The Circuits and Filters Handbook

Third Edition

Computer Aided Design and Design Automation

Edited by

Wai-Kai Chen

CRC Press
Taylor & Francis Group

Computer Aided Design and Design Automation

The Circuits and Filters Handbook

Third Edition

Edited by

Wai-Kai Chen

Fundamentals of Circuits and Filters

Feedback, Nonlinear, and Distributed Circuits

Analog and VLSI Circuits

Computer Aided Design and Design Automation

Passive, Active, and Digital Filters

The Circuits and Filters Handbook

Third Edition

Computer Aided Design and Design Automation

Edited by

Wai-Kai Chen

University of Illinois

Chicago, U. S. A.

CRC Press

Taylor & Francis Group

Boca Raton London New York

CRC Press is an imprint of the

Taylor & Francis Group, an **informa** business

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

Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number-13: 978-1-4200-5918-2 (Hardcover)

Library of Congress Cataloging-in-Publication Data

Computer aided design and design automation / Wai-Kai Chen.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-1-4200-5918-2
ISBN-10: 1-4200-5918-1
1. Engineering design. 2. Electric engineering--Computer-aided design. I. Chen, Wai-Kai, 1936- II. Title.

TA174.C58135 2009
621.3--dc22 2008048129

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

and the CRC Press Web site at
http://www.crcpress.com

Contents

Preface

Editor-in-Chief

Contributors

SECTION I Computer Aided Design and Optimization

1 Modeling of Circuit Performances
Sung-Mo Kang and Abhijit Dharchoudhury

2 Symbolic Analysis Methods
Benedykt S. Rodanski and Marwan M. Hassoun

3 Numerical Analysis Methods
Andrew T. Yang

4 Design by Optimization
Sachin S. Sapatnekar

5 Statistical Design Optimization
Maciej A. Styblinski

6 Physical Design Automation
Naveed A. Sherwani

7 Design Automation Technology
Allen M. Dewey

8 Computer-Aided Analysis
J. Gregory Rollins and Peter Bendix

9 Analog Circuit Simulation
J. Gregory Rollins

SECTION II Design Automation

10 Internet-Based Microelectronic Design Automation Framework
Moon-Jung Chung and Heechul Kim

11 System-Level Design
Alice C. Parker, Yosef Tirat-Gefen, and Suhrid A. Wadekar

12 Performance Modeling and Analysis Using VHDL and SystemC
Robert H. Klenke, Jonathan A. Andrews, and James H. Aylor

13 Embedded Computing Systems and Hardware/Software Codesign
Wayne Wolf

14 Design Automation Technology Roadmap
Donald R. Cottrell

Preface

As system complexity continues to increase, the microelectronic industry must possess the ability to adapt quickly to the market changes and new technology through automation and simulations. The purpose of this book is to provide in a single volume a comprehensive reference work covering the broad spectrum of computer aided design and optimization techniques, which include circuit performance modeling, design by optimization, statistical design optimization, physical design automation, computer aided analysis and circuit simulations; and design automation, which includes system-level design, performance modeling and analysis, and hardware/software codesign. This book is written and developed for the practicing electrical engineers and computer scientists in industry, government, and academia. The goal is to provide the most up-to-date information in the field.

Over the years, the fundamentals of the field have evolved to include a wide range of topics and a broad range of practice. To encompass such a wide range of knowledge, this book focuses on the key concepts, models, and equations that enable the design engineer to analyze, design, and predict the behavior of large-scale systems. While design formulas and tables are listed, emphasis is placed on the key concepts and theories underlying the processes.

This book stresses fundamental theories behind professional applications and uses several examples to reinforce this point. Extensive development of theory and details of proofs have been omitted. The reader is assumed to have a certain degree of sophistication and experience. However, brief reviews of theories, principles, and mathematics of some subject areas are given. These reviews have been done concisely with perception.

The compilation of this book would not have been possible without the dedication and efforts of Professor Steve Sung-Mo Kang, and most of all the contributing authors. I wish to thank them all.

Wai-Kai Chen

Editor-in-Chief

Wai-Kai Chen is a professor and head emeritus of the Department of Electrical Engineering and Computer Science at the University of Illinois at Chicago. He received his BS and MS in electrical engineering at Ohio University, where he was later recognized as a distinguished professor. He earned his PhD in electrical engineering at the University of Illinois at Urbana–Champaign.

Professor Chen has extensive experience in education and industry and is very active professionally in the fields of circuits and systems. He has served as a visiting professor at Purdue University, the University of Hawaii at Manoa, and Chuo University in Tokyo, Japan. He was the editor-in-chief of the *IEEE Transactions on Circuits and Systems, Series I and II*, the president of the IEEE Circuits and Systems Society, and is the founding editor and the editor-in-chief of the *Journal of Circuits, Systems and Computers*. He received the Lester R. Ford Award from the Mathematical Association of America; the Alexander von Humboldt Award from Germany; the JSPS Fellowship Award from the Japan Society for the Promotion of Science; the National Taipei University of Science and Technology Distinguished Alumnus Award; the Ohio University Alumni Medal of Merit for Distinguished Achievement in Engineering Education; the Senior University Scholar Award and the 2000 Faculty Research Award from the University of Illinois at Chicago; and the Distinguished Alumnus Award from the University of Illinois at Urbana–Champaign. He is the recipient of the Golden Jubilee Medal, the Education Award, and the Meritorious Service Award from the IEEE Circuits and Systems Society, and the Third Millennium Medal from the IEEE. He has also received more than a dozen honorary professorship awards from major institutions in Taiwan and China.

A fellow of the Institute of Electrical and Electronics Engineers (IEEE) and the American Association for the Advancement of Science (AAAS), Professor Chen is widely known in the profession for the following works: *Applied Graph Theory* (North-Holland), *Theory and Design of Broadband Matching Networks* (Pergamon Press), *Active Network and Feedback Amplifier Theory* (McGraw-Hill), *Linear Networks and Systems* (Brooks/Cole), *Passive and Active Filters: Theory and Implements* (John Wiley), *Theory of Nets: Flows in Networks* (Wiley-Interscience), *The Electrical Engineering Handbook* (Academic Press), and *The VLSI Handbook* (CRC Press).

Contributors

Jonathan A. Andrews
Department of Electrical and Computer Engineering
Virginia Commonwealth University
Richmond, Virginia

James H. Aylor
Department of Electrical Engineering
University of Virginia
Charlottesville, Virginia

Peter Bendix
Technology Modeling Associates, Inc.
Palo Alto, California

Wai-Kai Chen
Department of Electrical and Computer Engineering
University of Illinois at Chicago
Chicago, Illinois

Moon-Jung Chung
Department of Computer Science
Michigan State University
East Lansing, Michigan

Donald R. Cottrell
Silicon Integration Initiative, Inc.
Austin, Texas

Allen M. Dewey
Department of Electrical and Computer Engineering
Duke University
Durham, North Carolina

Abhijit Dharchoudhury
Beckman Institute
University of Illinois at Urbana–Champaign
Urbana, Illinois

Marwan M. Hassoun
Department of Electrical and Computer Engineering
Iowa State University
Ames, Iowa

Sung-Mo Kang
Office of Chancellor
University of California, Merced
Merced, California

Heechul Kim
Department of Computer Science and Engineering
Hankuk University of Foreign Studies
Yongin, Korea

Robert H. Klenke
Department of Electrical and Computer Engineering
Virginia Commonwealth University
Richmond, Virginia

Alice C. Parker
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, California

Benedykt S. Rodanski
Faculty of Engineering
University of Technology
Sydney, New South Wales, Australia

J. Gregory Rollins
Technology Modeling Associates, Inc.
Palo Alto, California

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

Naveed A. Sherwani
Intel Corporation
Hillsboro, Oregon

Maciej A. Styblinski
Department of Electrical Engineering
Texas A&M University
College Station, Texas

Yosef Tirat-Gefen
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, California

Suhrid A. Wadekar
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, California

Wayne Wolf
Department of Electrical Engineering
Princeton University
Princeton, New Jersey

Andrew T. Yang
Apache Design Solutions, Inc.
San Jose, California

I

Computer Aided Design and Optimization

Sung-Mo Kang
University of California, Merced

1 Modeling of Circuit Performances *Sung-Mo Kang and Abhijit Dharchoudhury* **1**-1
Introduction • Circuit Performance Measures • Input–Output Relationships • Dependency of Circuit Performances on Circuit Parameters • Design of Experiments • Least-Squares Fitting • Variable Screening • Stochastic Performance Models • Other Performance Modeling Approaches • Example of Performance Modeling • References

2 Symbolic Analysis Methods *Benedykt S. Rodanski and Marwan M. Hassoun* **2**-1
Introduction • Symbolic Modified Nodal Analysis • Solution Methods • Ideal Operational Amplifiers • Applications of Symbolic Analysis • Symbolic Analysis Software Packages • References

3 Numerical Analysis Methods *Andrew T. Yang* **3**-1
Equation Formulation • Solution of Linear Algebraic Equations • References

4 Design by Optimization *Sachin S. Sapatnekar* **4**-1
Introduction • Optimization Algorithms • Transistor Sizing Problem for CMOS Digital Circuits • Analog Design Centering Problem • References

5 Statistical Design Optimization *Maciej A. Styblinski* **5**-1
Introduction • Problems and Methodologies of Statistical Circuit Design • Underlying Concepts and Techniques • Statistical Methods of Yield Optimization • Conclusion • References

6 Physical Design Automation *Naveed A. Sherwani* **6**-1
Introduction • Very Large-Scale Integration Design Cycle • Physical Design Cycle • Design Styles • Partitioning • Other Partitioning Algorithms • Placement • Routing • Classification of Global Routing Algorithms • Classification of Detailed Routing Algorithms • Compaction • Summary • References

7 Design Automation Technology *Allen M. Dewey* **7**-1
Introduction • Design Entry • Conceptual Design • Synthesis • Verification • Test • Frameworks • Summary • References

8 Computer-Aided Analysis *J. Gregory Rollins and Peter Bendix* **8**-1
Circuit Simulation Using SPICE and SUPREM • Appendix A • References • Parameter Extraction for Analog Circuit Simulation • References

9 Analog Circuit Simulation *J. Gregory Rollins* **9**-1
Introduction • Purpose of Simulation • Netlists • Formulation of the Circuit Equations • Modified Nodal Analysis • Active Device Models • Types of Analysis • Verilog-A • Fast Simulation Methods • Commercially Available Simulators • References

1

Modeling of Circuit Performances

1.1 Introduction 1-1
1.2 Circuit Performance Measures 1-2
1.3 Input–Output Relationships 1-3
Memoryless (Static) Relationship • Dynamic Relationships
1.4 Dependency of Circuit Performances on Circuit Parameters 1-3
1.5 Design of Experiments 1-5
Factorial Design • Central Composite Design • Taguchi's Orthogonal Arrays • Latin Hypercube Sampling
1.6 Least-Squares Fitting 1-11
1.7 Variable Screening 1-13
1.8 Stochastic Performance Models 1-14
1.9 Other Performance Modeling Approaches 1-14
1.10 Example of Performance Modeling 1-15
References 1-16

Sung-Mo Kang
University of California, Merced

Abhijit Dharchoudhury
University of Illinois at Urbana–Champaign

1.1 Introduction

The domain of computer-aided design and optimization for circuits and filters alone is very broad, especially since the design objectives are usually multiple. In general, the circuit design can be classified into

- Electrical design
- Physical design

Physical design deals with concrete geometrical parameters while electrical design deals with the electrical performances of the physical object. Strictly speaking, both design aspects are inseparable since any physical design is a realization of a particular electrical design. On the other hand, electrical design without proper abstraction of the corresponding physical design can be futile in the sense that its implementation may not meet the design goals. Thus, electrical and physical design should go hand in hand. Most literature deals with both designs using weighted abstractions. For example, in the design of a digital or analog filter, various effects of physical layout are first captured in the parasitic models for interconnects and in the electrical performance models for transistors. Then, electrical design is performed using such models to predict electrical performances. In physical design optimization, geometrical parameters are chosen such that the predicted electrical performances can meet the design

objectives. For example, in the timing-driven physical layout of integrated circuits, circuit components will be laid out, placed and interconnected to meet timing requirements. In order to make the optimization process computationally efficient, performance models are used during the optimization process instead of simulation tools. In this chapter, the focus of design optimization is on the electrical performance. The computer-aided electrical design can be further classified into performance analysis and optimization. Thus, this chapter first discusses the modeling of electrical performances, symbolic and numerical analysis techniques, followed by optimization techniques for circuit timing, and yield enhancement with transistor sizing and statistical design techniques.

