be transferred between two processors are transferred collectively in one step (either one message or one access to a locked data structure). Note that each processor synchronizes with other processors which send signals to it or receive signals from it only once per simulation cycle. This is due to the fact that the processor graph is acyclic.

Pipeline is a set of data processing elements connected in series, where the output of one element is the input of the next one.

Splitting SCC's

The simple assignment approach shown is practical only if the SCC's in the circuit are not too unbalanced in size. This will be the case for designs based on logic gates or small standard cells, but not for general custom designs, in which single SCC's makeup substantial portions (20-60%) of the circuit. For the general case it is necessary to split up large SCC's. Then the assignment task is performed with the following two-step procedure:

1) For all SCC's: if size (scc) > [K (average size of a part)] - split the SCC into at most K parts (the so-called partial SCC's).
2) Assign the SCC's and partial SCC's to processors by rank as above, such that all partial SCC's of an SCC are assigned to different processors.

```
simulate_subcircuit {

    for all time steps {

                        read primary inputs;
                        receive signals on boundary from other processors;
                        for all ranks on this processor{
                            for all SCC's:evaluate_SCC;
                        }
                        send signals on boundary to other processors;
    }
} /* simulate_subcircuit */

evaluate_SCC {
    while groups to evaluate {
                        evaluate group;
                        update fanout of group internal to SCC
                        and schedule events;
    }
    update fanout to other SCC's ;
} /* evaluate_SCC */
```

Figure 3.8. Simulation Algorithm.

Note that the problem of partitioning the original circuit graph (consisting of electrical nodes as vertices and the transistors as edges) has been reduced to the problem of partitioning a number of SCC's, which is smaller in size. Each SCC to be split constitutes a sub graph of the group graph induced by the groups in the SCC. The problem of splitting an SCC can be formulated as an instance of the general graph partitioning problem: the sizes of the groups are assigned as weights to their corresponding vertices; then an optimal partition has a minimum number of cut edges and approximately the same sum of weights in each processor.

Balancing the size of the parts appears to be a good approximation to balancing the load on the processors. However, a small cut set does not guarantee an efficient distributed computation since a large number of iterations may still be needed between processors before the steady state is reached. Unfortunately, the amount of coupling between processors is data dependent and hard to predict. Thus, there is no simple

cost function for the partitioning problem. As a first approach, we consider a fan-out-based heuristic similar to, in which "strings" of groups are computed in a depth first search manner. This is outlined in Figure 3.9.

```
split_SCC (scc) {

    average_size = size (scc)/number of parts ;
    pick starting group g ;

    for all parts part_scc of scc {
                        while ( (size ( part_scc ) < average_size) &&
                             still unassigned groups left ) {
                            if ( size ( part_scc ) > MIN_SIZE  &&
                              (size ( part_scc ) + size ( g )) > MAX_SIZE ) break ;
                            add g to part_scc;
                            size ( part_scc ) += size ( g ) ;
                            if g has fanout:
                            g = first fanout of g ;
                            else: pick new unassigned starting group g;
    }

} /* split_SCC */
```

Figure 3.9. Compute parts of SCC.

3.3.2 Simulation Algorithm

The more general assignment scheme requires a new simulation algorithm that has a different evaluation and synchronization scheme.

Distributed Evaluation of SCC's

As mentioned earlier, finding the steady state of a set of DCC's that are connected as an SCC is an iterative process. If the complete SCC resides on one processor, this is achieved simply by iterating on the local list of events until the list is empty. In the distributed case each processor owns a partial SCC and the steady state of the whole SCC is reached when the following two conditions are fulfilled for all processors: (1) the processor's partial SCC is in a steady state; (2) no change of signal values occurred on the nets that lie on the boundary of the processor's partial SCC. The simulation algorithm for evaluating a partial SCC is shown in Figure 3.10.

```
evaluate_partial_SCC {

    while ( global_change ) {
                    while ( groups to evaluate in partial SCC ) {
                        evaluate groups ;
                        schedule internal fanout ;
                    }
                    global_change = TRUE, if change occurred on
                        boundary of any processor;
    }

} /* evaluate_partial_SCC */
```

Figure 3.10. Distributed evaluation of an SCC.

The global_change flag needs to be computed by all processors jointly, since it represents a distributed convergence check. Again here care has to be taken to ensure portability and scalability.

A global variable in shared memory is not a scalable solution since it would introduce a bottleneck. Instead, we view the global convergence check as a type of global sum problem, where each processor computes the logical OR of all local change flags.

A straightforward approach is to map the K processors that own parts of a split SCC onto a hypercube of dimension d = [log K]. If K is a power of two we obtain the distributed convergence check outlined in Figure 3.11.

```
find_global_change {
    change = TRUE, if any change on the boundary of this processor;
                    else FALSE ;
    for all dimensions, i = 0, ..., d-1 {
                    change = change OR change_flag on processor in dimension i;
    }
} /* find_global_change */
```

Figure 3.11. Distributed convergence check.

The exchange of local flags between a pair of processors is performed either by reading/writing from/to a shared monitored buffer or by message passing. log K simultaneous synchronizations are needed, instead of 2K serial ones when

using a global shared flag. Since K is not necessarily a power of two in our partitioning scheme, the d dimensional cube is incomplete. More precisely, if we assign processors to cube vertices in ascending order of the vertex labels (using the standard labeling for hyper cubes), we obtain a complete and an incomplete d-1 dimensional cube, where each unoccupied vertex v in the incomplete cube is connected to the corresponding occupied vertex w in the complete cube. Thus, we can map all buffering **data structures** need at v onto w in order to complete the cube using only the K processors. Then the scheme in Figure 3.11 will also work for the case that K is not a power of two.

Data structure is a particular way of organizing data in a computer so that it can be used efficiently

Synchronization

We observe that the distributed evaluation of SCC's introduces cyclic dependence into the processor graph. As a consequence the simple synchronization can lead to a deadlock in a situation of the type illustrated in Figure 3.12. Processors P1 and P2 jointly compute parts A and B of a split SCC at rank i. Processor P1 also owns an SCC at rank i+1 that depends on processor P2's output. The deadlock occurs because P 2 cannot deliver its output to the SCC on rank i+1 before completing iterating with P1 on rank i; P 1 cannot start the iteration because it is waiting for inputs from P 2.

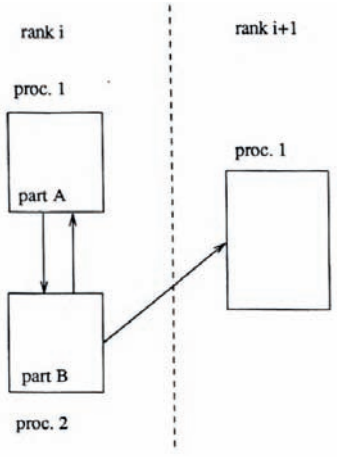


Figure 3.12. Deadlock situation.

The deadlock situation is avoided by introducing a rank by rank synchronization as shown in Figure 3.13.

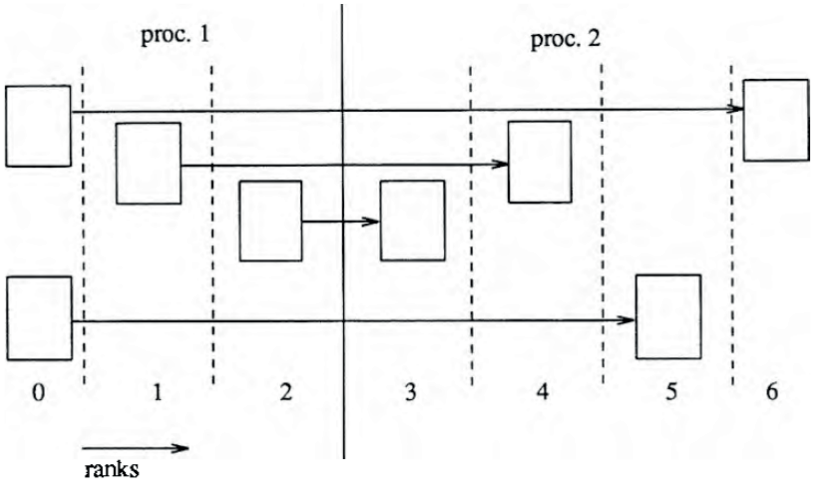
```
simulate_subcircuit {

    for all time steps {

                        read primary inputs;
                        for all ranks on this processor{
                            receive signals on boundary from other processors;
                            for all SCC's:evaluate_SCC;
                            send signals on boundary to other processors;
                        }
    }
} /* simulate_subcircuit */
```

Figure 3.13. General Simulation Algorithm.

Clearly, many additional synchronized connections between processors (also called channels here) are introduced. However, many of them can be eliminated by compressing several connections into one. This is illustrated in Figure 3.14. In the figure, the boxes represent SCC's and the arrows stand for channels along which a number of signals is transferred.



(a)

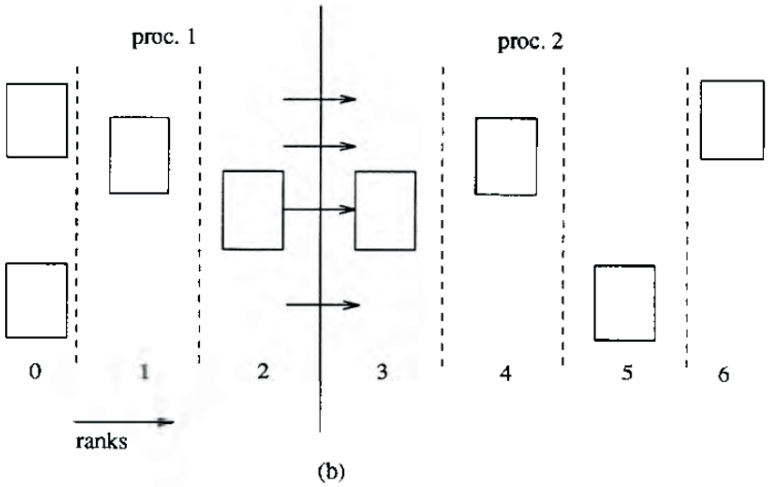


Figure 3.14. Compressing connections.

Figure 3.14 (a) shows the original connections between two processors; we observe that it is not necessary to deliver the signals from the source ranks 0 and 1 to the respective destination ranks 4, 5, and 6 immediately. Instead, all signals can be transferred after the computation at rank 2 is finished; as shown in Figure 3.14 (b), all four channels from Figure 3.14 (a) can be compressed into one channel, corresponding to one synchronization.

Here, the functions source_rank (channel) and dest_rank (channel) return the source and destination rank of the channel respectively.

3.4 MULTILEVEL SIMULATION: PREPROCESSING AND SIMULATION

The multi-level simulation technique presented herein analyzes circuits composed of blocks described at different levels of abstraction by using different simulation methods (circuit, functional or behavioral simulation) for each block of the circuit according to its abstraction level. For the analysis of circuit- and functional-level blocks (circuit and functional simulation), standard simulation techniques such as built-in models, Modified Nodal Analysis formulation and the Newton-Raphson nonlinear solution algorithm are used. However, for the analysis of behavioral-level blocks (behavioral simulation), an analysis methodology other than that of conventional simulation programs, such as SPICE and ASTAP, has to be used since the behavioral model of the blocks is not built in the simulation program, but is defined by the user in the form of 'C' language functions. This methodology includes a hybrid linearization technique,

which selects either the finite difference method or a modified Broyden's method according to predefined conditions, and a new behavioral model representation technique.

3.4.1 Switch-level and Multi-level Simulation

REMEMBER

The op-amp behavioral model developed in this dissertation is a generic model. All the parameters of the behavioral model are obtained from the electric performance characteristics of the op-amp being modeled. They are independent of the load conditions of the op-amp, which is a very convenient feature for system-level design.

Switch-level simulation, first introduced by Bryant, has become a widely used method in the design verification process of Very Large Scale Integrated (VLSI) MOS circuits. This is due to the fact that a gate-level circuit representation can be the inappropriate level of abstraction for MOS designs: circuits built in MOS technology typically use pass transistors and ratioed logic; consequently they exhibit bidirectional signal flow and charge sharing effects. Gate-level models for such phenomena tend to be cumbersome or inaccurate. Furthermore, they do not correspond to the physical implementation of the design. This modeling problem is particularly serious for the case of testing and fault simulation: gate-level models may not capture faults in the circuit and the gate-level stuck-at fault model does not represent failures in MOS circuits adequately.

Switch-level simulation, in contrast, can model electrical circuit behavior, such as bidirectional signal flow, charge storage and sharing, and resolution of conflicting signals. Here, a circuit is described as a netlist of pMOS and nMOS transistors. A transistor is modeled as a device with three nodes (source, drain, and gate). All transistors act as voltage-controlled switches which can be in one of three states: on, off, and undefined (on or off or intermediate). **Transistors** are assumed to have finite resistance between source and drain in the on-state and infinite resistance in the off -state. The resistance is modeled by assigning a discrete strength value to each transistor. The nodes of the circuit may assume one of three logic states: high (1), low (0), or undefined (X). Node capacitance is approximated with a discrete size value.

An nMOS (pMOS) transistor is on (off) when its gate node is high (low), off (on) when its gate node is low, and undefined when its gate node is undefined. All transistors are bidirectional as far a signal flow between source and drain is concerned, but there is no signal flow from either source or drain to the gate. The logic state of a node in the circuit is determined by all the paths reaching it from primary

inputs, power nodes, and other internal nodes. The steady state of the complete network is found either by computing the paths or by evaluating symbolic Boolean expressions that are found for the so-called dc- connected components (DCC) in the circuit. The DCC's are the maximal sets of transistors connected through their sources and drains.

Besides its powerful modeling capability, one of the main advantages of a switch- level representation is the close correspondence between model and physical design. Hence one application of switch-level simulation is to simulate a circuit description that is extracted directly from a layout and that includes detailed electrical information such as transistor and node capacitances. At the same time, due to its high level of detail the switch-level simulation of large designs (above 10,000 transistors) is time consuming and has substantial memory requirements (for example compared to gate-level simulation). For accelerated simulation one strategy is to simulate only part of the design at the most detailed level and replace other parts by a higher-level behavioral model that can be evaluated faster possibly at the cost of sacrificing some accuracy. A higher-level description can be generated either as a by-product of a top-down design approach or extracted automatically from a transistor netlist. As a result, circuit descriptions tend to be more heterogeneous and the process of creating efficient internal representations form the external ones is more involved.

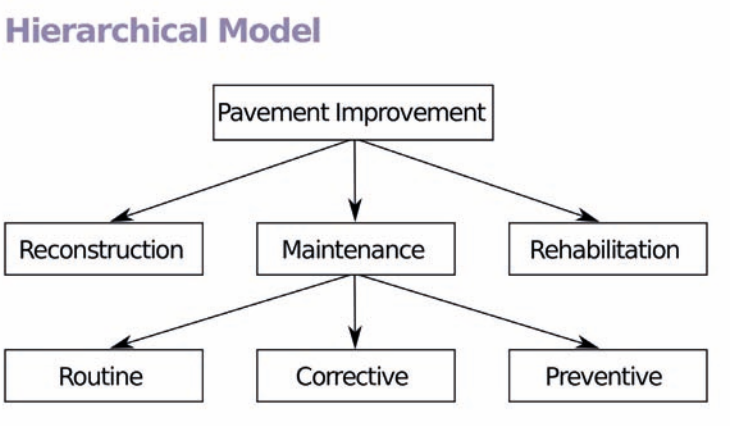
Transistor is a semiconductor device used to amplify or switch electronic signals and electrical power.

3.4.2 Hierarchical Design Description and Data Structures

As VLSI designs become larger and more complex hierarchical specifications are in use increasingly because they are more compact and more readable. Structural and behavioral hierarchy arises naturally in a structured, top-down design methodology and when using a high level description language such as VHDL. State of the art circuit extractors will also generate a hierarchical format.

The main feature of a hierarchical specification is that the circuit is described as collection of smaller less complex sub-circuits which are in turn made up of other sub-circuits or circuit primitives (such as transistors, gates, etc.). This

contrasts with a 'flat' description where a circuit is given as the net list of primitives.



DID YOU KNOW?

VHDL was originally developed at the behest of the U.S Department of Defense in order to document the behavior of the ASICs that supplier companies were including in equipment.

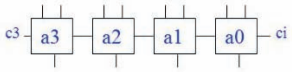
The hierarchical description requires less storage than the flat description since the interconnections of a repeatedly used substructure are saved only once and can be referenced. With memory requirements becoming a serious concern in the computer aided design of very large systems hierarchical formats are a necessity. The hierarchical description also has the advantage of preserving the structure inherent in a design. It is thus easier to understand and to manipulate, for example to modify incrementally.

Here, we consider design descriptions that exhibit structural hierarchy and that are composed of two types of building blocks: primitives and composite blocks. A typical list of primitives used here contains depletion mode, n-type, and p-type transistors and macros. The transistors will in general be bidirectional but can also be defined to be unidirectional pass transistors if information about signal flow is available. The macros can be simple logic gates described as lookup tables or more complex behavioral descriptions. Composite blocks are described as the interconnection of primitives and/or other composite blocks.

Examples

- 4-bit adder

```
module add4 (s,c3,ci,a,b)
    input [3:0] a,b ;  // port declarations
    input ci ;
    output [3:0] s :   // vector
    output c3 ;
    wire [2:0] co ;
        add a0 (co[0], s[0], a[0], b[0], ci) ;
        add a1 (co[1], s[1], a[1], b[1], co[0]) ;
        add a2 (co[2], s[2], a[2], b[2], co[1]) ;
        add a3 (c3, s[3], a[3], b[3], co[2]) ;
endmodule
```



Adder/Subtractor

```
// 4-bit full-adder (behavioral)

module fa(sum, co, a, b, ci) ;
  input   [3:0] a, b ;
  input   ci ;
  output  [3:0] sum ;
  output  co ;

  assign {co, sum} = a + (ci ? ~b : b) + ci ;

endmodule
```

Figure 3.15. 4-bit adder.

Intuitively, the hierarchical description of the circuit can be thought of as a tree where each vertex stands for a design entity and the descendants or children constitute its building blocks, as illustrated in Figure 3.16. The root of the tree is also referred to as top level. The leaves of the tree correspond to the primitive building blocks of the circuits.

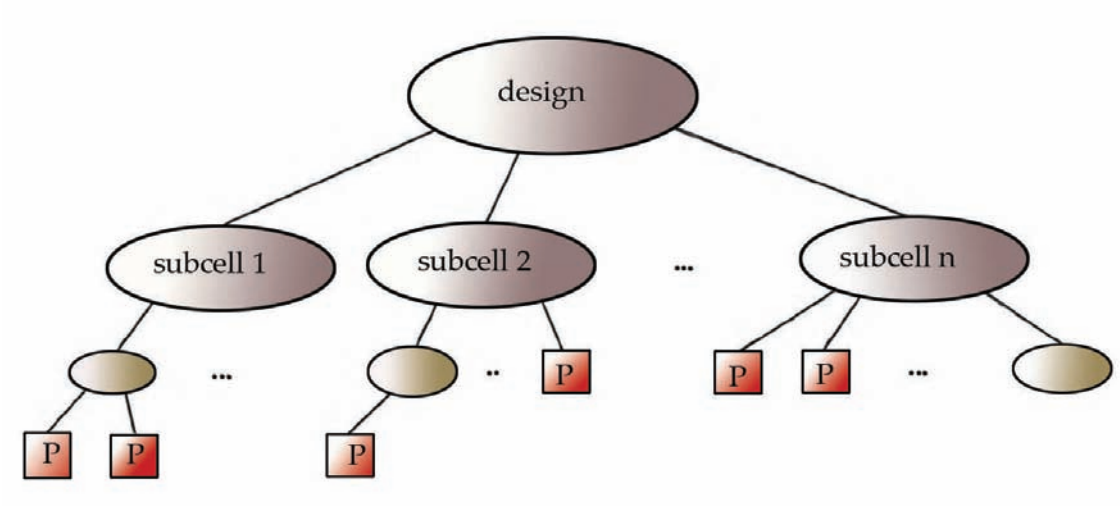


Figure 3.16. Tree structure of hierarchical description.

A more efficient approach is to organize the description into a multi graph as shown in Figure 3.17. Here, instead of replicating the structure of identical blocks, we use a single representative that is shared by all the identical subcells. Clearly, any vertex beside the root may now have more than one 'parent'. However, uniqueness is preserved through the different paths from the root to a particular subcell.

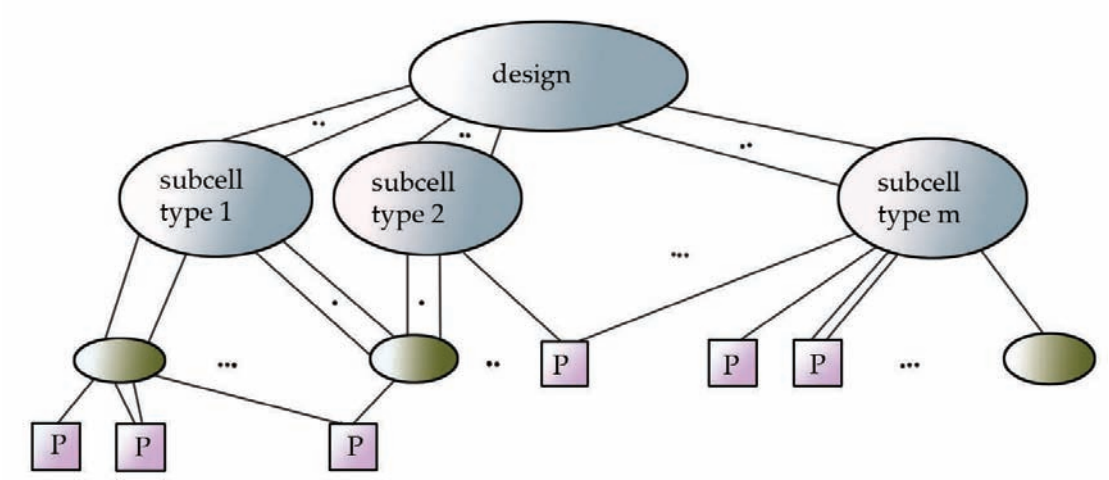
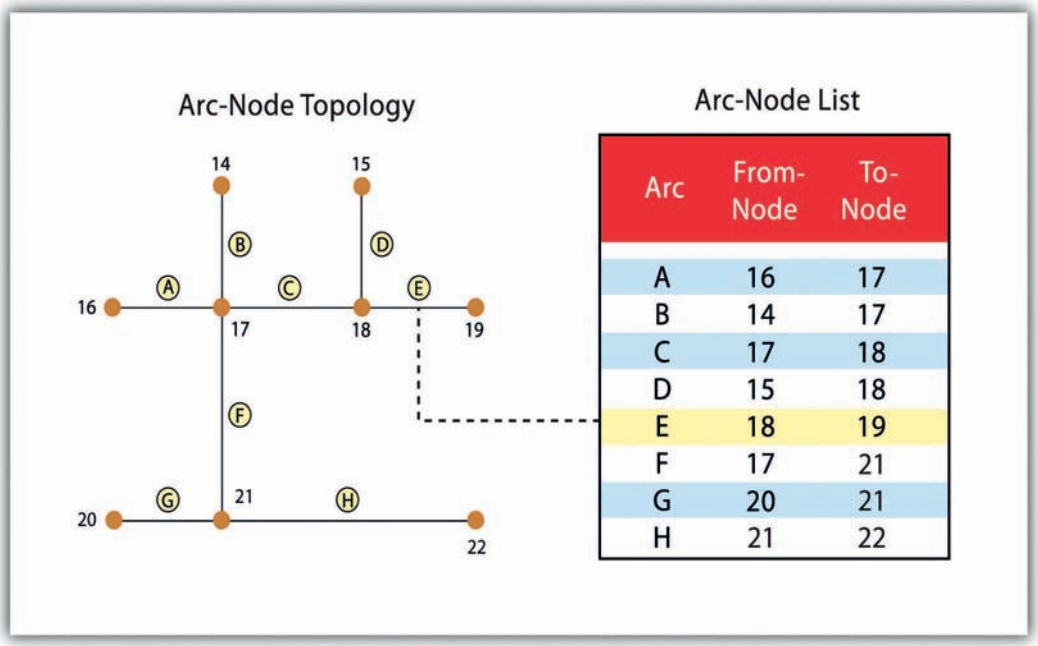


Figure 3.17. Multigraph structure of hierarchical description.

The cell construct contains a node data structure; both are outlined in Figure 3.18. In cell, pin_count gives the number of connections on the boundary of the cell while node_count indicates the number of nodes inside the cell. If the cell is of composite type, subcell_count gives the number of subcells it contains. For primitives the primitive_descriptor specifies the type of primitive and other information such as and index into a lookup table.

A node can be of type INPUT, OUTPUT, or IOPUT (for bidirectional signal flow). Fanin_count and fan-out count respectively give the number of subcells on the fan-in and fan-out lists of a node. The actual fan-in and fan-out lists are given as indices to the respective subcells.



Arc	From-Node	To-Node
A	16	17
B	14	17
C	17	18
D	15	18
E	18	19
F	17	21
G	20	21
H	21	22

Figure 3.18. Data structures cell and node.

The instance data structure is shown Figure 3.19. It contains a reference to its topology description cell. The instance pointers parent and children form the link to the next level of hierarchy up and down respectively. Inodes is a vector of structure inst_node; there is one inst_node per pin in the boundary of inst. Each inst_node contains a field uplink (an index to the node in the parent instance this node is connected to) and a field enode.

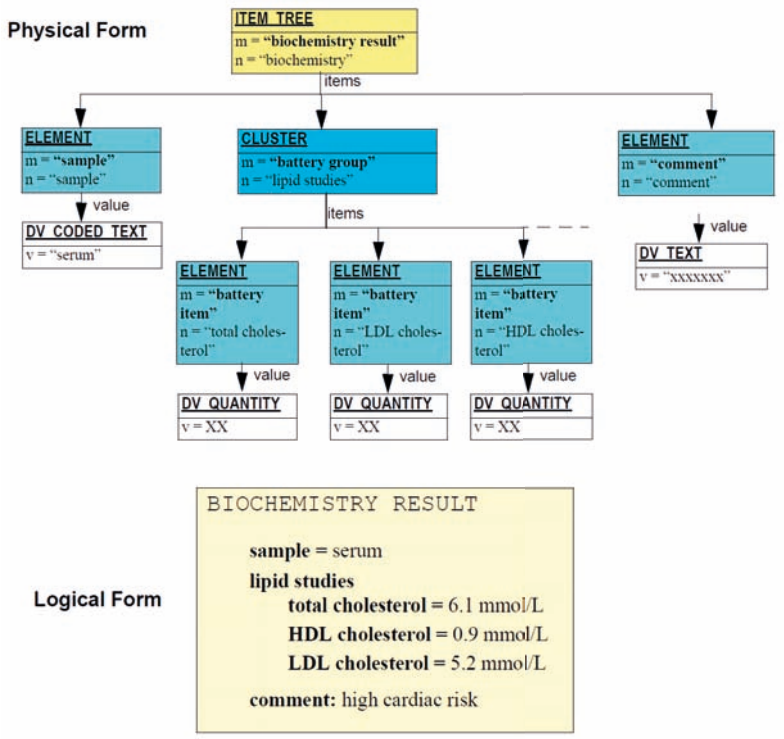


Figure 3.19. Instance and instance node structure.

Note that both the structures node and inst_node do not correspond Jo actual electrical nodes. In fact a whole set of nodes may map into one single electrical node.

ROLE MODEL

JACK KILBY: *PRESENTED HIS IDEA TO HIS SUPERIORS, AND WAS ALLOWED TO BUILD A TEST VERSION OF INTEGRATED CIRCUIT.*

Biography

Jack St. Clair Kilby (November 8, 1923 – June 20, 2005) was an American electrical engineer who took part (along with Robert Noyce) in the realization of the first integrated circuit while working at Texas Instruments (TI) in 1958. He was awarded the Nobel Prize in physics on December 10, 2000. To congratulate him, US President Bill Clinton wrote, "You can take pride in the knowledge that your work will help to improve lives for generations to come.

Education and Early Career

Kilby was the son of an electrical engineer and, like many inventors of his era, got his start in electronics with amateur radio. His interest began while he was in high school when the Kansas Power Company of Great Bend, Kansas, of which his father was president, had to rely on amateur radio operators for communications after an ice storm disrupted normal service. After serving as an electronics technician in the U.S. Army during World War II, Kilby enrolled in the electrical engineering program at the University of Illinois in Urbana-Champaign (B.S.E.E., 1947).

After graduation Kilby joined the Centralab Division of Globe Union, Inc., located in Milwaukee, Wisconsin, where he was placed in charge of designing and developing miniaturized electronic circuits. He also found time to continue his studies at the University of Wisconsin, Milwaukee Extension Division (M.S.E.E., 1950). In 1952 Centralab sent Kilby to Bell Laboratories' headquarters in Murray Hill, New Jersey, to learn about the transistor, which had been invented at Bell in 1947 and which Centralab had purchased a license to manufacture. Back at Centralab, Kilby began working on germanium-based transistors for use in hearing aids. He soon realized, however, that he needed the resources of a larger company to pursue the goal of miniaturizing circuits, and in 1958 he switched to another Bell licensee, Texas Instruments Incorporated of Dallas, Texas.

Career At Texas Instruments

Shortly after his arrival at Texas Instruments (TI), Kilby had his epoch-making "monolithic idea." Kilby realized that, instead of connecting separate components, an entire electronic assembly could be made as one unit from one semiconducting material by overlaying it with various impurities to replicate individual electronic components, such as resistors, capacitors, and transistors. Soon Kilby had a working postage-stamp-size prototype manufactured from germanium, and in February 1959 TI filed a patent application for this "miniaturized electronic circuit"—the world's first integrated circuit. Four months later, Robert Noyce of Fairchild Semiconductor Corporation filed a patent application for essentially the same device, but based on a different manufacturing procedure. Ten years later, long after their respective companies had cross-licensed technologies, the courts gave Kilby credit for the idea of the integrated circuit but gave Noyce the patent for his planar manufacturing process, a method for evaporating lines of conductive metal (the "wires") directly onto a silicon chip.

Although the original integrated circuit (IC) was Kilby's most important invention, it was only one of more than 50 patents that he was awarded. Many of those patents concerned improvements in IC design and manufacturing, including those for the first IC-powered experimental computer that TI built for the U.S. Air Force in 1961 and for the ICs that TI designed and delivered to the Air Force in 1962 for use in the Minuteman ballistic missile guidance system. In 1965 Kilby invented the semiconductor-based thermal printer. In 1967 he designed the first IC-based electronic calculator, the Pocketronic, gaining himself and TI the basic patent that lies at the heart of all pocket calculators. The Pocketronic required dozens of ICs, making it too complicated and expensive to manufacture for consumers, but by 1972 TI had reduced the number of necessary ICs to one. The introduction in that year of the TI Datamath pocket calculator marked the beginning of the IC's integration into the very fabric of everyday life. By 1976 the pocket calculator had made the slide rule a museum piece.

Honors and Awards

Kilby began a leave of absence from TI in 1970 to pursue independent research, particularly in solar power generation, although he continued as a semiconductor consultant on a part-time basis. He also served (1978–84) as a professor of electrical engineering at Texas A&M University in College Station. Among his many honours, Kilby was awarded the National Medal of Science in 1969, the Charles Stark Draper Medal in 1989, and the National Medal of Technology in 1990. In 1997 TI dedicated its new research and development building in Dallas, the Kilby Center. The Royal Swedish Academy of Sciences, breaking with a trend of recognizing only theoretical physicists, awarded half of the 2000 Nobel Prize for Physics to Kilby for his work as an applied physicist.

SUMMARY

- Simulation is more than a mere convenience it allows a designer to explore his circuit in ways which may be otherwise impractical or impossible.
- In fault simulation, one is interested in the behavior of a digital design in the presence of certain predefined faults.
- The parallel fault simulation differs from the straightforward data parallel approach where the faults are partitioned and each processor simulates a copy of the design for the same input data.
- The exchange of local flags between a pair of processors is performed either by reading/writing from/to a shared monitored buffer or by message passing.
- The multi-level simulation technique presented herein analyzes circuits composed of blocks described at different levels of abstraction by using different simulation methods (circuit, functional or behavioral simulation) for each block of the circuit according to its abstraction level.
- Structural and behavioral hierarchy arises naturally in a structured, top-down design methodology and when using a high level description language such as VHDL.

KNOWLEDGE CHECK

1. **In VLSI design, which process deals with the determination of resistance & capacitance of interconnections?**
   a. Floor planning
   b. Placement & Routing
   c. Testing
   d. Extraction
2. **Which among the following is a process of transforming design entry information of the circuit into a set of logic equations?**
   a. Simulation
   b. Optimization
   c. Synthesis
   d. Verification
3. **Among the VHDL features, which language statements are executed at the same time in parallel flow?**
   a. Concurrent
   b. Sequential

c. Net-list

d. Test-bench

4. Which data type in VHDL is non synthesizable & allows the designer to model the objects of dynamic nature?

a. Scalar

b. Access

c. Composite

d. File

5. In synthesis process, the load attribute specifies the existing amount of ..... load on a particular output signal.

a. Inductive

b. Resistive

c. Capacitive

d. All of the above

REVIEW QUESTIONS

1. What is logic and fault simulation?
2. Define the parallelism in simulation.
3. Discuss on logic simulation algorithm.
4. What is circuit model extension?
5. Differentiate between switch-level and multi-level simulation.

CHECK YOUR RESULT

1. (d) 2. (c) 3. (a) 4. (b) 5. (c)

REFERENCES

1. http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-304.pdf
2. https://www.ideals.illinois.edu/bitstream/handle/2142/88634/B55-UILU-ENG-90-2230-CRHC-90-2_opt.pdf?sequence=1
3. https://www.eecis.udel.edu/~elias/verilog/verilog_manuals/chap_6.pdf

CHAPTER

4

FLOORPLANNING AND PIN ASSIGNMENT

LEARNING OBJECTIVES

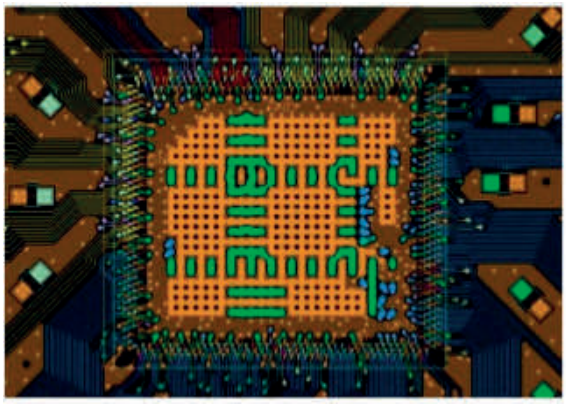
After studying this chapter, you will be able to:

1. Discuss about floorplanning
2. Explain the chip planning
3. Define pin assignment

INTRODUCTION

Floorplanning is the process of identifying structures that should be placed close together, and allocating space for them in such a manner as to meet the sometimes conflicting goals of available space (cost of the chip), required performance, and the desire to have everything close to everything else. After the circuit partitioning phase, the area occupied by each block (sub-circuit) can be estimated, possible shapes of the blocks can be ascertained and the number of terminals (pins) required by each block is known. In addition, the netlist specifying the connections between the blocks is also available. In order to complete the layout, we need to assign a specific shape to a block and arrange the blocks on the layout surface and interconnect their pins according to

the netlist. The arrangement of blocks is done in two phases; Floorplanning phase, which consists of planning and sizing of blocks and interconnect and the Placement phase, which assign a specific location to blocks. The inter-connection is completed in the routing phase. In the placement phase, blocks are positioned on a layout surface, in a such a fashion that no two blocks are overlapping and enough space is left on the layout surface to complete the inter connections. The blocks are positioned so as to minimize the total area of the layout. In addition, the locations of pins on each block are also determined.