1.2 Circuit Performance Measures

The performance of a particular circuit can be measured in many ways according to application and design goals. In essence, each performance measure is an indication of the circuit output which the designer is interested in for a particular operating condition. The circuit performance, in general, is dependent on the values of the designable parameters such as the transistor sizes, over which the designer has some degree of control, and the values of the noise parameters such as the fluctuation in power supply voltage or the threshold voltages of the transistors, which are random in nature. Also, the performance measure used is dependent on the type of the circuit. The performance measures of digital circuits are usually quite different from those of analog circuits or filters. Some of the important performance measures of digital circuits are

- Signal propagation delay from an input to an output
- Rise and fall times of signals
- Clock signal skew in the circuit
- Power dissipation
- Manufacturing yield
- Failure rate (reliability)

Important performance measures for analog circuits are

- Small-signal gain and bandwidth in frequency domain
- Sensitivities to various nonideal factors (noise)
- Slew rate of operational amplifiers
- Power dissipation
- Manufacturing yield
- Failure rate (reliability)

The modeling of digital circuit performances is usually more difficult to derive than the modeling of analog circuit performances, mainly due to the nonlinear nature of digital circuits. Thus, simple transistor models have been used often to obtain simple qualitative models for propagation delays, rise and fall times, and power dissipation. On the other hand, due to the small-signal nature, the device models required by analog designers need to be much more accurate. Another important performance measure is the manufacturing yield, which determines the product cost. The yield depends on both the quality of manufacturing process control and the quality of the design. A challenging design problem is how to come up with a design which can be robust to nonideal manufacturing conditions and thereby produce a good manufacturing yield. The model for failure rate is difficult to derive due to several complex mechanisms that determine the lifetime of chips. In such cases, empirical models, which are not physical, have been introduced.

In order to develop models for analog circuits, transistors in the circuit are first DC-biased at the proper biasing points. Then, small-signal circuit models are developed at those DC operating points. The resulting linear circuits can be analyzed by either using mathematical symbols (symbolic analysis) or by using simulators.

1.3 Input–Output Relationships

The circuit functionally depends on the input–output relationships and thus circuit specifications and test specifications are made on the basis of the input–output relationships. For modeling purposes, the input–output relationships can be classified into static relationships and dynamic relationships.

1.3.1 Memoryless (Static) Relationship

A given input–output pair is said to have a memoryless relationship if it can be described statically without the use of any ordinary or partial differential equation. It is important to note that the choice of input and output variables can determine the nature of the relationship. For instance, the relationship between input charge q and the output voltage v in a 1 F capacitor is static. On the other hand, the relationship between an input current i and the output voltage v, in the same capacitor is nonstatic since the voltage is dependent on the integral of the input current. Ideally, basic circuit elements such as resistor, capacitor, inductor, and source should be characterized by using static relationships, so-called constitutive relationships, with proper choice of input and output variables. However, for circuits and systems design, since their input–output relationships are specified in terms of specific variables of interest, such choice of variables cannot be made arbitrarily. Often, both input and output variables are voltages, especially in digital circuits.

1.3.2 Dynamic Relationships

A given input–output pair is said to have a dynamic relationship if the modeling of the relationship requires a differential equation. In most cases, since the circuits contain parasitic capacitors or inductors or both, the input–output relationship is dynamic. For instance, the input–output relationship of an inverter gate with capacitive load C_{load} can be described as

$$C_{\text{load}} \frac{\mathrm{d}V_{\text{out}}}{\mathrm{d}t} = i(V_{\text{out}}, V_{\text{in}}, V_{\text{DD}}) \tag{1.1}$$

where

V_{out} and V_{in} denote the output and input voltages
i denotes the current through the pull-up or pull-down transistor
V_{DD} denotes the power supply voltage

Although the relationship is implicit in this case, Equation 1.1 can be solved either analytically or numerically to find the relationship. Equation 1.1 can be generalized by including noise effects due to variations in the circuit fabrication process.

1.4 Dependency of Circuit Performances on Circuit Parameters

Circuit performances are functions of circuit parameters, which are usually not known explicitly. In order to design and analyze analog or digital circuits, the dependency of the circuit performances on the various circuit parameters need to be modeled.

The actual value of a circuit parameter in a manufactured circuit is expected to be different from the nominal or target value due to inevitable random variations in the manufacturing processes and in the environmental conditions in which the circuit is operated. For example, the actual channel width W of an MOS transistor can be decomposed into a nominal component W^0 and a statistically varying component ΔW, i.e., $W = W^0 + \Delta W$. The nominal component is under the control of the designer and can be set to a particular value. Such a nominal component is said to be designable or controllable, e.g., W^0.

The statistically varying component, on the other hand, is not under the control of the designer and is random in nature. It is called the noise component, and it represents the uncontrollable fluctuation of a circuit parameter about its designable component, e.g., ΔW. For certain circuit parameters, e.g., device model parameters (like the threshold voltages of MOS transistors) and operating conditions (like temperature or power supply voltage), the nominal values are not really under the control of the circuit designer and are set by the processing and operating conditions. For these circuit parameters, the nominal and random components are together called the noise component. In general, therefore, a circuit parameter x_i can be expressed as follows:

$$x_i = d_i + s_i \tag{1.2}$$

where
d_i is the designable component
s_i is the noise component

It is common to group all the designable components to form the set of designable parameters, denoted by **d**. Similarly, all the noise parameters are grouped together to form the noise parameters, denoted by the random vector **s**. Vectorially, Equation 1.2 can be rewritten as

$$\mathbf{x} = \mathbf{d} + \mathbf{s} \tag{1.3}$$

A circuit performance measure is a function of the circuit parameters. For example, the propagation delay of a CMOS inverter circuit is a function of the nominal channel lengths and widths of the NMOS and PMOS transistors. It is also a function of the threshold voltages of the transistors and the power supply voltage. Thus, in general, a circuit performance is a function of the designable as well as the noise parameters:

$$y = y(\mathbf{x}) = y(\mathbf{d}, \mathbf{s}) \tag{1.4}$$

If this function is known explicitly in terms of the designable and noise parameters, then optimization methods can be applied directly to obtain a design which is optimal in terms of the performances or the manufacturing yield. More often than not, however, the circuit performances cannot be expressed explicitly in terms of the circuit parameters. In such cases, the value of a circuit performance for given values of circuit parameters can be obtained by circuit simulation. Circuit simulations are computationally expensive, especially if the circuit is large and transient simulations are required. A compact performance model in terms of the circuit parameters is an efficient alternative to the circuit simulator. Two factors determine the utility of the performance model. First, the model should be computationally efficient to construct and evaluate so that substantial computational savings can be achieved. Second, the model should be accurate.

Figure 1.1 shows the general procedure for constructing the model for a performance measure y in terms of the circuit parameters **x**. The model-building procedure consists of four steps. In the first stem, *m training points* are selected from the **x**-space. The ith training point is denoted by $\mathbf{x}_i$, $i = 1, 2, \ldots, m$. In the second step, the circuit is simulated at these m training points and the values of the performance measure are obtained from the circuit simulation results as $y(\mathbf{x}_1)$, $y(\mathbf{x}_2)$, $\ldots, y(\mathbf{x}_m)$. In the third step, a preassigned function of y in terms of **x** is "fitted" to the data. In the fourth and final step, the model is validated for accuracy. If the model accuracy is deemed inadequate, the modeling procedure is repeated with a larger number of training points or with different models.

The model is called the response surface model (RSM) [1] of the performance and is denoted by $\hat{y}(\mathbf{x})$. The computational cost of modeling depends on the number of training points m, and the procedure of fitting the model to the data. The accuracy of the RSM is quantified by computing error measures which

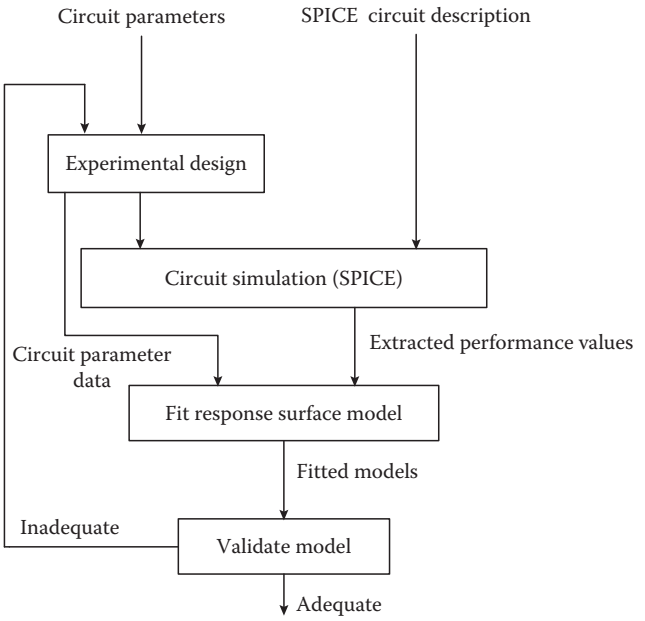


FIGURE 1.1 Performance modeling procedure.

quantify the "goodness of fit." The accuracy of the RSM is greatly influenced by the manner in which the training points are selected from the **x**-space. Design of experiment (DOE) [2] techniques are a systematic means of obtaining the training points such that the maximum amount of information about the model can be obtained from these simulations or experimental runs. In the next section, we review some of the most commonly used experimental design techniques in the context of performance optimization and statistical design. To illustrate certain points in the discussion below, we will assume that the RSM of the performance measure y is the following quadratic polynomial in terms of the circuit parameters:

$$\hat{y}(\mathbf{x}) = \hat{\alpha}_0 + \sum_{i=1}^{n} \hat{\alpha}_i x_i + \sum_{i=1}^{n} \sum_{j=1}^{n} \hat{\alpha}_{ij} x_i x_j \tag{1.5}$$

where the $\hat{\alpha}$s are the coefficients of the model. Note, however, that the discussion is valid for all RSMs in general.

1.5 Design of Experiments

1.5.1 Factorial Design

In this experimental design, each of the circuit parameters $x_1, x_2, \ldots, x_n$ is quantized into two levels or settings, which are denoted by -1 and $+1$. A full factorial design contains all possible combinations of levels for the n parameters, i.e., it contains 2^n training points or experimental runs. The design matrix for the case of $n = 3$ is shown in Table 1.1 and the design is pictorially described in Figure 1.2. The value of the circuit performance measure for the kth run is denoted by y_k.

Much information about the relationship between the circuit performance y and the circuit parameters x_i, $i = 1, 2, \ldots, n$ can be obtained from a full factorial experiment. The main or individual effect of a

TABLE 1.1 Full Factorial Design for $n = 3$

	Parameter Levels			Interaction Levels				
Run	x_1	x_2	x_3	$x_1 \times x_2$	$x_1 \times x_3$	$x_2 \times x_3$	$x_1 \times x_2 \times x_3$	y
1	−1	−1	−1	+1	+1	+1	−1	y_1
2	−1	−1	+1	+1	−1	−1	+1	y_2
3	−1	+1	−1	−1	+1	−1	+1	y_3
4	−1	+1	+1	−1	−1	+1	−1	y_4
5	+1	−1	−1	−1	−1	+1	+1	y_5
6	+1	−1	+1	−1	+1	−1	−1	y_6
7	+1	+1	−1	+1	−1	−1	−1	y_7
8	+1	+1	+1	+1	+1	+1	+1	y_8

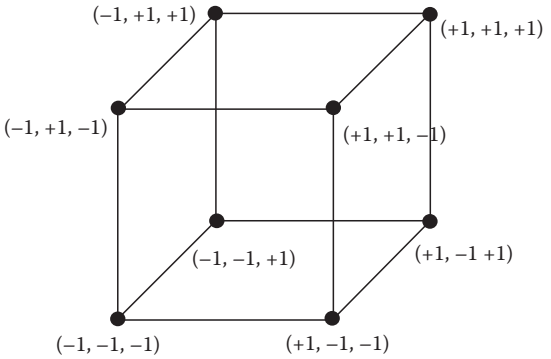


FIGURE 1.2 Pictorial representation of full factorial design for $n = 3$.

parameter quantifies how much a parameter affects the performance singly. The main effect of x_i, denoted by v_i, is given by

$$v_i = \frac{1}{2^n} \sum_{k=1}^{2^n} x_{ik} \times y_k \tag{1.6}$$

where

x_{ik} is the value of x_i in the kth run
y_k is the corresponding performance value

Thus, the main effect is the difference between the average performance value when the parameter is at the high level (+1) and the average performance value when the parameter is at the low level (−1). The main effect of x_i is the coefficient of the x_i term in the polynomial RSM of Equation 1.5. Similarly, the interaction effect of two or more parameters quantities how those factors jointly affect the performance. The two-factor interaction effect is computed as the difference between the average performance value when both factors are at the same level and the average performance value when they are at different levels. The two-factor interaction effect of parameters x_i and x_j, denoted by $v_{i\times j}$, $i \neq j$, is defined as follows:

$$\begin{aligned} v_{i\times j} &= \frac{1}{2}\left\{v_{i|j=+1} - v_{i|j=-1}\right\} \\ &= \frac{1}{2^n} \sum_{k=1}^{2^n} x_{ik} \times x_{jk} \times y_k \end{aligned} \tag{1.7}$$

The two-factor interaction effect of x_i and x_j, $v_{i\times j}$, is the coefficient of the $x_i x_j$ terms in the quadratic RSM of Equation 1.5. Note that $v_{i\times j}$ is the same as $v_{j\times i}$. Moreover, higher-order multifactor interaction effects can be computed recursively as before. Note that the interaction columns of Figure 1.2 are obtained by the aggregated ''sign-multiplication'' given in Equation 1.7.