The input to the Floorplanning phase is a set of blocks, the area of each block, possible shapes of each block and the number of terminals for each block and the netlist. If the layout of the circuit within a block has been completed then the dimensions (shape) of the block are also known. The blocks for which the dimensions are known are called fixed blocks and the blocks for which dimensions are yet to be determined are called flexible blocks. Thus we need to determine an appropriate shape for each block (if shape is not known), location of each block on the layout surface, and determine the locations of pins on the boundary of the blocks. The problem of assigning locations to fixed blocks on a layout surface is called the Placement problem. If some or all of the blocks are flexible then the problem is called the Floorplanning problem. Hence, the placement problem is a restricted version of the floorplanning problem. If one asks for planning of the interconnect in addition to floorplanning, then it is referred to as the chip planning problem. Thus floorplanning is a restricted version of chip planning problem. The terminology is slightly confusing as floorplanning problems are placement problems as well but these terminologies have been widely used and accepted. It is desirable that the pin locations are identified at the same time when the block locations are fixed. However, due to the complexity of the placement problem, the problem of identifying the pin locations for the blocks is solved after the locations of all the blocks are known. This process of identifying pin locations is called pin assignment.

Chip planning, Floorplanning and Placement phases are very crucial in overall physical design cycle. It is due to the fact, that an ill-floorplanned layout cannot be improved by high quality routing. In other words, the overall quality of the layout, in terms of area and performance is mainly determined in the chip planning, floorplanning and placement phases.

4.1 FLOORPLANNING

A floorplanning is the process of placing blocks/macros in the chip/core area, thereby determining the routing areas between them. Floorplan determines the size of die and creates wire tracks for placement of standard cells. It creates power ground (PG) connections. It also determines the I/O pin/pad placement information.

A good floorplanning should meet the following constrains.

- Minimize the total chip area,
- Make routing phase easy (routable),
- Improve the performance by reducing signal delays.

Floorplanning is the placement of flexible blocks, that is, blocks with fixed area but unknown dimensions. It is a much more difficult problem as compared to the placement problem. In floorplanning, several layout alternatives for each block are considered. Usually, the blocks are assumed to be rectangular and the lengths and widths of these blocks are determined in addition to their locations. The blocks are assigned dimensions by making use of the aspect ratios. The aspect ratio of a block is the ratio of the width of the block to its height. Usually, there is an upper and a lower bound on the aspect ratio a block

can have as the blocks cannot take shapes which are too long and very thin. Initial estimate on the set of feasible alternatives for a block can be made by statistical means, i.e., by estimating the expected area requirement of the block. Many techniques of general block placement have been adapted to floorplanning. The only difference between floorplanning and general block placement is the freedom of cells' interface characteristic. Like placement, inaccurate data partly affects floorplanning. In addition to the inaccuracy of the cost function that we optimize, the area requirements for the blocks may be inaccurate.

REMEMBER

A floorplanner should consider many factors, such as aspect ratio, rout ability, timing, packaging and pre-placed macros.

Floorplanning algorithms are typically used in hierarchical design. This is due to the fact that, although the dimensions of each leaf of the hierarchical tree may be known, the blocks at the node level in the tree are flexible, i.e., they can take any dimension. Hence, the floorplanning algorithms are used at each of the nodes in the tree so that the area of the layout is minimum and the position of all the blocks are identified.

There are several factors that are considered by the chip planning, floorplanning, pin assignment and placement algorithms. These factors are discussed below:

Shape of the blocks:

In order to simplify the problem, the blocks are assumed to be rectangular. The shapes resulting from the floorplanning algorithms are mostly rectangular for the same reason. The floorplanning algorithms use aspect ratios for determining the shape of a block. The aspect ratio of a block is the ratio between its height and its width. Usually there is an upper and a lower bound on the aspect ratios, restricting the dimensions that the block can have. More recently, other shapes such as L-shapes have been considered, however dealing with such shapes is computationally intensive.

Routing considerations:

In chip planning, it is required that routing is considered as an integral part of the problem. In placement and floor-planning algorithms it may be sufficient to estimate the area required for routing. The blocks are placed in a manner such that there is sufficient routing area between the blocks, so that routing

algorithms can complete the task of routing of nets between the blocks. If complete routing is not possible, placement phase has to be repeated.

Floorplanning and Placement for high performance circuits:

For high performance circuits the blocks are to be placed such that all critical nets can be routed within their timing budgets. In other words, the length of critical paths must be minimized. The floorplanning (placement) process for high performance circuits is also called as performance driven floorplanning (placement).

Packaging considerations:

All of these blocks generate heat when the circuit is operational. The heat dissipated should be uniform over the entire surface of the group of blocks placed by the placement algorithms. Hence, the chip planning, floorplanning and placement algorithms must place the blocks, which generate a large amount of heat, further apart from each other. This might conflict with the objective for high performance circuits and some tradeoff has to be made.

Pre-placed blocks:

In some cases, the locations of some of the blocks may be fixed, or a region may be specified for their placement. For example, in high performance chips, the clock buffer may have to be located in the center of the chip. This is done with the intention to reduce the time difference between arrival times of the clock signal at different blocks. In some cases, a designer may specify a region for a block, with in which the block must be placed.

4.1.1 Problem Formulation

The input consists of $B_1, B_2, \ldots, B_n$ circuit blocks, with area respectively. Associated with each block are two aspect ratios A_i^l and A_i^h which give the lower and the upper bound on the aspect ratio for that block. Thefloorplanning algorithm has to determine the width w_i and height, h_i of each block B_i such that $A_i^l \leq \frac{h_i}{w_i} \leq A_i^h$. In addition to finding the shapes of the blocks, the floorplanning algorithm has to generate a valid placement such that the area of the layout is minimized.

A slicing floorplan is a floorplan which can be obtained by recursively partitioning a rectangle into two parts either by a vertical line or a horizontal line. The cut tree obtained by min-cut algorithm is known as slicing tree. A slicing tree is a binary tree in which each leaf represents a partition and each internal node represents a cut. Consider the floorplan as shown in Figure 4.1. Partitions are labeled with letters and cut lines are labeled with numbers. Figure 4.1(b) shows the slicing tree for the floorplan

in Figure 4.1 (a). Figure 4.1(c) is the slicing tree indicating the cut direction. Figure 4.1(d) shows a floorplan for which there is no valid slicing tree.

A floorplan is said to be hierarchical of order k, if it can be obtained by recursively partitioning a rectangle into at most k parts. The hierarchy of a hierarchical floorplan can be represented by a floorplan tree. Figure 4.2 shows a hierarchical floorplan of order 5 and its floorplan tree. Each leaf in the tree corresponds to a basic rectangle and each internal node corresponds to a composite rectangle in the floorplan. An important class of hierarchical floorplans is the set of all slicing floorplans.

Design Style Specific Floorplanning Problems

Floorplanning is not carried out for some design styles. This is due to the fixed dimensions of blocks in some design styles.

- Full custom design style:

  Floorplanning for general cells is the same as discussed above.
- Standard cell design style:

  In standard cell design style, the dimensions of cells are fixed, and floorplanning problem is simply the placement problem. For large standard cell design, circuit is partitioned into several regions, which are floorplanned, before cells are placed in regions.
- Gate array design style:

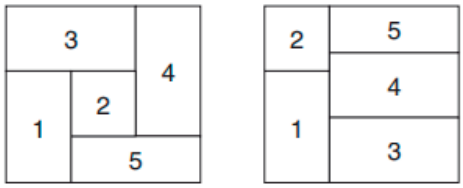
  Like standard cells, the floorplanning problem is same as placement problem.

4.1.2 Floorplanning Model

We can classify floorplans into two categories for discussions: (1) slicing floorplans and (2) non-slicing floorplans. A slicing floorplan can be obtained by repetitively cutting the floorplan horizontally or vertically, whereas a non-slicing floorplan cannot. The given dimension of each hard module must be kept. All modules are free of rotation; if a module is rotated, its width and height are exchanged.

Slicing Floorplans

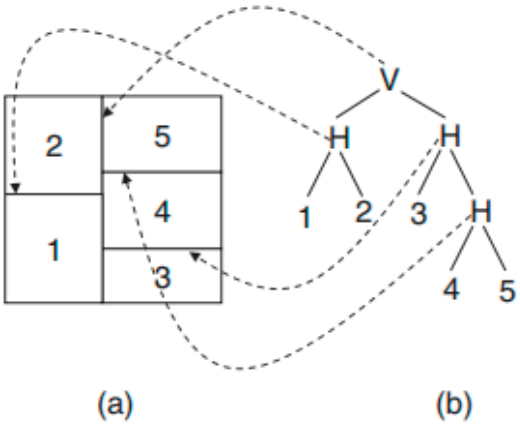
On the basis of the slicing property of a slicing floorplan, we can use a binary tree to represent a slicing floorplan. A slicing tree is a binary tree with modules at the leaves and cut types at the internal nodes. There are two cut types, H and V. The H cut divides the floorplan horizontally, and the left (right) child represents the bottom (top) sub-floorplan. Similarly, the V cut divides the floorplan vertically, and the left (right) child represents the left (right) sub-floorplan.



Note that a slicing floorplan may correspond to more than one slicing tree, because the order of the cut-line selections may be different. This representation duplication might incur a larger solution space and complicate the optimization process. Therefore, it is desirable to prune such redundancies to facilitate floorplan design. As such, we refer to a slicing tree as a skewed slicing tree if it does not contain a node of the same cut type as its right child.

Non-slicing Floorplans

The non-slicing floorplan is more general than the slicing floorplan. However, because of its non-slicing structure, we cannot use a slicing tree to model it. Instead, we can use a horizontal constraint graph (HCG) and a vertical constraint graph (VCG) to model a non-slicing floorplan. The horizontal constraint graph defines the horizontal relations of modules, and the vertical constraint graph defines the vertical ones. In a constraint graph, a node represents a module. If there is an edge from node A to node B in the HCG (VCG), then module A is at the left (bottom) of module B.



Floorplanning Cost

The goal of floorplanning is to optimize a predefined cost function, such as the area of

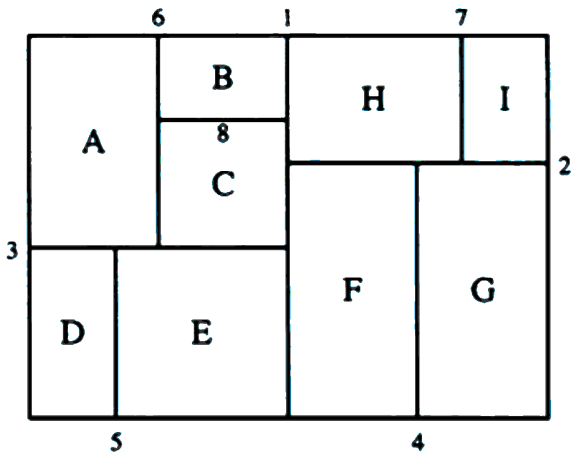
a resulting floorplan given by the minimum bounding rectangle of the floorplan region. The floorplan area directly correlates to the chip silicon cost. The larger the area, the higher the silicon cost. The space in the floorplan bounding rectangle uncovered by any module is called white space or dead space.

Other floorplanning cost, such as wire length, will also be considered. Shorter wire length not only can reduce signal delay but also can facilitate wire interconnection at the routing stage. The floorplanning objective can also be a combined cost, such as area plus wire length.

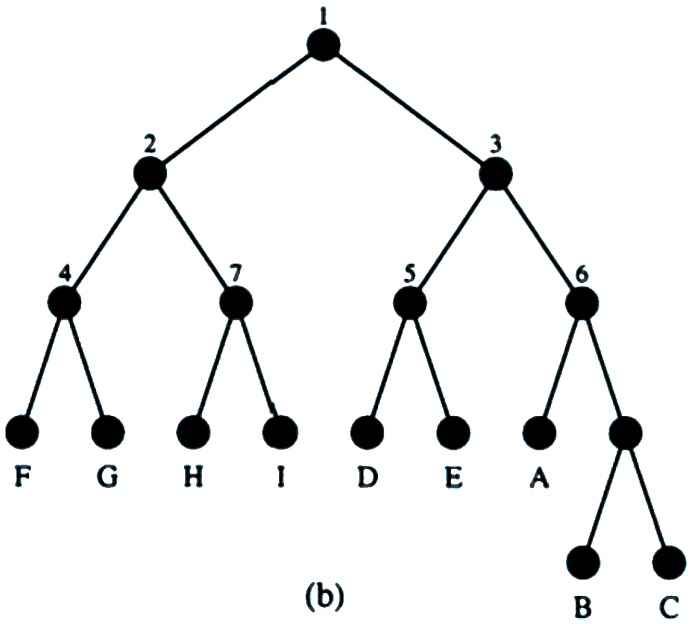
4.1.3 Classification of Floorplanning Algorithms

Floorplanning methods can be classified as follows:

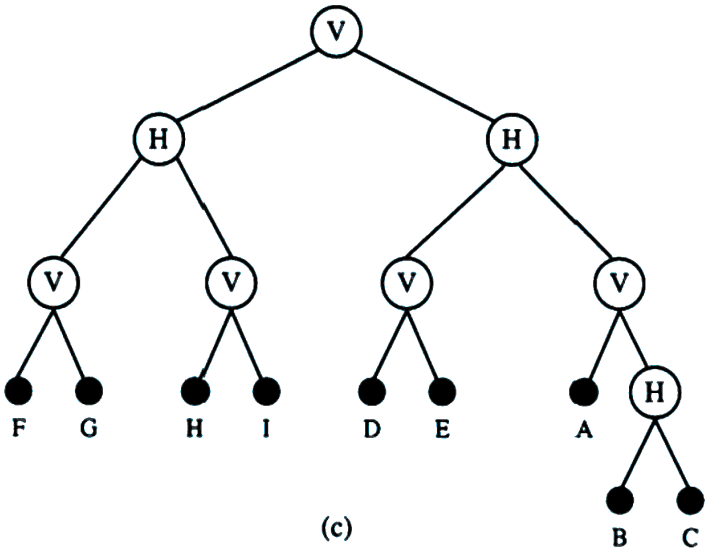
- Constraint based methods.
- Integer programming based methods.
- Rectangular dualization based methods.
- Hierarchical tree based methods.
- Simulated Evolution algorithms
- Timing Driven Floorplanning Algorithms



(a)



(b)



(c)

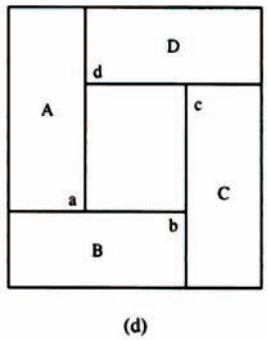
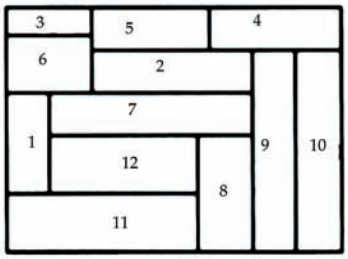


Figure 4.1: A floorplan with slicing tree and a non-slicing floorplan.



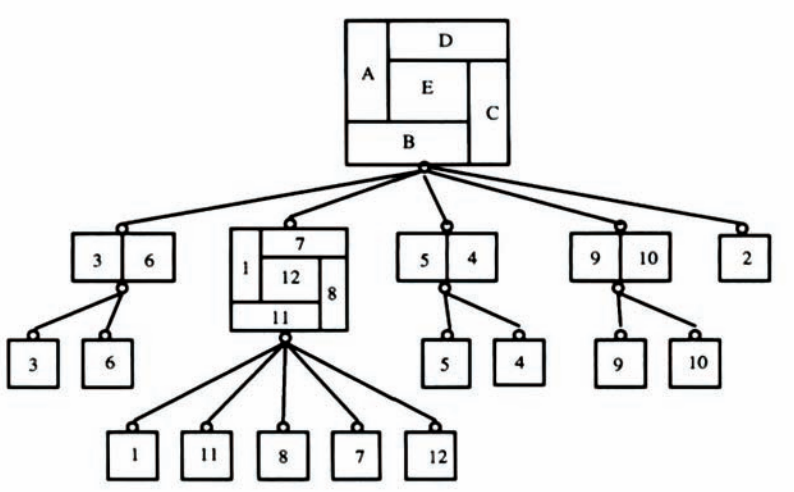


Figure 4.2: Hierarchical Floorplan.

Beside the methods stated above, several simple methods may be used, such as min-cut method. The process of min-cut can be used to construct a sized floorplan. The first phase of min-cut method i.e., bipartition of a weighted graph, helps in constructing the floorplan. The weight of the vertex roughly estimates the area taken up by the block. This weight may represent the area of the corresponding ceil in general-cell placement. The initial sized floorplan represents an empty base rectangle whose area is the total of all weights of the vertices of the weighted graph and each node in the tree represents a rectangular room in the layout area. All the floorplans that can be generated with min-cut bi-partitioning are slicing floorplans.

To make the bus routing easier, bus planning should be considered at the floorplanning stage, which is called bus-driven floorplanning.

4.1.4 Constraint Based Floorplanning

The constructs a floorplan of optimal area that satisfies (respects) a given set of constraints. A set of horizontal and vertical topological (i.e., ordering) constraints is derived from the relative placement of blocks. Given a constraint set, it is usually the case that there is no reason to satisfy all the constraints in the set. This is especially true when a majority of the blocks have flexible shapes. A floorplan is said to respect a constraint, if for each pair of blocks, the floorplan satisfies at least one constraint (horizontal or vertical). A constraint set is said to be overconstmined if it has many redundant constraints. It is desirable to derive a complete constraint set from the input relative placement and then to remove those redundant constraints that result in reduction of floorplan area.

A topological constraint set of a set of blocks is given by two directed acyclic graphs (G_H, G_V): G_H is the horizontal constraint graph and G_V is the vertical constraint graph. In order to reduce the floorplan area, the heuristic iterativelyremoves a redundant constraint from the critical path of either G_H or G_V and also iteratively reshapes the blocks on the critical paths of the two graphs. Critical path is the longest path in G_H or G_V. The input to the algorithm is a constraint set (G_H, G_V) of the set of blocks. To minimize the floorplan area, repeat steps 1 and 2 until there is no improvement in the floorplan area.

Step 1:

Repeat the following steps until no strongly redundant edges on P_H or P_v exist. G_H and G_V topologically sorted and swept. Either or

whichever is more critical is selected, where P_H and P_v are the critical paths of G_H and G_v respectively. The strongly redundant edge on the selected critical path is eliminated.

Step 2:

The current shapes of the blocks are stored and a path, either P_H or P_v is selected depending on which of the two is more critical. All theflexible blocks on the selected path are reshaped. G_H and G_v are scanned again to construct the new floorplan. If the newly generated floorplan is better than the previous one the stored block shapes are updated. All the steps described above are repeated, a specified number of times.

Each pass of the algorithm constitutes one execution of two steps. Constraint reduction takes place in step 1 and step 2 does the reshaping of the blocks. If the chip dimensions are fixed, the passes are repeated until the tar-get dimensions are reached. Otherwise the passes can be repeated until there is improvement in the floorplan area. Typically three or four passes are required.

The purpose of removing a redundant edge on the critical path is to break the path into two smaller paths. A good choice for such a redundant edge is the one which is nearest to the center point of the path. The above heuristic removes only one redundant constraint from a critical path at each iteration, and thus seeks to minimize the number of constraints removed. An edge can be checked for strong redundancy in constant time if we maintain the adjacency matrix G_H and G_V. It takes $O(n^2)$ time to set up adjacency matrices. A topological sort of a directed acyclic graph with n nodes and m edges takes O(m + n) time. The number of topological sorts executed depends on the number of redundant edges removed, the user-specified value for the number of reshaping iterations, and the number of passes.

4.1.5 Integer Programming Based Floorplanning

The floorplanning problem is modeled as a set of linear equations using 0 /1integer variables. Two types of constraints are considered: the overlap constraints and the rout ability constraints. The overlap constraints prevent any two blocks from overlapping whereas the rout ability constraints estimate the routing area required between the blocks. For the critical nets, net lengths are specified which should not be exceeded. The length of the net depends on the timing budget of that net. The critical net constraints ensure that the length of the critical nets does not exceed this specified value. We now describe how the constraints can be developed.

Block overlap constraints for fixed blocks:

Given two fixed (rigid) blocks, B_{ri} and B_{rj} which should not overlap, we have four

possible ways to position the two blocks so as to avoid overlap. Let $\{x_i, y_i, w_i, h_i\}$ and $\{x_j, y_j, w_j, h_j\}$ be the 4-tuples associated with blocks B_{ri} and B_{rj} respectively, where (x_i, y_i) gives the location of the block, h_i is the width of the block and hi is the height of the block. The block B_{rj} can be positioned to the right, left, above or below block B_{ri}. These conditions transformed into equations given below:

$$
\begin{aligned}
&x_i + w_i \le x_j \left(B_{rj} \text{is to the right of } B_{ri}\right), \text{ or}\\
&x_i - w_j \ge x_j \left(B_{rj} \text{is to the left of } B_{ri}\right), \text{ or}\\
&y_i + h_i \le y_j \left(B_{rj} \text{is to the above of } B_{ri}\right), \text{ or}\\
&y_i - h_j \le y_j \left(B_{rj} \text{is to the below of } B_{ri}\right)
\end{aligned}
\qquad (1)
$$

To satisfy one of these equations, two 0-1 integer variables x_{ij} and y_{ij} are used for each pair of blocks. Two bounding functions W and H are defined such that, $|x_i - x_j| \le W \text{ and } |y_i - y_j| \le H$. W can be equal to W_{max} which is the maximal allowed width of the chip or $W = \sum_{i=1}^{p+q+r} hi$. Similarly, H = H_{max}, the maximal allowed height of the chip or H = $H = \sum_{i=1}^{p+q+r} hi$. Equation set (1) can be rewritten with the introduction of the integer variables to generate the 'or' condition as,

$$
\begin{aligned}
&x_i + w_i \le x_j + W\left(x_{ij} + y_{ij}\right)\\
&x_i - w_j \ge x_j + W\left(1 - x_{ij} + y_{ij}\right)\\
&y_i + h_i \le y_j + w\left(1 + x_{ij} - y_{ij}\right)\\
&y_i - h_j \ge y_j + W\left(2 - x_{ij} - y_{ij}\right)
\end{aligned}
\qquad (2)
$$

As the integer variables x_{ij} and y_{ij} can take either 0 or 1 values, only one of the above equations in (2) will be active and other equations will be true depending on the value of x_{ij} and y_{ij}. For example, when x_{ij} and y_{ij} = 1, the first equation in (2) becomes active and all other equations are true.

$$
\begin{aligned}
&x_i \ge 0,\ y_i \ge 0\\
&x_i + w_i \le W,\\
&y^* \ge y_i + h_i
\end{aligned}
\qquad (3)
$$

Where y^* is the height to be minimized. To allow rotation of the blocks so as to optimize the solution, another integer variable z_i is used for each block. z_i is 0 when the block is in its initial orientation and 1 when the block is rotated by 90°. The constraints for the fixed blocks can be rewritten as:

$$
\begin{aligned}
&x_i + z_i h_i + \left(1 - z_i\right) w_i \le x_j + M\left(x_{ij} + y_{ij}\right)\\
&x_i - z_i h_i - \left(1 - z_j\right) w_j \ge x_j + M\left(1 - x_{ij} + y_{ij}\right)\\
&y_i + z_i w_i + \left(1 - z_i\right) h_i \le y_j + M\left(1 + x_{ij} - y_{ij}\right)\\
&y_i - z_i w_j - \left(1 - z_j\right) h_j \ge y_j + M\left(2 - x_{ij} - y_{ij}\right)
\end{aligned}
\qquad (4)
$$

Where, M = max (W, H). Constraints (3) are rewritten as:

$$x_i \geq 0,\ y_i \geq 0,$$
$$x_i + (1 - z_i) w_i + z_i h_i \leq W,$$
$$y^* \geq y_i + (1 - z_i) w_i + z_i h_i \qquad (5)$$

where y^* is the height to be minimized. The floorplanning problem, forfixed blocks without taking into consideration either routing areas or critical nets can be solved by finding the minimum y^* subject to constraints (4) and (5).

Block overlap constraints for flexible blocks:

So far we discussed about fixed blocks. We can now see how constraints for flexible blocks can be developed. The flexible blocks can take rectangular shapes within a limited aspect ratio range i.e. its width and height can be varied keeping the area fixed. The non-linear area relation is linearized about the point of maximum allowable width by applying the first two members of the Taylor series giving,

$$h_i = h_{io} + \Delta w_i \lambda_i$$

where,

$$h_{i0} = \frac{A_i}{w_{i\max}}$$
$$\lambda_i = \frac{A_i}{w_{i\max}^2},$$
$$\Delta w_i = w_{i\max} - w_i$$

where Δw_i is a continuous variable for block B_{fi}. The overlap constraintsfor a flexible block B_{fi} and a fixed block B_{rj} can be written as:

$$x_i + w_{i\max} - \Delta w_i \leq x_j, \quad (B_{fi} \text{ is to the right of } B_{ri}), \text{ or}$$
$$y_i + h_{io} + \Delta w_i \lambda_i \leq y_j, \quad (B_{fi} \text{ is above } B_{ri}), \quad \text{or}$$
$$x_i - w_j \geq x_j, \quad (B_{fi} \text{ is to the left of } B_{ri}) \text{ or}$$
$$y_i - h_j \geq y_j, \quad (B_{fj} \text{ is below } B_{ri}) \qquad (6)$$

Using two integer variables x_{ij} and y_{ij} per block pair as was done for fixed blocks, the 'or' condition between the equations can be satisfied.

The same set of equations can be extended to get overlap constraints be-tween

two flexible blocks. Using the same technique, the interconnection length constraints and routing area constraints can be developed. This set of equations are the input to any standard linear programming software package such as LINDO. The locations of the blocks and their dimensions are variables, the values of which are calculated by the software depending on the constraints and the objective function.

4.1.6 Rectangular Dualization

The partitioning process generates a group of sub circuits and their interconnections. This output from a partitioning algorithm can be represented as a graph G = (V, E) where the vertices of the graph correspond to the sub circuits and the edges represent the interconnections between the sub circuit. The floorplan can be obtained by converting this graph into its rectangular dual and this approach to floorplanning is called rectangular dualization. A rectangular dual of graph G = (V, E) consists of non-overlapping rectangles which satisfy the following properties:

- Each vertex $v_i \in V$ corresponds to a distinct rectangle $R_i 1 \leq i \leq |V|$.
- For every edge $(v_i, v_j) \in E$, the corresponding rectangles R_i and R_j are adjacent in the rectangular dual.

When this method is directly applied to the graph generated by partitioning, it may not be possible to satisfy the second property for generating the rectangular dual.

The problem of finding a suitable rectangular dual is a hard problem. In addition, there are many graphs which do not have rectangular duals. A further complication arises due to areas and aspect ratios of the blocks. In rectangular dualization, areas and aspect ratios are ignored to simplify the problem. As a result, the output cannot be directly used for floorplanning.

A planar triangular graph (PTG) G is a connected planar graph that satisfies the following properties:

- Every face (except the exterior) is a triangle.
- all the internal vertices have a degree ≥ 4
- all cycles that are not faces have length ≥ 4

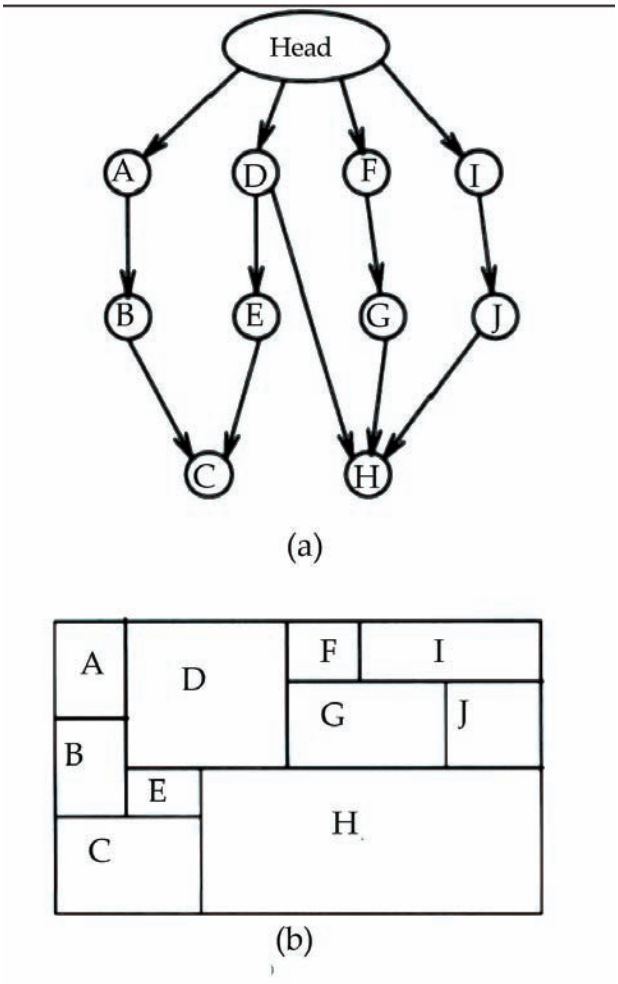


Figure 4.3: Conversion of planar digraph to a floorplan.

Given a PTG, a planar digraph is constructed which is a directed graph. Once a planar digraph is constructed, it can be converted into a floorplan as shown in Figure 4.3.

4.1.7 Hierarchical Tree Based Methods

Hierarchical tree based methods represent a floorplan as a tree. Each leaf in the tree corresponds to a block and each internal node corresponds to a composite block in the floorplan. A floorplan is said to be hierarchical of order k, if it can be obtained

by recursively partitioning a rectangle into at most k parts. Physical hierarchy can be generated in two ways: top-down partitioning or bottom-up clustering. Partitioning assumes that the relative areas (or number of nodes). Within partitions, at a given level of hierarchy, may be fixed during a top down construction of a decomposition tree (or partitioning tree). There is no justification, except convenience, for this assumption. The optimal choice of relative areas varies from problem instance to problem instance, but there is no way to determine a desirable ratio, in top-down construction. Placements performed by min-cut method, a popular partitioning algorithm, often creates lot of vacant space or white space. Clustering on the other hand is a bottom-up algorithm for constructing a decomposition tree (or cluster tree).

A hierarchical floorplanner for arbitrary size rectangular blocks using the clustering approach has been proposed. At each level of the hierarchy, highly connected blocks (or clusters of blocks) are grouped together into larger clusters. At each level, the number of blocks is limited to five so that simple pattern enumeration and exhaustive search algorithms can be used later. Blocks (or block clusters) which are connected by edges of greater than average edge weight are grouped into a single cluster, if the resulting cluster has less than five members.

After forming the hierarchical clustering tree, a floorplanner and a global router together perform a top-down traversal of the hierarchy. Given an overall aspect ratio goal and I/O pin goal, at each level of the hierarchy, the floorplanner searches a simple library of floorplan templates and considers all possible room assignments which meet the combined goals of aspect ratios and I/O pins. At each level, the global routing problem is formulated as a series of minimum Steiner tree problems in partial 3-trees. The global routing solution at the cur-rent level is used as the I/O pin goal for the floorplan evaluation, and as base for the global routing refinement at the next level. This floorplanning and global routing create constraints on the aspect ratio of the rooms, and gives assignments of I/O pins on the walls of the rooms, which are recursively transmitted downward as sub-goals to the floorplanner and global router. While evaluating the cost of a given floorplan template and room assignment, both chip area and net path length are considered. When undesirable block shapes and pin positions are detected, alternate floorplan templates and room assignments are tried by backtracking and using automatic module generators. This algorithm performs better than other well-known deterministic algorithms and generates solutions comparable to random-based algorithms.

4.1.8 Floorplanning Algorithms for Mixed Block and Cell Designs

All the algorithms discussed in the previous section can be used for floor-planning of Mixed Block and Cell (MBC) designs. These designs can be viewed as a set of blocks in a sea of cells. This is a popular ASIC layout design style. However, these algorithms were not implemented as a part of any tool which can generate floorplans for MBC

designs. In this section, we describe some of the algorithms that were developed as a part of a tool specifically designed for floorplanning of MBC designs.

In a heuristic algorithm has been developed for MBC designs. The algorithm employs a combined floorplanning, partitioning and global routing strategy. The main focus of the algorithm is in reducing the white space costs and the wiring cost. In the simulated annealing approach is used to solve the floorplanning problem for MBC designs.

4.1.9 Simulated Annealing Approach

Simulated annealing (SA) is probably the most popular method for floorplan optimization. It has the significant advantage of easily incorporating an optimizing goal into the objective function. To apply simulated annealing for floorplan design, it needs to first encode a floorplan as a solution, called a floorplan representation, which models the geometric relation of modules in a floorplan. A floorplan representation not only induces a solution space that contains all feasible solutions defined by the representation but also induces a unique solution structure that guides the search of simulated annealing to find a desired floorplan in the solution space.

Simulated Evolution (Genetic) Algorithm is based on suitable techniques for solution encoding and evaluation function definition, effective cross-over and mutation operators, and heuristic operators which further improve the method's effectiveness. An adaptive approach automatically provides the optimal values for the activation probabilities of the operators. Experimental results show that the proposed method is competitive with the most effective ones as far as the CPU time requirements and the result accuracy is considered, but it also presents some advantages. It requires a limited amount of memory, it is not sensible to special structures which are critical for other methods, and has a complexity which grows linearly with the number of implementations. Finally, it is demonstrated that the method is able to handle floorplans much larger (in terms of number of basic rectangles) than any benchmark.

Multi-Selection-Multi-Evolution (MSME) scheme for parallelizing a genetic algorithm for floorplan optimization is presented and its implementation with MPI and its experimental results are discussed. The experimental results on a 16 node IBM SP2 scalable parallel computer have shown that the scheme is effective in improving performance of floorplanning over that of a sequential implementation. The parallel version could obtain better results with more than 90 parallel program could reduce both chip area and maximum path delay by more than 8 also speed up the evolution process so that there could be higher probability of obtaining a better solution within a given time interval.

The genetic algorithm (GA) paradigm is a search procedure for combinatorial optimization problems. Unlike most of other optimization techniques, GA searches

the solution space using a population of solutions. Although GA has an excellent global search ability, it is not effective for searching the solution space locally due to crossover-based search, and the diversity of the population sometimes decreases rapidly.

4.1.10 Timing Driven Floorplanning

With increasing chip complexities and the requirement to reduce design time, early analysis is becoming increasingly important in the design of performance critical CMOS chips. As clock rates increase rapidly, interconnect delay consumes an appreciable portion of the chip cycle time, and the floorplan of the chip significantly affects its performance.

In presents a timing-influenced floorplanner for general cell IC de-sign. The floorplanner works in two phases. In the first phase the modules are restricted to be rigid and the floorplan to be slicing. The second phase of floorplanner allows modification to the aspect ratios of individual modules to further reduce the area of the overall bounding box. The first phase is implemented using genetic algorithm while in the second phase, a constraint graph based approach is adopted.

A timing driven floorplanning program for general cell layouts is presented. The approach used combines quality of force directed approach with that of constraint graph approach. A floorplan solution is produced in two steps. First a timing and connectivity driven topological arrangement is obtained using a force directed approach. In the second step, the topological arrangement is transformed into a legal floorplan. The objective of the second step is to minimize the overall area of the floorplan. The floorplanner is validated with circuits of sizes varying from 7 to 125 blocks.

The floorplanner is designed to be used in the early stages of system design, to optimize performance, area and wire ability targets before detailed implementation decisions are made. Unlike most floorplanners which optimize timing by considering only a subset of paths this floorplanner performs static timing analysis during the floorplan optimization process, instead of working on a subset of the paths. The floorplanner incorporates various interactive and automatic floorplanning capabilities.

DID YOU KNOW?

In 1959 Dijkstra published in a 3-page article 'A note on two problems in connexion with graphs' the algorithm to find the shortest path in a graph between any two given nodes, now called Dijkstra's algorithm.

4.2 CHIP PLANNING

Both floorplanning and placement problems either ignore the interconnect or consider it as a secondary objective. Chip planning is an attempt to integrate floorplanning and interconnect planning. The basic idea is to comprehend impact of interconnect as early as possible.

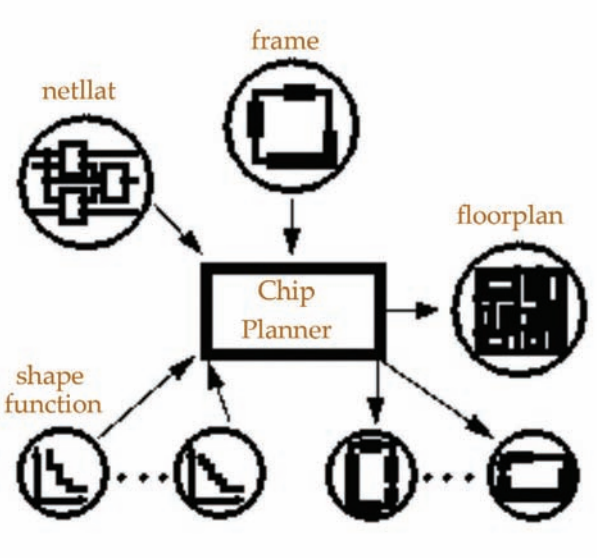
4.2.1 Problem Formulation

The input consists of $B_1, B_2 \ldots, B_n$ circuit blocks, with area $a_1, a_2 \ldots, a_n$ respectively. Associated with each block are two aspect ratios andwhich give the lower and the upper bound on the aspect ratio for that block.In addition, we have $S_1, S_2 \ldots, S_m$ signals. For each signal we have criticality, width, source and sink. The chip planning algorithm has to determine the width w_i *and height*, h_i *of each block* B_i and layout of each signal such that $A_i^l \leq \frac{h_i}{w_i} \leq A_i^h$. In addition to finding the shapes of the blocks, the chip planning algorithm has to generate a valid placement for blocks and interconnect such that the area of the layout is minimized.

4.2.2 Chip Planner of the Playout System

The chip planner is a central tool in hierarchical top-down VLSI design. The set of the transistors of a circuit is assumed to be divided into so-called `cells' in form of a hierarchical wiring diagram. The chip planner is meant to place the cells of one hierarchy level onto a given area nearly optimally. This is a top-down process. First, the whole chip area is divided for the cells on top-level. The area of each of these cells is divided recursively down to a level with primitive components. These primitives contain about 100 to 1000 transistors each and can be produced by a cell synthesis tool.

Input to a recursion step of the chip planner are the netlist and size/aspect ratio of the CUD (cell under design) as well as information about the area of the sub cells. In the playout system the latter data are given by the shape functions which have been computed by the shape function generator (SFG). The frame of the CUD further contains pin intervals, i.e. possible positions/intervals where nets can enter the cell. *Output* of the chip planner is a floorplan as `optimal' as possible as well as appropriate frames for the sub cells.



The chip planner computes the floorplan in *two steps*. The first step is *placement*, the determination of the topology, the second step is *routing*.

The *aims of a (optimal) Placement* are a total chip area which is as small as possible and short delay times. To reach this goal especially three sub goals must be reached:

- short connecting wires between the cells
- minimal cell area (of flexible cells)
- No creation of dead area during the composition of the cells.

The minimization of net lengths, i.e. of the connecting wires between the cells, is influenced by several, partially conflicting conditions. Normally, a minimal total area can be gained, if the average of all net lengths is minimal. On the other hand it may be necessary to keep certain time critical wires short enough to fulfil a prescribed timing behavior. The integration of timing preconditions in a subject of present research. Dead area can be avoided if during placement the shapes of the cell are not defined yet (flexible cells).

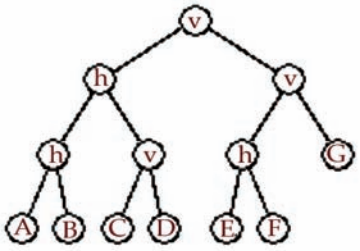
The first *task of the routing* in the chip planner is the planning of the clock and power wiring of the circuit (*clock and power routing*), the second is the assignment of the nets of the netlist to the single channels between the cells (*global routing*). In that, course, length and need of area of the connection wires are computed.

The problems the chip planning has to solve are NP-complete. An efficient computation of the optimal solution is not possible because of the combinatorial variety. Therefore a suboptimal solution has to be found heuristically, that is near to the global optimum. Even then computation time and need of memory still increase

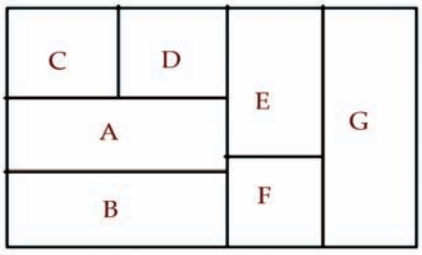
very fast with the number of cells to be placed. The strongest restriction for the number of cells is given by the search for an appropriate global routing. The number of cells that can be placed in one step (i. e. the number of sub cells per CUD) therefore should remain below an upper limit of 40 or 50. The search for solutions near to the optimum leads to different *chip planning strategies* which consider the whole top-down process across the hierarchy levels.

Placement

In the placement phase of the chip planner the CUD area is divided for the sub cells. For the division of a chip into rectangles we only allow a *slicing structure*. It is got by recursive division of the whole area into two parts each. The relative position of the cells one to another (topology) can be represented by a (binary) slicing tree in below figure.



a) h/v-oriented slicing tree



b) possible floorplan

The structure of the tree resembles the neighborhood relationship of the cells in the floorplan. Every inner node in the tree can uniquely be mapped to a slicing cut in the floorplan (and vice versa). Then the complete topology is defined by the orientation of the cuts and the relative order of two adjacent modules to a cut dividing them.

The fixing of floorplans to slicing structures is a limitation of cell placement possibilities. This limitation, though, will lead to worse results only in a few cases. On the other hand essential advantages for planning algorithms and in the chip assembly are gained by the slicing structure. The oriented slicing tree is the result of the now described partitioning and placement steps.

Partitioning

The partitioning tries to connect such cells rather deep in the slicing tree that are connected by many resp. time critical wires. Cells that have only a few common wires should own only the root node as a common super node. This procedure tries

to minimize the average net length, i.e. the routing area. In slicing structures besides the connection weight of the modules (i.e. the number of connecting nets) also their relative size must be considered. As the size of a (flexible) cell grows with increasing deformation the partitioning must try to keep the area of the two sub trees of a parent node nearly the same.

The chip planner consists of three different partitioning algorithms: Cluster, Min-cut and Inplace partitioning. A fourth method, Ratio-Cut partitioning, currently is implemented.

- The *Cluster algorithm* works bottom-up, distributing the sub cells to clusters and combining these cluster to even larger clusters until one big cluster is produced containing all sub cells of the CUD. The heuristic of this algorithm consists in *maximization* of the connection costs *inside* a cluster.
- The *Min-cut algorithm* contrarily works top-down. It recursively divides the set of all cells into smaller partitions until such a partition only contains one cell. This heuristic *minimizes* the connection costs *between* the partitions (weight of the cut).
- The *Ratio-Cut algorithm* essentially is an extension of the Min-cut algorithm. The cost function of the Ratio-Cut partitioning not only considers the number of cut nets but also the balance of the partitions. Thus the Ratio-Cut partitioning tends to `natural› partitions, more than the Min-cut heuristic for example. Here the heuristic is the *minimization* of the quotient of the cut weight and the product of the sub tree sizes.

Every procedure computes the connection strength of cells and clusters respect partitions (and thus an un-oriented slicing tree), but the relative order of the cells remains undefined. This requires additional ordering steps. Because of its global sight in the first recursion steps the Min-cut algorithm in most cases yields better results than the Cluster method. As already said the Ratio-Cut is still worked on.

The third algorithm performs an *Inplace partitioning* of the modules to be placed and generates a slicing tree in this manner. The word `inplace' describes the fact that this partitioning of a subset of modules considers the location of the CUD pins as well as the connections to modules that do not belong to the set to be partitioned. In order to consider the location of the CUD pins the designer has to define the orientation of the cuts on top level.

Sizing and H/V-Orientation

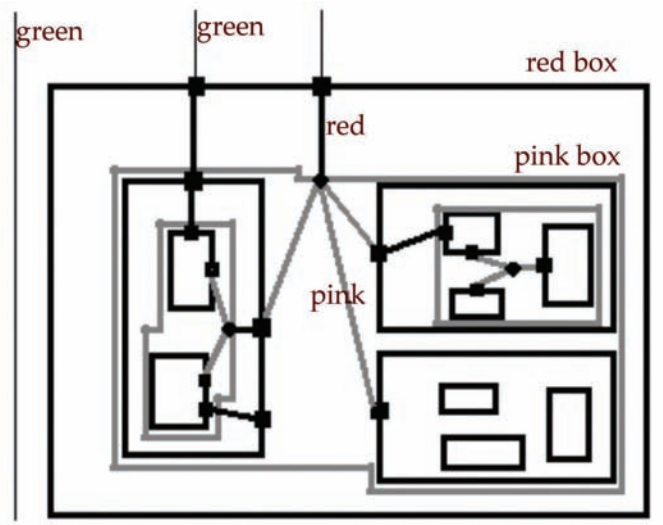
After the partitioning the generated un-oriented slicing tree has to be oriented and for the flexible cells a form has to be chosen. The sizing-h/v-orientation algorithm processes the slicing tree bottom-up. With the input to each sub cell of the CUD, i.e. to each leaf cell of the slicing tree, the chip planner is already given a shape function.

The shape function of an inner node in the slicing tree is computed as follows: The shape functions of both sons are added `horizontally' and `vertically'. Thus two shape functions are created out of which one shape function is generated by optimal superposition. The procedure goes on until the root node is reached.

For the root node one point on the shape function is chosen (determination of the external frame of the CUD). For each point on the shape function it has been saved whether it is the result of vertical or horizontal addition of the sons. So the orientation of the cut (h/v-orientation) as well as the shape of both sons can be determined (sizing). The procedure is continued until the leafs are reached. The **algorithm** aims at determining the cuts of a given partitioning in a way that the area is as small as possible.

Algorithm is a procedure or formula for solving a problem, based on conducting a sequence of specified actions.

A special topic of sizing is the consideration of the wiring. The area needed for the wiring of the sub cells is estimated and partially added to the area of the concerned sub cells. This results in a shift (right and up) of the shape function. As the nets are globally defined in the net list, a net may connect cells of different hierarchy levels and so may exist at severy hierarchy levels. For the purpose of estimation we divide nets into segments that are fully contained in one level only. Here we use the following color model:



Internal net segments, i.e. segments that are only connections between sub cells and do not leave the CUD, are displayed *pink*; *red* segments are the connections to the frame of the CUD, i.e. to the external pins; the *green* color is assigned to the external wiring (below figure). Depending on which wiring area has been included we get shape functions for the enclosing rectangles (pink, red and green box).

The additional wiring area is determined by the Track Demand method. Depending on the orientation of an edge cutting a net, track demands in orthogonal direction are necessary. A horizontal cut certainly causes a track demand in vertical direction, a vertical edge requires wiring area in X-direction. The quantity of the shift in X- and Y-direction is computed with the aid of statistically determined track demand factors which depend on the actual slice orientation.

To minimize the clock skew among all leaf nodes, the clock delay for each sub-block must be determined and the design of the clock planned accordingly.

L/R-Ordering

The l/r-ordering decides whether a module is to be placed left or right to a vertical respect above or below a horizontal cut. The algorithm processes the slicing tree top-down. It is based on the floorplan topography and therefore needs additional data on the need of area for the sub cells and their routing.

The (topographic) l/r-ordering only examines a certain section (window) of the slicing tree at a time. The depth of the search window is configurable. Within the search window all possible arrangements of the modules are computed by mirroring the modules at the cuts. Call such an arrangement a configuration. The best configuration of a window is taken as the start orientation for the next (deeper) window.

The first objective of l/r-ordering is the minimization of the sum of net lengths over the configurations. Considering a given configuration, for each net a (heuristic) Steiner tree is built between the centers of the modules the net connects (because the exact pin positions are not known yet).

To consider timing issues several approaches are workable:

- weighting of critical nets
- limits of net lengths, capacities or delays (based on a timing analysis)
- Incremental timing analysis.

After the determination of the geometrical ordering manual adjustments by exchange or movement of sub cells can be performed. This interaction of designer and chip planner is supported by graphical output and extensive analysis. Since the chip planner essentially contains only heuristic algorithms this interaction (together with analysis) is a main part of chip planning.

Clock and Power Routing

The routing sets up an electrical connection between a numbers of cells belonging together. As soon as more than two cells are to be connected this connection is no longer a simple `wire', you speak of a `net'. Those nets can follow certain `channels' between the cells or, in some cases, through the cells. Take the points that are to be wired with a net as vertices and the possible wiring routes between single points as edges and it gets clear that the routing can be considered as a *graph problem*.

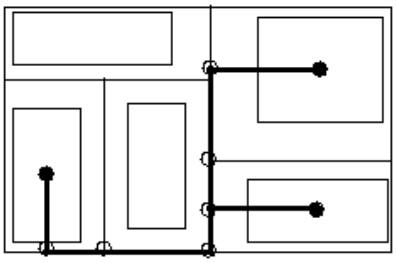
Figure 4.4: Minimal Spanning Tree (a) and Steiner Tree (b).

A first theoretical approach to routing problem therefore would be the search for a minimum spanning tree for the graph spanned by the vertices to be connected (figure 4.4a). If the possibility exists to use further vertices besides the points to be connected and if we compute the **minimum spanning tree** of this extended graph, we get a *Steiner Tree* whose total net length often is smaller (figure 4.4b).

Definition. Let G = (V, E) be a connected graph with vertex set V and edge set E. Let λ : E > **R**+ be a cost function and R the subset of the *required vertices* of V.

A *Steiner Tree* is a sub tree of G that connects all vertices in R and all of whose leaves are vertices in R. A *minimum Steiner Tree* is a Steiner tree whose costs concerning λ are minimal.

For the global routing in the chip planner we use the following *channel intersection graph* model. Since the exact positions of the cell pins is not determined yet, the vertices to be wired (required vertices) are identified with the centers of these cells. The edges essentially consist of the slicing lines of the (partitioned and oriented) floorplan. Additional vertices are the corners of the cells and the projections of the module centers to the slicing lines. Therefore, a net that connects the required vertices is a Steiner tree on this channel intersection graph (below figure).



Now the chip planner performs the global routing sequentially. One net after the other is routed with the aid of Steiner tree heuristics, under consideration of restrictions given by the already routed nets. A heuristic method has to be used because the problem complexity - not at least because of the variable additional vertices - does not allow an exact computation.

The global routing assigns channels between the cells (through the cells) to the nets. The length and need of area of the connecting wires are computed. The channel area is added to the adjacent cells proportionally. Since the sizes of the sub cells change by this another sizing step must follow. The phases global routing and sizing have to be repeated as long as major changes in topography take place.

There is the dilemma that the computations of the shortest routes are always based on the old channels and new routes imply new channels again. Furthermore it cannot be guaranteed that after any number of iterations the total area converges at a minimum. The designer himself/herself has to decide after which iteration he or she wants to accept the result. In practice it has turned out that the results normally are already satisfying after two or three iterations.

Minimum spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

Chip Planning Strategies

The chip planner is a tool for hierarchical top-down design of a circuit. The design steps described above therefore take place top-down on different levels. It has to be guaranteed that the realizations of the lower levels and eventually the physical layout fit to the intended frame of the floorplan of the current CUD. The presupposition for such a top-down approach therefore is a good area estimation.

Before the chip planner is used the sizes of the sub cells were computed by an area estimation step. The result of the estimation is a shape function that represents the minimal area of a cell for all aspect ratios. It is obvious that the quality of a top-down design depends very much on the quality of the estimation phase. However, it is not possible to estimate the area of a cell exactly. All estimation methods have tolerances because they do not have all information the synthesis tools will have. The analysis of many test designs has shown that even small changes in the shape or size of a (sub) cell may result area disadvantages. These differences between estimation and final layout make an improved top-down chip planning method necessary.

Sibling is one of two or more individuals having one or both parents in common.

Simple top-down chip planning computes a floorplan based on the shape functions in one isolated step per cell and determines a frame for the sub cells in the sizing sub step. These frames are input to the planning of the sub cells and then obliging. The procedure starts on top-level with the whole chip, afterwards the sub cells on the next lower level can be processed analogously and independently. So the computation of the sub trees can be performed parallel.

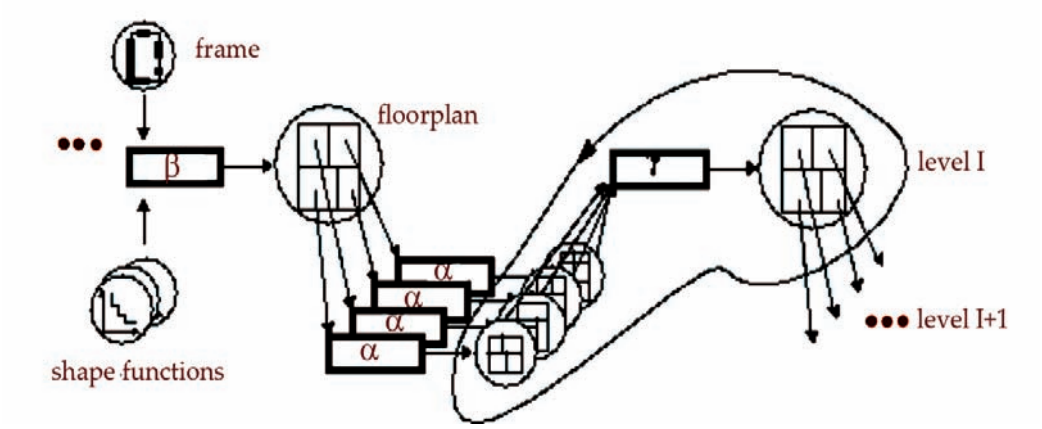
A problem is here the unique determination of the cell frame on the basis of an estimated shape function (additionally the determination of pin is problematic). If later realizations cannot fulfil the frame restrictions the chip assembly phase has to compensate by means of cell enlargement, normally resulting in dead area. Without going into detail concerning possible kinds of errors it can be said that extensive tests have shown the impossibility of area estimation with a deviation of less than 10 - 15%. In the worst-case it can be even greater. In most cases this is inacceptable. One possibility to react upon deviations in top-down chip planning is to return to the shape function estimation phase and to change estimation parameters to gain shape functions of better quality. This would be a design iteration step which should have been avoided. A second possibility is to use realizations (layouts) of critical cells (using macros or multi-macros instead of flexible cells). The giving up of flexibility leads to loss of design quality, too.

A solution is the try to balance the deviations across several hierarchy levels in several phases. This approach is pursued in the Three-Phase chip planning method as described in the following.

Three-Phase Chip Planning

If we are not planning the top-most cell, it may be possible to compensate the deviation of the current cell shape with the deviations of the **sibling** cells. The area of the super cell should not change if the sum of all area estimations of the sibling cells is similar to the areas of the realizations. Furthermore, all planned cells are lightly flexible because their sub cells are still flexible. This flexibility increases the chance of a good balancing.

So, it is useful after computing a floorplan of the CUD to perform an adjustment planning step for the super cell before continuing the top-down planning process at the sub cell level. The deviations at the current level can be compensated at the super cell level (below figure).



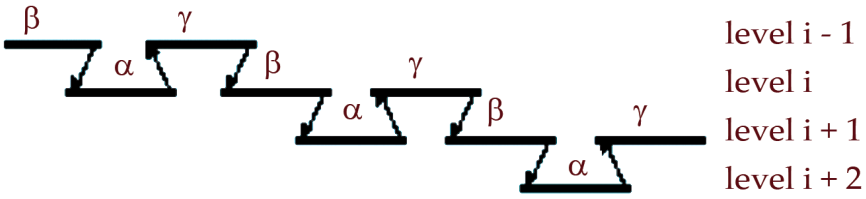
There are three steps in the Three-Phase planning which we denote by Greek letters ALPHA, BETA, and GAMMA. After planning the sub cells at level i+1 (phase ALPHA), we perform an adjustment step at level i (phase GAMMA) before continuing the top-down planning of all sub cells at level i+1 (phase BETA). Of course, the cell at level i itself is part of an adjustment process for its super cell on the level i-1. Thus, the three fundamental planning phases ALPHA, BETA, and GAMMA are performed with each cell except for those on top and bottom level. This gave the method its name.

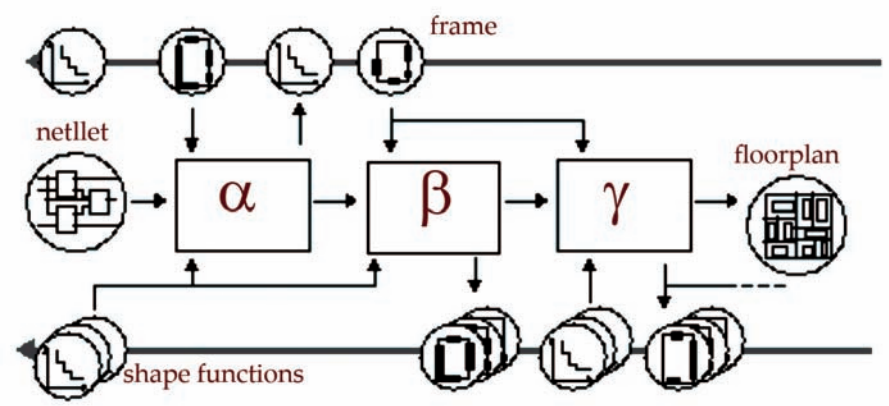
- In *phase* ALPHA we compute an initial floorplan whose topology will be fixed for all future planning steps. Since the sub cells are still flexible many floorplans with different shapes but the same topology are feasible. We generate a refined shape function that describes all possible shapes of the same topology. This refined shape functions are more precise than shape

functions of flexible cells because they rely on a particular topology and a global routing (not only on a rough wiring area estimation). They are input to the super cell adjustment (phase GAMMA).

- *Phase* BETA adjusts the geometry of phase ALPHA to a new input frame. This frame stems from phase GAMMA of the super cell. After this phase the phase ALPHA of the sub cell level can start.
- *Phase* GAMMA is used for an adjustment to more precise sub cell information which are given by refined shape functions and computes a new frame for each cell.

In below figure the process of Three-Phase chip planning on several hierarchy levels is shown in an abstract notation. In below figure you see the data flow during the three phases. Not drawn is a floorplan which is also passed from ALPHA to BETA and from BETA to GAMMA.





In the meaning of stepwise refinement extensions of this method are possible, e. g. splitting of phase GAMMA into several adjustment phases or the inclusion of more levels. Thus the procedure becomes more and more complex and eventually you have to weigh up speed and quality of the design process. It must be considered further that increasing vertical and horizontal dependencies of the phases decrease the possibilities of parallelization.

4.3 PIN ASSIGNMENT

The purpose of pin assignment is to define the signal that each pin will receive. Pin assignment may be done during floorplanning, placement or after placement is fixed. If the blocks are not designed then good assignment of nets to pins can improve the placement. If the blocks are already designed, it may be possible to exchange a few pins. This is because some pins are function-ally equivalent and some are equipotential. Two pins are called functionally equivalent, if exchanging the signals does not affect the circuit. For example, exchanging two inputs of a gate does not affect the output of the gate. Two pins are equipotential if both are internally connected and hence represent the same net. The output of the gate may be available on both sides, so the output signal can be connected on any side. Figure 4.5 shows both functionally equivalent pins and equipotential pins.

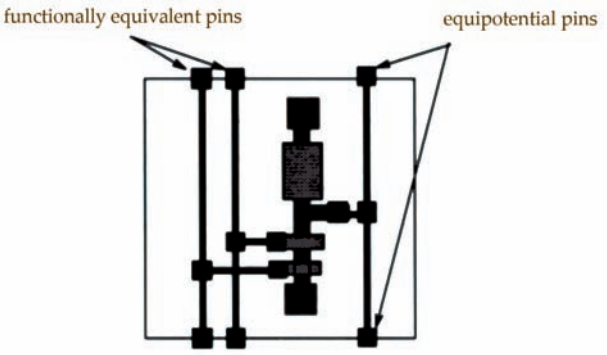


Figure 4.5: Functionally equivalent and equipotential pins.

In order to fit each macro instance at the chip top level in an effective manner, the automatic floorplan algorithm needs to have a range of legal shapes that is derived from aspect ratio bounds for each partition in the design.

4.3.1 Problem Formulation

The purpose of pin assignment is to optimize the assignment of nets within a functionally equivalent pin groups or assignment of nets within an equipotential pin group. The objective of pin assignment is to reduce congestion or reduce the number of crossovers. Figure 4.6 illustrates the effectiveness of pin assignment. Note that a net can be assigned to any equipotential pin within a set of functionally equivalent pins. The pin

assignment problem can be formally stated as follows: Given a set of terminals $T_1, T_2, \ldots, T_n$ and a set of pins $P_1, P_2, \ldots, P_m$, $m > n$. Each T_i is assigned to pin P_i, $i = 1, 2, \ldots, n$. Let $\mathcal{E}_{P_i}$ be the set of pins which are equipotential and equivalent to P_i, the objective of pin assignment is to assign each T_i to a pin in $\mathcal{E}_{P_i}$ such that a specific objective function is minimized. The objective functions are typically routing congestions. For standard cell design, it may be the channel density.

Design Style Specific Pin Assignment Problems

Pin assignment problems in different design styles have different objectives.

- Full custom design style:

In full custom, we have two types of pin assignment problems. At floorplanning level, the pin location along the boundary of the block can be changed as the block is assigned a shape. This assignment of pins can reduce routing congestions. Thus, not only we can change pin assignment of pins, we can also change the location of pins along the boundary. At placement level, the options are limited to assigning the nets to pins. Notice that in terms of problem formulation, we can declare all pins of a flexible block as functionally equivalent to achieve pin assignment in floorplanning.

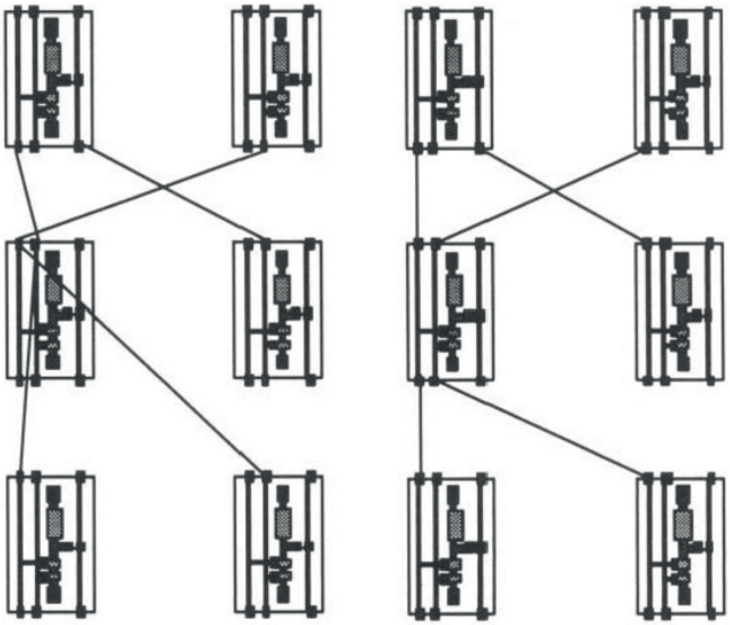


Figure 4.6: Impact of pin assignment.

- Standard cell design style:

  The pin assignment problem for standard cells is essentially that of permuting net assignment for functionally equivalent pins or switching equipotential pins for a net.

- Gate array design style:

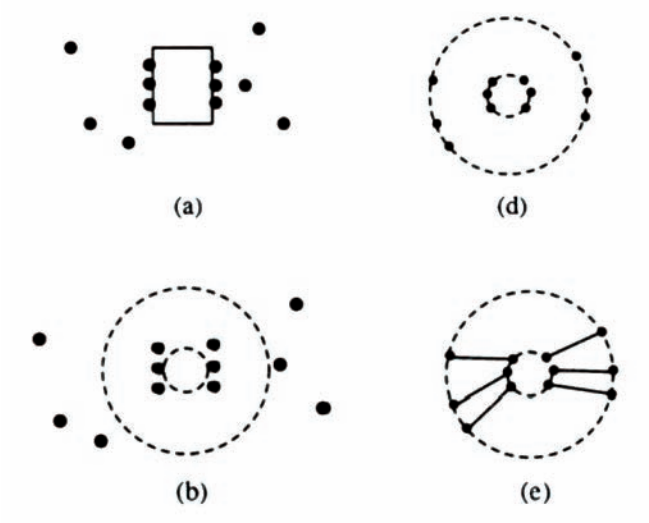
The pin assignment problem for gate array design style is the same as that of standard cells.

Assignment problems mostly occur in semi-custom design styles such as gate arrays or standard cells.

In gate array design, the cells are pre-fabricated and are arranged on the master. Pin assignment problem in this type of design style is to assign to each terminal a functionally equivalent slot such that wiring cost is minimized. Slots in this case are the pin locations on pre-designed (library) cells. In standard cells, however, equipotential pins appear as feedthroughs. Since no wiring around the cell is needed, the wire length decreases with the use of feedthroughs.

4.3.2 Classification of Pin Assignment Algorithms

The pin assignment techniques are classified into general techniques and special pin assignment techniques. General techniques are applicable for pin assignment at any level and any region within a chip. Such techniques are applied at board level as well as chip level. On the other hand, the special pin assignment technique can be used for assignment of pins within a specific region such as channel or a switchbox.



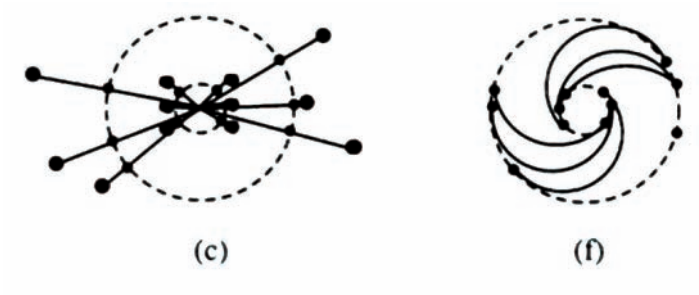


Figure 4.7: Concentric circle mapping.

REMEMBER

To avoid area segmentation, macros should be positioned such that the standard cell area is continuous.

4.3.3 General Pin Assignment

There are several methods in this category as discussed below:

Concentric Circle Mapping:

To planarize the interconnections, this method models the pin assignment problem by using two concentric circles. The pins on the component being considered are represented as points on the inner circle whereas the points on the outer circle represent the interconnections to be made with other components. The concentric circle mapping technique solves the pin assignment problem by breaking it into two parts. The first part is the assignment of pins to points on the two circles and in the second part the points on the inner and outer circles are mapped to give the interconnections.

For example, consider the component and the pins shown in Figure 4.7(a).The two circles are drawn so that the inner circle is inside all the pins on the component being considered while the outer circle is just inside the pins that are to be connected with the pins of this component. This is shown in Figure 4.7(b). Lines are drawn from the component center to all these pins as shown in Figure 4.7(c). The points on the inner and outer circle are defined by the intersection of these lines with the circles (Figure 4.7(d)). The pin assignment is completed by mapping the points on the outer circle to those on the inner circle in a cyclic fashion. The worst and the best case assignment are shown in Figure 4.7(e) and (f).

Topological Pin Assignment:

Brady developed a technique which is similar to concentric circle mapping and has certain advantages over the concentric circle mapping method. With this method it is easier to complete pin assignment when there is interference from other components and barriers and for nets connected to more than two pins. If a net has been assigned to more two pins than the pin closest to the center of the primary component is chosen and all other pins are not considered. Hence in this case only one pin external to the primary component is chosen. The pins of the primary component are mapped onto a circle as in the concentric circle method. Then beginning at the bottom of the circle and moving clockwise the pins are assigned to nets and hence they get assigned in the order in which the external pins are encountered. For nets with two pins the result is the same as that for concentric circle mapping.

Nine Zone Method:

The nine zone method, developed by Mory-Rauch, is a pin assignment technique based on zones in a Cartesian coordinate system. The center of the coordinate system is located inside a group of interchangeable pins on a component. This component is called pin class.

A net rectangle is defined by each of the nets connected to the pin class. There are nine zones in which this rectangle can be positioned as shown in Figure 4.8. The positions of these net rectangles are defined relative to the coordinate system defined by the current pin class.