Thus, the full factorial design allows us to estimate the coefficients of all the first-order and cross-factor second-order coefficients in the RMS of Equation 1.5. However, it does not allow us to estimate the coefficients of the pure quadratic term x_i^2. Moreover, the number of experimental runs increases exponentially with the number of circuit parameters; this may become impractical for a large number

TABLE 1.2 Half Fraction of a Full Factorial Design for $n = 3$

	Parameter Levels			Interaction Levels				
Run	x_1	x_2	x_3	$x_1 \times x_2$	$x_1 \times x_3$	$x_2 \times x_3$	$x_1 \times x_2 \times x_3$	y
1	−1	−1	+1	+1	−1	−1	+1	y_1
2	−1	+1	−1	−1	+1	−1	+1	y_2
3	+1	−1	−1	−1	−1	+1	+1	y_3
4	+1	+1	+1	+1	+1	+1	+1	y_4

of circuit parameters. In many cases, high-order multifactor interactions are small and can be ignored. In such cases, it is possible to reduce the size of the design by systematically eliminating some interactions and considering only some of the runs. The accuracy with which the main effects and the low-order interaction effects are estimated does not have to be compromised. Such designs are called fractional factorial designs. The most popular class of fractional factorial designs are the 2^{n-p} designs, where a $1/2^p$th fraction of the original 2^n design is used. One of the half fractions of the 2^3 design shown in Table 1.1, i.e., a 2^{3-1} design is shown in Table 1.2.

We observe that some of the columns of Table 1.2 are now identical. It can be seen from Equations 1.6 and 1.7 that we cannot distinguish between effects that correspond to identical columns. Such effects are said to be confounded or aliased with each other. From Table 1.2, we see that v_3, the main effect of x_3, and $v_{1\times2}$, the interaction effects of x_1 and x_2, are confounded with each other. Moreover the three-factor interaction effect $v_{1\times2\times3}$ is confounded with the grand average of the performance (corresponding to a column of 1's). Confounding, however, is not really a problem, since in most applications, high-order interaction effects are negligible. For example, in the quadratic RSM of Equation 1.5, only main effects and two-factor interaction effects are important, and it can be assumed that all higher-order interaction effects are absent.

There is a systematic way of obtaining fractional factorial designs by first isolating a set of basic factors (circuit parameters) and then defining the remaining nonbasic factors in terms of the interaction of the basic factors. These interactions that define the nonbasic factors in terms of the basic factors are called generators. The set of basic factors and the generator functions completely characterize a fractional factorial design. For example, the generator function used in the example of Table 1.2 is

$$x_3 = x_1 \times x_2 \tag{1.8}$$

Note that, once the generator function is known for a fractional design, the confounding pattern is also completely known. Moreover, note that the generator functions are not unique. The generator functions $x_1 = x_2 \times x_3$ and $x_2 = x_1 \times x_3$ also give rise to the design of Table 1.2. This can be understood by realizing that a column corresponding to a parameter sign-multiplied by itself results in a column of all 1s, i.e., the identity column, denoted by I. Thus, multiplying both sides of Equation 1.8 by x_2, we get $x_3 \times x_2 = x_1 \times x_2 \times x_2 = x_1 \times I = x_1$, which is also a generator function. Thus, an appropriate set of generator functions must be defined.

Another important concept related to confounding is called the resolution of a design, which indicates the clarity with which individual effects and interaction effects can be evaluated in a design. A *resolution III* design is one in which no two main effects are confounded with each other (but they may be confounded with two-factor and/or higher-order interaction effects). An important class of resolution *III* designs is the saturated fractional factorial designs. Suppose that the number of factors (parameters), n, can be expressed as $n = 2^q - 1$, for some integer q. Then, a resolution *III* design can be obtained by first selecting q basic factors and then saturating the 2^q design by assigning a nonbasic factor to each of the possible combinations of the q basic factors. There are $p = n - q$ nonbasic factors and generator functions, and these

designs are called 2^{n-p}_{III} designs, since $2^q = n+1$ runs are required for n factors. Note that the fractional factorial design of Table 1.2 is a 2^{n-p}_{III} design, with $n=3$ and $p=1$. In the saturated design for $n=2^q-1$ factors, the confounding patterns for the basic and nonbasic factors are the same, which implies that any of the q factors may be chosen as the basic factors. However, if $n+1$ is not an integral power of 2, then q is chosen as $q=[\log_2(n+1)]$. Next, we define q basic factors and generate a 2^q design. Next, we assign $p=n-q$ nonbasic factors to p generators and form the 2^{n-p}_{III} design. Note that, in this case, only p of the 2^q-q-1 available generators are used, and the confounding pattern depends on the choice of the q basic factors and the p generators. The number of runs in the saturated design is considerably less than 2δ.

Another important class of designs commonly used is called *resolution V* designs. In these designs, no main effect or two-factor interaction effect is confounded with another main effect or two-factor interaction effect. Such designs are denoted by 2^{n-p}_{V} designs, where as before, n is the number of factors and p is the number of generators. These designs can be used to estimate the coefficients of all but the pure quadratic terms in the RSM of Equation 1.5. Since the number of such coefficients (including the constant term) is $C=1+n+n(n-1)/2$, there must be at least C runs. Note that C is still substantially less than 2^n. Another important advantage of the factorial designs is that they are orthogonal, which implies that the model coefficients can be estimated with minimum variances (or errors). An algorithmic method for deriving resolution V designs is given in [3].

1.5.2 Central Composite Design

As mentioned previously, one of the problems of factorial designs in regard to the RMS of Equation 1.5 is that the coefficients of the pure quadratic terms cannot be estimated. These coefficients can be estimated by using a central composite design. A central composite design is a combination of a "cube" and a "star" subdesign. Each factor in a central composite design takes on five values: 0, ± 1, and $\pm\gamma$. Figure 1.3 shows the central composite design for the case of $n=3$. The cube subdesign, shown in dotted lines in Figure 1.3, is a fractional factorial design with the factor levels set to ± 1. This design is of resolution V so that linear and cross-factor quadratic terms in the RSM of Equation 1.5 can be estimated. The star

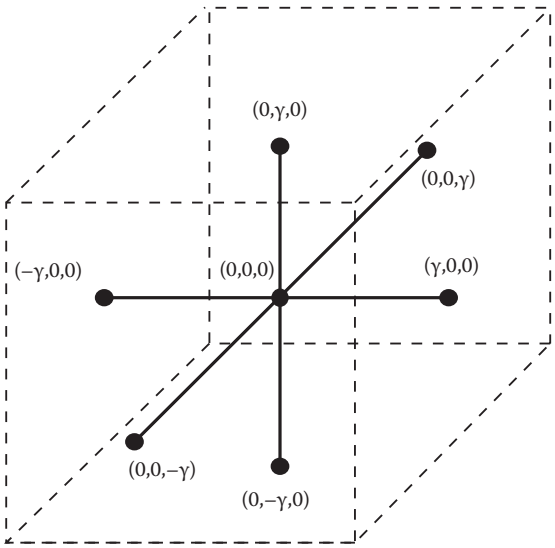


FIGURE 1.3 Central composite design for $n=3$.

subdesign, shown in solid lines in Figure 1.3, is used to estimate the pure quadratic terms, x_1^2, in the RMS, and consists of

- One *center* point, where all the parameters are set to 0
- $2n$ *Axial* points, one pair for each parameter by setting its value to $-\gamma$ and $+\gamma$ and setting all other parameters to 0

The parameter γ (chosen by the user) is selected so that the composite plan satisfies the rotatability property [1]. The main advantage of the central composite design is that all the coefficients of Equation 1.5 can be estimated using a reasonable number of simulations.

1.5.3 Taguchi's Orthogonal Arrays

Taguchi's method using orthogonal arrays (OA) [4] is another popular experimental design technique. An orthogonal array is a special kind of fractional factorial design. These orthogonal arrays can be classified into two types. The first category of OAs corresponds to two-level designs, i.e., the parameters are quantized to two levels each, while the second corresponds to three-level designs, i.e., the parameters are quantized to three levels each. These arrays are often available as tables in books which discuss Taguchi techniques [4]. As an example, we show the L18 OA in Table 1.3. The number in the designation of the array refers to the number of experimental runs in the design. The L18 OA belongs to the second category of designs, i.e., each parameter has three levels. In the Taguchi technique, the experimental design matrix for the controllable or designable parameters is called the inner array, while that for the noise parameters is called the outer array. The L18 design of Table 1.3 can be used as an inner array as well as an outer array.

1.5.4 Latin Hypercube Sampling

The factorial, central composite, and Taguchi experimental designs described earlier set the parameters to certain levels or quantized values within their ranges. Therefore, most of the parameters space remains

TABLE 1.3 Taguchi's L18 OA

Run	Parameter Levels							
1	1	1	1	1	1	1	1	1
2	1	1	2	2	2	2	2	2
3	1	1	3	3	3	3	3	3
4	1	2	1	1	2	2	3	3
5	1	2	2	2	3	3	1	1
6	1	2	3	3	1	1	2	2
7	1	3	1	1	1	3	2	3
8	1	3	2	3	2	1	3	1
9	1	3	3	1	3	2	1	2
10	2	1	1	3	3	2	2	1
11	2	1	2	1	1	3	3	2
12	2	1	3	2	2	1	1	3
13	2	2	1	2	3	1	3	2
14	2	2	2	3	1	2	1	3
15	2	2	3	1	2	3	2	1
16	2	3	1	3	2	3	1	2
17	2	3	2	1	3	1	2	3
18	2	3	3	2	1	2	3	1

unsampled. It is therefore desirable to have a more ''space-filling'' sampling strategy. The most obvious method of obtaining a more complete coverage of the parameter space is to perform random or Monte Carlo sampling [5]. In random sampling, each parameter has a particular statistical distribution and the parameters are considered to be statistically independent. The distribution of the circuit parameters can often be obtained from test structure measurements followed by parameter extraction. Often, however, based on previous experience or reasonable expectation, the circuit parameters are considered to be normal (or Gaussian). Since the designable parameters are deterministic variables, they are considered to be independent and uniformly distributed for the purposes of random sampling. Random samples are easy to generate and many inferences can be drawn regarding the probability distribution of the performances. The problem with random sampling, however, is that a large sample is usually required to estimate quantities with sufficiently small errors. Moreover, circuit parameters are usually not statistically independent. In particular, the noise parameters are statistically correlated because of the sequential nature of semiconductor wafer processing. Further, a random sample may not be very space filling either. This can be understood by considering the bell-shaped Gaussian density curve. In random sampling, values near the peak of the bell-shaped curve would be more likely to occur in the sample since such values have higher probability of occurrence. In other words, values away from the central region would not be well represented in the sample.

Latin hypercube sampling (LHS) [6,7] is a sampling strategy that alleviates this problem to a large extent. For each parameter x_i, all portions of its distribution are represented by sample values. If S is the desired size of the sample, then the range of each x_i, $i = 1, 2, \ldots, n$ is divided into S nonoverlapping intervals of equal marginal probability $1/S$. Each such interval is sampled once with respect to the probability density in that interval to obtain S values for each parameter. Next, the S values for one parameter are randomly paired with the S values for another parameter, and so on. Figure 1.4 illustrates this process for the case of two circuit parameters x_1 and x_2, where x_1 is a uniform random variable, and x_2 is a Gaussian random variable, and $S = 5$. The marginal probability in an interval is defined as the area under the probability density curve for that interval. Therefore, intervals with equal probability are intervals with equal areas under the probability density curve. For the uniformly distributed x_1, equal probability intervals are also of equal length. For x_2, however, the intervals near the center are smaller (since the density there is higher) than the intervals away from the center (where the density is lower). Figure 1.4 shows that $S = 5$ values are chosen from each region for x_1 and x_2 (shown as open circles). These values are then randomly paired so as to obtain the sample points (shown as filled dots). If the circuit parameters are correlated, then the pairing of the sample values can be controlled so that

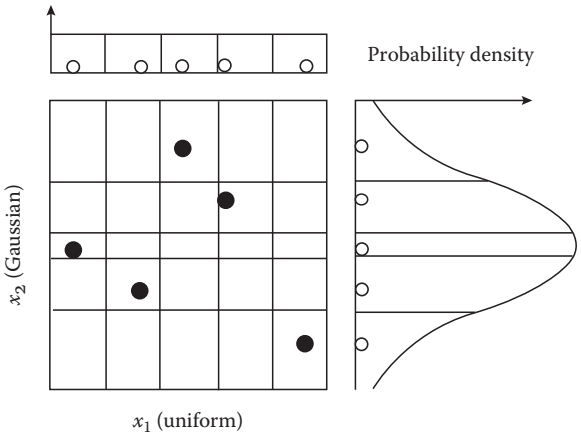


FIGURE 1.4 Latin hypercube sampling for a uniform and a Gaussian random variable.

the sample correlation matrix is close to the user-provided correlation matrix for the circuit parameters. Note that LHS will provide a more uniform coverage of the parameter space than other experimental design techniques. Moreover, a sample of any size can be easily generated and all probability densities are handled.

1.6 Least-Squares Fitting

After the experimental design has been used to select the training points in the parameter space, the circuit is simulated at the training points and the values of the performance measure are extracted from the simulation results. Let S denote the number of experimental runs, and y_k, $k = 1, \ldots, S$ denote the corresponding performance values. The aim in least-squares fitting is to determine the coefficients in the RSM so that the fitting error is minimized. Let C denote the number of coefficients in the RSM. For example, the value of C for the RSM of Equation 1.5 is $C = (n+1)(n+2)/2$, where n is the number of circuit parameters. There are interpolation-based methods that can be used to determine the coefficients when there are a smaller number of data points than coefficients, i.e., when $S < C$. One such method is called maximally flat quadratic interpolation and is discussed later in this chapter. We will restrict our discussion to the case of $S \geq C$. If $S = C$, then there are as many data points as coefficients, and simple simultaneous equation solving will provide the values of the coefficients. In this case, the model interpolates the data points exactly, and there is no fitting error to be minimized. When $S > C$, the coefficients can be determined in such a way that the fitting error is minimized. This method is called least-squares fitting (in a numerical analysis context) or linear regression (in a statistical context). The error measure is called the sum of squared errors and is given by

$$\in = \sum_{i=1}^{S} (y_k - \hat{y}_k)^2 \tag{1.9}$$

where

y_k denotes the simulated performance value
$\hat{y}_k$ denotes the model-predicted value at the kth data point

Least-squares fitting can then be stated as

$$\min_{\hat{\alpha}_i} \in = \sum_{i=1}^{s} (y_k - \hat{y}_k)^2 \tag{1.10}$$

where $\hat{\alpha}_i$ represents the (unknown) coefficients of the RSM. The error ε is used to determine the adequacy of the model. If this error is too large, the modeling procedure should be repeated with a larger number of training points or a different sampling strategy, or a different model of the performance may have to be hypothesized.

From a statistical point of view (linear regression), the actual performance value is considered to be

$$y_i = \hat{y}_i + e_i, \quad i = 1, 2, \ldots, S \tag{1.11}$$

where e_i denotes the random error in the model. In least-squares fitting, this random error is implicitly assumed to be independent and identically distributed normal random variables with zero mean and constant variance. In particular, we assume that

$$\begin{aligned} E(e_i) &= 0 \\ \mathrm{Var}(e_i) &= \sigma^2 \\ \text{and} \quad \mathrm{Cov}(e_i, e_j) &= 0, \quad i \neq j \end{aligned} \tag{1.12}$$

If the RSM of the performance measure is linear, then it can be written as

$$\hat{y}(\mathbf{x}) = \sum_{i=1}^{C} \alpha_i f_i(\mathbf{x}) \tag{1.13}$$

where

$f_i(\mathbf{x})$ are the known basis functions of the circuit parameters
C denotes the number of basis functions

Note that the RSM is said to be linear (and least-squares fitting can be used) if it is linear in terms of the model coefficients α_i. The basis functions $f_i(\mathbf{x})$ need not be linear. For instance, the basis functions used in the quadratic RSM of Equation 1.5 are 1, x_i, and $x_i x_j$.

Let $\mathbf{y}$ and $\mathbf{e}$ be S-vectors: $\mathbf{y} = [y_1, \ldots, y_S]^{\mathrm{T}}$ and $\mathbf{e} = [e_1, \ldots, e_S]^{\mathrm{T}}$, where S is the number of data points. Also, let us define the model coefficients as $\alpha = [\alpha_1, \ldots, \alpha_C]$. Finally, let $\mathbf{X}$ denote the following $S \times C$ matrix:

$$X = \begin{bmatrix} f_1(\mathbf{x}_1) & \cdots & f_C(\mathbf{x}_1) \\ \vdots & \vdots & \vdots \\ f_1(\mathbf{x}_S) & \cdots & f_C(\mathbf{x}_S) \end{bmatrix} \tag{1.14}$$

Then, Equation 1.11 can be vectorially written as

$$\mathbf{y} = \mathbf{X}\alpha + \mathbf{e} \tag{1.15}$$

We choose α to minimize the function

$$\mathrm{RSS}(\alpha) = (\mathbf{y} - \mathbf{X}\alpha)^{\mathrm{T}}(\mathbf{y} - \mathbf{X}\alpha) \tag{1.16}$$

The least squares estimate $\hat{\alpha}$ of α is given by the following formula:

$$\hat{\alpha} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y} \tag{1.17}$$

The function Equation 1.16 evaluated at $\hat{\alpha}$ is called the residual sum of squares, abbreviated RSS, and is given by

$$\mathrm{RSS} = (\mathbf{y} - \mathbf{X}\hat{\alpha})^{\mathrm{T}}(\mathbf{y} - \mathbf{X}\hat{\alpha}) \tag{1.18}$$

It can be shown that an estimate of the variance of the random error is

$$\hat{\sigma}^2 = \frac{\mathrm{RSS}}{S - C} \tag{1.19}$$

and that $\hat{\alpha}$ is unbiased, i.e., $E(\hat{\alpha}) = \alpha$, and the variance of the estimate is

$$\mathrm{var}(\hat{\alpha}) = \hat{\alpha}^2(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1} \tag{1.20}$$

Another figure of merit of the linear regression is called the coefficient of determination, denoted by R^2, and given by

$$R^2 = 1 - \frac{\text{RSS}}{\sum_{i=1}^{s} (\text{y}_1 - \bar{y})^2} \tag{1.21}$$

where $\bar{y}$ is the sample average of the y_is, $i = 1, \ldots, S$. The coefficient of determination measures the proportion of variability in the performance that is explained by the regression model. Values of R^2 close to 1 are desirable.

1.7 Variable Screening

Variable screening is used in many performance modeling techniques as a preprocessing step to select the significant circuit parameters that are used as independent variables in the performance model. Variable screening can be used to substantially reduce the number of significant circuit parameters that must be considered, and thereby reduce the complexity of the analysis. The significant circuit parameters are included in the performance model, while the others are ignored. Several variable screening approaches have been proposed; in this chapter, we discuss a simple yet effective screening strategy that was proposed in [8].

The variable screening method of [8] is based on the two-level fractional factorial designs discussed earlier. We assume that the main effects (given by Equation 1.6) of the circuit parameters dominate over the interaction effects and this assumption enables us to use a resolution *III* design. Suppose that there are n original circuit parameters, denoted by $x_1, \ldots, x_n$. Then, we can use the simulation results from the fractional factorial design to compute the following quantities for each of the circuit parameters x_i, $i = 1, \ldots, n$:

1. Main effect v_i computed using Equation 1.6
2. *High deviation* d_i^{h} from the nominal or center point, computed by

$$d_i^{\text{h}} = \frac{2}{n_{\text{r}}} \sum_{k \in K_i^+} y_k - y_{\text{c}} \tag{1.22}$$

   where
   n_{r} is the number of simulations
   K_i^+ is the set of indices where x_i is $+1$
   y_{c} is the performance value at the center point

3. The *low deviation* d_i^{l} from the center point, computed using a formula similar to Equation 1.22, with K_i^+ replaced by K_i^-

Note that only two of the preceding quantities are independent, since the third can be obtained as a linear combination of the other two. A statistical significance test is now used to determine if x_i is significant. To this end, we assume that the quantities v_i, d_i^{h}, and d_i^{l} are sampled from a normal distribution with a sample size of n (since there are n circuit parameters). Denoting any one of the above quantities v_i, d_i^{h}, or d_i^{l} by δ_i, the hypothesis for the significance test is

$$\begin{aligned} &\text{Null hypothesis } H_0: \quad |\delta_i| = 0, \text{ i.e., } x_i \text{ is insignifiant} \\ &\qquad\qquad\qquad\quad H_0 \text{ is rejected if } |\delta_i|/\sigma_\delta > t_{\alpha_s/2, n-1} \end{aligned} \tag{1.23}$$

In Equation 1.23, a two-sided t-test is used [9] with significance level α_s and σ_δ denotes the estimate of the standard deviation of $\delta_i \cdot t_{\alpha_s/2, n-1}$ denotes the value of the t statistic with significance level $\alpha_s/2$ and degrees of freedom $n - 1$. We declare a parameter x_i to be insignificant if it is accepted in the tests for at least two of v_i, d_i^{h}, and d_i^{l}. This screening procedure is *optimistic* in the sense that an insignificant

parameter may be wrongly deemed significant from the preceding significance tests. The values of the t-variable can be read off from standard t-distribution tables available in many statistics text books.

1.8 Stochastic Performance Models

As stated earlier, the objective of performance modeling is to obtain an inexpensive surrogate to the circuit simulator. This performance model can then be used in a variety of deterministic and statistical optimization tasks in lieu of the expensive circuit simulator. The circuit simulator is a computer model, which is deterministic in the sense that replicate runs of the model with identical inputs will always result in the same values for the circuit performances. This lack of random error makes computer-based experiments fundamentally different from physical experiments. The experimental design and response surface modeling techniques presented earlier have the tacit assumption that the experiments are physical and the difference between the observed performance and the regression model behaves like white noise. In the absence of random error, the rationale for least-squares fitting of a response surface is not clear even though it can be looked upon solely as curve fitting. Stochastic performance modeling [10,11] has been used recently to model the outputs (in our case, the circuit performances) of computer codes or models (in our case, the circuit simulator). The basic idea is to consider the performances as realizations of a stochastic process and this allows one to compute a predictor of the response at untried inputs and allows estimation of the uncertainty of prediction.

The model for the deterministic performance y treats it as a realization of a stochastic process $Y(\mathbf{x})$ which includes a regression model:

$$Y(\mathbf{x}) = \sum_{i=1}^{k} \alpha_i f_i(\mathbf{x}) + Z(\mathbf{x}) \tag{1.24}$$

The first term in Equation 1.24 is the simple regression model for the performance and the second term represents the systematic departure of the actual performance from the regression model as a realization of a stochastic process. This model is used to choose a design that predicts the response well at untried inputs in the parameter space. Various criteria based on functionals of the means square error matrix are used to choose the optimal design. One the design is selected, the circuit simulator is exercised at those design points, and the circuit performance values are extracted. Next, a linear (in the model coefficients $\hat{\alpha}_i s$) predictor is used to predict the values of the performance at untried points. This predicted value can be shown to be the sum of two components: the first component corresponds to the first term in Equation 1.24 and uses the generalized least squares estimate of the coefficients α, and the second component is a smoothing term, obtained by interpolating the residuals at the experimental design points. The main drawback of the stochastic performance modeling approach is the excessive computational cost of obtaining the experimental design and of predicting the performance values.

1.9 Other Performance Modeling Approaches

The most popular performance modeling approach combines experimental design and least-squares-based response surface modeling techniques described previously [8,12–15]. A commonly used assumption valid for MOS circuits that has been used frequently is based on the existence of four critical noise parameters [16–18]. These four critical parameters are the flat-band voltage, the gate-oxide thickness, the channel length reduction, and the channel width reduction.

Several other performance modeling approaches have been proposed, especially in the context of statistical design of integrated circuits. One such approach is based on self-organizing methods [19] and is called the group method of data handling (GMDH) [20]. An interesting application of GMDH for performance modeling is presented in [21,22]. Another interesting approach is called maximally flat

quadratic interpolation (MFQI) [23]. A modeling approach combining the advantages of MFQI and GMDH is presented in [24]. It has been tacitly assumed in the discussion so far that a single RSM is sufficiently accurate to approximate the circuits performances over the entire circuit parameter space. This may not be true for many circuits. In such cases, a piecewise modeling approach may be used advantageously [8]. The circuit parameter space is divided into smaller regions, in each of which a single performance model is adequate. The continuous performance models in the various regions are then *stitched* together to preserve continuity across the regions.

1.10 Example of Performance Modeling

Below, we show an application of performance modeling applied to the clock skew of a clock distribution circuit. Consider the clock driver circuit shown in Figure 1.5. The top branch has three inverters and the lower branch has two inverters to illustrate the problem of signal skew present in many clock trees. We define the clock skew to be the difference between the times at which CLK and its complement cross a threshold of 0.5 V_{DD}. Two skews occur, as shown in Figure 1.6, one corresponding to the rising edge of the CLK, ΔS_r, and another corresponding to the falling edge of CLK, ΔS_f. The performance measure of interest is the signal skew ΔS defined as the larger of the two skews.

The designable circuit parameters of interest in this example are the nominal channel widths of each of the transistors in the second and third inverters in the top branch and the last inverter in the lower

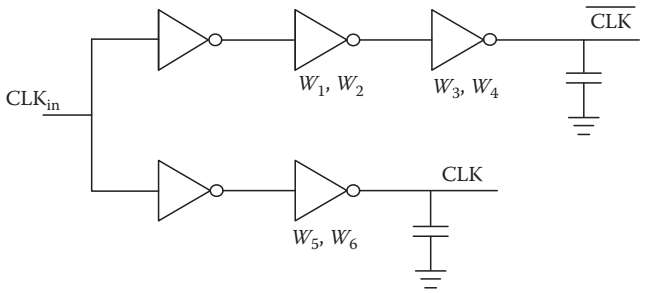


FIGURE 1.5 Clock distribution circuit.

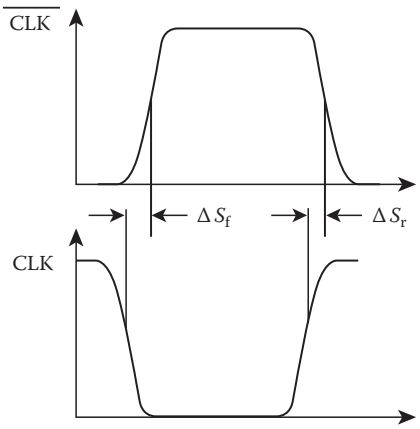


FIGURE 1.6 Definition of rising and falling clock skews.