4.3.4 Channel Pin Assignment

In design of VLSI circuits, a significant portion of the chip area is used for channel routing. Usually, after the placement phase, the positions of the terminals on the boundaries of the blocks are not completely fixed and they still have some degree of freedom to move before the routing phase begins. Figure 4.9 shows how channel density could be reduced by moving the terminals. Figure 4.9(a) shows a channel which needs three tracks. By moving the pins, the routing can be improved such that it requires one track as shown in Figure 4.9 (b). The channel pin assignment problem is the problem of assigning positions for the terminals, subject to constraints imposed by design rules and the designs of the previous phases, so as to minimize the density of the channel. The problem has various versions depending on how the pin assignment constraints are specified.

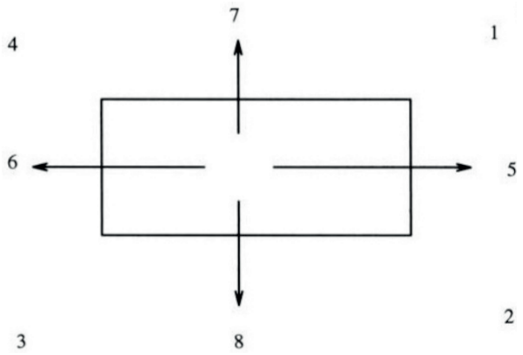


Figure 4.8: The nine pin zones.

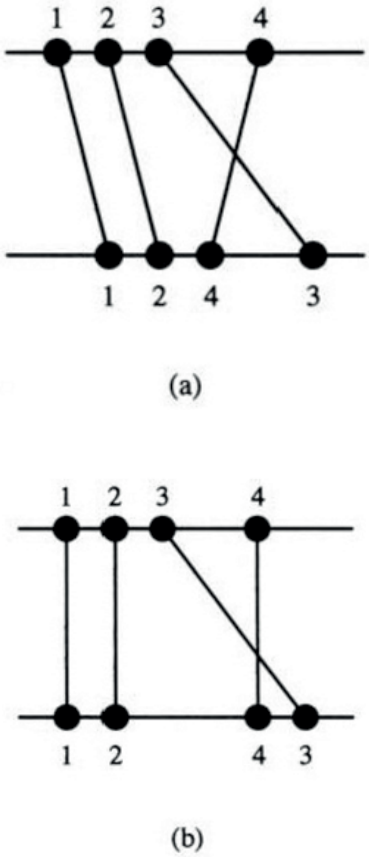


Figure 4.9: Reducing channel density by moving terminals.

4.3.5 Integrated Approach

The various stages in the physical design cycle evolved as the entire problem is extremely complex to be solved altogether at once. But over the years, with better understanding of the problems, attempts are made to merge some steps of the design cycle. For example, floorplanning was considered as a problem of just finding the shapes of the blocks without considering routing areas. Over the years, the floorplanning problem has been combined with the placement problem. The placement problem is sometimes combined with the routing problem giving rise to the 'place and route' algorithms.

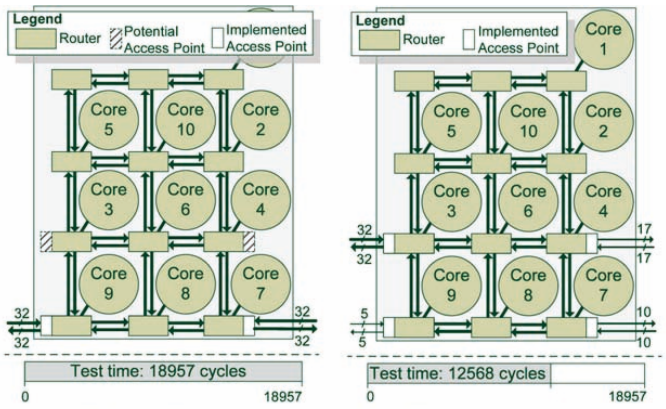
The process of floorplanning is carried out in the following three steps:

- Clustering:

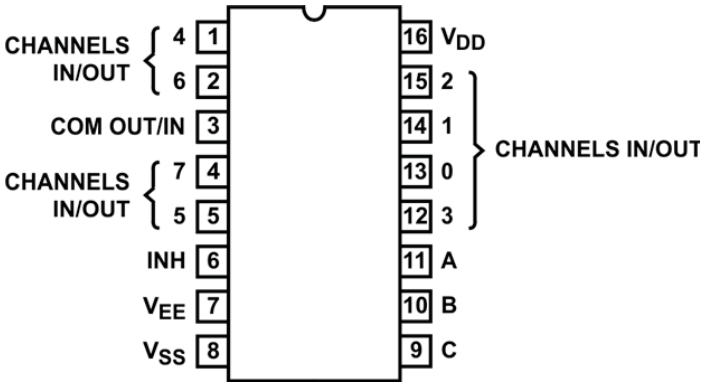
  In this step, a hierarchical tree is constructed. Blocks that are strongly connected are grouped together in a cluster. Each cluster can have a limited number of blocks within it. The clustering process considers the shapes of the blocks to avoid a mismatch within the cluster. This step is repeated to build the cluster tree.

- Placement:

  In this step, the tree is traversed top-down. The target shape and terminal goals for the root of the cluster tree is specified. This information is used to identify the topological possibility for the clusters at the level below.



Figure



Figure

This in turn sets the shape and terminal goals for the immediate lower levels in the hierarchy till at the leaf level the orientations of the blocks are determined. For each of the topologies, the routing space is determined. The selection of a particular topology is based on the area and the shape of the resulting topology and the connection cost. The system is developed so as to allow the user to control the tradeoff between the shape, the area and the connection costs. This strategy works well in case the blocks at the leaf level are flexible so that the shapes of these blocks can be adjusted to the shape of the cluster. On the other hand, if the leaf level blocks are fixed then this top-down approach can give unfavorable results. This is due to the fact that the information of the shape of these blocks at the leaf level are not considered by the objective function when determining the cluster shapes at higher levels of the cluster tree. This is rectified by passing the shape information from the leaves towards the root of the tree during the clustering phase. In addition, during the top-down placement step, a look-ahead is added so that the objective function can examine the shapes generated during **clustering**, at a level below the immediate level for which the shape is being determined.

Clustering is the task of grouping a set of objects in such a way that objects in the same group are more similar to each other than to those in other groups.

Floorplan optimization:

This an improvement step that resizes selected blocks iteratively. The blocks to be resized are identified by computing the longest path through the layout surface using the routing estimates done in the above step.

SUMMARY

- Floorplanning is the process of identifying structures that should be placed close together, and allocating space for them in such a manner as to meet the sometimes conflicting goals of available space (cost of the chip), required performance, and the desire to have everything close to everything else.
- In the placement phase, blocks are positioned on a layout surface, in a such a fashion that no two blocks are overlapping and enough space is left on the layout surface to complete the inter connections.
- In order to simplify the problem, the blocks are assumed to be rectangular. The shapes resulting from the floorplanning algorithms are mostly rectangular for the same reason.
- The heat dissipated should be uniform over the entire surface of the group of blocks placed by the placement algorithms.
- The goal of floorplanning is to optimize a predefined cost function, such as the area of a resulting floorplan given by the minimum bounding rectangle of the floorplan region.
- The initial sized floorplan represents an empty base rectangle whose area is the total of all weights of the vertices of the weighted graph and each node in the tree represents a rectangular room in the layout area.

KNOWLEDGE CHECK

1. In VLSI design, which process deals with the determination of resistance & capacitance of interconnections?

a. Floorplanning
b. Placement & Routing
c. Testing
d. Extraction

2. Chip utilization depends on ___.

a. Only on standard cells
b. Standard cells and macros
c. Only on macros
d. Standard cells macros and IO pads

3. Which configuration is more preferred during floorplaning?

a. Double back with flipped rows
b. Double back with non-flipped rows

c. With channel spacing between rows and no double back
d. With channel spacing between rows and double back

4. Routing congestion can be avoided by ___.

a. placing cells closer
b. Placing cells at corners
c. Distributing cells
d. None

5. In Physical Design following step is not there ___.

a. Floorplaning
b. Placement
c. Design Synthesis
d. CTS

REVIEW QUESTIONS

1. What do you mean by floorplanning?
2. Define the difference between slicing and non- slicing floorplans.
3. Why is the chip planner a central tool in hierarchical top-down VLSI design?
4. Discuss about constraint based floorplanning.
5. What do you understand by pin assignment?

CHECK YOUR RESULT

1. (d) 2. (b) 3. (a) 4. (c) 5. (c)

REFERENCES

1. Springer, http://link.springer.com/chapter/10.1007%2F0-306-47509-X_6#page-1
2. http://vlsibyjim.blogspot.in/2015/03/floorplanning.html
3. Scribd, https://www.scribd.com/doc/51093768/Algorithms-for-VLSI-Physical-Design-Automation-Third-Edition
4. http://cc.ee.ntu.edu.tw/~ywchang/Courses/PD_Source/EDA_floorplanning.pdf
5. http://www.schoelzke.info/mirror/galway/projects/ChipPlanner-Description.htm

GLOBAL ROUTING

CHAPTER 5

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Understand the overview of global routing
2. Define the problem formulation
3. Describe the classification of global routing algorithms
4. Explain the maze routing algorithms and line-probe algorithms
5. Discuss the steiner tree based algorithms

INTRODUCTION

Global routing in VLSI (very large scale integration) design is one of the most challenging discrete optimization problems in computational theory and practice. VLSI physical design is a multi-phase process, where each phase typically falls into one of the following three classes: partitioning, placement, and routing. In the partitioning phase, we split the chip into smaller, more manageable pieces. The assumption is that each of these pieces may be designed independently of one another. In the placement phase, we fix the locations of all blocks within the chip, as well as produce a list of blocks which need to be connected with wires. In the routing phase, the goal is to find a

realization of the connections provided from the placement phase. Typically, routing is broken into two distinct processes: global routing, and detailed routing. In global routing, we wish to find the approximate interconnections between the blocks. Detailed routing takes the output from the global router and produces the exact geometric layout of the wires to connect the blocks.

In the global routing phase of VLSI design, we assume that the circuits are in a one-layer frame. We model the chip as a lattice graph, where each channel in the chip corresponds to an edge in the lattice graph. Pins of the chip components are found at the intersections of these edges, which correspond to vertices in the lattice graph. We define a net to be a group of pins which are to be connected. In an instance, we are given a set of nets, each of which has pins that must be connected by wires. Additionally, there are constraints to the number of wires that may pass through any given channel. A solution is a set of trees in the lattice graph, one for each net, corresponding to the wires in the chip routing the given nets, while the constraints are satisfied. The goal is to minimize some cost of these connections (such as wire-length or edge congestion).

Short circuit is an electrical circuit that allows a current to travel along an unintended path with no or a very low electrical impedance.

5.1 OVERVIEW OF GLOBAL ROUTING

In the placement phase, the exact locations of circuit blocks and pins are determined. A netlist is also generated which specifies the required inter-connections. Space not occupied by the blocks can be viewed as a collection of regions. These regions are used for routing and are called as routing regions. The process of finding the geometric layouts of all the nets is called routing. Nets must be routed within the routing regions. In addition, nets must not **short-circuit**, that is, nets must not intersect each other.

The input to the general routing problem is:

1. Netlist,
2. Timing budget for nets, typically for critical nets only,
3. Placement information including location of blocks, locations of pins on the block boundary as well as on top due to ATM model (sea-of-pins model), location

of I/O pins on the chip boundary as well as on top due to C4 solder bumps,

4. RC delay per unit length on each metal layer, as well as RC delay for each type of via.

The objective of the routing problem is dependent on the nature of the chip. For general purpose chips, it is sufficient to minimize the total wire length, while completing all the connections. For high performance chips, it is important to route each net such that it meets its timing budget. Usually routing involves special treatment of such nets as clock nets, power and ground nets. In fact, these nets are routed separately by special routers. A VLSI chip may contain several million transistors. As a result, tens of thousands of nets have to be routed to complete the layout. In addition, there may be several hundreds of possible routes for each net. This makes the routing problem computationally hard.

One approach to the general routing problem is called Area Routing, which is a single phase routing technique. This technique routes one net at a time considering all the routing regions. However, this technique is computationally infeasible for an entire VLSI chip and is typically used for specialized problems, and smaller routing regions.

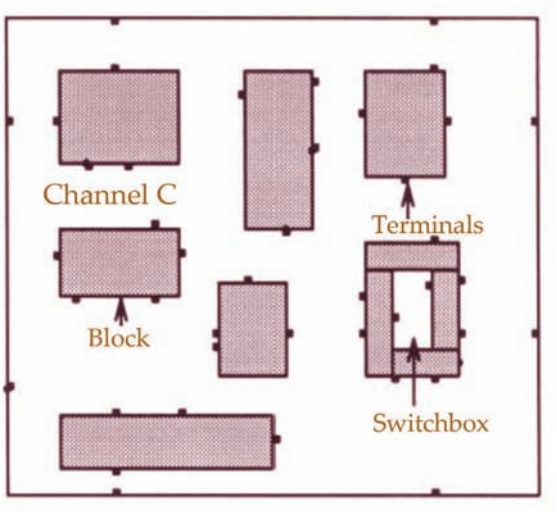


Figure 5.1: Layout of circuit blocks and pins after placement.

The traditional approach to routing, however, divides the routing into two phases. The first phase is called global routing and generates a 'loose' route for each net. In fact it assigns a list of routing regions to each net without specifying the actual geometric layout of wires (see Figure 5.2(a)). The second phase, which is called detailed routing, finds the actual geometric layout of each net within the assigned routing regions (see Figure 5.2(b)).

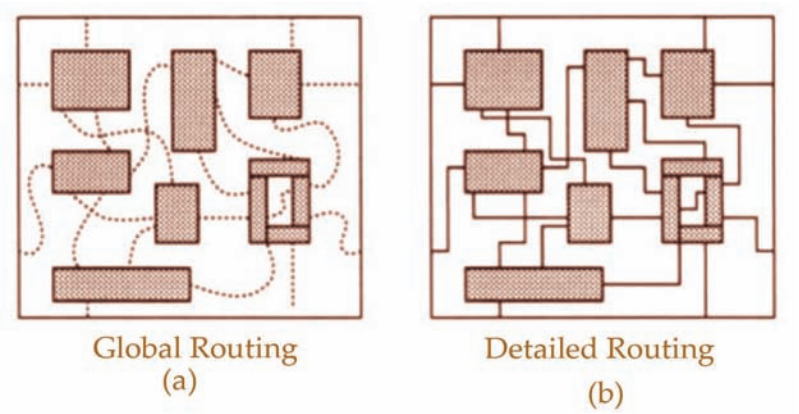


Figure 5.2: Two phases of Routing.

Unlike global routing, which considers the entire layout, a detailed router considers just one region at a time. The exact layout is produced for each wire segment assigned to a region, and vias are inserted to complete the layout. In fact, even when the routing problem is restricted to a routing region, such as channels, it cannot be solved in polynomial time, i.e., the channel routing problem is NP-complete. Figure 5.3 shows a typical two phase routing approach.

The global routing consists of three distinct phases; Region definition, Region Assignment, and Pin assignment. The first phase of global routing is to partition the entire routing space into routing regions. This includes spaces between blocks and above blocks, that is, OTC areas. Between blocks there are two types of routing regions: channels and 2D-switchboxes. Above blocks, the entire routing space is available, however, we partition it into smaller regions called 3D-switchboxes. Each routing region has a capacity, which is the maximum number of nets that can pass through that region. The capacity of a region is a function of the design rules and dimensions of the routing regions and wires. A channel is a rectangular area bounded by two opposite sides by the blocks. Capacity of a channel is a function of the number of layers (l), height (h) of the channel, wire width (w) and wire separation (s), i.e., $\text{Capacity} = \frac{l \times h}{w + s}$. For example, if for channel C shown in Figure 5.1, l= 2, $h = 18\lambda, w = 3\lambda, s = 3\lambda,$ then the capacity is $\frac{2 \times 18}{3+3} = 6.$ In a five layer process, only M1, M2 and M3 are used for channel routing. Note that channel may also have pins in thmiddle. The pins in the middle are actually used to make connections to nets routed in 3D-switchboxes. A 2D-switchbox is a rectangular area bounded on all sides by blocks. It has pins on all four sides as well as pins in the middle. The pins in the middle are actually used to make connections to nets route din 3D-switchboxes. A 3D-switchbox is a rectangular

area with pins on all six sides. The pins on the bottom are the pins which allow for connections to nets in channels, 2D-switchboxes and nets using ATM (sea-of-pins) on top of blocks. The pins on the top may be required to connect to C4 solder bumps.

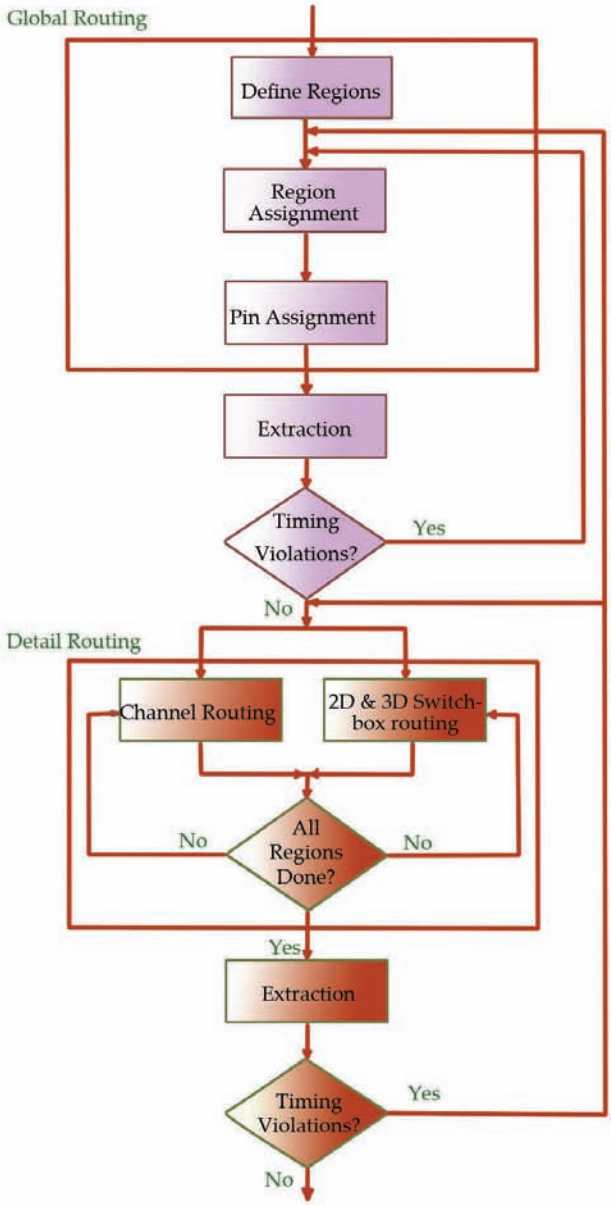


Figure 5.3: The two phase routing flow.

Consider the five metal layer process and assume that blocks use up to third metal layer for internal routing. In this case, channel and 2D-switchboxes will be used in Ml, M2 and M3 to route regions between the blocks. Furthermore, the M4 and M5 routing space will be partitioned into several smaller routing regions. The three different routing regions are shown in Figure ref3dswitchbox-6. Another approach to region definition is to partition the M4 and M5 along block boundaries. In this case, channels and 2D-switchboxes will be routed in five metal layers. In addition the regions on top of blocks will be 3D-switchboxesand need to be routed in M4 and M5.

The second phase of global routing can be called region assignment. The purpose of this phase to identify the sequence of regions through which a net will be routed. This phase must take into account the timing budget of each net and routing congestion of each routing region. After the region assignment, each net is assigned a pin on region boundaries. This phase of global routing is called pin assignment. The region boundaries can be between two channels, channel and 3D-switchbox, 2D-switchbox and a 3D-switchbox among others. The pin assignment phase allows the regions to be somewhat independent.

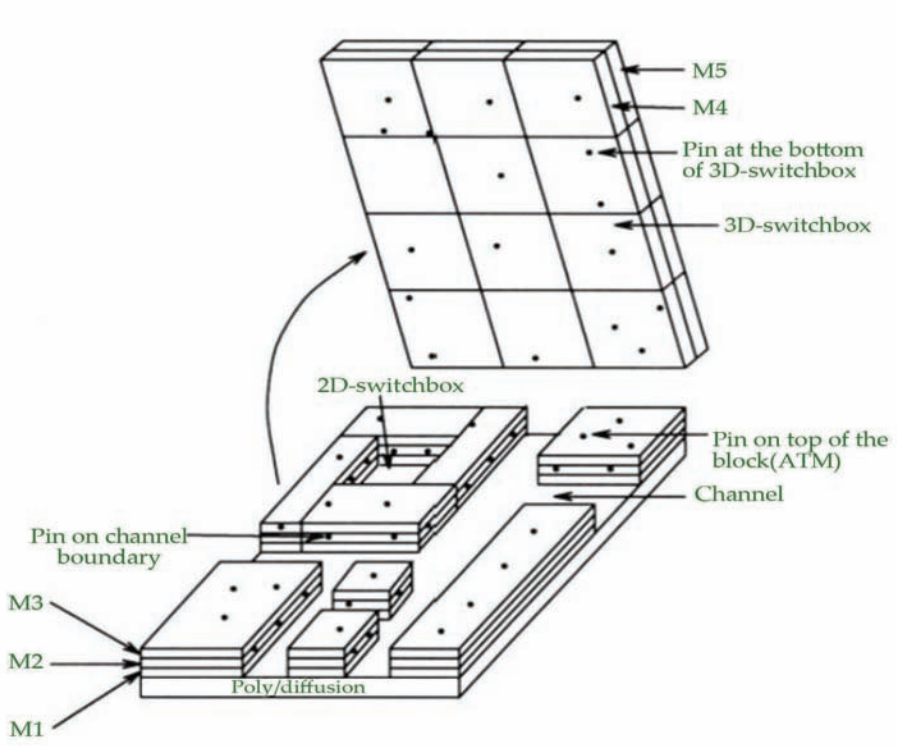


Figure 5.4: Three different routing regions.

After global routing is complete, the output is pin locations for each net on the all the region boundaries it crosses. Using this information, we can extract the length of the net and estimate its delay. If some net fails to meet its timing budget, it needs to be ripped-up or global routing phase needs to be repeated.

Detailed routing includes channel routing, 2D-switchbox and 3D-switchboxrouting. Typically channels and 2D-switchboxes should be routed first, since channels may expand. After channels and 2D-switchboxes have been routed, the pin locations for 3D-switchboxes are fixed and then their routing can be completed. Channels are routed in a specific order to minimize the impact of channel expansion on the floor plan.

After detailed routing is completed, exact wire geometry can be extracted and used to compute RC delays. The delay model not only considers the geometry (length, width, layer assignments and vias) of a net, but also the relationship of this net with other nets. If some nets fail to meet their timing constraints, they need to be ripped-up or detailed routing of the specific routing region needs to be repeated.

Global routing has to deal with two types of nets. Other nets which may not need use of M4and M5 can be routed through a sequence of channels. Global router must notallocate more nets to a routing region than the region capacity. Let us illustrate this concept of global routing by an example. Suppose that each channel in Figure 5.5 has unit capacity. We consider routing of two nets $N_1 = \{t_{11}, t_{12}\}$ and $N_2 = \{t_{21}, t_{22}\}$. There are several possible routes for net N_1. Two such routes R_1 and R_2 are shown in Figure 5.6. If the objective is to route just N_1 obviously R_1 is a better choice. However, if both N_1 and N_2 are to berouted, it is not possible to use R_1 for N_1 since it would make N_2 unroutable. Thus global routing is computationally hard since it involves trade-offs between routability of all nets and minimization of the objective function. We will consider global routing with channels and 2D-switchboxes. We will note exceptions for 3D-switchboxes, as and when appropriate. In addition, we will assume that timing constraints are translated into length constraints, hence the objective is to route each net within its length budget.

Critical nets, which must be routed in high performance layers and other nets. Fir very critical nets, global router must use a path which takes the nets from its channel (or2D-switchbox) through 3D-switchboxes to the termination point in a channel or a 2D-switchbox or C4 bump.

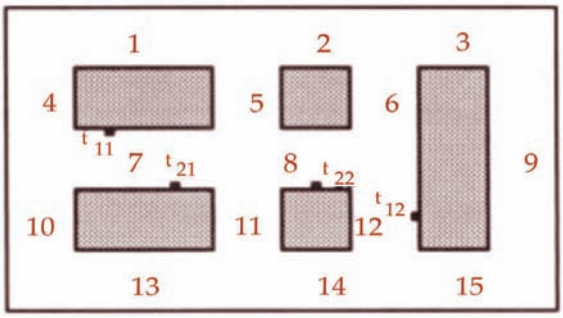


Figure 5.5: The horizontal and vertical channels.

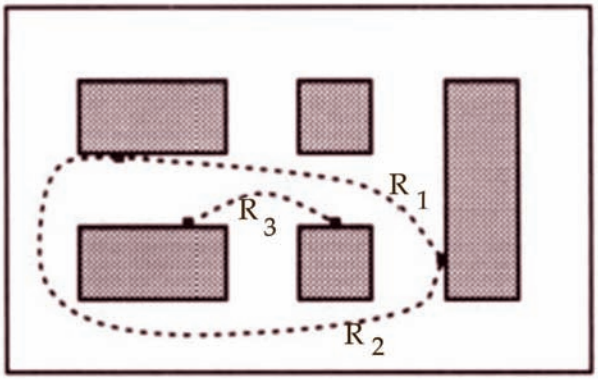


Figure 5.6: Two possible connections between source and target.

5.2 PROBLEM FORMULATION

The global routing problem is typically studied as a graph problem. The routing regions and their relationships and capacities are modeled as graphs. However, the design style strongly effects the graph models used and as a result, there are several graph models. Before presenting the problem formulation of global routing, we discuss three different graph models which are commonly used. The graph models for area routing capture the complete layout information and are used for finding exact route for each net. On the other hand, graph models for global routing capture the adjacencies and routing capacities of routing regions. We discuss three graph models viz; grid graph model, checkerboard model and the channel intersection graph model. Grid graphs are most suitable for area routing while the channel intersection graphs are most suitable for **global routing**.

Grid Graph Model

The simplest model for routing is a grid graph. The grid graph $G_1 = (V_1, E_1)$ is a representation of a layout. In this model, a layout is considered to be a collection of unit side square cells arranged in a h × w array. Each cell c_i is represented by a vertex v_i, and there is an edge between two vertices v_i, and v_j, if cells c_i and c_j are adjacent. A terminal in cell c_i is assigned to the corresponding vertex v_i. The capacity and length of each edge is set equal to one, i.e., c(e) = 1, l(e) = 1. It is quite natural to represent blocked cells by setting the capacity of the edges incident on the corresponding vertex to zero. Figure 5.7(b) shows a grid graph model for a layout in Figure 5.7(a).

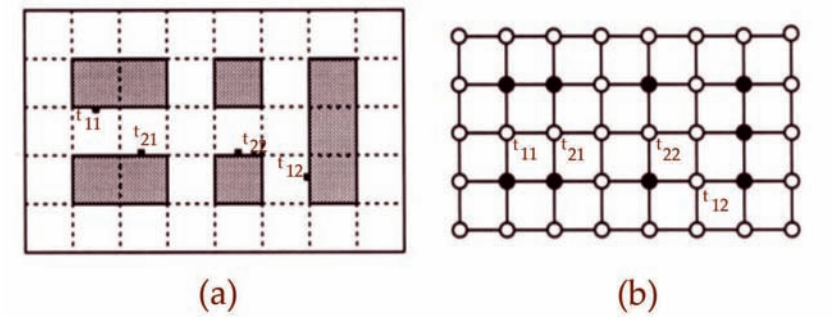


Figure 5.7: Grid Graph Model.

Given a grid graph, and a two terminal net, the routing problem is simply to find a path connecting the vertices, corresponding to the terminals, in the grid graph. Whereas, for a multi-terminal net, the problem is to find a Steiner tree in the grid graph.

The more general routing problems may consider k -dimensional grid graphs, however, the general techniques for routing essentially remain the same in all grids. In fact, routing in grids, should be considered as area routing, since the actual detailed route of the net is determined.

KEYWORD

Global routing is done to provide instructions to the detailed router about where to route every net. It provides the channels for interconnect to be routed.

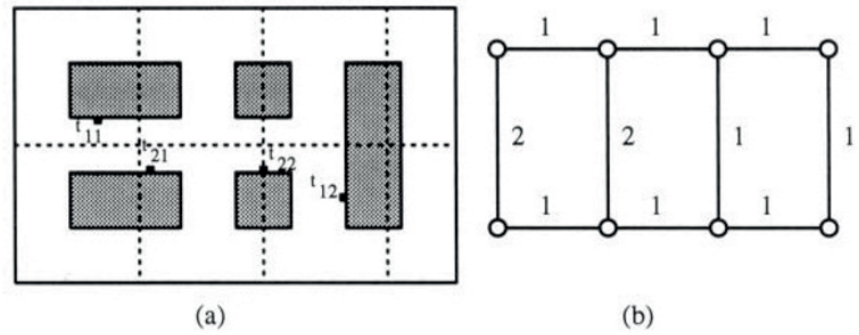


Figure 5.8: Checker Board Graph.

Checker Board Model

Checker board model is a more general model than the grid model. It approximates the entire layout area as a 'coarse grid' and all terminals located inside a coarse grid cell are assigned thatcell number. The checker board graph $G_2 = (V_2, E_2)$ is constructed in a manner analogous to grid graph. The edge capacities are computed based on the actual area available for routing on the cell boundary. Figure 5.8(b) shows a checker board graph model of a layout in Figure 5.8(a). Note that the partially blocked edges have unit capacity, whereas, the unblocked edges have a capacity of 2. Given the cell numbers of all terminals of a net, the global routing routing problem is to find a routing in the coarse grid graph.

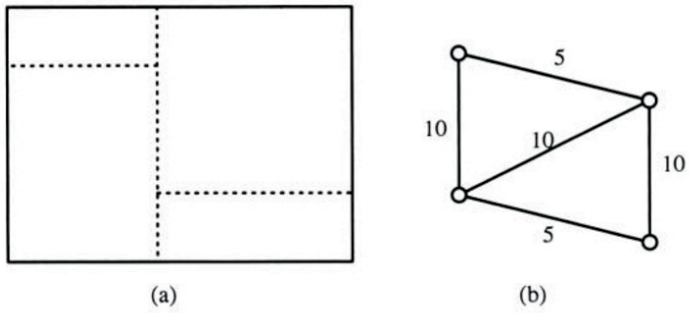


Figure 5.9: Checker Board Graph of a Floor plan.

A checker board graph can also be formed from a cut tree of floor plan. A block b_i in a floorplan is represented by a vertex and there is an edge between vertices v_i and v_j if the corresponding blocks b_i and b_j are adjacent to each other. Note that, unlike

the cells in a grid, two adjacent modules in a cut tree of a floor plan may not entirely share a boundary with each other. Figure 5.9(b) shows an example of a checker board graph for a cut tree of a floor plan in Figure 5.9(a).

Channel Intersection Graph Model

The most general and accurate model for global routing is the channel intersection model. Given a layout, we can define a channel intersection graph $G_3 = (V_3, E_3)$ where each vertex $v_i \in V_3$ represents a channel intersection CI_i. Two vertices v_i and v_j are adjacent in G_3 if there exists a channel between CI_i and CI_j. In other words, the channels appear as edges in G_3. Figure 5.10(b) shows a channel intersection graph for a layout in Figure 5.10(a). Let $c(e)$ and $l(e)$ be the capacity and length of a channel associated with edge $e \in E_3$. For example, the extended channel intersection graph in Figure 5.11(b) is obtained by adding vertices representing terminals to the channel intersection graph in Figure 5.10(b). The type of routing graph will be clear from the context and will be denoted as G =(V , E).

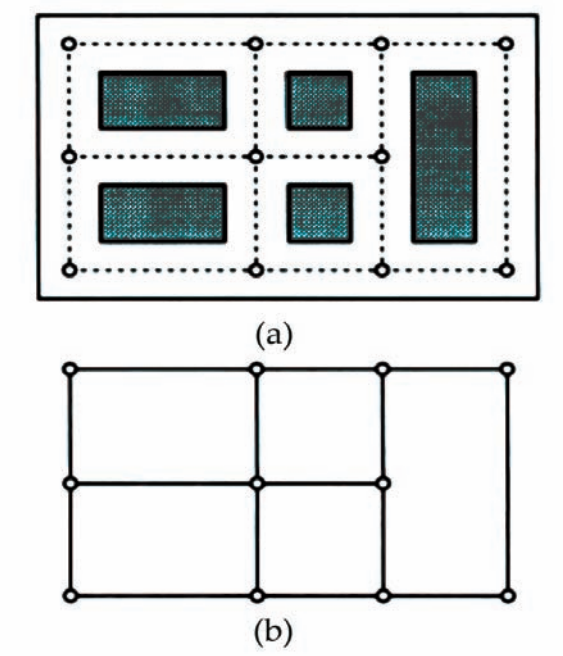


Figure 5.10: Channel intersection graph.

The channel intersection graph should be extended to include the pins as vertices so that the connections between the pins can be considered in this graph.

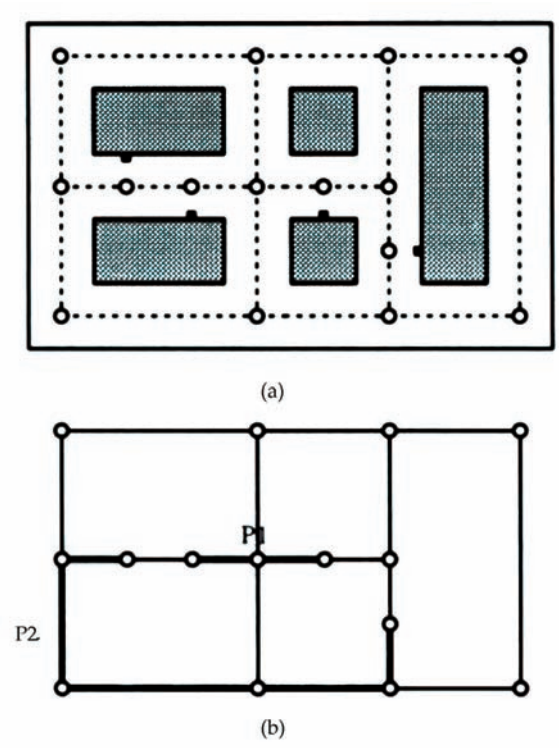


Figure 5.11: Extended channel intersection graph.

REMEMBER

The global routing problem can be viewed as a problem of finding a Steiner tree for each net in the routing graph such that the desired objective function is optimized. In addition, the capacity of the edges must not be violated.