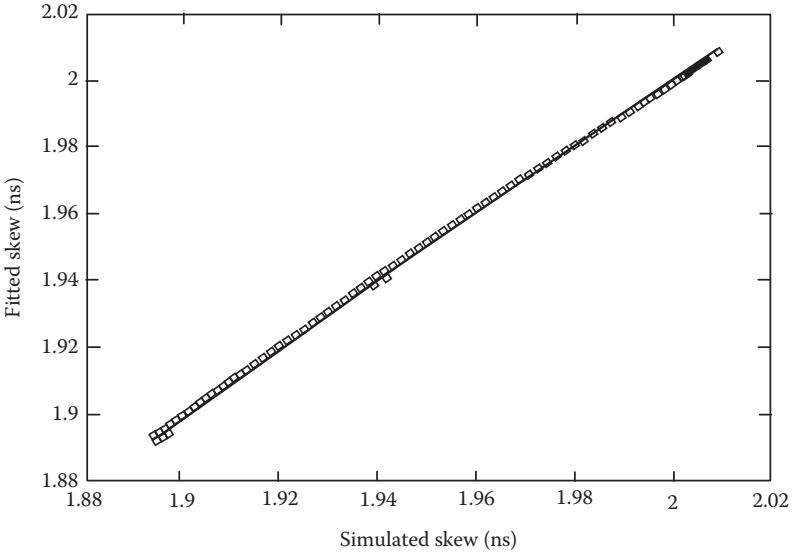


FIGURE 1.7 Plot of predicted versus simulated values for RSM of Equation 1.25.

branch. The designable parameters are denoted by W_1, W_2, W_3, W_4, W_5, and W_6 and are marked in Figure 1.5. The noise parameters of interest are the random variations in the channel widths and lengths of all nMOS and pMOS transistors in the circuit, and the common gate oxide thickness of all the transistors. These are denoted by ΔW_n (channel width variation of nMOS transistors), ΔL_n (channel length variation nMOS transistors), ΔW_p (channel width variation of pMOS transistors), ΔL_p (channel length variation of pMOS transistors), and t_{ox} (gate oxide thickness of all nMOS and pMOS transistors). We hypothesize a linear RSM for the clock skew in terms of the designable and noise parameters of the circuit. To this end, we generate an experimental design of size 30 using Latin hypercube sampling. The circuit is simulated at each of these 30 training points, and the value of the clock skew ΔS is extracted from the simulation results. Then, we use regression to fit the following linear RSM:

$$\begin{aligned}\Delta S = {} & 1.93975 - 0.0106W_1 + 0.0121W_2 + 0.0045W_3 - 0.1793W_4 \\ & + 0.0400W_5 + 0.0129W_6 - 0.0149\Delta W_n - 0.0380\Delta W_p \\ & - 0.0091\Delta L_n + 0.0115\Delta L_p + 0.161t_{ox}\end{aligned} \tag{1.25}$$

This RSM has an R^2 of 0.999. The prediction accuracy for the preceding RSM is in Figure 1.7 where the values of ΔS predicted from the RSM are plotted against the simulated values for a set of checking points (different from the training points used to fit the RSM). Most of the points lie near the 45° line indicating that the predicted values are in close agreement with the simulated values.

References

1. G. E. P. Box and N. R. Draper, *Empirical Model Building and Response Surfaces*, New York: Wiley, 1987.
2. G. E. P. Box, W. G. Hunter, and J. S. Hunter, *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building*, New York: Wiley, 1987.
3. N. R. Draper and T. J. Mitchell, The construction of saturated 2_R^{k-p} designs, *Ann. Math. Stat.*, 38: 1110–1126, August 1967.

4. P. J. Ross, *Taguchi Techniques for Quality Engineering*, New York: McGraw-Hill, 1988.
5. J. M. Hammersley and D. C. Handscomb, *Monte Carlo Methods*, London: Methuen and Co. Ltd., 1964.
6. M. D. McKay, R. J. Beckman, and W. J. Conover, A comparison of three methods for selecting values of input variables in the analysis of output from a computer code, *Technometrics*, 21: 239–245, May 1979.
7. R. L. Iman and W. J. Conover, A distribution-free approach to inducing rank correlations among input variables, *Commn. Stat.*, B11(3): 311–334, 1982.
8. K. K. Low and S. W. Director, An efficient methodology for building macromodels of IC fabrication processes, *IEEE Trans. Comput. Aided Des.*, 8: 1299–1313, December 1989.
9. D. C. Montgomery and E. A. Peck, *Introduction to Linear Regression Analysis*, New York: Wiley, 1982.
10. J. Sacks, W. J. Welch, T. J. Mitchell, and H. P. Wynn, Design and analysis of computer experiments, *Stat. Sci.*, 4(4): 409–435, 1989.
11. M. C. Bernardo, R. Buck, L. Liu, W. A. Nazaret, J. Sacks, and W. J. Welch, Integrated circuit design optimization using a sequential strategy, *IEEE Trans. Comput. Aided Des.*, 11: 361–372, March 1992.
12. T. K. Yu, S. M. Kang, I. N. Hajj, and T. N. Trick, Statistical modeling of VLSI circuit performances, *Proceedings of the IEEE International Conference Computer-Aided Design*, Santa Clara, CA, November 1986, pp. 224–227.
13. T. K. Yu, S. M. Kang, I. N. Hajj, and T. N. Trick, Statistical performance modeling and parametric yield estimation of MOS VLSI, *IEEE Trans. Comput. Aided Des.*, CAD-6: 1013–1022, November 1987.
14. T. K. Yu, S. M. Kang, W. Welch, and J. Sacks, Parametric yield optimization of CMOS analog circuits by quadratic statistical performance models, *Int. J. Circuit Theory Appl.*, 19: 579–592, November 1991.
15. A. Dharchoudhury and S. M. Kang, An integrated approach to realistic worst-case design optimization of MOS analog circuits, *Proceedings of the 29th ACM/IEEE Design Automation Conference*, Anaheim, CA, June 1992, pp. 704–709.
16. P. Cox, P. Yang, S. S. Mahant-Shetti, and P. Chatterjee, Statistical modeling for efficient parametric yield estimation, *IEEE J. Solid State Circuits*, SC-20: 391–398, February 1985.
17. P. Yang and P. Chatterjee, Statistical modeling of small geometry MOSFETs, *IEEE International Electron Device Meeting*, San Franscisco, CA, December 1982, pp. 286–289.
18. P. Yang, D. E. Hocevar, P. Cox., C. Machala, and P. Chatterjee, An integrated and efficient approach for MOS VLSI statistical circuit design, *IEEE Trans. Comput. Aided Des.*, 5: 5–14, January 1986.
19. S. J. Farlow, Ed., *Self-Organizing Methods in Modeling: GMDH Type Algorithms*, New York: Marcel Dekker, 1984.
20. A. G. Ivakhnenko, The group method of data handling, a rival of the method of stochastic approximation, *Sov. Autom. Control*, 13(3): 43–45, 1968.
21. A. J. Strojwas and S. W. Director, An efficient algorithm for parametric fault simulation of monolithic IC's, *IEEE Trans. Comput. Aided Des.*, 10: 1049–1058, August 1991.
22. S. Ikeda, M. Ochiai, and Y. Sawaragi, Sequential GMDH and its application to river flow prediction, *IEEE Trans. Syst. Man Cybern.*, SMC-6: 473–479, July 1976.
23. R. M. Biernacki and M. A. Styblinski, Statistical circuit design with a dynamic constraint approximation scheme, *Proceedings of the IEEE ISCAS*, San Jose, CA, 1986, pp. 976–979.
24. M. A. Styblinski and S. A. Aftab, Efficient circuit performance modeling using a combination of interpolation and self organizing approximation techniques, *Proceedings of the IEEE ISCAS*, New Orleans, LA, 1990, pp. 2268–2271.

2

Symbolic Analysis Methods

2.1 Introduction 2-1
2.2 Symbolic Modified Nodal Analysis 2-3
2.3 Solution Methods 2-7
Determinant-Based Solutions • Parameter Reduction Solutions
2.4 Ideal Operational Amplifiers 2-12
Nullator, Norator, and Nullor • Ideal Op-Amp Model
2.5 Applications of Symbolic Analysis 2-16
Frequency Response Evaluation • Circuit Response Optimization • Sensitivity Analysis • Circuit Sizing • Parameter Extraction in Device Modeling • Statistical Analysis • Fault Diagnosis of Analog Circuits • Insight into Circuit Operation • Education
2.6 Symbolic Analysis Software Packages 2-18
References 2-19

Benedykt S. Rodanski
University of Technology, Sydney

Marwan M. Hassoun
Iowa State University

2.1 Introduction

The late 1980s and 1990s saw a heightened interest in the development of symbolic analysis techniques and their applications to integrated circuits analysis and design. This trend continues in the new millennium. It resulted in the current generation of symbolic analysis methods, which include several software implementations like ISSAC [Gie89], SCAPP [Has89], ASAP [Fer91], EASY [Som91], SYNAP [Sed88], SAPWIN [Lib95], SCYMBAL [Kon88], GASCAP [Hue89], SSPICE [Wie89], STAINS [Pie01], and Analog Insydes [Hen00]. This generation is characterized by an emphasis on the production of usable software packages and a large interest in the application of the methods to the area of analog circuit design. This includes extensive efforts to reduce the number of symbolic expression generated by the analysis through hierarchical methods and approximation techniques [Fer92,Sed92,Wam92,Hsu94, Yu96]. This generation also includes attempts at applying symbolic analysis to analog circuit synthesis [Gie91,Cab98], parallel processor implementations [Has93b,Mat93,Weh93], time-domain analysis [Has91,Als93,Lib93], and behavioral signal path modeling [Ley00].

The details of traditional symbolic analysis techniques as they apply to linear circuit theory are given in [Rod08]. This chapter addresses symbolic analysis as a computer-aided design tool and its application to integrated circuit design. Modern symbolic analysis is mainly based on either a topological approach or a modified nodal analysis (MNA) approach. The basic topological (graph-based) methodologies are discussed in [Rod08] because the methods are part of traditional symbolic analysis techniques. The methodology addressed in this chapter is based on the concept of MNA [Ho75] as it applies to symbolic analysis.

Traditional symbolic circuit analysis is performed in the frequency domain where the results are in terms of the frequency variable s. The main goal of performing symbolic analysis on a circuit in the frequency domain is to obtain a symbolic transfer function of the form:

$$H(s,\mathbf{x}) = \frac{N(s,\mathbf{x})}{D(s,\mathbf{x})}, \quad \mathbf{x} = \begin{bmatrix} x_1 & x_2 & \dots & x_p \end{bmatrix}, \quad p \leq p_{\text{all}} \tag{2.1}$$

The expression is a rational function of the complex frequency variable s, and the variables x_1 through x_p representing the variable circuit elements, where p is the number of variable circuit elements and p_{all} is the total number of circuit elements. Hierarchical symbolic analysis approaches are based on a decomposed form of Equation 2.1 [Sta86,Has89,Has93a,Pie01]. This hierarchical representation is referred to as a sequence of expressions representation to distinguish it from the single expression representation of Equation 2.1. The major advantage of having a symbolic transfer function in single expression form is the insight that can be gained by observing the terms in both the numerator and the denominator. The effects of the different terms can, perhaps, be determined by inspection. This process is very valid for the cases where there are relatively few symbolic terms in the expression.

The heart of any circuit simulator, symbolic or numeric, is the circuit analysis methodology used to obtain the mathematical equations that characterize the behavior of the system. The circuit must satisfy these equations. These equations are of three kinds: Kirchhoff's current law (KCL), Kirchhoff's voltage law (KVL), and the branch relationship (BR) equations [Chu75]. KCL and KVL are well known and have been discussed earlier in this book. The definition of BR is ''For independent branches, a branch relationship defines the branch current in terms of the branch voltages.'' The most common independent branches are resistors, capacitors, and inductors. For the case of dependent branches, the branch current or voltage is defined in terms of other branches' currents or voltages. The four common types of dependent branches are the four types of controlled sources: voltage-controlled voltage source (VCVS), voltage-controlled current source (VCCS), current-controlled voltage source (CCVS), and current-controlled current source (CCCS). A linear element would produce a BR equation that is linear, and a nonlinear element would produce a nonlinear BR equation. The scope of this section is linear elements. If a nonlinear element is to be included in the circuit, a collection of linear elements could be used to closely model the behavior of that element around a dc operating point (quiescent point) in a certain frequency range [Gie94]. BR equations are not used on their own to formulate a system of equations to characterize the circuit. They are, however, used in conjunction with KCL or KVL equations to produce a more complete set of equations to characterize the circuit. Hybrid analysis [Chu75] uses a large set of BR equations in its formulation.

A system of equations to solve for the circuit variables, a collection of node voltages and branch currents, does not need to include all of the above equations. Several analysis methods are based on these equations or a subset thereof. Mathematically speaking, an equal number of circuit unknowns and linearly independent equations is required in order for a solution for the system to exist. The most common analysis methods used in circuit simulators are as follows:

1. A modification of nodal analysis as used in the numeric circuit simulators SPICE2 [Nag75], and SLIC [Idl71], and in the formulation of the symbolic equations in [Ald73]. Nodal analysis [Chu75] uses KCL to produce the set of equations that characterize the circuit. Its advantage is its relatively small number of equations. This results in a smaller matrix, which is very desirable in computer implementations. Its limitation, however, is that it only allows for node-to-datum voltages as variables. No branch current variables are allowed. Also the only type of independent power sources it allows are independent current sources. However, voltage sources are handled by using their Norton equivalent. A further limitation to this method is that only VCCSs are allowed, none of the other dependent sources are allowed in the analysis. To overcome these limitations, the MNA method was proposed in [Ho75]. It allows branch current as variables in the analysis,

which in turn leads to the ability to include voltage sources and all four types of controlled sources in the analysis. The MNA method is described in more detail in the next section because it is used in symbolic analysis after further modification to it. These modifications are necessary in order for the analysis to accept ideal operational amplifiers (op-amps), and to produce a compact circuit characterization matrix. This matrix is referred to as the reduced MNA matrix (RMNAM).

2. Hybrid analysis [Chu75] is used in the numeric simulators ASTAP [Wee73] and ECAP2 [Bra71]. The method requires the selection of a tree and its associated co-tree. It uses cutset analysis (KCL) and loop analysis (KVL) to formulate the set of equations. The equations produced solve for the branch voltages and branch currents. The method is able to accommodate voltage sources and all four types of dependent sources. It utilizes a much larger matrix than the nodal analysis methods; however, it is a very sparse matrix (lots of zero entries). The main drawback is that the behavior of the equations is highly dependent on the tree selected. A bad tree could easily result in an ill-conditioned set of equations. Some methods of tree selection can guarantee a well-conditioned set of hybrid equations [Nag75]. Therefore, special rules must be observed in the tree selection process that constitutes a large additional overhead on the analysis.
3. Sparse tableau analysis [Hac71] has been used in symbolic analysis to employ the parameter extraction method (this chapter). It uses the entire set of circuit variables: node voltages, branch voltages, and branch currents, for the symbolic formulation in addition to capacitor charges and inductor fluxes in the case of numeric simulations. This results in a huge set of equations and in turn a very large matrix but a very sparse one. This method uses all the above stated equation formulation methods: KCL, KVL and the BR. The entire system of equations is solved for all the circuit variables. The disadvantage of the method is its inherent problem of ill-conditioning [Nag75].

It has been found that the sparse tableau method only produces a very minor improvement in the number of mathematical operations, the nodal analysis methods despite its large overhead, stemming from a large matrix, and the large and rigid set of variables that have to be solved for. The MNA method gives the choice of what branch current variables to be solved for and equations are added accordingly. The tableau method would always have a fixed size matrix and produce unneeded information to the user.

For the purpose of symbolic analysis, as highlighted in this chapter, the entire analysis is performed in the frequency domain with the aid of the complex frequency variable s. The Laplace transform representation of the value of any circuit element is therefore used in the formulation. Also, since a modification of the nodal admittance matrix is used, the elements may be represented by either their admittance or impedance values.

2.2 Symbolic Modified Nodal Analysis

The initial step of MNA is to formulate the nodal admittance matrix $\mathbf{Y}$ [Chu75] from the circuit. The circuit variables considered here are all the node-to-datum voltages, referred to simply as node voltages; there are n_v of them. They are included in the variable vector $\mathbf{V}$. So $\mathbf{V}$ has the dimensions of $(n_v \times 1)$. The vector $\mathbf{J}$, also of dimensions $(n_v \times 1)$, represents the values of all independent current sources in the circuit. The ith entry of $\mathbf{J}$ is the sum of all independent current sources entering node i. The linear system of node equations can be represented in the following matrix form:

$$\mathbf{YV} = \mathbf{J} \tag{2.2}$$

Row i of $\mathbf{Y}$ represents the KCL equation at node i. The coefficient matrix, $\mathbf{Y}$, called the node admittance matrix, is constructed by writing KCL equations at each node, except for the datum node. The ith

equation then would state that the sum of all currents leaving node *i* is equal to zero. The equations are then put into the matrix form of Equation 2.2. The following example illustrates the process.

Example 2.1

Consider the circuit in Figure 2.1.

Collecting the node voltages would produce the following **V**:

$$\mathbf{V} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \tag{2.3}$$

Considering all the independent current sources in the circuit, the vector **J** becomes

$$\mathbf{J} = \begin{bmatrix} J_1 \\ J_4 \\ 0 \end{bmatrix} \tag{2.4}$$

Notice that the last entry of **J** is a zero because no independent current sources are connected to node 3.

Now, writing KCL equations at the three non-datum nodes produces

$$\begin{aligned} &G_2 v_1 + G_3(v_1 - v_2) + \frac{1}{sL_7}(v_1 - v_3) = J_1 \\ &G_5 v_2 + G_3(v_2 - v_1) + sC_6(v_2 - v_3) = J_4 \\ &sC_6(v_3 - v_2) + \frac{1}{sL_7}(v_3 - v_1) + g_8 V_{R_3} = 0 \end{aligned} \tag{2.5}$$

Substituting $V_{R_3} = v_1 - v_2$ in the third equation of the set Equation 2.5 and rearranging the variables results in

$$\begin{aligned} &\left(G_2 + G_3 + \frac{1}{sL_7}\right)v_1 - G_3 v_2 - \frac{1}{sL_7} v_3 = J_1 \\ &-G_3 v_1 + (G_3 + G_5 + sC_6)v_2 - sC_6 v_3 = J_4 \\ &\left(g_8 - \frac{1}{sL_7}\right)v_1 - (g_8 + sC_6)v_2 + \left(\frac{1}{sL_7} + sC_6\right) = 0 \end{aligned} \tag{2.6}$$

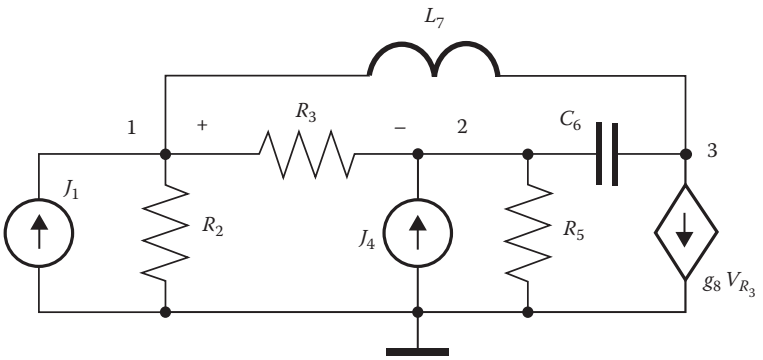


FIGURE 2.1 MNA example circuit.

Finally, rewriting Equation 2.6 in a matrix form of Equation 2.2 yields

$$\begin{bmatrix} G_2+G_3+\dfrac{1}{sL_7} & -G_3 & -\dfrac{1}{sL_7} \\ -G_3 & G_3+G_5+sC_6 & -sC_6 \\ g_8-\dfrac{1}{sL_7} & -(g_8+sC_6) & \dfrac{1}{sL_7}+sC_6 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} J_1 \\ J_4 \\ 0 \end{bmatrix} \tag{2.7}$$

An automatic technique to construct the node admittance matrix is the element stamp method [Ho75]. The method consists of going through each circuit component and adding its contribution to the nodal admittance matrix in the appropriate positions. It is an easy way to illustrate the impact of each element on the matrix.* Resolving Example 2.1 using the element stamps would produce exactly the same **Y** matrix, **V** vector, and **J** vector as in Equation 2.7.

The MNA technique, introduced in [Ho75], expands on the above nodal analysis in order to readily include independent voltage sources and the three other types of controlled sources: VCVS, CCCS, and CCVS. This is done by introducing some branch currents as variables into the system of equations, which in turn allows for the introduction of any branch current as an extra system variable. Each extra current variable introduced would need an additional equation to solve for it. The extra equations are obtained from the BRs for the branches whose currents are the extra variables. The effect on the nodal admittance matrix is the deletion of any contribution in it due to the branches whose currents have been declared as variables. This matrix is referred to as $\mathbf{Y}_n$. The addition of new variables and a corresponding number of equations to the system results in the need to append extra rows and columns to $\mathbf{Y}_n$. The augmented $\mathbf{Y}_m$ matrix is referred to as the MNA matrix (MNAM). The new system of equations, in matrix form, is

$$\begin{bmatrix} \mathbf{Y}_n & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{V} \\ \mathbf{I} \end{bmatrix} = \begin{bmatrix} \mathbf{J} \\ \mathbf{E} \end{bmatrix} \tag{2.8}$$

where

I is a vector of size n_i whose elements are the extra branch current variables introduced
E contains the values of the independent voltage sources
C and **D** correspond to BR equations for the branches whose currents are in **I**

An important feature of the MNAM approach is its ability to include impedance elements in the formulation. Let Z_x be an impedance connected between nodes i and j. Introduce the current through Z_x as new variable i_x. The extra equation needed is provided by the corresponding BR: $v_i - v_j - Z_x i_x = 0$. The usefulness of this approach can be appreciated when one needs to handle short circuits ($R = 0$) or inductors at dc. It becomes indispensable in approximation techniques that require the circuit matrix to contain only entries of the type $a = \alpha + s\beta$.

According to [Ho75], the MNA current variables should include all branch currents of independent voltage sources, controlled voltage sources, and all controlling currents. Further modifications of this procedure, like RMNA formulations [Has89], compacted MNA (CMNA) formulations [Gie89], and supernode analysis [Som93] relax these constraints and result in fewer MNA equations.

Example 2.2

As an example of a MNAM formulation, consider the circuit of Figure 2.2. The extra current variables are the branch currents of the independent voltage source E_1 (branch 1), the CCVS (branch 5), and the

* Program STAINS has the animate option that allows viewing the animated process of matrix formulation.

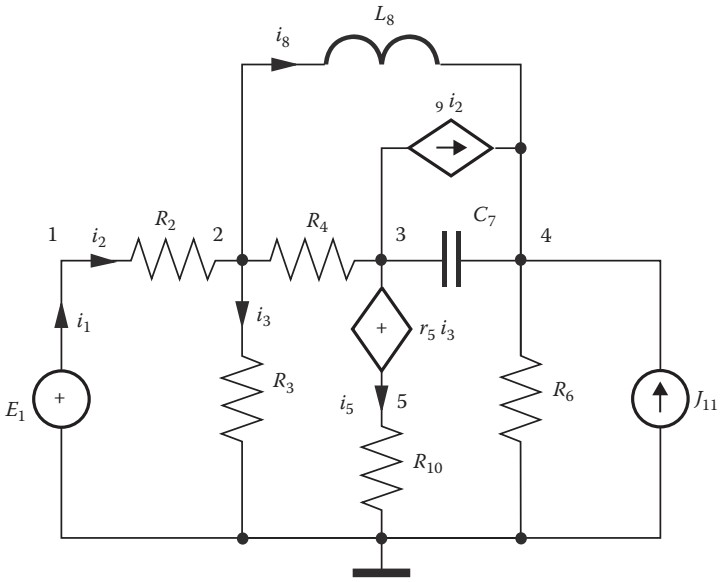


FIGURE 2.2 Circuit of Example 2.2.

inductor L_8 (branch 8). They are referred to as i_1, i_5, and i_8, respectively. The CCCS $\beta_9 i_2$ can be transformed to a VCCS by noticing that $i_2 = G_2(v_1 - v_2)$, so we have $\beta_9 i_2 = \beta_9 G_2(v_1 - v_2)$. Similarly, the CCVS $r_5 i_3$ becomes a VCVS $r_5 G_3 v_2$.

Using element stamps, the MNA system of equations becomes

$$\left[\begin{array}{ccccc|ccc}
G_2 & -G_2 & 0 & 0 & 0 & 1 & 0 & 0 \\
-G_2 & G_2+G_3+G_4 & 0 & 0 & 0 & 0 & 0 & 1 \\
\beta_9 G_2 & -G_4-\beta_9 G_2 & G_4+sC_7 & -sC_7 & 0 & 0 & 1 & 0 \\
-\beta_9 G_2 & \beta_9 G_2 & -sC_7 & G_6+sC_7 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & G_{10} & 0 & -1 & 0 \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -r_5 G_3 & 1 & 0 & -1 & 0 & 0 & 0 \\
0 & 1 & 0 & -1 & 0 & 0 & 0 & -sL_8
\end{array}\right]
\left[\begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ \hline i_1 \\ i_5 \\ i_8 \end{array}\right]
=
\left[\begin{array}{c} 0 \\ 0 \\ 0 \\ J_{11} \\ 0 \\ \hline E_1 \\ 0 \\ 0 \end{array}\right]$$

The above system of linear algebraic equations can be written in the following matrix form:

$$\mathbf{Ax} = \mathbf{b} \tag{2.9}$$

where

A is a symbolic matrix of dimension $n \times n$
x is a vector of circuit variables of length n
b is a symbolic vector of constants of length n
n is the number of circuit variables: currents, voltages, charges, or fluxes

The analysis proceeds by solving Equation 2.9 for **x**.

2.3 Solution Methods

2.3.1 Determinant-Based Solutions

Here the basic idea is to apply an extension of Cramer's rule, symbolically, to find a transfer function from the coefficient matrix of Equation 2.9. According to Cramer's rule, the value of a variable x_i in Equation 2.9 is calculated from the following formula:

$$x_i = \frac{\left|\mathbf{A}^{(i)}\right|}{|\mathbf{A}|} \tag{2.10}$$

where $\mathbf{A}^{(i)}$ is obtained from matrix $\mathbf{A}$ by replacing column i with the vector $\mathbf{b}$ and $|\bullet|$ denotes the determinant. Any transfer function, which is a ratio of two circuit variables, can be found by application of Equation 2.10:

$$\frac{x_i}{x_j} = \frac{\left|\mathbf{A}^{(i)}\right|/|\mathbf{A}|}{\left|\mathbf{A}^{(j)}\right|/|\mathbf{A}|} = \frac{\left|\mathbf{A}^{(i)}\right|}{\left|\mathbf{A}^{(j)}\right|} \tag{2.11}$$

Several symbolic algorithms are based on the concept of using Cramer's rule and calculating the determinants of an MNAM. Most notably is the program ISSAC [Gie89], which uses a recursive determinant expansion algorithm to calculate the determinants of Equation 2.11. Although there are many other algorithms to calculate the determinant of a matrix, like elimination algorithms and nested minors algorithms [Gie91], recursive determinant expansion algorithms were found the most suitable for sparse linear matrices [Wan77]. These algorithms are cancellation-free given that all the matrix entries are different, which is a very desirable property in symbolic analysis.