The global routing problem of two terminal nets is to find path for each net in the routing graph such that the desired objective function is optimized. In addition, the number of nets using each edge (traffic through the corresponding channel) should not violate the capacity of that edge. For example, the global routes for nets N_1 and N_2 are shown as the paths P_1 and P_2 in Figure 5.11 (b).It is obvious from the example that routing of one net at a time causes ordering problem for nets. It is important to note that the overall optimal solution may consist of suboptimal solutions of individual nets. For a net with more than two terminals. In fact, global routing of multi-terminal nets can be formulated as a Steiner tree problem. A Steiner tree is a tree interconnecting a set of specified points called demand points and some other points called Steiner points. The number of Steiner points is arbitrary. A typical objective function is to minimize the total length of selected Steiner trees. In high-performance circuits, the

objective function is to minimize the maximum wire length of selected Steiner trees. A more precise objective function for high-performance circuits is to minimize the maximum diameter of selected Steiner trees. The diameter of a Steiner tree is defined as the maximum length of a path between any two vertices in the Steiner tree. If there is no feasible solution to an instance of a global routing problem, then the net list is not routable as the capacity constraints of some edges cannot be satisfied. In such cases, the placement phase has to be carried out again.

The formal statement of global routing problem is as follows: Given, a net-list $\mathcal{N} = \{N_1, N_2, \ldots, N_n\}$, the routing graph G = (V, E), find a Steiner tree T_i for each net $N_i, 1 \leq i \leq n$, such that, the capacity constraints are not violated, i.e., $U(e_j) \leq c(e_j)$ for all $e_j \in E$, where $U(e_j) = \sum_{i=1}^{n} x_{ij}$ is the number of wires that pass through the channel corresponding to edge e_j ($x_{ij} = 1$ if e_j in T_i it is 0 otherwise). A typical objective function is to minimize the total wire length $(\sum_{i=1}^{n} L(T_i))$, where is the length of Steiner tree T_i.

In the case of high-performance chips the objective function is to minimize the maximum wire length $(\max_{i=1}^{n} L(T_i))$ Note that minimization of maximum wire length may not directly reduce the diameter of the Steiner trees. Consider the example shown in Figure 5.12. The two Steiner trees are both of length 30,but the Steiner tree shown in Figure 5.12(b) has diameter equal to 20, which is much smaller that the diameter of the tree shown in Figure 5.12(a).

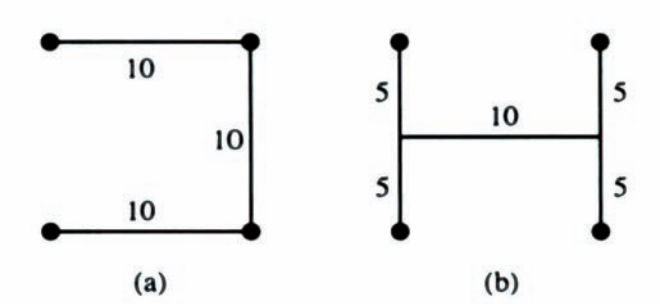


Figure 5.12: Difference between diameter and length of a net.

5.2.1 Design Style Specific Global Routing Problems

The objective of global routing in each design style is different.

Full custom

The global routing problem formulation for full custom design style is similar to the general formulation. The only difference is how capacity constraints guide the global

routing solution. In the general formulation the edge capacities cannot be violated. In full custom, since channels can be expanded, some violation of capacity constraints is allowed. However, major violation of capacities which leads to significant changes in placement are not allowed. In such case, it may be necessary to carry out the placement again.

Standard cell

In the standard cell design style, at the end of the placement phase, the location of each cell in a row is fixed. In addition, the capacity and location of each feed through is fixed. However, the channel heights are not fixed. They can be changed by varying the distance between adjacent cell rows to accommodate wires assigned by a global router. As a result, they do not have a predetermined capacity. On the other hand, feedthroughs have predetermined capacity. The area of a standard cell layout is determined by the total cell row height and the total channel height, where the total cell row height is the summation of all cell row heights and the total channel height is the summation of all channel heights. As the total cell row height is fixed, the layout area could only be minimized by minimizing the total channel height. Standard cell global routers attempt to minimize the total channel height. Other optimization functions include the minimization of the total wire length and the minimization of the maximum wire length.

The edge set of G = (V, E) are partitioned into two disjoint sets E^v and E^h i.e., $E = E^v \cup E^h$, Edges in E^v represent feedthroughs, whereas, edges in E^h represent channels. Capacity of each edge $e_j \in E^v$ is equal to the number of wires that can pass through the corresponding feedthrough. Whereas, the capacity of an edge $e \in E^h$ is set to infinity. Let E^h_{ij} represent a j^{th} edge i^{th} in channel and let $E^h_i = \cup_{\forall j} E^h_{ij}$ for all i =1, 2,…., p, where p is the total number of channels in the layout.

Thus, the global routing problem is to find a Steiner tree T_i for each net $N_i, 1 \le i \le n$, such that, the capacity constraints are not violated, i.e., $U(e_j) \le c(e_j)$ for all $e_j \in E^v$, where $U(e_j) = \sum_{i=1}^{n} x_{ij}$ is the number of wires that go through the feedthrough corresponding to edge e_j ($x_{ij} = 1$ if e_j is in T_i it is 0 otherwise). The optimization function is either tominimize the total wire length $(\sum_{i=1}^{n} L(T_i))$ or to minimize the maxi-mum wire length $(\max_{i=1}^{n} L(T_i))$ or to minimize the total channel height $\sum_{i=1}^{p} \max\{U(e)|e \in E^h_i\}$. $L(T_i)$ is the length of Steiner tree T_i.

If there is no feasible solution for a global routing problem, feedthrough capacities are not sufficient, (see Figure 5.13.) Additional feedthroughs should be inserted in order to allow global routing. Recently, a new approach, called over-the-cell routing, has been presented for standard cell design, in which, in addition to the channels and feedthroughs the over-the-cell areas are available for routing.

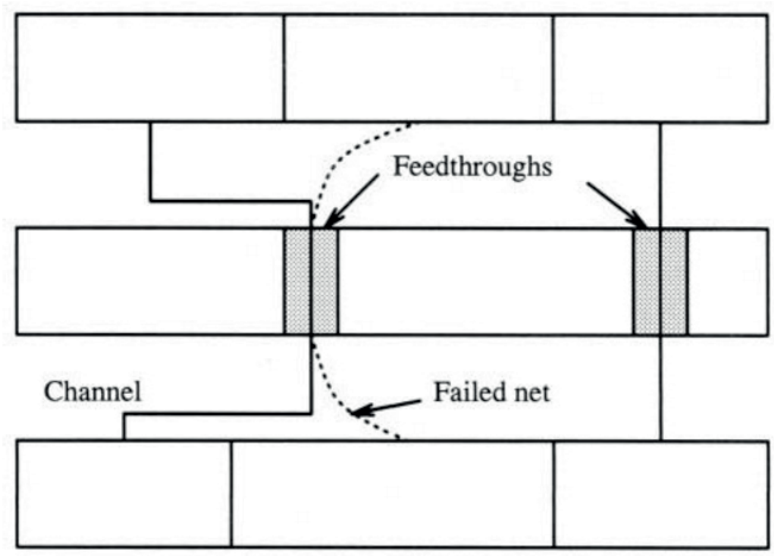


Figure 5.13: Not enough feedthroughs in standard cell design style.

Gate Array

In gate array design style, the size and location of all cells and the routing channels and their capacities are fixed by the architecture. This is the key difference between gate array and other design styles. Unlike the full custom design style and standard cell design style the primary objective of the global routing in gate arrays is to guarantee routability.

The secondary objective may be to minimize the total wire length or to minimize the maximum wire length. Other than these objectives, the formulation of global routing problem in **gate array** design style is same as the general global routing formulation. If there is no feasible solution to a given instance of global routing problem, the net list cannot be routed (see Figure 5.14). In this case, the placement phase has to be carried out again as the capacity of routing channels is fixed in gate array design style.

Gate array is an approach to the design and manufacture of application-specific integrated circuits (ASICs), using a prefabricated chip with active devices like NAND-gates, that are later interconnected according to a custom order by adding metal layers in the factory.

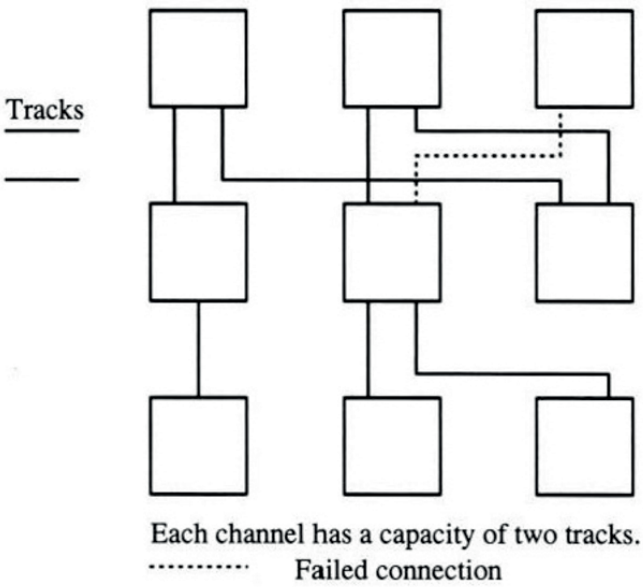


Figure 5.14: Not all nets are routable in gate array design style.

5.3 CLASSIFICATION OF GLOBAL ROUTING ALGORITHMS

Basically, there are two kinds of approaches to solve global routing problem; the sequential and the concurrent.

Sequential Approach

In this approach, as the name suggests, nets are routed one by one. However, once a net has been routed it may block other nets which are yet to be routed. As a result, this approach is very sensitive to the order in which the nets are considered for routing. Usually, the nets are sequenced according to their criticality, half perimeter of the bounding rectangle and number of terminals. The criticality of a net is determined by the importance of the net. For example, clock net may determine the performance of the circuit and therefore it is considered to be a very important net. As a result, it is assigned a high criticality number. The nets on the critical paths are assigned high criticality numbers since they also play a key role in determining the performance of the circuit. The criticality number and other factors can be used to sequence nets. However, sequencing techniques do not solve the net ordering problem satisfactorily. In a practical router, in addition to a net ordering scheme an improvement phase is used to remove blockages when further routing of nets is not possible. However, this also

may not over-come the shortcoming of sequential approach. One such improvement phase involves 'rip-up and reroute' technique, while other involves 'shove-aside' technique. In 'rip-up and reroute', the interfering wires are ripped up, and rerouted to allow routing of the affected nets. Whereas, in 'Shove-Aside' technique, wires that will allow completion of failed connections are moved aside without breaking the existing connections. Another approach is to first route simple nets consisting of only two or three terminals since there are few choices for routing such nets. Usually such nets comprise a large portion of the nets (up to 75%) in a typical design. After the simple nets have been routed, a Steiner tree algorithm is used to route intermediate nets. Finally, a maze routing algorithm is used to route the remaining multi-terminal nets (such as power, ground, clock etc.) which are not too numerous.

The sequential approach includes:

- Two-terminal algorithms:
  - Maze routing algorithms
  - Line-probe algorithms
  - Shortest path based algorithms
- Multi-terminal algorithms:
  - Steiner tree based algorithms

Concurrent Approach

This approach avoids the ordering problem by considering routing of all the nets simultaneously. The concurrent approach is computationally hard and no efficient polynomial algorithms are known even for two-terminal nets. As a result, integer programming methods have been suggested. The corresponding integer program is usually too large to be employed efficiently. Hence, hierarchical methods that work top down are employed to partition the problem into smaller subproblems, which can be solved by integer programming.

5.4 MAZE ROUTING ALGORITHMS

Lee introduced an algorithm for routing a two terminal net on a grid in 1961. Since then, the basic algorithm has been improved for both speed and memory requirements. Lee's algorithm and its various improved versions form the class of maze routing algorithms.

Maze routing algorithms are used to find a path between a pair of points, called the source(s) and the target (t) respectively, in a planar rectangular grid graph. The geometric regularity in the standard cell and gate array design style lead us to model the whole plane as a grid. The areas available for routing are represented as unblocked

vertices, whereas, the obstacles are represented as blocked vertices. The objective of a maze routing algorithm is to find a path between the source and the target vertex without using any blocked vertex. The process of finding a path begins with the exploration phase, in which several paths start at the source, and are expanded until one of them reaches the target. Once the target is reached, the vertices need to be retraced to the source to identify the path. The retrace phase can be easily implemented as long as the information about the parentage of each vertex is kept during the exploration phase. Several methods of path exploration have been developed.

DID YOU KNOW?

The Lee algorithm is one possible solution for maze routing problems based on Breadth-first search. It always gives an optimal solution, if one exists, but is slow and requires considerable memory.

5.4.1 Lee's Algorithm

This algorithm, which was developed by Lee in 1961, is the most widely used algorithm for finding a path between any two vertices on a planar rectangular grid. The key to the popularity of Lee's maze router is its simplicity and its guarantee of finding an optimal solution if one exists.

The exploration phase of **Lee's algorithm** is an improved version of the breadth-first search. The search can be visualized as a wave propagating from the source. The source is labeled '0' and the wave front propagates to all the unblocked vertices adjacent to the source. Every unblocked vertex adjacent to the source is marked with a label '1'. Then, every unblocked vertex adjacent to vertices with a label '1' is marked with a label '2', and so on. This process continues until the target vertex is reached or no further expansion of the wave can be carried out. An example of the algorithm is shown in Figure 5.15. Due to the breadth-first nature of the search, Lee's maze router is guaranteed to find a path between the source and target, if one exists. In addition, it is guaranteed to be the shortest path between the vertices.

The input to the Lee's Algorithm is an array B, the source (s) and target (t) vertex. B[v], denotes if a vertex v is blocked or unblocked. The algorithm uses an array L, where L[v] denotes the distance from the source to the vertex v. This array will be used in the procedure RETRACE that retraces the vertices to form a path P, which is the output of the Lee's Algorithm. Two linked lists plist (Propagation list) and nlist (Neighbor list) are used to keep track of the vertices on the wave front

and their neighbor vertices respectively. These two lists are always retrieved from tail to head. We also assume that the neighbors of a vertex are visited in counter-clockwise order that is top, left, bottom and then right.

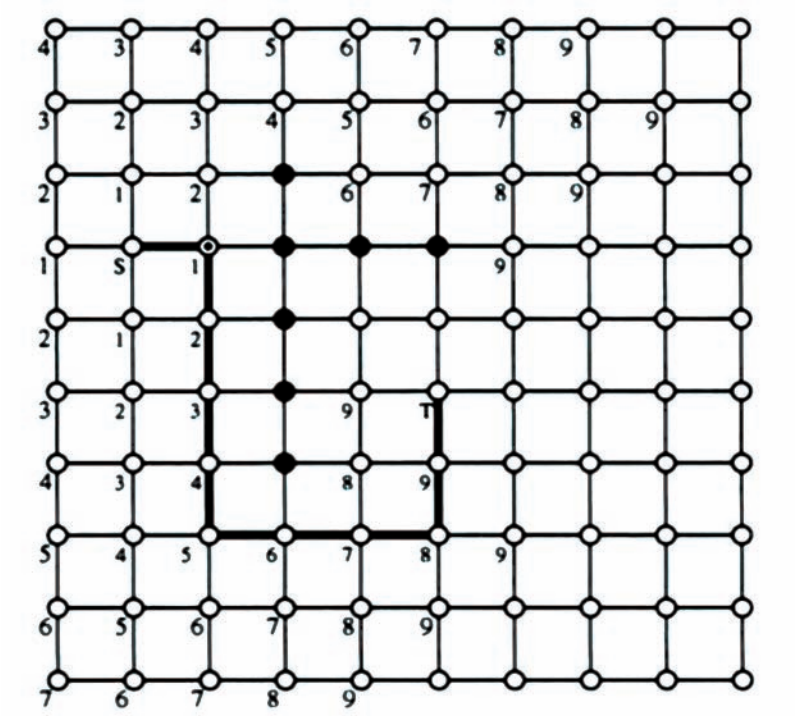


Figure 5.15: A net routed by Lee's algorithm

The formal description of the Lee's Algorithm appears in Figure 5.16. The time and space complexity of Lee's algorithm O (h × w) is for a grid of dimension h × w.

The Lee's routing algorithm requires a large amount of storage space and its performance degrades rapidly when the size of the grid increases. There have been numerous attempts to modify the algorithm to improve its performance and reduce its memory requirements.

KEYWORD

Lee's algorithm is a path finding algorithm and one possible solution for maze routing, which is often used in computer-aided design systems to route wires on printed circuit boards and .

```
Algorithm LEE-ROUTER (B, s, t, P)
    input: B, s, t
    output: P
begin
    plist = s;
    nlist = φ;
    temp = 1;
    path_exists = FALSE;
    while plist ≠ φ do
        for each vertex v_i in plist do
            for each vertex v_j neighboring v_i do
            if B[v_j] = UNBLOCKED then
                L[v_j] = temp;
                INSERT(v_j,nlist);
                if v_j = t then
                    path_exists = TRUE;
                    exit while;
        temp=temp + 1;
        plist = nlist;
        nlist = φ;
    if path_exists = TRUE then RETRACE (L, P);
    else path does not exist;
end.
```

Figure 5.16: Algorithm LEE-ROUTER.

Lee's algorithm requires up to k+1 bits per vertex, where k bits are used to label the vertex during the exploration phase and an additional bit is needed to indicate whether the vertex is blocked. For an h × w grid, $k = \log_2(h \times w)$ Acker [Ake67] noticed that, in the retrace phase of Lee's algorithm, only two types of neighbors of a vertex need to be distinguished; vertices toward the target and vertices toward the source. This information can be coded in a single bit for each vertex. The vertices in wave front L are always adjacent to the vertices in wave front L –1 and L + 1. Thus, during wave propagation, instead of using a sequence 1, 2, 3,..., the wave fronts are labeled by a sequence like 0, 0, 1, 1, 0, 0, …. The predecessor of any wave front is labeled differently from its successor. Thus, each scanned vertex is either labeled '0' or '1'. Besides these two states, additional states ('block' and 'unblocked') are needed for each vertex. These four states of each vertex can be represented by using exactly two bits, regardless of the problem size. Compared to Acker's scheme, Lee's algorithm requires at least 12 bits per vertex for a grid size of 2000 × 2000.

It is important to note that Acker's coding scheme only reduces the memory requirement per vertex. It inherits the search space of Lee's original routing algorithm, which is O(h × w) in the worst case.

5.4.2 Soukup's Algorithm

Lee's algorithm explores the grid symmetrically, searching equally in the directions away from target as well as in the directions towards it. Thus, Lee's algorithm requires a large search time. In order to overcome this limitation, Soukup proposed an iterative algorithm in 1978. During each iteration, the algorithm explores in the direction toward the target without changing the direction until it reaches the target or an obstacle, otherwise it goes away from the target. If the target is reached, the exploration phase ends. If the target is not reached, the search is conducted iteratively. If the search goes away from the target, the algorithm simply changes the direction so that it goes towards the target and a new iteration begins. However, if an obstacle is reached, the breadth-first search is employed until a vertex is found which can be used to continue the search in the direction toward the target. Then, a new iteration begins. Figure 5.17 illustrates the Soukup's algorithm with an example. In Figure 5.17, the number near a vertex indicates the order in which that vertex was visited.

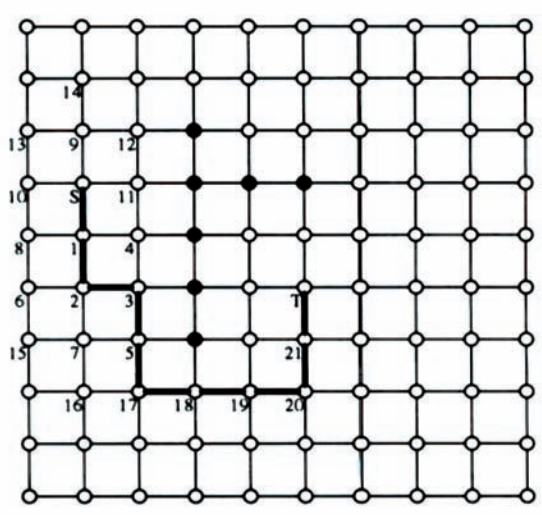


Figure 5.17: A net routed by Soukup's router.

Figure 5.18 contains the formal description of Soukup's Algorithm. The notation used in the algorithm is similar to that used in the Lee's algorithm except for the array *L*.

We use L[v] to denote the order in which the vertex v is visited during the exploration phase in this algorithm. Function DIR(v_1, v_2) returns the direction from v_1 to v_2 Function NGHBR-IN-DIR(v_1,v_2) returnsthe neighbor of which is in the direction from v_1 to v_2.

The Soukup's Algorithm improves the speed of Lee's algorithm by a factor of 10 to 50. It guarantees finding a path if a path between source and target exits. However, this path may not be the shortest one. The search method for this algorithm is a combined breadth-first and depth-first search. The worst case time and space complexities for this algorithm are both O(h × w), for a grid of size h × w.

5.4.3 Hadlock's Algorithm

An alternative approach to improve upon the speed was suggested by Hadlock in 1977. The algorithm is called Hadlock's minimum detour algorithm. This algorithm uses A^* search method.

```
Algorithm SOUKUP-ROUTER (B, s, t, P)
    input: B, s, t
    output: P
begin
    plist = s;
    nlist = φ;
    temp = 1;
    path_exists = FALSE;
    while plist ≠ φ do
        for each vertex v_i in plist do
            for each vertex v_j neighboring v_i do
                if v_j = t then
                    L[v_j] = temp;
                    path_exists = TRUE;
                    exit while;
                if B[v_j] = UNBLOCKED then
                (* If the direction of the search is toward the
                target, the search continues in this direction *)
                    if DIR(v_i, v_j) = TO-TARGET
                    then L[v_j] = temp;
                        temp = temp + 1;
```

```
                    INSERT (v_j, plist);
                    while B[NGHBR-IN-DIR(v_i, v_j)]=
                        UNBLOCKED do
                        v_j = NGHBR-IN-DIR(v_i, v_j);
                        L[v_j] = temp;
                        temp = temp + 1;
                        INSERT (v_j, plist);
                else
                    L[v_j] = temp;
                    temp = temp + 1;
                    INSERT (v_j, nlist);
            plist = nlist;
            nlist = φ;
        if path_exists = TRUE then RETRACE (L, P);
        else path does not exist;
    end.
```

Figure 5.18: Algorithm SOUKUP-ROUTER.

```
Algorithm HADLOCK-ROUTER(B, s, t, P)
    input: B, s, t
    output: P
begin
    plist = s;
    nlist = φ;
    detour = 0;
    path_exists = FALSE;
    while plist ≠ φ do
        for each vertex v_i in plist do
            for all vertices v_j neighboring v_i do
                if B[v_j] = UNBLOCKED then
                    D[v_j] = DETOUR-NUMBER(v_j);
                    INSERT (v_j,nlist);
                    if v_j = t then
                        path_exists = TRUE;
                        exit while;
        if nlist = φ then
            path_exists = FALSE;
            exit while;
        detour = MINIMUM-DETOUR( nlist );
        for each vertex v_k in nlist do
            if D[v_k] = detour then INSERT(v_k, plist);
        DELETE (nlist, plist);
    if path_exists = TRUE then RETRACE (L, P);
    else path does not exist;
end.
```

Figure 5.19: Algorithm HADLOCK-ROUTER.

Hadlock observed that the length of a path (P) connecting source and target can be given by $M(s,t)+2d(P)$, where M(s, t) is Manhattan distance between source and target and d(P) is the number of vertices on path P that are directed away from the target. The length of P is minimized if and only if d is minimized as M(s, t) is constant for given pair of source and target. This is the essence of Hadlock's algorithm. The exploration phase, instead of labeling the wave front by a number corresponding to the distance from the source, uses the detour number. The detour number of a path is the number of times that the path has turned away from the target. Figure 5.20 illustrates the Hadlock's algorithm with an example. In Figure 5.20, the number near a vertex indicates the order in which that vertex was visited.

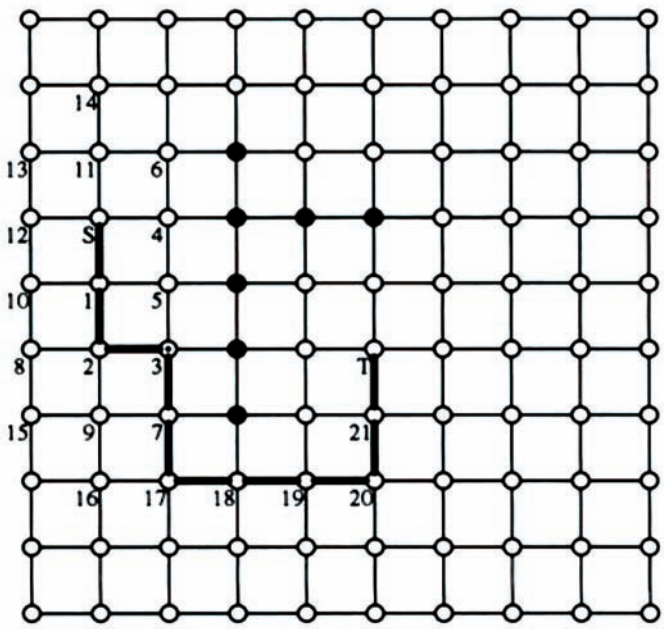


Figure 5.20: A net routed by Hadlock's Algorithm.

A formal description of Hadlock's Algorithm is given in Figure 5.19. Function DETOUR-NUMBER (v) returns detour number of a vertex v. Procedure DELETE (*nlist, plist)* deletes the vertices which are in *plist* from *nlist.* Function MINIMUM-DETOUR (nlist) returns the minimum detour number among all vertices in the list *nlist.*

The worst case time and space complexity of Hadlock's algorithm is O (h × w) for a grid of size h ×w.

5.4.4 Comparison of Maze Routing Algorithms

Maze routing algorithms are grid based methods. The time and space required by these algorithms depend linearly on their search space.

The search in Lee's algorithm is conducted by using a wave propagating from the source. The algorithm searches symmetrically in every direction, using the breath-first search technique. Thus, it guarantees finding a shortest path between any two vertices if such a path exists. However, the worst case happens when the source is located at the center and the target is located at a corner of routing area, in which all the vertices have to be scanned before the target is reached, (see Figure 5.21.)

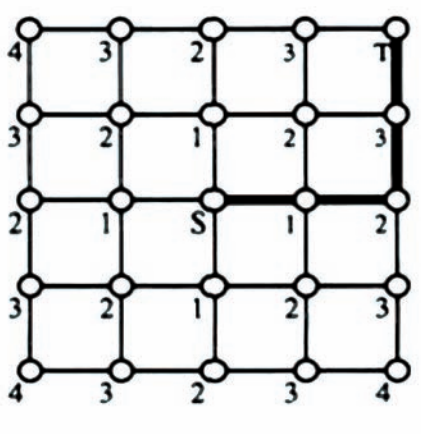


Figure 5.21: Lee's algorithm in the worst case.

The Soukup's algorithm remedies the shortcoming of the breadth-first search method by using a depth-first search until an obstacle is encountered. If an obstacle is encountered, a breadth-first search method is used to get around the obstacle. The search time in Soukup's algorithm is usually smaller than the Lee's algorithm due to the nature of depth-first search method. However, this algorithm may not find a shortest path between the source and target. In Figure 5.22, the Soukup's algorithm explores all the vertices and does not find the shortest path between s and t.

The worst case of Soukup's algorithm occurs when the search goes in the direction of the target, which is opposite the direction of the passageway through the obstacle. Figure 5.23 shows an example in which Soukup's algorithm scans all vertices while finding a path between s and t.

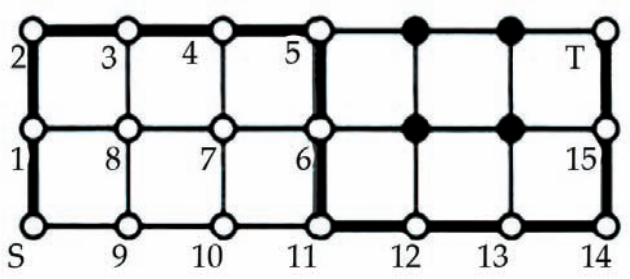


Figure 5.22: Soukup's algorithm does not find the shortest path.

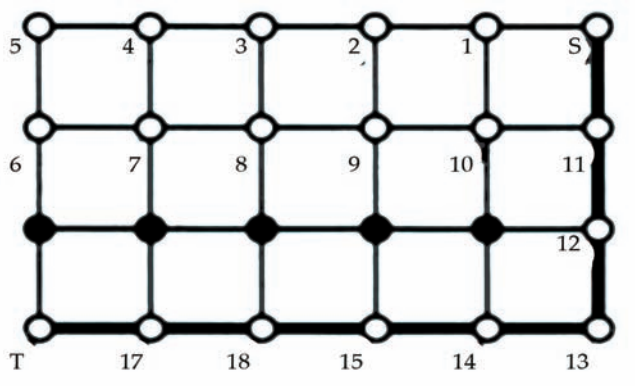


Figure 5.23: Soukup's algorithm in the worst case.

The Hadlock's algorithm aims at both reducing the search time and finding an optimal path between given two vertices. Basically, the Hadlock's algorithm is a breadth-first search method. As a result, it finds a shortest path if one exists. The difference between the Hadlock's algorithm and Lee's algorithm is the way in which the wave front is labeled. The Hadlock's algorithm label the wave front by the detour number instead of the distance from the source used in the Lee's algorithm. In this way, the search can prefer the direction toward the target to the direction away from the target. This search time is shorter than the Lee's algorithm. When the direction of the search goes toward the target and opposite the passageway through the obstacle, the worst case happens (see Figure 5.24).

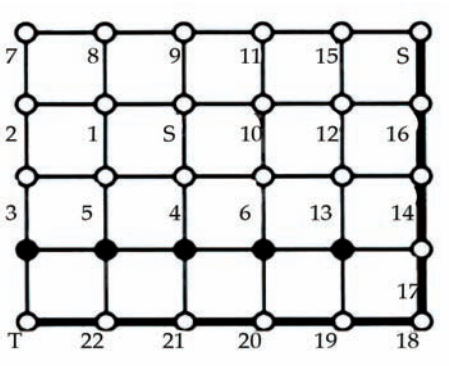


Figure 5.24: Hadlock's algorithm in the worst case.

All the maze routers and many of their variations are grid based methods. Information must be kept for each grid node. Thus, a very large memory space is needed to implement these algorithms for a large grid. To give anapproximate estimate, a chip of size $10000\lambda \times 10000\lambda$ requires as much as 350MBytes of memory and 66 seconds to route one net on a 15MIPS workstation. There may be 5000 to 10000 nets in a typical chip. Such numbers make these maze routing algorithms infeasible for large chips. In order to reduce the large memory requirements and run times, line-probe algorithms were developed.

5.5 LINE-PROBE ALGORITHMS

The line-probe algorithms were developed independently by Mikami and Tabuchi in 1968, and Hightower in 1969. The basic idea of a line probe algorithm is to reduce the size of memory requirement by using line segments instead of grid nodes in the search. The time and space complexities of these line-probe algorithms is O (L), where L is the number of line segments produced by these algorithms.

The basic operations of these algorithms are as follows. Initially, lists slist and tlist contain the line segments generated from the source and target respectively. The generated line segments do not pass through any obstacle. If a line segment from slist intersects with a line segment in tlist, the exploration phase ends; otherwise, the exploration phase proceeds iteratively. During each iteration, new line segments are generated. These segments originate from 'escape' points on existing line segments in slist and tlist. The new line segments generated from slist are appended to slist. Similarly, segments generated from a segment in tlist are appended. To tlist. If a line segment from slist intersects with a line segment from tlist, then the exploration phase ends. The path can be formed by retracing the line segments in set tlist, starting from the target, and then going through the intersection, and finally retracing the line segments in set slist until the source is reached.

The data structures used to implement these algorithms play an important role in the efficiency considerations of the search for obstructions to probes. Typically two lists, one for the horizontal lines and one for the vertical lines are used. The use of two separate lists allows lines parallel to the direction of the probe to be ignored, thus expediting the search.

The Mikami and the Hightower algorithms differ only in the process of choosing escape points. In Mikami's algorithm, every grid node on the line segment is an 'escape' point, which generates new perpendicular line segments. This search is similar to the breadth-first search, and is guaranteed to find a path if one exists. However, the path may not be the shortest one. Figure 5.25 shows a path generated by Mikami's algorithm. On the other hand, Hightower's algorithm makes use of only a single 'escape' point on each line segment. In the simple case of a probe parallel to the blocked vertices, the escape point is placed just past the endpoint of the segment. Figure 5.26 shows a path generated by Hightower's algorithm. Designed to help the router find a path around different types of obstacles. The disadvantage in generating fewer escape points in Hightower's algorithm essentially means that it may not be able to find a path joining two points even when such a path exists.

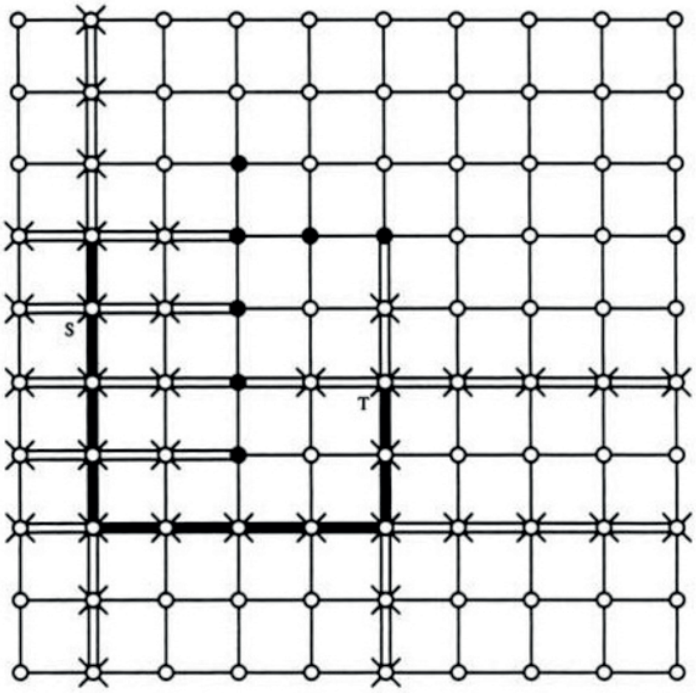


Figure 5.25: A net routed by Mikami-Tabuchi's Algorithm.

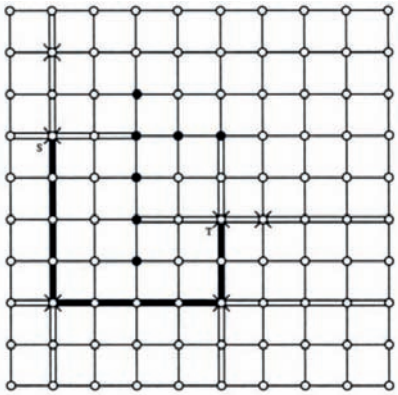


Figure 5.26: A net routed by Hightower's Algorithm.