Let $\mathbf{A}$ be a square matrix of order n. Delete row i and column j from $\mathbf{A}$ and denote the resulting matrix of order $n-1$ by $\mathbf{A}_{-i-j}$. Its determinant, $M_{ij} = |\mathbf{A}_{-i-j}|$, is called a *minor* of $\mathbf{A}$ of order $n-1$. The quantity $\Delta_{ij} = (-1)^{i+j} M_{ij}$ is called a *cofactor* of a matrix element a_{ij}. The determinant of $\mathbf{A}$ can be calculated by recursive application of the following formula:*

$$|\mathbf{A}| = \begin{cases} \sum_{j=1}^{n} a_{ij}\Delta_{ij} = \sum_{j=1}^{n} a_{ji}\Delta_{ji} & \text{for } n > 1 \\ a_{11} & \text{for } n = 1 \end{cases} \tag{2.12}$$

where i is an arbitrary row or column of matrix $\mathbf{A}$.

As an example, the determinant of a 3×3 matrix is calculated by this method as

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = (-1)^{1+1} a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} + (-1)^{1+2} a_{12} \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + (-1)^{1+3} a_{13} \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}$$

$$= a_{11}a_{22}a_{33} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31}$$

Here, i was chosen to be, arbitrarily, row 1. The expression is cancellation-free in this example because all the matrix entries are unique. Note that for matrices with duplicated entries, the algorithm is not cancellation-free.

* Single application of Equation 2.12 is known as the Laplace expansion of the determinant.

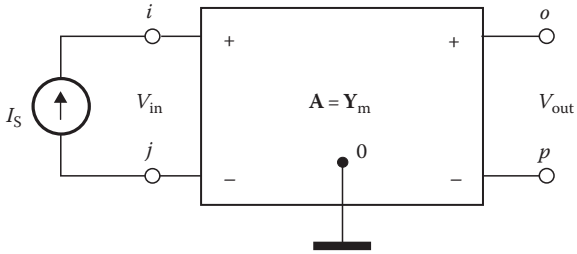


FIGURE 2.3 Two-port representation of a circuit.

Most symbolic analysis problems deal with two-port circuits of the form shown in Figure 2.3, where the signal source, I_S, is the only independent source in the circuit.

Let the two-port be described by the modified nodal equation of the form as in Equation 2.8. The right-hand side of this equation will have only two nonzero entries: $+I_S$ at position i and $-I_S$ at position j. Applying Cramer's rule to calculate potential at any node k, we get

$$v_k = \frac{I_S \Delta_{ik} - I_S \Delta_{jk}}{\Delta} \tag{2.13}$$

Now, if we wish, for example, to calculate the voltage transmittance of the two-port in Figure 2.3, using Equation 2.13, we obtain

$$T_v = \frac{v_o - v_p}{v_i - v_j} = \frac{\Delta_{io} - \Delta_{jo} - \Delta_{ip} + \Delta_{jp}}{\Delta_{ii} - \Delta_{ji} - \Delta_{ij} + \Delta_{jj}} \tag{2.14}$$

Equation 2.14 simplifies considerably if, as it is often the case, nodes j and p are identical with the reference node. We then have

$$T_v = \frac{v_o}{v_i} = \frac{\Delta_{io}}{\Delta_{ii}} \tag{2.15}$$

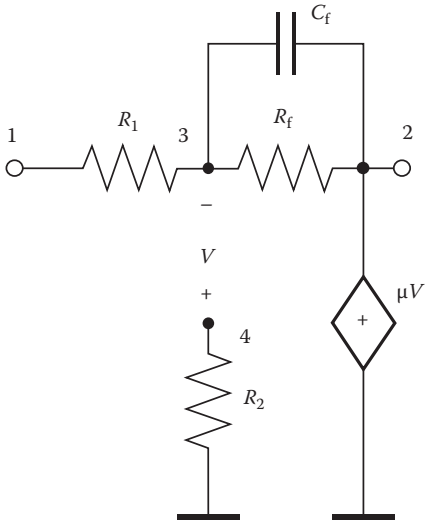


FIGURE 2.4 Circuit for Example 2.3.

Example 2.3

As an example, the voltage transfer function $T_v = v_2/v_1$ for the circuit in Figure 2.4 is to be calculated. The MNAM of this circuit is

$$\mathbf{Y}_m = \left[\begin{array}{cccc|c} G_1 & 0 & -G_1 & 0 & 0 \\ 0 & G_f + sC_f & -G_f - sC_f & 0 & 1 \\ -G_1 & -G_f - sC_f & G_1 + G_f + sC_f & 0 & 0 \\ 0 & 0 & 0 & G_2 & 0 \\ \hline 0 & 1 & \mu & -\mu & 0 \end{array}\right]$$

Using Equation 2.15 and expanding the cofactors according to Equation 2.12, we get

$$T_v = \frac{v_2}{v_1} = \frac{\Delta_{12}}{\Delta_{11}} = \frac{(-1)^{1+2}\begin{vmatrix} 0 & -G_f - sC_f & 0 & 1 \\ -G_1 & G_1 + G_f + sC_f & 0 & 0 \\ 0 & 0 & G_2 & 0 \\ 0 & \mu & -\mu & 0 \end{vmatrix}}{(-1)^{1+1}\begin{vmatrix} G_f + sC_f & -G_f - sC_f & 0 & 1 \\ -G_f - sC_f & G_1 + G_f + sC_f & 0 & 0 \\ 0 & 0 & G_2 & 0 \\ 1 & \mu & -\mu & 0 \end{vmatrix}}$$

$$= \frac{-G_1 G_2 \mu}{-G_2[\mu(-G_f - sC_f) - (G_1 + G_f + sC_f)]} = -\frac{G_1 \mu}{G_1 + G_f + sC_f + \mu(G_f + sC_f)}$$

Note that cancellation of a common term (G_2) in the numerator and denominator occurred. Although not present in the above example, cancellation of terms in each determinant expansion may occur in general if matrix entries are not unique.

2.3.2 Parameter Reduction Solutions

These methods use basic linear algebra techniques, applied symbolically, to find the solution to Equation 2.9. The goal here is to reduce the size of the system of equations by manipulating the entries of matrix **A** down to a 2×2 size. The two variables remaining in **x** are the ones for which the transfer function is to be calculated. The manipulation process, which in simple terms is solving for the two desired variables in terms of the others, is done using methods like Gaussian elimination [Nob77]. This methodology is used in the symbolic analysis programs SCAPP [Has89] and STAINS [Pie01]. Both programs use successive application of a modified Gaussian elimination process to produce the transfer function of the solution. The main difference between the techniques used in SCAPP and STAINS is in pivot selection strategy. While SCAPP selects pivots from the diagonal of **A**, STAINS employs a locally optimal pivot selection [Mar57,Gus70] that minimizes the number of arithmetic operations at each elimination step, thus producing more compact sequences of expressions. The process is applied to the MNAM and the result is a RMNAM ($\mathbf{Y}_R$). The process of reducing a circuit variable, whether a node voltage or a branch current, is referred to as variable suppression.

If the signal (input) source is the only independent source in the circuit, then the reduction process leads to the final RMNAM being identical with the well-known two-port admittance matrix. Its elements are known as the short-circuit admittance parameters or simply the *y*-parameters and have been used in circuit analysis from the 1940s. Since the two-port representation is the most commonly encountered in symbolic analysis problems, for the rest of this section we concern ourselves with terminated two-ports, depicted in Figure 2.5. The advantage of such a representation is the fact that once the *y*-parameters and the terminating admittances, Y_S and Y_L, are known, all other parameters of interest, like input/output impedances and various gains, can be readily calculated (e.g., [Gha65]). With no loss of generality we can assign numbers 1 and 2 to the input and output nodes, respectively. All other circuit variables are numbered from 3 to *n*. To focus our attention, we will only consider a (very typical) case when the input and output ports have a common terminal, which is the reference node.

The circuit of Figure 2.5 can be described by the set of symbolic equations of the form as in Equation 2.9, rewritten here more explicitly:

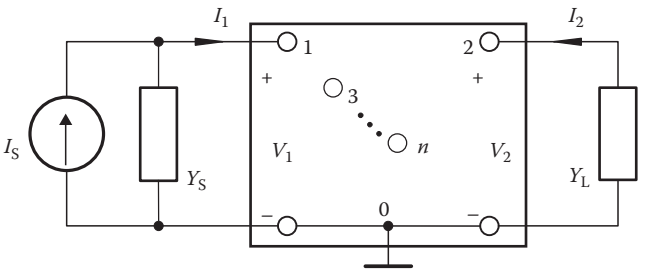


FIGURE 2.5 Terminated two-port for symbolic analysis.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} I_1 \\ I_2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{2.16}$$

The elements a_{ij} are symbolic entries of general form: $a = \alpha + s\beta$.

Since we are only interested in the relationship between the input and output variables (V_1, I_1 and V_2, I_2 in our case), all other variables can be suppressed. Suppose that we wish to suppress the variable x_p. To achieve this, we can use any equation from the set (Equation 2.16), except the first two equations that has a nonzero coefficient at x_p. Let us choose it to be equation $q > 2$. The qth equation can be written in the expanded form as follows:

$$a_{q1}V_1 + a_{q2}V_2 + \cdots + a_{qp}x_p + \cdots + a_{qn}x_n = 0 \tag{2.17}$$

Provided that $|a_{qp}| \neq 0$, we can calculate x_p from Equation 2.17 as

$$x_p = -\frac{a_{q1}}{a_{qp}}V_1 - \frac{a_{q2}}{a_{qp}}V_2 - \frac{a_{q3}}{a_{qp}}x_3 - \cdots - \frac{a_{qn}}{q_{qp}}x_n \tag{2.18}$$

Substituting Equation 2.18 into Equation 2.16 will eliminate the variable x_p and equation q from the set. During the elimination, each element a_{ij} of **A** undergoes the transformation:

$$a_{ij} \leftarrow a_{ij} - \frac{a_{qj}a_{ip}}{a_{qp}}; \quad i, j = 1, 2, \dots, n; \quad i \neq q, \quad j \neq p \tag{2.19}$$

This process of suppression of a variable is the very well-known Gaussian elimination. The only difference from the usual appearance of the elimination formula (Equation 2.19) in the literature is the fact that the pivot, a_{qp}, may be off-diagonal ($p \neq q$). In practice, the transformation (Equation 2.19) is only applied to the matrix elements a_{ij} for which $|a_{qj}||a_{ip}| \neq 0$. When any of the updated elements of **A**

is initially zero, a new nonzero element, a fill-in, is created in **A**. The pivot is chosen as to minimize either the number of arithmetic operations* or the number of fill-ins created at each elimination step.

All internal variables are successively eliminated using identical procedure. At the end of this process, we are left with a set of two equations:

$$\begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \begin{bmatrix} I_1 \\ I_2 \end{bmatrix} = \begin{bmatrix} I_S - Y_S V_1 \\ -Y_L V_2 \end{bmatrix} \tag{2.20}$$

Note that the two entries in the right-hand side vector, I_1 and I_2, have not been updated during the elimination process, thus coefficients y_{ij} in Equation 2.20 are the short-circuit admittance parameters of the original two-port, described by Equation 2.16. Various gain and immittance relations (network functions) can be expressed in terms of the two-port parameters and the termination admittances [Gha65]. For example, the voltage transmittance, V_2/V_1, and the current transmittance, $-I_2/I_1$, can be calculated as

$$T_v = \frac{V_2}{V_1} = -\frac{y_{21}}{y_{22} + Y_L}, \quad T_i = \frac{-I_2}{I_1} = \frac{-y_{21} Y_L}{y_{11}(y_{22} + Y_L) - y_{12} y_{21}}$$

A simple example serves to illustrate and easily verify the formulation of the RMNAM and advantages of a proper pivoting strategy.

Example 2.4

Consider the circuit in Figure 2.6.

Using element stamps, the modified nodal equation of the circuit in Figure 2.4 can be written as

$$\begin{bmatrix} G_1 & 0 & -G_1 & 0 \\ 0 & G_4 + sC_3 & -sC_3 & 0 \\ -G_1 & -sC_3 & G_1 + sC_3 & 1 \\ -\mu & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ v_3 \\ i_4 \end{bmatrix} = \begin{bmatrix} I_1 \\ I_2 \\ 0 \\ 0 \end{bmatrix} \tag{2.21}$$

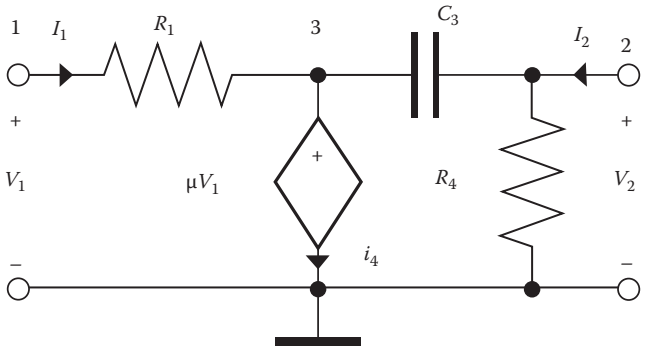


FIGURE 2.6 Circuit of Example 2.4.