A formal description of these two algorithm is given in Figure 5.27. (As these two algorithms basically are the same, we just use one description for both of them.) Procedure GENERATE (l, e) generates a line-probe l from an escape point e, whereas, INSERT (l, list) adds a line-probe l to the list. Function $\text{INTERSECT}(l_i, l_j)$ returns TRUE if line-probes l_i and l_i intersect, it returns FALSE otherwise.

```
Algorithm LINE-PROBE-ROUTER(s, t, P)
    input: s, t
    output: P
begin
    new_slist = line segments generated from s;
    new_tlist = line segments generated from t;
    while new_slist ≠ φ and tlist ≠ φ do
        slist = new_slist;
        tlist = new_tlist;
        for each line segment l_i in slist do
            for each line segment l_j in tlist do
                if INTERSECT(l_i, l_j)=TRUE then
                    path_exists = TRUE;
                    exit while;
        new_slist = φ;
        for each line segment l_i in slist do
            for each escape point e on l_i do
                GENERATE(l_k, e);
                INSERT(l_k, new_slist);
        new_tlist = φ;
        for each line segment l_i in tlist do
            for each escape point e on l_i do
                GENERATE(l_k, e);
                INSERT(l_k, new_tlist);
    if path_exists=TRUE then RETRACE;
    else a path can not be found;
end.
```

Figure 5.27: Algorithm LINE-PROBE-ROUTER.

Maze routers and many of their variations are grid based methods. Information must be kept for each grid node. Thus, a very large memory space is needed to implement these algorithms for a large grid. The line-probe algorithms, however, require the information to be kept for each line segment. Since the number of line segments is very small compared to the nodes in a grid, the required memory is greatly reduced. The key difference between the two line probe algorithms is that, the Mikami's algorithm can find a path between any two vertices if one exists. This path may not be the shortest path. Hightower's algorithm may not be able to find a path joining two vertices even if such a path exists. A comparison of the maze routing algorithms and line-probe algorithms in their worst cases is given in Table 5.1. ($h \times w$ denotes the size of grid and L denotes the number of line segments generated in line-probe algorithms).

Table 5.1: Comparison of different algorithms

	Algorithms				
	Maze Routing			Line-Probe	
	Lee	Soukup	Hadlock	Mikami	Hightower
Time complexity	$h \times w$	$h \times w$	$h \times w$	L	L
Space complexity	$h \times w$	$h \times w$	$h \times w$	L	L
Finds path if one exists?	yes	yes	yes	yes	no
Is the path shortest?	yes	no	yes	yes	no

5.5.1 Shortest Path Based Algorithms

A simple approach to route a two-terminal net uses Dikjstra's shortest algorithm. Given, a routing graph $G = (V, E)$, a source vertex $s \in V$ and a target vertex $t \in V$ a shortest path in G joining s and t can be found in $O(|V|^2)$ time. The algorithm in Figure 5.28 gives formal description of analgorithm based on Dijkstra's shortest path algorithm for global routing a set N of two-terminal nets in a routing graph G.

The output of the algorithm is a set P of paths for the nets in N. A path $P_i \in \mathcal{P}$ gives a path for net $N_i \in \mathcal{N}$. The time complexity of the algorithm SHORT-PATH-GLOBAL-ROUTER is $O(\mathcal{N}|V|^2)$.

Note that the length of an edge is increased by a factor $\alpha > 1$ whenever a congested edge is utilized in the path of a net. This algorithm is suitable for channel intersection graph, since it assumes that congested channels can be expanded. If the edge congestions are strict, the algorithm can be modified to use 'rip-up and reroute' or 'shove aside' techniques.

```
Algorithm SHORT-PATH-GLOBAL-ROUTER(G, N, P)
    input: G, N
    output: P
begin
    for each N_i in N do
        P_i = DIJKSTRA-SHORT-PATH(G, N_i);
        for each e_j in P_i do
            c(e_j) = c(e_j) − 1;
            if c(e_j) < 0 then l(e_j) = α × l(e_j);
end.
```

Figure 5.28: Algorithm SHORT-PATH-GLOBAL-ROUTER.

5.6 STEINER TREE BASED ALGORITHMS

Global routing algorithms presented so far are not suitable for global routing of multi-terminal nets. Several approaches have been proposed to extend maze routing and line-probe algorithms for routing multi-terminal nets. In one approach, the multi-terminal nets are decomposed into several two-terminal nets and the resulting two-terminal nets are routed by using a maze routing or line-probe algorithm. The quality of routing, in this approach, is dependent on how the nets are decomposed. This approach produces suboptimal results as there is hardly any interaction between the decomposition and the actual routing phase. In another approach, the exploration can be carried out from several terminals at a time. It allows the expansion process to determine which pairs of pins to connect, rather than forcing a predetermined net decomposition. However, the maze routing and line-probe algorithms cannot optimally connect the pins. In addition, these approaches inherit the large time and space complexities of maze routing and line-probe algorithms.

The natural approach for routing multi-terminal nets is Steiner tree approach. Usually *Rectilinear Steiner Trees* (RST) are used. A rectilinear Steiner tree is a Steiner tree with only rectilinear edges. The length of a tree is the sum of lengths of all the edges in the tree. It is also called the cost of the tree. The problem of finding a minimum cost RST is NP-hard. In view of NP-hardness of the problem, several heuristic algorithms have been developed. Most of the heuristic algorithms depend on minimum cost spanning tree. This is due to a special relationship between Steiner trees and minimum cost spanning trees. Hwang has shown that the ratio of the cost of a **minimum spanning tree (MST)** to that of an optimal RST is no greater than $\frac{3}{2}$.

Minimum Spanning Tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight

Let S be a net to be routed. We define an underlying grid *G*(*S*) of *S* (on an oriented plane) as the grid obtained by drawing horizontal and vertical lines through each point of *S*. Let be a complete graph for *S*. An MST for net *S* is a minimum spanning tree. Note that, there may be several MST's for a given net and they can be foundeasily. Using Hwang's result, an approximation of the optimal RST can be obtained by recti linearizing each edge of an MST. Different ways of recti linearizing the edges of *T* give different approximations. If an edge) of *T* isrectilinearized as a shortest path between and on the underlying grid *G*(*S*), then it is called as a staircase edge layout. For example, all the edge layouts are staircase layouts. A staircase layout with exactly one turn on the grid G(S) is called as an L-shaped layout. A staircase layout having exactly two turns on the grid *G*(*S*) is called as a Z-shaped layout. For example, theedge layout of are L-shaped and Z-shaped layoutsrespectively. An RST obtained from an MST *T* of a net S, by rectilinearizing each edge of *T* using staircase layouts on G(S) is called S-RST. An S-RST of T, in which the layout of each MST edge is an L-shaped layout is called an L-RST of *T.* An S-RST of *T*, in which the layout of each MST edge is a Z-shaped layout is called a Z-RST of *T*. An optimal S-RST (Z-RST, L-RST) is an S-RST(Z-RST, L-RST) of the least cost among all S-RST's (Z-RST's, L-RST's). It is easy to see that an optimal L-RST may have a cost larger than an optimal S-RST, which in turn may have a cost larger than the optimal RST. Obviously, least restriction on the edge layout gives best approximation. However, as the number of steps allowed per edge is increased it becomes more difficult to design an efficient algorithm to find the optimal solution.

5.6.1 Non-Rectilinear Steiner Tree Based Algorithm

Burman, Chen, and Sherwani studied the problem of global routingof multi-terminal nets in a generalized geometry called δ-geometry in order toimprove the layout and consequently enhance the performance. The restriction of layout to rectilinear geometry, and thus only rectilinear Steiner trees, in the previous Steiner tree based global routing algorithms was necessary to account for restricted computing capabilities. Recently, because of enhanced computing capabilities and the need for design

of high performance circuit, non-rectilinear geometry has gained ground. In order to obtain smaller length Steiner trees, theconcept of separable MST's in δ-geometry was introduced. In edges with angles $i\pi/\delta$ for all i, are allowed, where $\delta\ (\geq 2)$ is a positive integer. $\delta = 2$, 4 and ∞ correspond to rectilinear, 45° and Euclidean geometries respectively.Obviously, we can see that δ-geometry always includes rectilinear edges and is a useful with respect to the fabrication technologies. It has been proved that for an even $\delta \geq 4$, all minimum cost spanning trees in δ-geometry satisfy the separability property.

```
Function LEAST-COST(z, T_e, CostM_z[e], M_z[e])
    input: z, T_e
    output: CostM_z[e], M_z[e]
begin
    if CHILD-EDGES-NUM(e) ≠ 0 then
        for each child edge e_i of e do
            for each layout z_ij of e_i do
                LEAST-COST(z_ij, T_e_i, CostM_z_ij[e_i], M_z_ij[e_i]);
        for (each combination of the layouts containing one
            optimal Z-RST layout for each T_e_i) do
            merge layouts in the combination with the layout z
            of edge e;
            calculate the resulting cost of merged layout;
        CostM_z[e] = minimum cost among all merged layouts;
        M_z[e] = the layout corresponding to the minimum cost;
    else (* The bottom edge is reached *)
        M_z[e] = z;
        CostM_z[e] = cost of z;
end.
```

Figure 5.29: Function LEAST-COST.

```
Algorithm Z-RST (r, T_r, CostM, M)
    input: r, T_r
    output: CostM, M
begin
    CostM = ∞;
    for each z-shaped layout z of r do
        LEAST-COST (z, T_r, CostTempM, TempM);
        if CostTempM < CostM then
            M = TempM;
            CostM = CostTempM;
end.
```

Figure 5.30: Algorithm Z-RST.

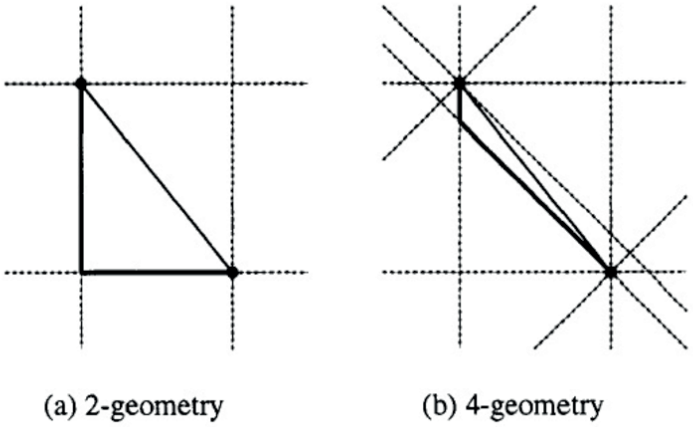


Figure 5.31: Comparison of the rectilinearizationin 2-geometry and in 4- geometry.

Theorem 7 Any minimum spanning tree for a given point set in the plane is δ-geometry for any even $\delta \geq 4$.

Therefore, there exists a polynomial time algorithm to find an optimal Steiner tree in δ-geometry which is derivable from the separable minimumspanning tree. The experiments have shown that tree length can be reducedup to 10-12% by using 4-geometry as compared to rectilinear geometry (2-geometry). Moreover, length reduction is quite marginal for higher geometries. As a consequence, it is sufficient and effective to consider layouts in 4-geometryin the consideration of global routing problem. An example of the derivation of a Steiner tree for a simple two-terminal net in 4-geometry is shown in Figure 5.31(b) while Figure 5.32(a) shows the derivation of a Steiner tree for the same net in rectilinear geometry. Clearly, the tree length in 4-geometry is shorter than the one in the rectilinear geometry.

5.6.2 Steiner Min-Max Tree based Algorithm

The approach in uses a restricted case of Steiner tree in global routing problem, called Steiner Min-Max Tree (SMMT) in which the maximum weight edge is minimized (real vertices represent channels containing terminals of a net, Steiner vertices represent intermediate channels, weights correspondto densities). They give an $O(\min\{|E| \log\log |E|, |V|^2\})$ time algorithm for obtaining a Steiner min-max tree in a weighted coarse grid graph G = (V, E). The weight of an edge in E is a function of current density, capacity, and measures crowdedness of a border. Each vertex in V is labeled with demand or (potential) Steiner depending on whether it is respectively a terminal of net N_i or not. A Steiner min-max tree of *G* dictates a global routing that

minimizes traffic in the densest channel. While the Steiner min-max tree method tends toroute nets through less crowded channels, it is also desirable to have nets withshort length. Therefore, among all Steiner min-max trees of the given net, weare interested in those with minimum length. The problem of finding a Steinermin-max tree whose total length is minimized is NP-hard.

```
Algorithm SMMT(G, d, T)
    input:G, d
    output:T
begin
    T = An MST of G;
    while EXIST-ODSV(T, d)=TRUE do
        v = GET-ODSV(T, d); (* a Steiner leaf*)
        REMOVE(v, T);
    output T;
end.
```

Figure 5.33: Algorithm SMMT.

Given, a weighted coarse grid graph G = (V, E) and a boolean array d such that d[v] is true if the vertex $v \in V$ corresponds to a terminals of N_i. An SMMT T of N_i can be obtained using algorithm in Figure 5.33. Function EXIST-ODSV (T, d) returns TRUE if there exists a one-degree Steiner vertexin T. Function GET-ODSV (T, d) returns a one-degree Steiner vertex from T. REMOVE(v, T) removes vertex v and edges incident on it from T.

Theorem 8 Algorithm SMMT correctly computes a Steiner min-max tree of net in weighted grid graph G = (V, E) in $O(\min\{|E| \log\log |E|, |V|^2\})$ time.

A number of heuristics have been incorporated in the global router based on the min-max Steiner trees. The nets are ordered first according to their priority, length and multiplicity numbers. The global routing is then performed in two phases: the SMMT-phase and the SP-phase. The SP-phase is essentially a minimum-spanning tree algorithm. The SMMT-phase consists of J_1 steps and the SP-phase consists of J_2 steps, where J_1 and J_2 are heuristic parameters based on the importance of density and length minimization in a problem, respectively.

In the SMMT-phase, the nets are routed one by one, using the algorithm SMMT. At the j-th step of the SMMT-phase, if the length of routing of N_i is within a constant factor, c_j of its minimum length then it is accepted, otherwise, the routing is rejected. Once a net is routed during SMMT-phase, it will not be routed again.

```
Algorithm LAYOUT-WRST (R, N_i)
begin
    T = an MST of N_i;
    L_0 = φ;
    for j = 1 to n − 1 do
        min_cost = ∞;
        for i = 1 to k do
            FIND-P(i, e_j, R);
            Q_{i,j} = MERGE(L_{j−1}, P_i(e_j));
            CLEANUP(Q_{i,j});
            if WT(Q_{i,j}) < min_cost then L_j = Q_{i,j};
end.
```

Figure 5.34: Algorithm LAYOUT-WRST.

In the SP-phase, the nets are routed one by one by employing a shortest-path heuristic and utilizing the results from the SMMT-phase. At the j-th step we accept a routing only if it is better than the best routing obtained so far.

ROLE MODEL

GEOFFREY DUMMER: THE FIRST PERSON TO CONCEPTUALIZE AND BUILD A PROTOTYPE OF THE INTEGRATED CIRCUIT

Name: Geoffrey Dummer

Born: February 25, 1909, Hull, Yorkshire, England

Death: September 9, 2002 (Age: 93)

Geoffrey Dummer was educated at the Manchester College of Technology. In 1931 he started work with the Mullard Radio Valve Company. In 1935 he moved to A C Cossor Ltd working on Cathode Ray Tubes (CRTs), time bases and circuits, and in 1938 moved again, this time to Salford Electrical Instruments.

In September 1939 he joined the radar research in the Air Ministry Research Establishment (AMRE), then at Dundee, where he worked in a team under Bob Dippy on Cathode Ray Tube (CRT) displays. It was here that he (with Franklin) started work on radial time base displays, which were the genesis of the first Plan Position Indicator (PPI). This is the classic map like radar display with a line sweeping around its centre. In mid 1940, the first Plan Position Indicator was demonstrated to Air Marshal Joubert at Worth Matravers on C-site using a specially modified Chain Home Low radar.

It was not practical to train radar operators on operational equipment, so training equipment was needed to simulate the real radars. Dummer became responsible for the design, manufacture and installation and servicing of many equipments for training radar operators. He also visited the United States and Canada to advise on setting up similar trainers there.

He was later in charge of a group dealing with the development of components for electronic circuits. The drive for reliability made it desirable to reduce the number of unreliable electrical connections between components - especially for operation in adverse climatic conditions. This led him to conceive the idea of multiple components on a single semiconductor chip. In May 1952 he presented this idea at the US Electronic Components Symposium. This was six years before Jack Kilby of Texas Instruments was awarded a patent for essentially the same idea. As a result he has been called "The Prophet of the Integrated Circuit". Dummer was unable to persuade either the government or industry in the United Kingdom to fund development of these ideas.

SUMMARY

- VLSI physical design is a multi-phase process, where each phase typically falls into one of the following three classes: partitioning, placement, and routing. In the partitioning phase, we split the chip into smaller, more manageable pieces.
- The objective of the routing problem is dependent on the nature of the chip. For general purpose chips, it is sufficient to minimize the total wire length, while completing all the connections.
- Global routing has to deal with two types of nets. Other nets which may not need use of M4and M5 can be routed through a sequence of channels. Global router must notallocate more nets to a routing region than the region capacity.
- The global routing problem is typically studied as a graph problem. The routing regions and their relationships and capacities are modeled as graphs.
- Checker board model is a more general model than the grid model. It approximates the entire layout area as a 'coarse grid' and all terminals located inside a coarse grid cell are assigned thatcell number.
- The global routing problem of two terminal nets is to find path for each net in the routing graph such that the desired objective function is optimized.
- In gate array design style, the size and location of all cells and the routing channels and their capacities are fixed by the architecture. This is the key difference between gate array and other design styles.
- Maze routing algorithms are used to find a path between a pair of points, called the source(s) and the target (t) respectively, in a planar rectangular grid graph. The geometric regularity in the standard cell and gate array design style lead us to model the whole plane as a grid.
- Maze routing algorithms are grid based methods. The time and space required by these algorithms depend linearly on their search space.

KNOWLEDGE CHECK

1. Routing congestion can be avoided by .....

a. placing cells closer

b. Placing cells at corners

c. Distributing cells

d. None

2. After the final routing the violations in the design .....

a. There can be no setup, no hold violations

b. There can be only setup violation but no hold

c. There can be only hold violation not Setup violation

d. There can be both violations.

3. What is routing congestion in the design?

a. Ratio of required routing tracks to available routing tracks

b. Ratio of available routing tracks to required routing tracks

c. Depends on the routing layers available

d. None of the above

4. The graph models for area routing capture the complete layout information and are used for finding exact route for each net.

a. True

b. False

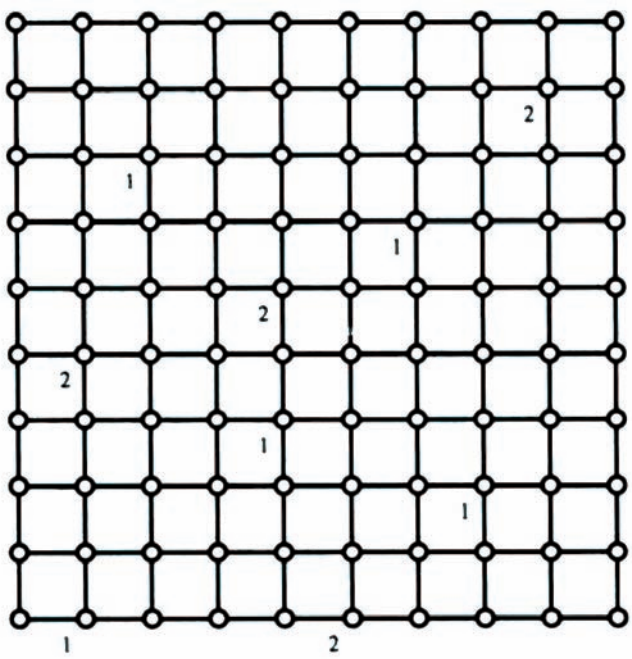
5. The objective of a maze routing algorithm is to find a path between the source and the target vertex without using any blocked vertex.

a. True

b. False

REVIEW QUESTIONS

1. What is global routing? Explain the Design style specific global routing problems.
2. Give an example for which the Hightower line-probe algorithm does not find a path even when a path exists between the source and the target.
3. Extend Lee's maze router so that it generates a shortest path from source to target with the least number of bends.
4. In Figure, terminals of two nets N_1 and N_2 are shown on a grid graph. Terminals of net N_1 are marked by '1' and that of N_2 are marked by '2'. Find an MRST for N_1.



5. Design an efficient heuristic algorithm based on maze routing to simultaneously route two 2-terminal nets on a grid graph. Compare the routing produced by this algorithm with that produced by Lee's maze router by routing one net at a time.

CHECK YOUR RESULT

1. (c) 2. (d) 3. (a) 4. (a) 5. (a)

REFERENCES

1. http://cc.ee.ntu.edu.tw/~ywchang/Courses/PD_Source/EDA_routing.pdf
2. http://users.eecs.northwestern.edu/~haizhou/357/lec6.pdf
3. http://www.cas.mcmaster.ca/~deza/jcmcc2012.pdf
4. linkedin, https://www.linkedin.com/pulse/maze-routing-lees-algorithm-vlsi-system-design
5. http://web.eecs.umich.edu/~mazum/ClassDescriptions/Routing.pdf

HIGH-LEVEL SYNTHESIS

CHAPTER 6

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Explain high-level synthesis in VLSI
2. Define the scheduling in VLSI synthesis
3. Describe the data path allocation and binding
4. Know about controller synthesis

INTRODUCTION

The term synthesis is used to denote the process of transforming a digital system from a behavioral specification into an implementation structure. Generally speaking, the specification includes some form of abstractions, i.e., some of the design decisions are not bound. The implementation, on the other hand, has to describe in detail the complete design at a given level of abstraction. Thus, synthesis can be seen as a process of creating implementation details which are left out of the specification. For example, a purely behavioral specification of a microprocessor may specify only what should be done in a typical instruction cycle and leaves it to the synthesis procedure to decide whether a centralized bus should be used, which technique should be employed to implement the control function, and how much parallelism should be supported. Due

to the complexity of digital systems, especially those implemented in VLSI technology, the synthesis process is usually divided into several steps. These steps usually include system-level synthesis, high-level synthesis, logic synthesis and physical design. System-level synthesis deals with the formulation of the basic architecture of the implementation.

The input to this synthesis step is a system-level specification which describes the behavior of the entire system in terms of a set of interacting processes. Such a system can be implemented by a set of cooperating processors such as ASICs dedicated controllers. FPGAs and DSP processors. The allocation of the set of physical processors and the mapping of processes in the behavioral specification onto these processors are the most critical design decisions to be made during the system-level synthesis step. An important feature of the system level is that the synthesis techniques and design requirements are highly application dependent. For example. System-level synthesis techniques used in the real-time embedded-controller application area will be very different from those used in DSP systems. This feature results in also the different definitions of what design tasks constitute a system-level synthesis step. These issues are system partitioning, hardware/software co-design and interconnection-structure design. The output of the system-level synthesis step is a set of processes with well-defined interfaces. Each of the processes is specified by a behavioral specification.

6.1 HIGH-LEVEL SYNTHESIS IN VLSI

High-level synthesis will then translate the behavioral specification of a process into a structural description that is still technology independent. This structural description is usually given in terms of a net list at the register transfer level. System-level synthesis and high-level synthesis form the front-end of a synthesis approach to digital system design and are together called system synthesis. The issue of controller synthesis which is related to both high-level synthesis and system-level synthesis will be presented at the end of this chapter. After system-level synthesis and high-level synthesis have been performed logic synthesis and physical design are used to map the structural implementation at register-transfer level into a layout description which is the final implementation.

Logic synthesis and physical design form the back-end of the synthesis approach to digital system design. One important reason for the separation of the synthesis process into frontend (system) synthesis and back-end synthesis can be attributed to the good general property possessed by front-end synthesis and the short-lived nature of the target semiconductor process associated with the back-end synthesis. As front-end synthesis is not bound to a particular technology, it can be used in several different design environments or adopted quickly to new technologies as needed. The back-end however has a very short life cycle because technologies change. In recent years, there has been a clear trend toward automating the system synthesis process and there are several reasons for this:

Shorter design cycle. The use of automation in the synthesis process reduces the design time and provides better chances for a company to hit the market window for the products. Automation reduces also the cost of the products significantly since in many cases the design cost dominates the product development cost.

- The ability to explore a much larger design space. An efficient synthesis technique can produce sever designs from the same specification in a short period of time. This allows the designer to explore different trade-offs between cost performance power consumption testability etc.
- Support for design verification. One prerequisite for automating the hardware/software co-design process, for example is to start the synthesis process with a joint specification of both the hardware and software. This makes it possible to verify the complete design consisting of both hardware components and software procedures.
- Fewer errors. The reduction of manual design activities means that the number of human errors will be decreased. If the synthesis algorithms can be validated we can also be more confident that the final design will correctly implement the given specification.
- Increased availability of IC technology. As more design knowledge is captured in the synthesis **algorithms** it is much easier for people who are not IC-technology experts to design chips.

Algorithm is an effective method that can be expressed within a finite amount of space and time and in a well-defined formal language for calculating a function.

6.1.1 The High-Level Synthesis Tasks

The input to the high-level synthesis process is given in an algorithmic-level specification such as behavioral VHDL. This type of specification gives the required mapping from sequences of inputs to sequences of outputs. The specification should constrain the internal structure of the system to be designed as little as possible. From the input specification, a synthesis system produces a description of a data path, that is, a network of registers functional units, multiplexers and buses. A control part should also be produced if it is

not integrated into the data path. In a synchronous design the control part can be given as microcode. PLA profiles or random logic. The basic components of the data path will eventually be implemented by some physical modules available in a given technology. The technological parameters, such as silicon area, operation delay and power consumption of the physical modules are usually stored in a module library and made available to the high-level synthesis algorithms. In this way the same high-level synthesis algorithm can be used to synthesize design based on different technologies by using different module libraries.

The basic tasks to perform in high-level synthesis are behavioral analysis design-style selection, operation scheduling, 'data-path allocation, control allocation module binding, and optimization.

6.1.2 Basic Synthesis Techniques

To carry out a synthesis task means to make design choices. There is usually a set of alternative structures that can be used to realize a given behavior For example, an addition operation in a given behavioral specification can be implemented tither by a dedicated adder or share an ALU with several other operations. The function of a synthesis algorithm is to analyze all or a subset of these alternatives and to choose the best structure which meets given design constraints, such as limitations on cycle time, area, or power, while minimizing a cost function. The difficulty of synthesis is that trade-offs of different design aspects are highly dependent on each other. Thus, when making design decisions for a synthesis task, other tasks have to be taken into account in order to optimize some design criterion, for example, the implementation cost. Consequently each of the synthesis tasks cannot be carried out independently without reducing the possibility of global optimization of the design. Another difficult factor of synthesis is the immense gap between the input specifications and the implementation results. Many design decisions are not bound in the specification and are left for the synthesis algorithms to decide. For example the synthesizer has to consider whether to multiplex a set of operations onto a single ALU or to implement several dedicated operators and the consequences of such a decision in terms of device timing, power consumption chip size, pin-out count, and other low-level parameters.

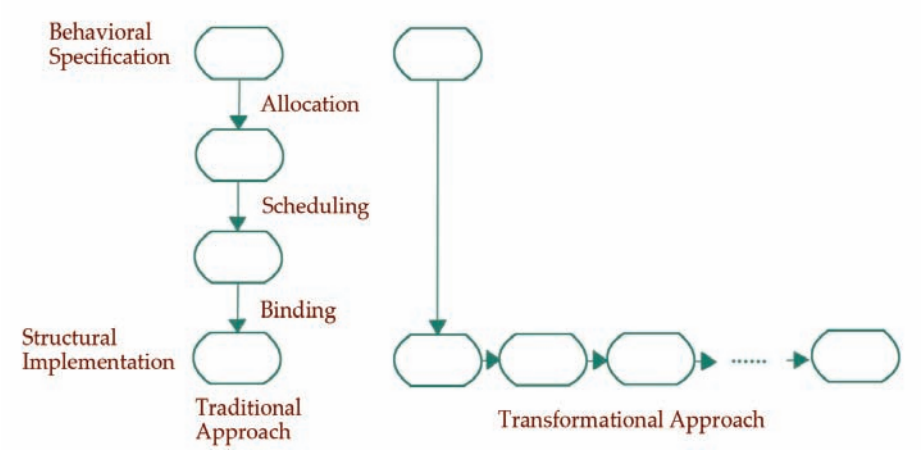


Figure 6.1: Comparison between the traditional and the transformational approaches to high level synthesis.

The final source of difficulty of automated system synthesis is the parallelism inherent in a digital system. To organize the available hardware resources to efficiently and reliably perform the desired operation, the synthesis algorithms have to automatically generate parallel structures as well as their synchronization/communication schemes. They are also responsible for the scheduling of operations so as to ensure sufficient parallelism in the implementation. It is obvious that system synthesis is a very complex problem. Many of the synthesis tasks have also been proved to be NP-complete. Therefore, it is often necessary to divide a design into several modules and apply synthesis algorithms to one module at a time. To further reduce the complexity, a synthesis approach either partitions the synthesis task into several sub-tasks or perform one sub-task at a time, or partition the task into a sequence of transformation steps each of which makes a small change to the intermediate result of the earlier steps.

The basic idea of the transformational approach can be illustrated with Figure 6.1, where the traditional approach to high-level synthesis divides the main synthesis task into three sub-tasks: allocation, scheduling, and binding. The transformational approach first moves the design to a structural implementation in one single step by a relatively naïve mapping. The implementation produced by this step is then transformed using the iterative application of semantic-preserving transformations. These transformations do not change the level of abstraction, rather they explore the implementation space, looking for good solutions. The use of the transformational approach has two main advantages. The first one IS that it facilitates the correctness-by-construction method. Since the synthesis process is divided into a sequence of small transformations, if the transformations can be proved to be semantics-preserving, the synthesis will produce a design which correctly implements the specification.

The second advantage of the transformational approach is that it makes it more efficient to employ optimization heuristics in the synthesis process. A transformational approach can be viewed as a neighborhood-search optimization method. The synthesis process starts with an initial implementation which is generated by a naive mapping and similar 10 an initial solution of an optimization problem. The synthesis process then makes small changes to transform the current solution to a neighborhood solution, which is the same as the neighborhood moves. The objective of the transformations is to reach the optimal design, which is the same as finding the optimal solution in the solution space. Therefore, we can employ existing neighborhood search techniques in the transformational approach. Techniques such as simulated annealing, tabu search, and genetic algorithms have been reported to perform wen in different synthesis processes.

An important component of the transformational approach is the design representation which is used to capture the intermediate results of the synthesis process. The representation model must be able to represent the design with different degrees of completion. That is, it should be able to describe a very abstract design with a lot of unspecified information, for example a purely behavioral specification. At the same time, it should also be able to describe a very detailed implementation with physical parameters which is, for example, produced by the synthesis process. Thus, it is not necessary or always possible to have just one design representation model; a lot of synthesis systems actually use several different representation models during different stages of the synthesis process. .

In most design environments, however, it is very useful to have an unfired design representation which can be used to represent the design at different levels of abstraction. The main application of a design representation is to explicitly capture the intermediate result of a design so as to allow the design algorithm to make appropriate design decisions. This is very important in the transformational approach to synthesis.

DID YOU KNOW?

In 2004, there emerged a number of next generation commercial high-level synthesis products (also called behavioral synthesis or algorithmic synthesis at the time) which provided synthesis of circuits specified at C level to a register transfer level (RTL) specification.

6.2 SCHEDULING IN VLSI SYNTHESIS

Operation scheduling, or in short scheduling, deals with the assignment of each operation to a time slot corresponding

to a clock cycle or time interval. Typically, the input to this task consists of a control and data flow graph (CDFG), a set of available hardware resources and a performance constraint. A schedule will be generated such that the data control dependency defined by the CDFG will not be violated and the performance constraint is satisfied. Since scheduling determines which operations can be assigned to the same time slot, it affects the degree of concurrency of the resulting design and thus its performance. Further the maximum number of concurrent operations of a given type in a schedule is a lower bound on the number of required hardware resources for that operation. Therefore, the choice of a schedule affects the cost of the implementation and consequently scheduling plays an important role in high-level synthesis.

The scheduling problem can be formulated in several ways depending on the basic assumptions made. One straightforward way is to assume that the behavioral descriptions do not contain conditional or loop constructs, that each operation takes exactly one control step to execute, and that each type of operation can be performed by one and only one type of functional unit. In this case, we have the following problem:

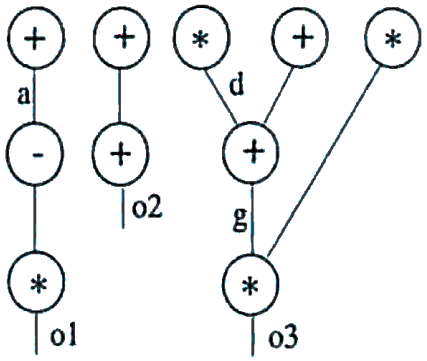
- Given: a set 0 of operations with a partial ordering which determines the precedence relations, a set K of functional unit types, a type function, "τ: O→K, to map the operations into the functional unit types and resource constraints m_k for each functional unit type.
- Find: a (optimal) schedule for the set of operations that obeys the partial ordering and utilizes only the available functional units.

The above stated problem is called resource-constrained scheduling, since a set of constraints regarding the number of functional units (hardware resources). For example, in a design where only two multipliers can fit within the available chip area it is necessary to impose a resource constraint to limit the number of multipliers to two. The other scheduling problems are time-constrained or time- and resource-constrained.

Example 6.1: Figure 6.2 gives a simple behavioral specification and its corresponding data flow graph (DFG). The DFG is generated by an algorithm which analyzes the data dependency relation of the different operations. The data dependency analysis is performed based on the following principle: Let O be the set of all operations in the DFG. If the result of operation o_i O is used by operation o_j O, then operation o_i must finish its execution before operation o_j can begin, and we say that there is a data dependency between these two operations. This data dependency is represented during the synthesis process as a precedence constraint (partial ordering) between the operations, which must be satisfied by the schedule. Figure 6.2b illustrates the data flow graph which is generated from the VHDL code shown in Figure 6.2a. We have assumed that all data flows downwards and have therefore drawn the directed edges only as edges without arrows to indicate directions.

```
a := i1 + i2;
o1 := (a - i3) * 3;
o2 := i4 + i5 + i6;
d := i7 * i8;
g := d + i9 + i10;
o3 := i11 * 7 * g;
```

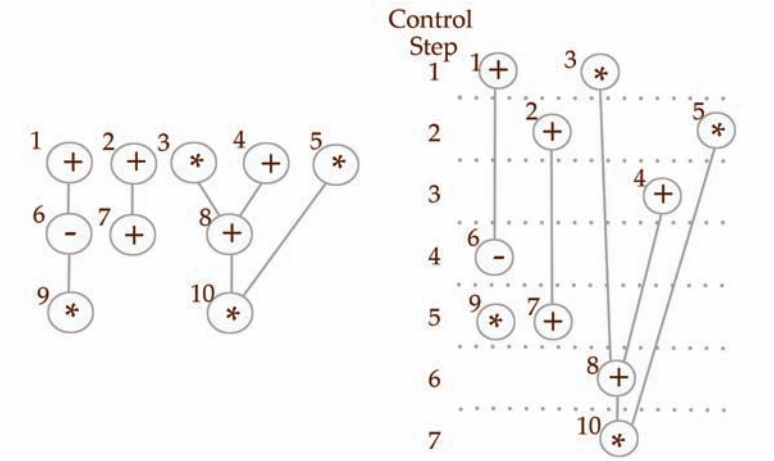
(a) Behaviroal specification



(b) DFG

Figure 6.2: A behavioral specification and its data flow graph.

The simplest scheduling technique is a greedy heuristics based on the "as soon as possible" (ASAP) principle. To schedule the DFG given in Figure 6.2b, using the ASAP algorithm, the operations are first sorted topologically according to their partial ordering; that is, if there is a partial order from o_i to o_j, o_i will be sorted before o_j" The algorithm then schedules operations one by one in the topologically sorted order by placing them in the earliest possible control step. For the DFG example given in Figure 6.2b, the topologically sorted order is illustrated in Figure 6.3a by the number used to label the operations.



(a) Sorted DFG (b) ASAP schedule

Figure 6.3: An ASAP scheduling example.

Let us assume that the available functional units include one adder and one multiplier. Both the addition and subtraction operations are mapped into the adder, and the multiplication operations are mapped into the multiplier. When operation 1 (the operation labeled with 1) is considered for scheduling, it is scheduled in control step 1, since the addition operation is mapped to the adder and there is an adder available. When operation 2 is considered, however, since the adder is already occupied, it cannot be scheduled in control step 1. Operation 2 will be scheduled in control step 2 instead. The algorithm will then examine operation 3. Since the multiplication operation is mapped to the multiplier and the multiplier is available, it is scheduled in control step 1. This process will continue until each operation is assigned to a control step.

Example 6.2: Figure 6.3b illustrates the result of applying the ASAP scheduling algorithm to the DFG graph of Example 6.1, which is shown In Figure 6.3a. In this case, seven control steps are needed to execute the given behavioral specification, provided that one adder and one multiplier are used.

A similar approach to the simple scheduling problem is to use the "as late as possible" (ALAP) principle. However, the operations will be scheduled backwards by placing them in the latest possible control step. To schedule the DFG graph of Example 6.1, operation 10 IS considered first. Since there is a multiplier available, operation 10 is scheduled in the last control step, which is illustrated in. Figure 6.4. The next operation to be considered is operation 9, another multiplication. It cannot be scheduled in the last control step, since there is no more multiplier available. Operation 9 will therefore be scheduled in the last but one control step. When operation 8, an addition, is considered, even though there is an adder available in the last control step, it cannot be scheduled there, since there is a data dependence between operation 8 and 10. Operation 8 must be completed before operation 10 can start. Since operation 10 has been scheduled in the last control step. The latest possible control step to perform operation 8 is the last but one control step. The algorithm will then exam operation 7, which has no data dependence with any operation already scheduled. Operation 7 can therefore be scheduled in the latest possible control step where the needed hardware resource to perform it is available, which is the last control step.

Example 6.3: Figure 6.4b illustrates the results of applying the ALAP algorithm to the DFG graph of Example 6.1, which is also depicted in Figure 6.4a. The ALAP algorithm generates a schedule which consists of six control steps, which is one step less than the schedule generated by the ASAP algorithm. Since the two designs need to have the same amount of functional units, the ALAP algorithm generates a better schedule in this particular case.

Usually, we would like the generated schedule to be an optimal one, i.e., it takes the minimal number of control steps to execute the specified behavior. The general scheduling problem is however NP-complete, and heuristics that do not guarantee optimal results are widely used to generate satisfactory solutions. A few of the most widely used algorithms will be described. For a complete treatment of scheduling techniques.

6.2.1 List Scheduling

With an ALAP algorithm, all the operations are also first sorted topologically according to their data/control dependencies, as in the case of ASAP.

The ASAP and ALAP algorithms are examples of constructive techniques for solving the scheduling problem. In such techniques a schedule is constructed step by step until all operations are scheduled. Another constructive approach is list scheduling, which proceeds from control step to control step. In each control step, the operations that are available to be scheduled are kept in a list, ordered by some priority function. Each operation on the list is then scheduled if the resources needed are free: otherwise .it is deferred to the next control step.

An .important decision for a list-scheduling approach is therefore to select the priority function. Since it has a strong impact on the final results. Some systems give higher Priority to operations with low mobility; here mobility of an operation is defined as the number of control steps from the earliest up to the latest feasible control step in which the operation can be scheduled. Others give higher priority operations with more immediate successors, arguing that **scheduling** them 10 the current control step would make the largest number of operations ready, thereby allowing the earliest possible consideration of each operation. Unfortunately there is no agreement on which priority function is best, and the selection of such a function usually depends on the application.

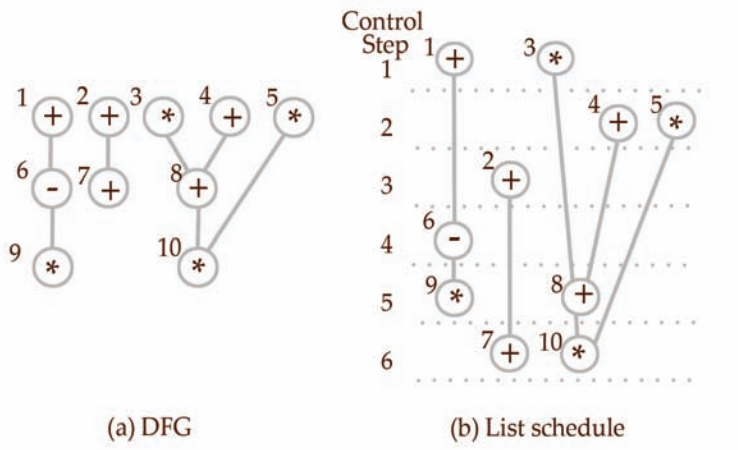


Figure 6.5: A list-scheduling example.

Let us consider the DFG of Example 6.1, which is illustrated in Figure 6.5a. Assume that the priority function is defined as the length of the path from the operation to the end of the block; an operation associated with a high number has high priority. In the first step of the list scheduling algorithm, operation 1, 2, 3, 4: and 5 are ready to be scheduled since none of them depend on any other operation to be completed. These ready operations are ordered by the priority function as follows:

Priority (o_1) = 2;

Priority (o_3) = 2;

Priority (o_4) = 2;

Priority (o_2) = 1;

Priority (o_5) =1.

These operation will be scheduled one by one in the above order. Assume again that there are only one adder and one multiplier available. Operation 1 (o_1) and 3 will be scheduled in the first control step. The algorithm will then proceed to the second control step. Now we have operation 2, 4, S, and 6 ready to be scheduled, since operation 2, 4, and 5 do not have any predecessors and the only predecessor of operation 6 has been scheduled in the previous step. Based on the priority function, we have the following order:

Priority (o_4) = 2;

Priority (o_2) = 1

Priority (o_5) = 1;

Priority (o_6) = 1;

Therefore operation 4 and 5 will be scheduled in control step 2 and the algorithm proceed to control step 3. Operation 2. 6, and 8 are now ready to be scheduled and they will be ordered in the following way:

Priority (o_2) = 1

Priority (o_6) = 1;

Priority (o_8) = 1

Since all the three ready operations are of type addition and there is only one adder available only one of them can be scheduled. The three operations have also the same priority value; this provides room for a secondary priority function to be used to select one of them to be scheduled. We assume however that only one priority scheme is used. Therefore, the selection is based on the topological order and operation 2 is scheduled. This process continues until all operations are scheduled. The final schedule for this example is illustrated in Figure 6.5b. This gives a schedule with six control steps, which is the optimal solution for this example.

6.2.2 Force-Directed Scheduling

Scheduling is the method by which work specified by some means is assigned to resources that complete the work.

The scheduling problems discussed so far are called resource-constrained scheduling (RCS). A RCS algorithm tries to find a schedule with the smallest number of control steps under some resource constraints. The resource constraints are typically given in the form of the number of functional units. In the more general form, the resource constraints can be given in terms of a complex cost function such as the estimated area of the silicon implementation of the whole design.

There is another type of scheduling problems which are called time constrained scheduling (TCS). A TCS algorithm tries to minimize the resources required to meet a specified global time constraint. The time constraint is typically given in terms of the number of control steps allowed for the execution of the specified behavior. The TCS problems occur very often in digital signal processing applications in which the system throughput is fixed and the silicon area must be minimized.

The force-directed scheduling algorithm is one of the most widely used techniques for the TCS problem. The basic strategy is to place similar operations in different control steps so as to balance the concurrency of the operations .assigned to the functional units without increasing the total execution time. By balancing the concurrency of operations, it is ensured that each functional unit has a high utilization and therefore the total number of units required is decreased.

The forced-directed scheduling algorithm consists of three main steps: determine the time frame of each operation, create a distribution graph, and calculate the force associated with each assignment. The first step of the force-directed scheduling algorithm is to determine the time frame of each operation. A good approximation of the time frame can be determined by constructing the ASAP and ALAP schedules without any resource constraints and then combining the result of both schedules.

Example 6.4: Let us consider the example given in Figure 6.2 again. Assume that the time constraint indicates that four control steps are allowed for this example. Figure 6.6 illustrates the ASAP and ALAP schedule of the DFG given in Figure 6.2b, assuming no resource constraints. From the two schedules it is easy to identify the time frame for each

operation. For example; operation 1 can be either scheduled in control step (c-step) 1 or 2 and therefore its time frame spans from c-step I to c-step 2. Operation 2, on the other hand, has a time frame from c-step 1 to c-step 3. The question that remains to be answered is in which c-step of the time frame the operation should be scheduled in order to reduce the number of functional units needed. It can be observed that if the ASAP schedule is directly used, 3 adders and two multipliers are needed, which is definitely not optimal for this example.

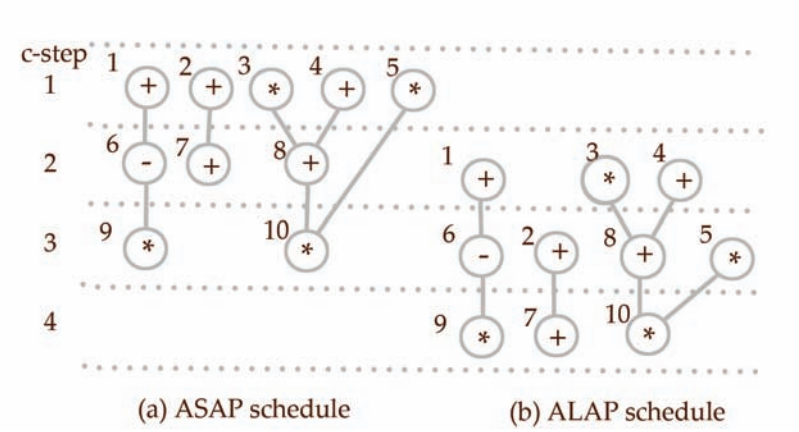


Figure 6.6: ASAP and ALAP schedules of Example 6.1.

The time frames for all the operations can be collected in a graph, such as the one given in Figure 6.7 for Example 6.1. The width of the box in Figure 6.7 containing an operation in a c-step represents the probability that the operation will be eventually placed in that c-step (time slot). It is assumed that the probability distribution for each operation is uniform. The width of an operation box therefore equals 1 divided by the number of c-steps of its time frame.

The next step of the force-directed scheduling algorithm is to add the probabilities of each type of operations for each c-step and build a distribution graph for it. The distribution graph shows, for each c-step, how heavily loaded that step is, provided that all possible schedules are equally likely. If an operation could be done in any of the k steps in a time frame, 1/k is added to each of the c-steps in the graph. Using the information captured in Figure 6.7, we can calculate the value of the distribution graph, or DG, for the multiplication operations.

The results are DG_{mult} (l) = 1/2 + 1/3 = 0.833, DG_{mult} (2) = 1/2 + 1/ 3 = 0.833, DG_{mult} (3) = 1/2 + 1/2 + 1/3 = 1.333, and DG_{mult} (4):= 1/2 + 1/2 = 1, as illustrated in Figure 6.8. The results for the addition/subtraction operation DG are $DG_{a/s}$,(l) = 1/2 + 1/3 + 1/2 = 1.333, $DG_{a/s}$,(2) = 1/2 + 1/2 + 1/3 + 1/3 + 1/2 + 1/2 = 2.667, $DG_{a/s}$,(3) = 1/2 + 1/3 + 1/3 + 1/2 = 1.667, and $DG_{a/s}$,(4) = 1/3 = 0.333.

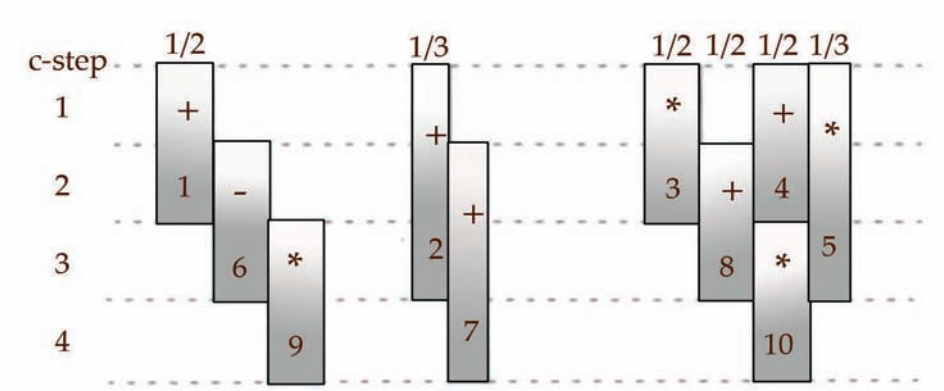


Figure 6.7: Time frames for Example 6.1.

The third step of the force-directed scheduling algorithm is to calculate the force associated with every feasible c-step assignment of each operation. For an operation with a time frame that spans from c-steps f to t, the force associated with its assignment to c-step j ($j \le j \le t$) is

$$Force(j) = DG(j) - \sum_{i=f}^{t} \left[\frac{DG(i)}{(t-f+1)} \right]$$

In other words, the force associated with the tentative assignment of an operation to c-step j is equal to the difference between the distribution value in that c-step and the average of the distribution values for the c-steps bounded by the operation's time frame.

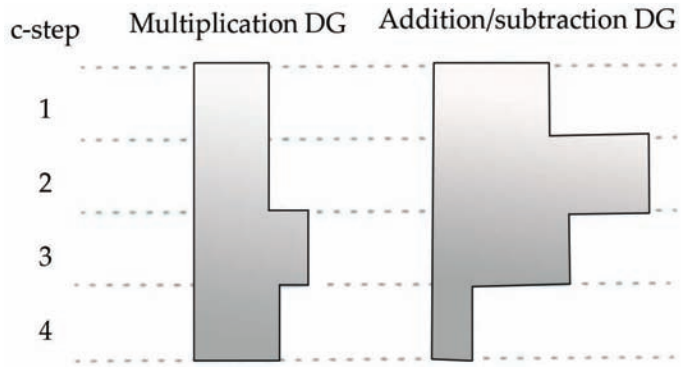


Figure 6.8: Distribution graphs for Example 6.1.

For example with the assignment of operation 10 to c-step 3, we have

Force (3) = DG_{mult} (3) = average DG_{mult} value over time frame of operation 10

= 1.333 - (1.333 + 1)/2 = 0.167.

On the other hand, the assignment of operation 10 to c-step 4 yields

Force (4) = DG_{mult} (4) = average DG_{mult} value over time frame of operation 10

= 1-(1.333+ 1)/2--0.167.

As the DG shows, if operation 10 is assigned to c-step 3, the distribution is not very well balanced, while the assignment of operation 10 to c-step 4 will generate a better result. This is reflected by the negative value of the force associated with the latter assignment.

We must also calculate the force for all predecessors and successors of the current operation whenever their time frames are affected. These additional forces are called indirect forces. The total force is the sum of the direct and indirect forces.

In Example 6.1, operation 10 has four predecessors, operations 3, 4, 5, and 8. The assignment of operation 10 to c-step 4 will not affect the time frame of any of its predecessors. Therefore the total force of assigning operation 10 to c-step 4 equals the direct force calculated above, namely -0.167. The assignment of operation 10 to c-step 3, on the other hand, will affect the time frames of operations 3, 4, 5, and 8. For example, operation 8 will now only be able to be performed in c-step 2 instead of both c-steps 2 and 3. Therefore, the assignment of operation 10 to c-step 3 implies that operation 8 will be assigned to c-step 2 and the indirect force of the latter assignment must be calculated and added to the total force of the former assignment. The indirect force of assigning operation 8 to c-step 2 equals $DG_{a/s}$ (2) - average $DG_{a/s}$ value over the time frame of operation 8 (c-steps 2 and 3), i.e., 2.667 - (2.667 + 1.667)/2, which is 0.5. The same calculation should be performed for operations 3, 4, and 5. All the indirect forces will then be added to the direct force to yield the total force of assigmning operation 10 to c-step 3.

Once all the forces are calculated, the operation-control step pair with the largest negative force (or least positive force) is scheduled. The distribution graphs and forces are then updated and the above process is repeated until all operations are scheduled.

Since the force-directed scheduling algorithm schedules one operation in each iteration, it is also constructive. Different from other constructive approaches, however, force-directed scheduling makes global analysis of the operations and control steps when selecting the next operation to be scheduled. Therefore force-directed scheduling is more expensive computationally than, for example, list scheduling. Force-directed scheduling has complexity O (cN^2), while list scheduling O (cN logN), and where c is the number of control steps and N the number of operations.

6.2.3 Transformation-Based Scheduling

Another basic class of scheduling algorithms is based on transformations. A transformation-based algorithm begins with an initial schedule, usually either maximally serial or maximally parallel, and applies transformations to it to obtain other schedules. The basic transformations are converting serial operations, or blocks of operations, into parallel ones and the inverse, converting parallel operations into series ones. Transformation-based algorithms differ in how they choose what transformations to apply and in which order these are applied. One extreme technique is to use exhaustive search. That is, all possible combinations of serial and parallel conversions are tried and the best design will be chosen. This method has the advantage that it looks through all possible designs and guarantee the optimal solution. However, it is computationally very expensive and not practical for large designs. **Exhaustive search** can be improved, to reduce the computation time needed, somewhat by using branch and-bound techniques, which cut off the search along any path that can be recognized to be suboptimal.

Exhaustive search is a very general problem-solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.

Another approach to transformation-based scheduling is to use heuristics to guide the process. Transformations are chosen that promise to move the design closer to the optimal design. Both the CAMAD system and the Yorktown Silicon Compiler use this approach. One important advantage of the transformation-based approach is that in each iteration, a complete schedule exists and accurate estimation of the design in terms of different criteria can be made. It is therefore straightforward to extend this type of algorithms to handle many advanced issues related to. Further combination of scheduling and allocation can also be achieved using the transformation-based approach, as in the case of the CAMAD system,

6.2.4 Advanced Scheduling Topics

The scheduling problems discussed so far are simplified versions of the real problems. We will now discuss several advanced topics which must be considered by any practical scheduling algorithm.

Control constructs

Until now we have assumed that a CDFG corresponds to a single basic block, i.e., one section of straight-line code with only one entry and one exit point. However, most hardware description languages such as YHDL support conditionals, loops and other control structures. The scheduling algorithm must consider these constructs during the scheduling process.

When scheduling conditional branches, the scheduling algorithm should make use of the possibility to share functional units between mutually exclusive branches as much as possible. For example the same adder can be used in both the "then" and "else" clauses of an "if' statement.

Chaining and multi-cycling

We have up till now assumed that all operations requires the same amount of time to execute, and this time is the control-step length or clock cycle time. In practice different operation types may have different execution times and the above assumption results in the situation that the clock cycle time is dictated by the most time consuming operation. In order to avoid the above problem, chaining and multi-cycling techniques can be used.

Example 6.5: Figure 6.9a illustrates a design where the addition and multiplication operations are mapped into an adder and a multiplier with 50ns and 100ns delay, respectively. With the single operation per cycle assumption, the clock cycle time is determined to be 100ns and the overall schedule length is 200ns.

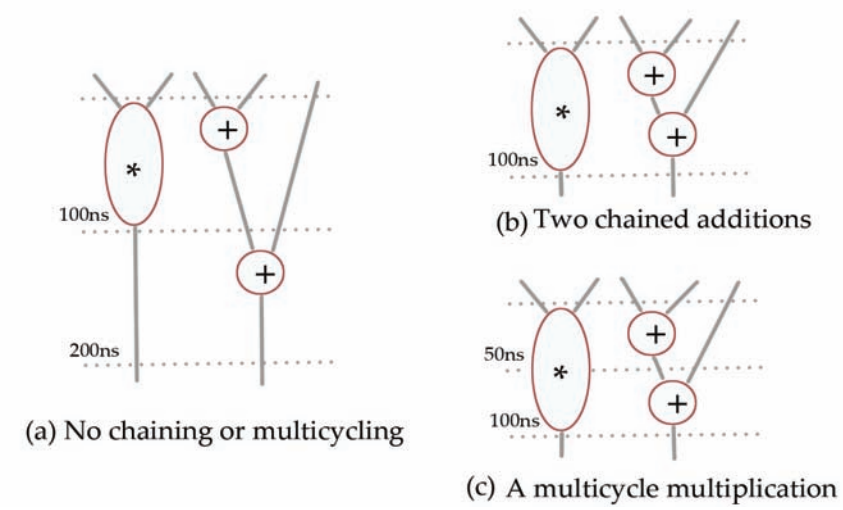


Figure 6.9: Chained and multi-cycle operations.

Chaining is the task of combining more than one operation in a control step. Figure 6.9 illustrates the case when chaining is used in Example 6.5, where the two addition operations are chained and scheduled in the same control step. In this way, the overall schedule length becomes 100ns. However, an extra adder is needed in this example since the two addition operations cannot longer be mapped into the same adder because they are performed in the same control step. Chaining can be performed before scheduling, and then the combined operations can be considered as a single entity in the scheduling process. Chaining can also be performed together with scheduling.

Another alternative to improve the schedule of Example 6.5 is to set the clock period to 50ns, and to execute the multiplication over two control steps, as illustrated in Figure 6.9c. An operation which is to be executed continuously over several clock cycles is called a multi-cycle operation. In Example 6.5, multi-cycling decreases the schedule length to 100ns, just like chaining, but without the cost of another adder. However, multi-cycling uses twice as many control steps as chaining, which may result in a larger controller. Further, multi-cycling usually needs additional registers to latch the results from one cycle to the other. For example, the result produced by the first addition operation must be latched while this is not the case in the chaining solution.

Scheduling with Timing Constraints

Most scheduling techniques try to find a schedule which has the shortest schedule length while satisfying a set of constraints or one which requires the least resources and has a shorter schedule length than a given constraint. In both cases the global performance of the design, in terms of the overall schedule length, is used as one design criterion. However, few digital systems work in isolation, so there may also be a need to specify more detailed timing constraints on certain operations, or sets of operations. For example we might want to give a minimum timing constraint, which specifies that one operation must be executed less than a specified amount of time after another operation, or a maximum timing constraint, which specifies that one operation must be executed at least a specified amount of time after another operation.

6.3 DATA PATH ALLOCATION AND BINDING

In general, data path allocation and binding deal with the problem of which resources are used to realize in the physical implementation. Such resources include registers memory units and different functional units as well as their communication channels. The basic principle is to share resources as much as possible provided that the performance and other design criteria can be satisfied.

Allocation and binding carry out selection and assignment of hardware resources for a given design. Allocation determines the type and number of hardware resources

for a given design. Binding assigns the instance of an allocated hardware resource to a given data path node. Different data path operations can share the same hardware resource if they are not executed at the same time. For example, an adder can be shared by two additions if they are not executed during the same clock cycle. A register can also be used to store the values of two variables if the life times of these variables do not overlap. As pointed out earlier, the term allocation is sometimes used to denote both the allocation and binding tasks.

Example 6.6: An allocation for the example depicted in Figure 6.2 selects two adders and one multiplier. A possible schedule for this allocation is depicted in Figure 6.10. Note that the scheduling assumes, in this case that at most two adders and one multiplier in every clock cycle can be used, as determined by the allocation. The binding step assigns every operation of the scheduled graph to a physical hardware component. A possible binding for this example is also depicted in Figure 6.10. Addition/subtraction nodes 1, 2, 6 and 7 are assigned to adder #1, addition nodes 4 and 8 are assigned to adder #2, and multiplication nodes 3, 5, 9 and 10 are assigned to the multiplier.

The selection of the type and number of hardware resources during the allocation step is usually formulated as an optimization problem. The main goal is to find the minimum number of resources while fulfilling given area/ performance constraints.

The basic assumption made by many high-level synthesis systems regarding binding is that each data path node has at least one module in a module library which implements the function of the data path node. For example, a node which performs an addition operation may correspond to an adder or an ALU in the module library. The different modules can have different areas and/or latencies. This gives the possibility to make trade-offs between different implementations. The binding problem is also an optimization problem and can be formulated using existing optimization methods. For example an Integer Linear Programming method or a graph clustering technique can be used to solve it. It can also be solved using heuristic methods.

Most scheduling techniques handle these timing constraints by adding additional constraint edges to the CDFG, and then treating those additional edges in much the same way as other constraints. Special techniques based on heuristics which deal with local timing constraints in a systematic manner have also been developed.

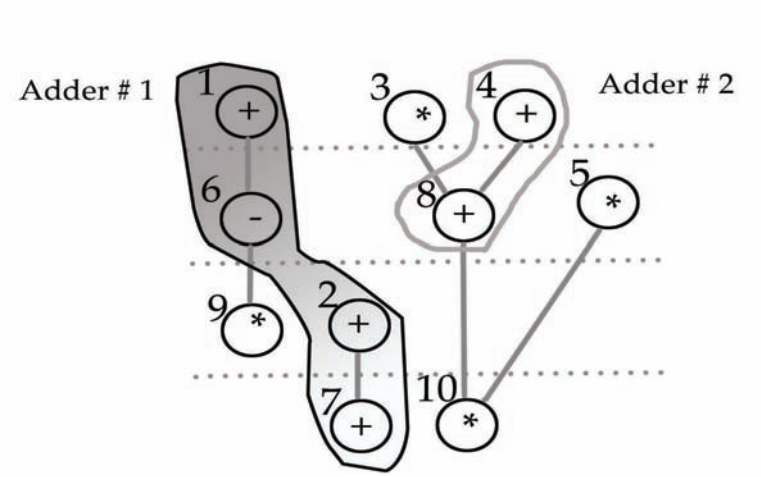


Figure 6.10: Binding for Example 6.6 with 2 adders and 1 multiplier.

6.3.1 Integer Linear Programming

Integer linear programming (ILP) is a subclass of the linear programming problems where the decision variables are of integer values. If we assume that decision variables are represented by a vector X the ILP problem can be formally stated as:

Maximize (or Minimize)	C^TX	(1)
Subject to	$AX=B$	(2)
	$X \geq O$	(3)
	X integer	(4)

In this formulation, the maximization/minimization criterion is defined as (1) and the constraints for the optimization problem are defined as equations (2) and inequalities (3). In addition all decision variables are constrained to be integers (4). If binary decision variables are used instead of integer ones the problem is called 011 linear programming problem. Once our problem is defined as an ILP problem known methods for solving it can be directly applied.

As an example we will define the problem of finding a binding for a given schedule as a 011 linear programming problem. We will concentrate on the formulation of binding constraints. The reader can imagine different types of cost functions which can be used to minimize different aspects of the design for example, interconnection cost or power consumption. For the purpose of this presentation we also make the simplifying assumption that every operation is executed in exactly one clock cycle,

i.e., no chaining or multi-cycle operations are possible.

We define the following decision variables and constants:

- $binding_{ij}$ where i = 1. 2..... ops and denotes operation number, and j = 1, 2 ..... r and denotes component number; the decision variable $binding_{ij}$ is 1 if operation i is bound to hardware resource j.
- $schedule_{ik}$ where i = 1.2..... ops and denotes operation number and k = 1, 2..... max_step and denotes the number of the steps in a given schedule.

Since we assume that the schedule is already decided, the constant $schedule_{ik}$ is 1 if operation number i is scheduled in step number k.

The binding problem can be defined as a solution satisfying the following constraints:

$$\sum_{j=l}^{r} binding_{ij} = 1, \qquad i = 1,2....,ops; \tag{5}$$

$$\sum_{i=1}^{ops} binding_{ij}.schedule_{ik} \leq 1, \qquad j = 1,2,...,r; k = 1,...,\max\_step; \tag{6}$$

$$bonding_{ij} \in \{0,1\},\ i = 1,2,...,ops; j = 1,2,...,r. \tag{7}$$

The constraint (5) denotes that an operation can only be assigned to one resource while the constraint (6) requires that at most one operation can be executed on a hardware recourse during an execution step. The above stated constraints can usually be satisfied by several solutions. Since we are looking for a solution which minimizes a given design criterion, a cost function should be defined to guide the selection of the best solution.

Example 6.7: Let us consider the problem of finding a binding for the example given in Figure 6.2 with the schedule presented in Figure 6.10. For this example the binding of the two adders has to fulfill the following constraints:

$$binding_{ij} + binding_{i2} = 1, \text{ for } i = 1,2,4,6,7,8;$$
$$\sum_{i\in\{1,2,4,6,7,8,\}} binding_{il}.schedule_{ik} \leq 1, \text{ for } k = 1,2,3,4;$$
$$\sum_{i\in\{1,2,4,6,7,8,\}} binding_{i2}.schedule_{ik} \leq 1, \text{ for } k = 1,2,3,4;$$

There are 16 different solutions which satisfy these constraints. One possible solution is:

$$binding_{11} = 1 \quad binding_{12} = 0$$
$$binding_{21} = 1 \quad binding_{22} = 0$$
$$binding_{41} = 0 \quad binding_{42} = 1$$
$$binding_{61} = 1 \quad binding_{62} = 0$$
$$binding_{71} = 1 \quad binding_{72} = 0$$
$$binding_{81} = 0 \quad binding_{82} = 1$$

The above binding solution is depicted in Figure 6.10.

The ILP-based definition of the binding problem can be extended in several ways to include more complex and realistic requirements. It can also be defined to include both scheduling and binding.

6.3.2 Clique Partitioning and Graph Coloring

Allocation and binding can also be defined as graph problems. They can be formulated either as the problem of finding cliques in a compatibility graph or that of coloring vertices in a conflict graph. A compatibility graph is used to represent information on resource sharing. Two operations are compatible, and can share the same hardware resource, if they are of the same type and are not executed at the same time. A compatibility graph G_{comp} (V, E) is built of vertices V denoting operations and edges E denoting the compatibility relation between the operations. A **vertex** v_i is connected to a vertex v_j if the two operations they represent can be assigned to the same resource. An example of a compatibility graph is depicted in Figure 6.11.

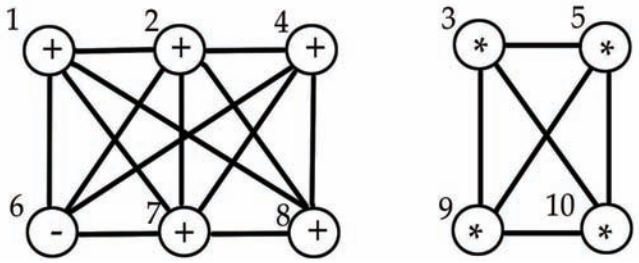


Figure 6.11: A resource-compatibility graph example.

To solve the binding problem using compatibility graphs, we have to find a maximal set of compatible operations. This can be formulated as a maximal clique partitioning

problem. A clique is defined as a sub graph where all nodes are connected to each other. It is maximal if it is not contained in any other clique.

Example 6.8: Let us consider the example depicted in Figure 6.10. Based on the given schedule and the assumption that addition and subtraction operations can share the same resource, a compatibility graph can be built as shown in Figure 6.11. Three maximal cliques can be identified as depicted by bold edges in Figure 6.11. Two cliques (1, 2, 6. 7} and {4, 8} represent the binding of the addition/subtraction operations to two adders while the clique (3. 5, 9, 10) represents the binding of the multiplication operations to one multiplier. Note that the solution is not unique and other solutions are also possible. For example, we can also identify other two cliques. (2. 4, 7, 8) and {1, 6}, for the addition/ subtraction operations.

Vertex is the fundamental unit of which graphs are formed: an undirected graph consists of a set of vertices and a set of edges, while a directed graph consists of a set of vertices and a set of arcs.

A conflict graph, on the other hand, captures the opposite information as the compatible groups. It denotes explicitly the operations that cannot share the same resource. A conflict graph $G_{conflict}$ (V, E) is built of vertices V denoting operations and edges E denoting the conflict relation between them. A vertex V_i is connected to a vertex V_j if the two operations they represent cannot be assigned to the same resource. An example of a conflict graph is depicted in Figure 6.12.

The binding problem in this case is solved using a graph coloring algorithm. The algorithm assigns different colors to vertices connected by edges while minimizing the number of colors.

Example 6.9: The resource conflict graph, depicted in Figure 6.12, has only two edges representing two resource conflicts for addition/subtraction operations. Addition operations 1 and 4 are executed in parallel during the first control step and addition/subtraction operations 6 and 8 are executed in parallel in control step 2. In this case, the sub graph for addition/subtraction needs be colored using two colors while the multiplication sub graph requires only one color. A coloring scheme is captured in Figure 6.12 which again is not unique.

Both the maximal clique-partitioning problem and the minimal graph coloring problem are intractable for realistic-size examples. Thus heuristics which generate solutions quickly without guaranteeing optimality, have widely been used. We

can also make use of the fact that these two problems form each other's dual. The two graphs are complementary to each other and the complexity of one formulation could be much less than the other. This makes it possible to select the formulation with a less complex graph for a given binding problem in order to speed up the binding process.

6.3.3 Left Edge Algorithm

To show other approaches to allocation and binding we discuss the left-edge algorithm which is used very often for register allocation and binding. The left-edge algorithm was originally introduced to perform the channel routing task. It was used for assigning interconnections (trunks) into a number of tracks in a channel. In the original formulation of the algorithm, the channel was oriented horizontally and the algorithm sorted trunks in an increasing order of their left end-points. The algorithm assigned trunks into successive tracks starting from the first one. For every track it scanned the ordered list of unplaced trunks and placed them one by one into successive available parts of the track. We will use this algorithm for the assignment of variables into registers.

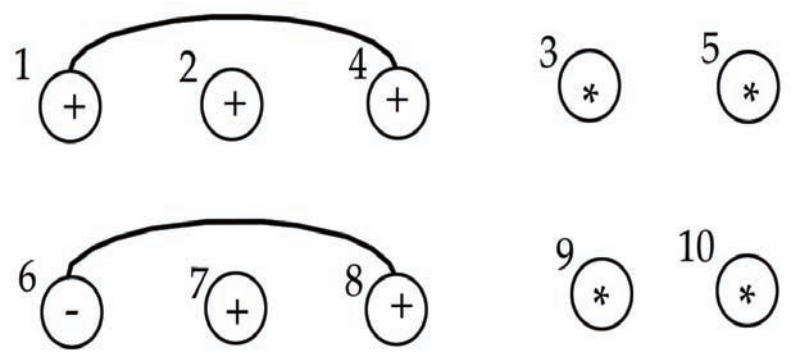


Figure 6.12: A conflict-graph example.

The starting point of the bar represents the time when the variable is set and the end of the bar represents the time when the variable is released and its value is not used any longer. In other words, the bar represents define-use time for the variable. On the right hand side of Figure 6.13 there are bars representing the life-times of the 21 variables used in the example given on the left hand side.

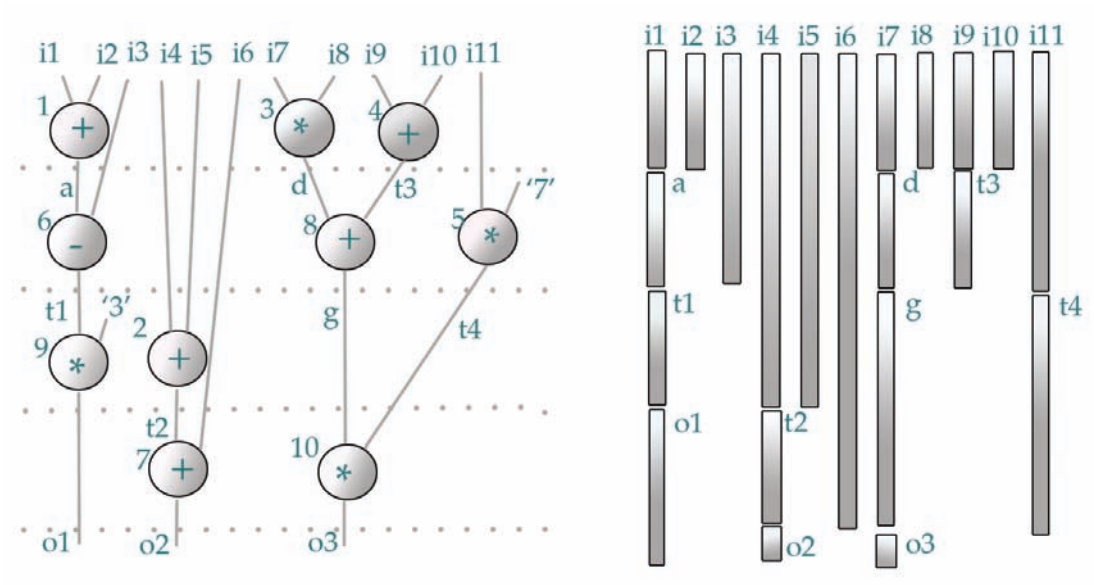


Figure 6.13: Variable life-times

A possible solution to the register-assignment problem can use the original formulation of the left-edge algorithm. In this solution all bars representing the life-times are sorted in increasing order of their starting points. The algorithm proceeds then with one register at a time. It assigns the first variable from the list, represented by a bar, into the first register. It then scans the sorted bars and the first encountered unbound bar which has start time higher than the end-time of the already assigned bar is placed into the same register. II continues this process by assigning the next variables into the same register. If the life-time of this register reaches the last scheduled step the algorithm starts to assign variables into the next register using the same method.

Example 6.10: Consider the assignment of the 21 variables used in the example depicted in Figure 6.13. The first step sorts all bars and the resulting sorted list of variables is depicted in Figure 6.14a. The final assignment of variables into registers is depicted in Figure 6.14b.

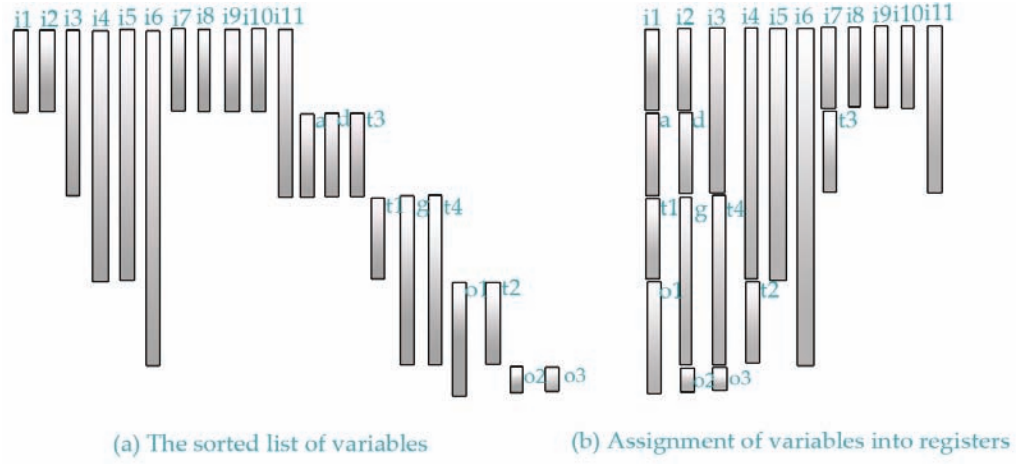


Figure 6.14: Applying the left-edge algorithm for register allocation

The left-edge algorithm will allocate the minimum number of registers, but has two disadvantages. First, not all life-time tables might be interpreted as intersecting intervals on a line. For example the existence of conditional branches prohibits the interpretation of intersecting intervals on a line, since values occurring in mutual exclusive branches may share a register although they seem to overlap in life-time. Second, the allocation produced is neither unique nor necessarily optimal in terms of, for example, the number of multiplexors required.

Example 6.11: Consider the sequencing graph depicted in Figure 6.15a. Figure 6.15b depicts its variable life-times while in Figure 6.15c the conflict graph for all variables is shown. This graph can be colored with three colors which gives an assignment of three registers.

6.4 CONTROLLER SYNTHESIS

Controller synthesis is a field which fits into this vision but has been mainly oriented towards hardware engineering (a highly componentized engineering discipline). Very abstractly, controller synthesis, given a model of the assumed behavior of the environment and a system goal, produces an operational behavior model for a component that when executing in an environment consistent with the assumptions results in a system that is guaranteed to satisfy the goal. In particular we focus on defining control problems for behavior models expressed as Labelled Transition Systems (LTS) and parallel composition defined broadly as synchronous product.

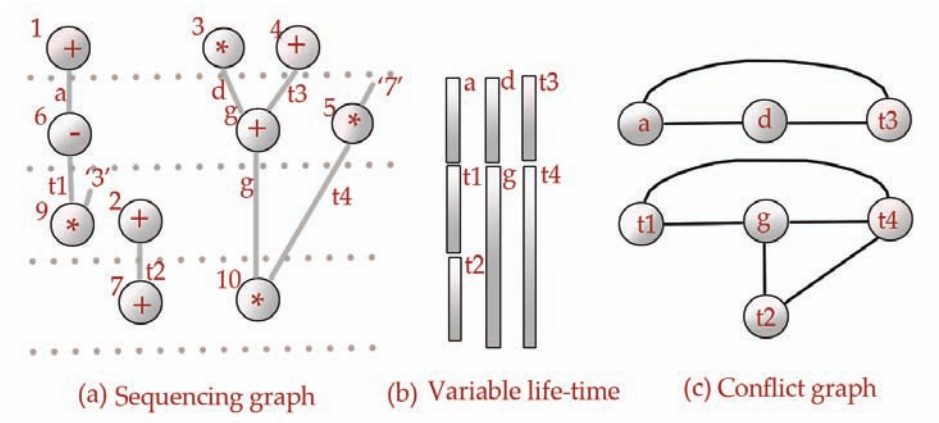


Figure 6.15: Variable life-times and conflict graph.

System synthesis deals with systems usually specified as a set of communicating concurrent processes. In this case the controller synthesis is highly dependent not only on the functionality of the controllers but also on the interaction and synchronization requirements. Traditional high-level synthesis methods, which 3rc limited to synthesis of one concurrent process at a time, produce poor results or cannot deal with such designs. One important system synthesis task is thus to propose an efficient control engine which may include several cooperating controllers.

An efficient method to implement a controller is to use a **finite state machine** (FSM) notation. The FSM formalism makes it possible to specify a number of states together with transitions between them. An FSM can be defined using either Mealy or Moore machine style.

Formally, an FSM can be defined by the following 5-tuple:

$<S, I, O, \delta, \lambda>$, where

S is a set of states;

I is a set of inputs (conditions);

O is a set of outputs (control signals);

δ is a next-state function, $\delta: S \times I \rightarrow S$; and

λ is an output function, $\lambda: S \times I \rightarrow O$ for Mealy machine or $\lambda: S \rightarrow O$ for Moore machine.

The above formulation defines an FSM as a machine which has a number of states. The machine can go from one state to another by executing the next state function. The next state is determined based on the current state and the input signals. The output signals are determined by the output function. For Mealy machines the output signals are generated based on the current state and the input signals while for Moore

machines they depend only on the current state.

A controller is usually graphically represented by a state diagram. The state diagram is a directed graph with nodes denoting states and arcs denoting transitions from one state to another. The control signals can be assigned either to states or arcs depending on the machine style while guarding conditions are assigned to arcs. A transition from a given state to the next state takes place if and only If the controller is in the given state and the guard assigned to the transition arc connecting both states is true.

Finite-state machine (FSM) is a mathematical model of computation used to design both computer programs and sequential logic circuits.

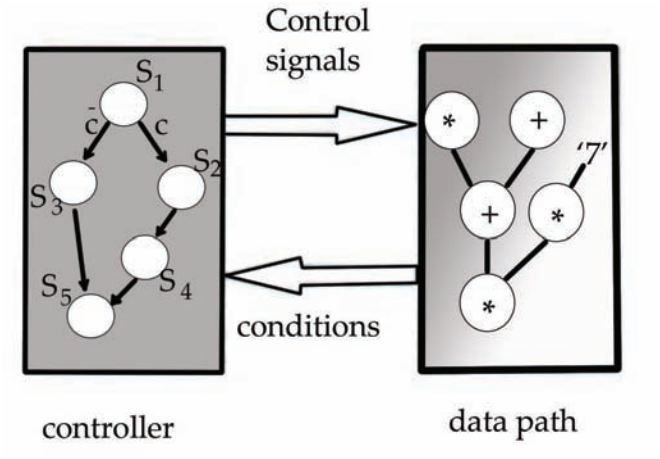


Figure 6.16: A controller/data path architecture.

In our formulation the controller, consisting of one or several FSMs, is used to control data path activities. The output signals are used to control data path operations while the input signals are conditions generated from the data path to influence the execution of the controller. Figure 6. 16 represents schematically the general view of the controller and its relation to the data path.

After generating the FSM specification it is a logic synthesis task to perform further optimization steps, such as state minimization and state encoding, using standard FSM synthesis methods. However, some decisions regarding the selection of the control structure have to be made during system synthesis. At this stage it should be decided if a single controller will be used to control the whole design or rather

several cooperating controllers will be used. It can also be decided if a hierarchical controller can be used. The selection of controller style depends on the synthesis problem formulation and design criteria, such as performance, area and testability. These decisions deal mainly with the overall architecture of the controller rather than its detailed implementation.

REMEMBER

The left-edge algorithm requires information about the life-time of variables. The life-time of a variable is the time interval when the variable is used by the computation. It can be represented as a bar drawn in parallel to our scheduled design.

6.4.1 Controller-Style Selection

The main idea of the presented approaches is to implement a complex control structure by several smaller controllers in order to simplify the design of the generated FSMs.

Single Controller

In this style, the controller is modeled by a single FSM, which is the simplest solution to the controller-style selection problem. Since an FSM is a sequential machine parallelism is only allowed for operations assigned to the same state. Otherwise operations have to be statically scheduled and assigned to consecutive states. The method is very efficient for simple computations but in the case of several parallel execution threads it can lead to the state explosion problem, especially when parallel loops are involved.

Hierarchical Controller

A hierarchical controller consists of a number of simple controllers organized in a hierarchy. It is assumed that the hierarchy of controllers is ordered by their parent-child relation. The parent controller distributes an execution task among a number of child controllers. Every child controller starts its execution upon receiving the activation signal and it sends a completion signal upon execution termination. The parent controller continues its execution when the child controller sends a completion signal. The child controller can be a parent controller for another lower-level child controller.

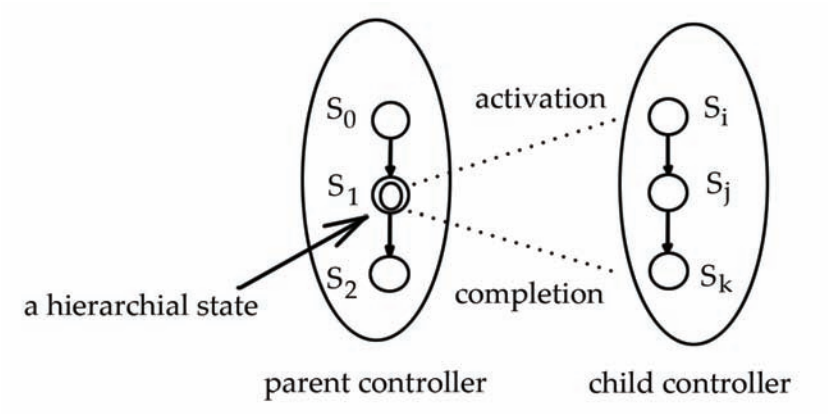


Figure 6.17: An example of a hierarchical controller.

Hierarchical controllers usually make use of two special internal control signals called activation and completion. The activation signal is used to start another controller and the completion signal provides information about termination of the execution of a controller. They are only used for controller synchronization.

Example 6.12: Figure 6.17 presents an example of a hierarchical controller. In this example state S_1 is hierarchical and is implemented by a child controller consisting of three states S_i, S_j and S_k. It is activated by the parent controller, with an activation signal. The parent controller will continue its execution upon receiving the completion signal from the child controller.

Hierarchical controllers can be used to implement in a natural way several control structures which are commonly used in algorithmic languages, such as procedure and loop constructs. Procedures are well suited for this because their purpose is to create a hierarchy of subprograms. A process calls a procedure and continues upon the procedure return. It matches the hierarchical controller principles very well. A large program can also be restructured by partitioning it into loop-free parts by structuring loops as subprograms. These structures can later use the hierarchical controller concept to implement efficient control engines.

Parallel Controller

The function of a controller can also be distributed into several smaller controllers which are executed in parallel. These parallel controllers can communicate to exchange data or synchronize their execution. Generally speaking, decomposition of a controller into a number of parallel controllers reduces the overall controller complexity and solves some design problems.

However, it requires new design methods.

In some cases, several parallel controllers do not need to synchronize. They are executed independently of each other. The activation/completion signals can then be used for correct sequencing of operations between a master controller and these parallel controllers, which is illustrated in the following example.

Example 6.13: A design specification includes two consecutive loops as depicted in Figure 6.18a. The computations of the loops are data independent and can be executed in parallel to speed up the computation. The generation of one sequential controller for the parallel computations is impossible in practice since the generated controller should include all combination of states in the two loops which results in a combinational explosion of states. It is possible, however to generate one master controller which activates two independent parallel controllers, one for each loop, as illustrated in Figure 6.18b. This solves the performance problem while keeping the controller size small.

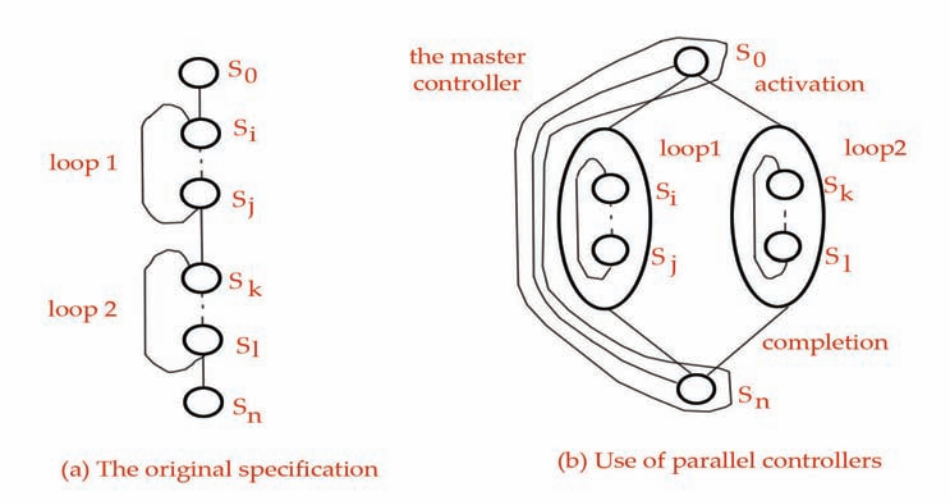


Figure 6.18: Using parallel controllers for two loops.

Conditions are used to specify synchronization between parallel controllers, when needed. The controller which is to be synchronized has a conditional state transition. This transition from one state to another will take place only in the case when the condition assigned to the edge connecting these two states is TRUE. The condition can be set by another controller. Using conditions different communication protocols between two or more parallel controllers can be implemented. Using this strategy we can specify hierarchical controllers with very complex synchronization schemes.

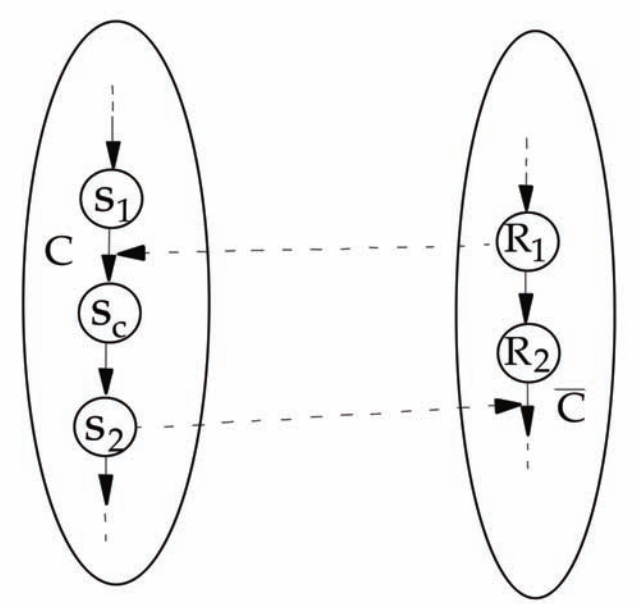


Figure 6.19: Two parallel controllers communicating using condition C.

Communication between parallel controllers can be achieved using a special protocol implemented using the basic technique described above together with data exchange through shared data path objects.

Example 6.14: Let us consider two parallel controllers which communicate as depicted in Figure 6.19. They use a handshake protocol to synchronize their execution and exchange data. Controller #1 waits in state S_1 for condition C to become TRUE. The condition is set by controller #2 in state R_1. In state S_e controller #1 can fetch data set by controller #2. Then controller #1 resets condition C and controller #2 continues its execution.

Many current design methods offer efficient algorithms for the synthesis of synchronous FSMs. In many cases, however this limits possible design space exploration and results in over-synchronized controllers. It should be noted that a proper combination of asynchronous and synchronous design styles can solve some of these problems. For example, basic controllers can be implemented as synchronous machines controlled by one clock or several independent clocks while their communication can follow an asynchronous protocol. This solution gives the freedom of using asynchronous design rules for controller synchronization and communication while using well-known synchronous design methods for single controllers.

It should be noted that this solution with activation and completion signals is the only possible one for the class of designs which have operations with unbounded-

delays. We cannot schedule these operations to any particular clock cycle or a number of clocks cycles since the execution time cannot be determined beforehand.

6.4.2 Controller Generation

When the controller style has been selected and the number of FSMs is decided we need to generate these FSMs. The FSMs can be represented for example as state diagrams or symbolic FSMs and later implemented using standard FSM synthesis methods.

The controller generation task has to decide whether the Moore or the Mealy machine should be used. This decision is mainly dependent on the design representation used and the available back-end logic synthesis tools. However every Mealy machine can be converted into an equivalent Moore machine and vice versa..

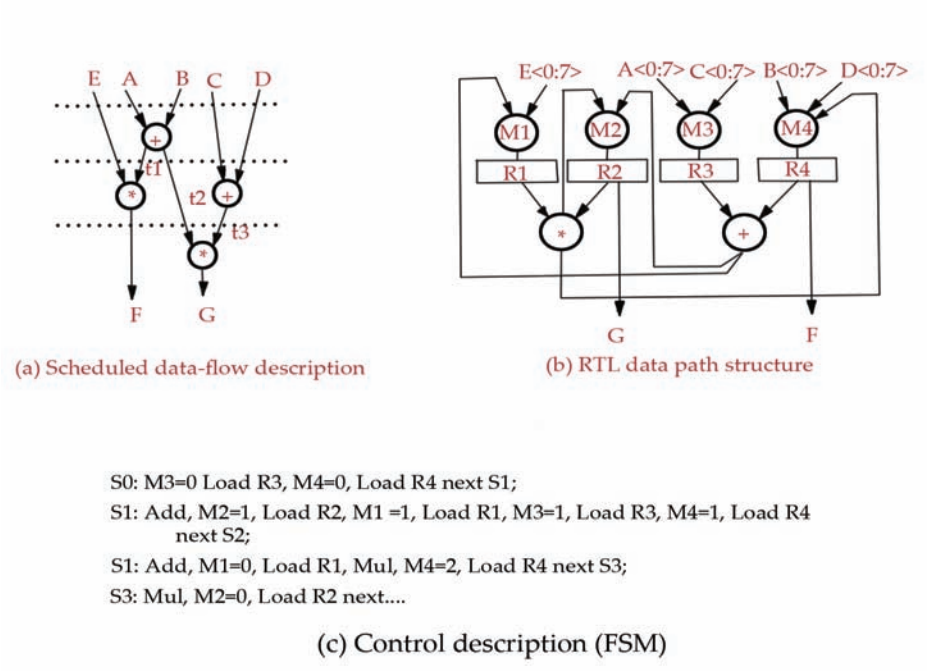


Figure 6.20: A design example with the generated FSM.

Example 6.15: Consider the scheduled data-flow graph given in Figure 6.20a which assumes the use of one adder and one multiplier. The schedule for this graph consists of three control steps. A data path implementation after the binding of functional units, registers and multiplexors is depicted in Figure 6.20b. Variables E and t_3 are assigned to register R_1 variables G. t_1 and t_2 to register R_2, variables A and C to register R_3, and

variables B. D and F to register R_4. Modules M_1. M_2. M_3 and M_4 represent multiplexors. The controller is given in Figure 6.20c in the form of a symbolic FSM instead of a state diagram. It has four states. The first state, S0, loads registers R_3 and R_4 with the values of variables A and B respectively. Control signals for multiplexors and registers are generated in this state. The signals M_3=0 and M_4=0 open the first input of the respective multiplexors while control signals Load R_3 and Load R_4 trigger the register loading. The next states perform the computations specified in the data-flow graph.

The VHDL description depicted on Figure 6.21 gives a possible synthesizable code for this FSM. The FSM has two input signals, reset and clock, and a number of control output signals. In this case, there are no input conditions which decide about the next-state selection. The description contains three processes: The process state_decade_logic implements the next-state function and the process output_decade_logic implements the output function. If the design would have had input condition signals the process state_decade_logic would have implemented the next-state selection as additional conditional statements, such as if-statement, instead of the main case-statement. The process state register implements the change of the state every clock cycle as well as the reselling of the state register to state SO on the reception of the reset signal.

6.4.3 Controller Implementation

Depending on the selected style of the controller a different controller structure has to be implemented. However, the basic structure is based on the implementation of a single FSM. This basic implementation, together with additional hardware for complex controller synchronization and communication, is used for other types of controllers.

The single FSM controller can be implemented using random logic, microcode or PLAs. The general implementation structure is very similar, not matter which technique is used. A state register is used to store the current state while combinational logic is used to generate the next state based on the current state and the conditions coming from the data path. The current state is also used to generate control signals for the data path since we assume the use of Moore machines. In some cases, additional decoding and coding can be used for the control signals and the conditions respectively. Figure 6.22 illustrates the general implementation structure of the basic controller.

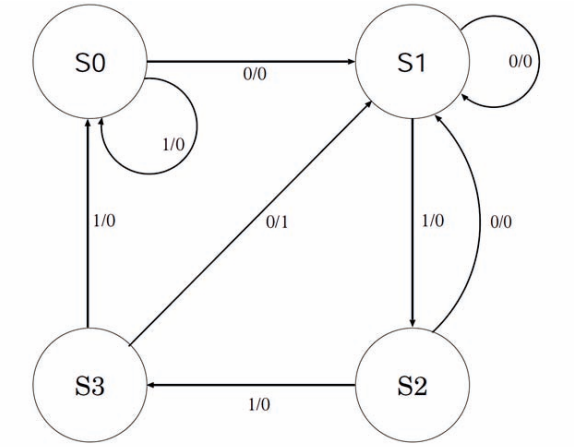


Figure 6.21: VHDL specification of the FSM given in Figure 6.20c.

The basic controller can then be used as a building block to create complex controllers. Both control signals and conditions generated by the basic controller are used for synchronization purpose. In addition, data path elements can be used to create complex communication facilities required by the given controller.

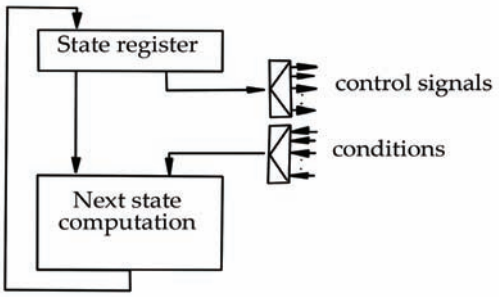


Figure 6.22: The general structure for the controller implementation.

Example 6.16: Figure 6.23 depicts a possible implementation of hierarchical controllers introduced earlier. The parent controller uses one of the control signals as an activation signal for a child controller. The child controller gets this signal as a condition signaling the start of the controller. After finishing its execution the child controller sends another control signal which is interpreted as a completion signal by the parent controller. Both the parent and the child controller have to use the same clock in this case.

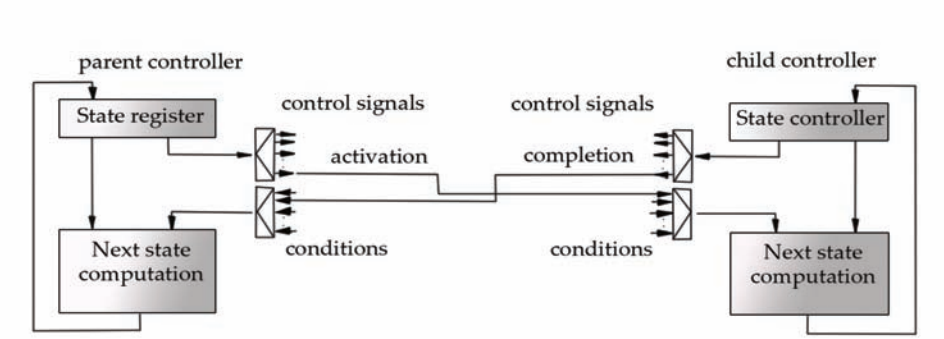


Figure 6.23: A hierarchical controller implementation.

Parallel controllers can synchronize using a method similar to the one sketched for hierarchical controllers. In more complex situations they require additional hardware for synchronization and data exchange. In general a protocol must be implemented between parallel controllers to provide correct means for communication. The most frequently used protocol is called the handshaking protocol. It uses request-acknowledge signals to establish communication. The controller which starts communication sends a request signal to the other controller and waits for the acknowledge signal. When the other controller receives the request signal it executes the needed operations and after that sends back the acknowledge signal. Both controllers continue their normal execution after the communication.

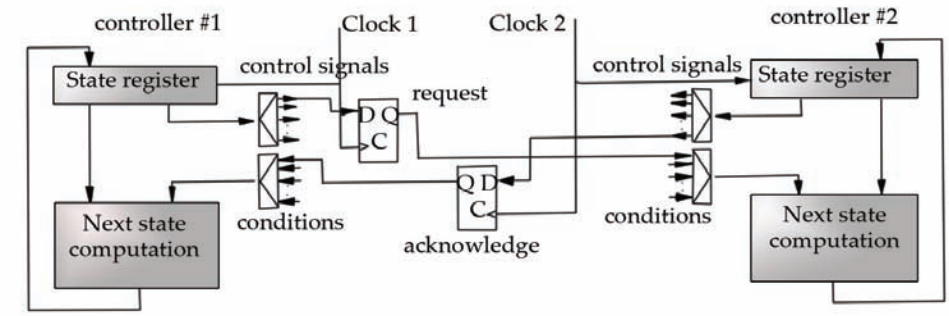


Figure 6.24: An implementation of a handshaking protocol between two parallel controllers.

Example 6.17: Figure 6.24 presents a possible implementation of the request-acknowledge protocol between two parallel controllers using two D flip-flops. Controller #1 sends a request signal by setting a D flip-flop to 1. It stays in this state until controller #2 sends an acknowledge signal by setting another D flip-flop to 1. This

state is recognized by the first controller which continues its execution. It can be noted that this solution makes it possible to use independent clocks for controllers. For example, controller #1 uses Clock 1 while controller #2 uses Clock 2, as illustrated in the figure. This means that although both controllers are synchronous themselves the communication between them are performed asynchronously.

SUMMARY

- The term synthesis is used to denote the process of transforming a digital system from a behavioral specification into an implementation structure.
- High-level synthesis will then translate the behavioral specification of a process into a structural description that is still technology independent.
- Logic synthesis and physical design form the back-end of the synthesis approach to digital system design.
- The basic components of the data path will eventually be implemented by some physical modules available in a given technology.
- Operation scheduling, or in short scheduling, deals with the assignment of each operation to a time slot corresponding to a clock cycle or time interval.
- The simplest scheduling technique is a greedy heuristics based on the "as soon as possible" (ASAP) principle.
- An .important decision for a list-scheduling approach is therefore to select the priority function.
- The forced-directed scheduling algorithm consists of three main steps: determine the time frame of each operation, create a distribution graph, and calculate the force associated with each assignment
- A transformation-based algorithm begins with an initial schedule, usually either maximally serial or maximally parallel, and applies transformations to it to obtain other schedules.
- Data path allocation and binding deal with the problem of which resources are used to realize in the physical implementation.
- Allocation and binding carry out selection and assignment of hardware resources for a given design.
- Controller synthesis is a field which fits into this vision but has been mainly oriented towards hardware engineering
- An efficient method to implement a controller is to use a finite state machine (FSM) notation.

KNOWLEDGE CHECK

1. In logic synthesis ..... is an EDIF that gives the description of logic cells & their interconnections.

a. Netlist

b. Checklist

c. Shitlist

d. Dualist

2. Which among the following operation/s is/are executed in physical design or layout synthesis stage?

a. Placement of logic functions in optimized circuit in target chip

b. Interconnection of components in the chip

c. Both a and b

d. None of the above

3. In VHDL, which class of scalar data type represents the values necessary for a specific operation?

a. Integer types

b. Real types

c. Physical types

d. Enumerated types

4. Which among the following is/are regarded as the function/s of translation step in synthesis process?

a. Conversion of RTL description to Boolean unoptimized description

b. Conversion of an unoptimized to optimized Boolean description

c. Conversion of unoptimized Boolean description to PLA format

d. All of the above

5. In synthesis flow, which stage/s is/are responsible for converting an unoptimized Boolean description to PLA format?

a. Translation

b. Optimization

c. Flattening

d. All of the above

REVIEW QUESTIONS

1. Define the task of high-level synthesis.
2. What is list scheduling? Explain.
3. Discuss on force-directed scheduling.
4. What do you understand by the integer linear programming?
5. Demonstrate the synthesis of controller generation and implementation.

CHECK YOUR RESULT

1. (a) 2. (c) 3. (d) 4. (a) 5. (c)

REFERENCES

1. https://ocw.tudelft.nl/wp-content/uploads/petru-3.pdf
2. https://pdfs.semanticscholar.org/312e/2aac37e8af4193a355878c3b2a1482697c7a.pdf
3. http://www.inf.ufrgs.br/~fglima/projeto/paper1.pdf
4. http://sportlab.usc.edu/download/qufd/qufd.pdf

INDEX

Level: Beginner to Advanced
Subject: Engineering and Technology

Algorithms for VLSI Design Automation

Gone are the days when huge computers made of vacuum tubes sat humming in entire dedicated rooms and could do about 360 multiplications of 10 digit numbers in a second. Though they were heralded as the fastest computing machines of that time, they surely don't stand a chance when compared to the modern day machines. Modern day computers are getting smaller, faster, and cheaper and more power efficient every progressing second. Very-large-scale integration (VLSI) is the process of creating an integrated circuit (IC) by combining thousands of transistors into a single chip. VLSI began in the 1970s when complex semiconductor and communication technologies were being developed. Electronic design automation (EDA), also referred to as electronic computer-aided design (ECAD), is a category of software tools for designing electronic systems such as integrated circuits and printed circuit boards. The tools work together in a design flow that chip designers use to design and analyze entire semiconductor chips. Since a modern semiconductor chip can have billions of components, EDA tools are essential for their design.

Key features:

- This Text covers all stages of design from layout synthesis through logic synthesis to high-level synthesis.
- Topics are logically arranged according to the course outline of the subject.
- The use of simple language to facilitate a better understanding of each chapter.
- Illustrative examples of varying difficulties, wherever needed, are presented to enhance the interest of students.
- The provision of knowledge check test for each chapter is given at the end of chapter to strengthen the learning process of students.
- Case Study, Role Model, Keywords, and Examples are also provided to enrich the knowledge of students.

The Book comes with a companion DVD for rich learning experience, which includes:

1. E-Book with further reading and learning links.
2. Interactive E-lecture of each chapter. E-lectures are expressive, informational, entertaining and persuasive, it uses the tool of self-exploration, which makes it easy to learn and understand each topic in detail. It is very informative as concrete details are provided and also entertaining, as graphics and other visuals are provided to make the learning process more interactive.
3. Video Lecture of each chapter, which explains each topic in detail with examples, animations, images and text and makes it easy to understand the topics in easier, simpler and better way.
4. Huge Database of Interactive Assessments for each chapter, which is also printable.
5. Further reading and learning links for each topic.
6. Glossary and Notes for each chapter to understand each chapter with to the point information.
7. The DVD also includes a printable workbook, which walks through with a various sets of questions and choices and assists in completing the curriculum. The workbook covers; Learning Objectives, Essential Concepts, Matching Definitions, Study Problem, Questions, Fill in the Blanks and Answers.
8. Review Questions for each chapter are also given in the DVD, which are also printable.

The DVD is also a useful tool for teachers to teach with digital resources in classroom and do a great job of illustrating skills and techniques that are otherwise difficult to explain.

ISBN 978-1-68095-509-5

3G E-LEARNING
www.3ge-learning.com
email: info@3ge-learning.com