* Minimization of the number of arithmetic operations is equivalent with minimization of the number of matrix updates via Equation 2.19. If the numbers of nonzero elements in the pivot row and column are n_q and n_p, respectively, then the number of updates required to suppress x_p using equation q is equal to $(n_q - 1)(n_p - 1)$.

To reduce the coefficient matrix in Equation 2.21 to the two-port admittance matrix, the variables v_3 and i_4 must be eliminated. If we select pivots from the main diagonal only, we have no choice but to first take the element $a_{33} = G_1 + sC_3$. Since row 3 and column 3 have four nonzero elements each, elimination of v_3 using the third equation requires $(4 - 1)(4 - 1) = 9$ updates. Moreover, six fill-ins will be created, making the matrix full. Four more updates are required to suppress the current variable, i_4, increasing the total number of updates to 13. On the other hand, if we are not restricted to diagonal pivots, the best choice for the first one is the element $a_{34} = 1$. Inspection of Equation 2.21 reveals that elimination of the fourth variable (i_4) using equation 3 requires no updates at all ($n_p - 1 = 0$). With no computational effort, we obtain

$$\begin{bmatrix} G_1 & 0 & -G_1 \\ 0 & G_4 + sC_3 & -sC_3 \\ -\mu & 0 & 1 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} I_1 \\ I_2 \\ 0 \end{bmatrix} \tag{2.22}$$

Now, using the element $a_{33} = 1$ of Equation 2.22 as a pivot, only two updates are required to suppress variable v_3. The final equation becomes

$$\begin{bmatrix} G_1 - \mu G_1 & 0 \\ -\mu sC_3 & G_4 + sC_3 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \begin{bmatrix} I_1 \\ I_2 \end{bmatrix}$$

By unrestricted pivot selection, we have achieved the desired result with only 2 updates, compared to 13 updates if the pivot was chosen from the main diagonal only.

2.4 Ideal Operational Amplifiers

Neither the classical nodal analysis, nor the MNA in its original form [Ho75], accounts for ideal op-amps in their equation formulation. The way op-amps have usually been handled in those analyses is by modeling each op-amp by a VCVS with a large gain in an attempt to characterize the infinite gain of the ideal op-amp.

The ideal op-amp is a two-port device for which one of the output terminals is usually assumed to be grounded.* Op-amps are used extensively in the building of a large number of analog circuits, especially analog filters, where symbolic circuit simulators have found extensive applications in obtaining transfer functions for the filters. It is therefore necessary for a modern symbolic analysis program to handle ideal op-amps. Both SCAPP and STAINS include ideal op-amps in their lists of available circuit components.

There are two ways by which the ideal op-amps can be handled in an automatic formulation of circuit equation. The first method is simply an extension of the MNA, which accounts for op-amps by adding the constraining equations to the MNA set, introducing an element stamp for the op-amp [Vla94]. The other approach reduces the size of the MNA set by removing variables that are known to be identical (node voltages at both op-amp inputs) and by removing unnecessary equations (KCL equation at the op-amp output) [Has89,Lin91]. The latter method is known as compacting of the MNAM due to op-amps; the resulting matrix is termed compacted MNA matrix (CMNAM). Both approaches are presented below.

2.4.1 Nullator, Norator, and Nullor

Before the characterization of the ideal op-amp is attempted, the concepts of the nullator, the norator, and the nullor are explored [Bru80]. They are not real elements. These ideal components are tools to introduce some mathematical constraints into a circuit. They are used as an aid to the development of insight into the behavior of ideal devices like the ideal op-amp.

The symbol for the nullator is shown in Figure 2.7. A nullator is defined as follows:

* If neither of the output terminals of the op-amp are grounded, such op-amp is said to be floating [Vla94].

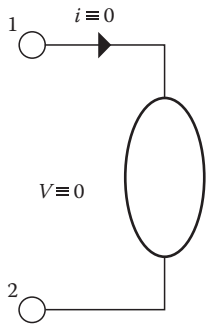


FIGURE 2.7 Nullator.

Definition 2.1: A nullator is a two-terminal element defined by the constraints

$$\begin{aligned} v_1 - v_2 &\equiv 0 \\ i &\equiv 0 \end{aligned} \tag{2.23}$$

The symbol for the norator is presented in Figure 2.8. A norator is defined as follows:

Definition 2.2: A norator is a two-terminal element for which the voltage and current are not constrained. That is,

$$\begin{aligned} v_1 - v_2 &= \text{arbitrary} \\ i &= \text{arbitrary} \end{aligned} \tag{2.24}$$

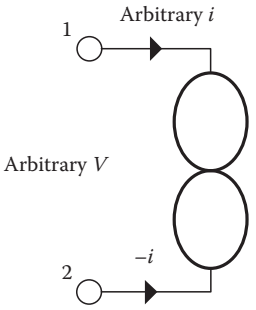


FIGURE 2.8 Norator.

A norator in a circuit introduces freedom from some constraints on the nodes it is connected to.

A circuit must contain an equal number of nullators and norators; in other words, a nullator and a norator must appear in a circuit as a pair. The combination of a nullator and a norator to produce a two-port, as shown in Figure 2.9, is referred to as a nullor. The equations characterizing the nullor are represented then by Equations 2.23 and 2.24.

In the MNA formulation, each nullator introduces additional equation: $v_i - v_j = 0$ into the set. The required extra variable is the corresponding norator's current. Nullor's contribution to the MNAM can be represented as its unique element stamp in similar way as it was done for VCVS, CCVS, etc. This is best illustrated in an example.

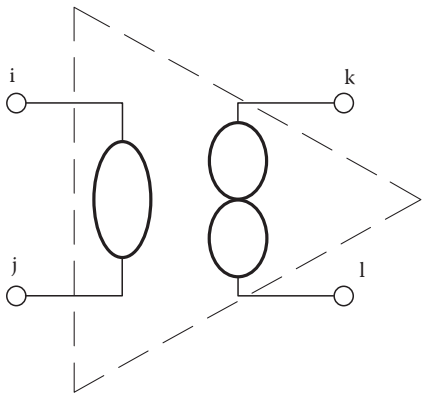


FIGURE 2.9 Nullor.

Example 2.5

Write the MNA equation for the circuit containing a nullor, as shown in Figure 2.10.

To formulate the MNAM, needed in the equation, we start with a 6×6 matrix and use element stamps to include the capacitor and the conductances in the usual manner. The nullator equation, $v_2 - v_5 = 0$, creates two entries in row 6. The norator current, our extra variable i_6, must be included in the KCL equations for the nodes 3 and 4, creating two entries in column 6. This results in the equation

$$\left[\begin{array}{ccccc|c} G_1 + sC_1 & -G_1 & 0 & 0 & -sC_1 & 0 \\ -G_1 & G_1 + G_2 & -G_2 & 0 & 0 & 0 \\ 0 & -G_2 & G_2 & 0 & 0 & 1 \\ 0 & 0 & 0 & G_3 & -G_3 & -1 \\ -sC_1 & 0 & 0 & -G_3 & G_3 + G_4 + sC_1 & 0 \\ \hline 0 & 1 & 0 & 0 & -1 & 0 \end{array}\right] \left[\begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ i_6 \end{array}\right] = \left[\begin{array}{c} I_S \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{array}\right] \tag{2.25}$$

The sixth row and the sixth column of the MNAM in Equation 2.25 represent the element stamp of the nullor.

2.4.2 Ideal Op-Amp Model

The two characteristics of an ideal op-amp are

1. Zero voltage differential and no current flowing between its input nodes (the virtual short-circuit)
2. Arbitrary current that the output node supplies (to satisfy the KCL at this node)

Both properties are perfectly modeled using a nullor (Figure 2.9), with terminal l grounded. Adding a nullor stamp to our repertoire of element stamps creates a natural extension of the MNA method.

There is, however, another way of looking at ideal op-amp's (nullor's) contribution to the set of nodal equations, describing a circuit. Consider first the input of an ideal op-amp. Due to property 1, potentials at both input nodes must be identical. So, one of the two variables (node potentials) can be eliminated without any loss of information about the circuit. With one less variable, we must also remove one equation, for the system to have a unique solution. Since the ideal op-amp will always supply the exact current required to satisfy the KCL at the output node, the KCL equation at this node is unnecessary and can be simply deleted from the set. This process results in a MNAM that is more compact than the matrix obtained with the use of ideal op-amp stamp. Thus, the resulting matrix is called the CMNAM.

The rules for formulating CMNAM for a circuit containing ideal op-amps are as follows:

1. Remove the ideal op-amps from the circuit.
2. Write the MNAM $\mathbf{Y}_m$ for this circuit; label all columns and rows of $\mathbf{Y}_m$ with numbers corresponding to variables.

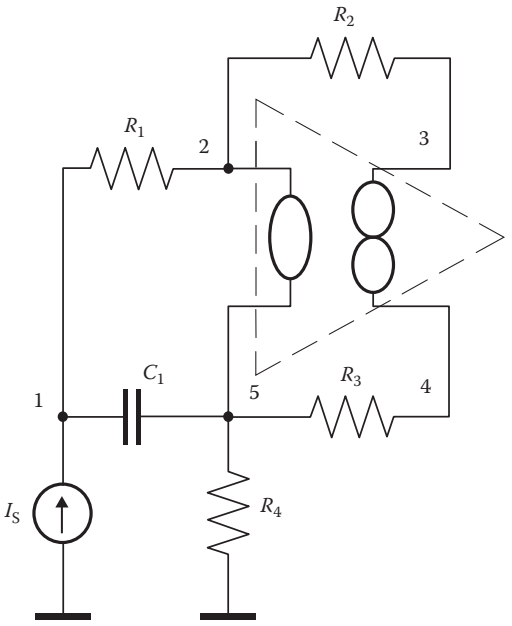


FIGURE 2.10 Circuit with nullor.

3. Repeat for every ideal op-amp in the circuit:
   a. For an ideal op-amp with the input terminals connected to nodes i $(-)$ and j $(+)$, add the column containing label j to the column containing label i and delete the former column; append all labels of the former column to the labels of the latter column. If one of the input nodes is the reference node, simply delete the column and the variable corresponding to the other node.
   b. For an ideal op-amp with the output terminal connected to node k, delete row with label k from the matrix.

Example 2.6

We wish to obtain the CMNAM of the gyrator circuit with two ideal op-amps, as shown in Figure 2.11. Following the compacting rules, we first formulate the NAM of the circuit with op-amps removed and label the rows and columns as shown below:

$$\mathbf{Y}_m = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} G_4 & 0 & 0 & 0 & -G_4 \\ 0 & G_1+G_2 & -G_2 & 0 & 0 \\ 0 & -G_2 & G_2+G_3 & -G_3 & 0 \\ 0 & 0 & -G_3 & G_3+sC_1 & -sC_1 \\ -G_4 & 0 & 0 & -sC_1 & G_4+sC_1 \end{bmatrix} \end{array}$$

Next, according to Rule 3a, for the first op-amp, we add the column containing label 1 to the column containing label 2, delete the former column, and append its labels to the labels of the latter column. Output of O_1 is connected to node 3. Rule 3b requires that we delete row with label 3. This results in the following matrix:

$$\mathbf{Y}_m = \begin{array}{c} \\ 1 \\ 2 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccc} 2,1 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} G_4 & 0 & 0 & -G_4 \\ G_1+G_2 & -G_2 & 0 & 0 \\ 0 & -G_3 & G_3+sC_1 & -sC_1 \\ -G_4 & 0 & -sC_1 & G_4+sC_1 \end{bmatrix} \end{array}$$

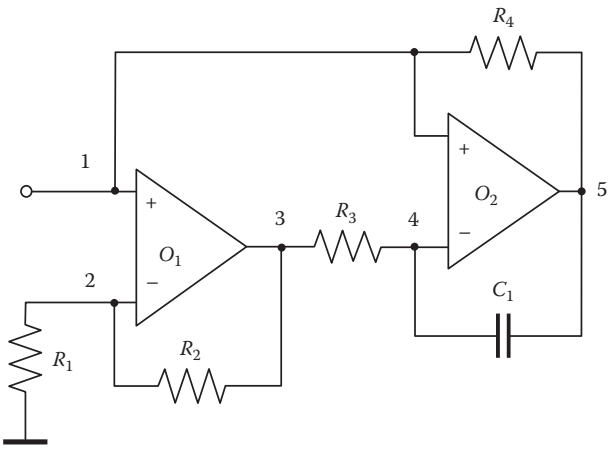


FIGURE 2.11 Gyrator circuit with two ideal op-amps for Example 2.6.

Applying Rule 3 to the op-amp O_2, we add column containing label 1 to the column containing label 4, delete the former column, and append its labels (2,1) to the labels of column 4. According to Rule 3b, we delete row with label 5. The resulting matrix is shown below:

$$\mathbf{Y}_{CM} = \begin{array}{c} \\ 1 \\ 2 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccc} 3 & 4,2,1 & 5 \end{array} \\ \begin{bmatrix} 0 & G_4 & -G_4 \\ -G_2 & G_1+G_2 & 0 \\ -G_3 & G_3+sC_1 & -sC_1 \end{bmatrix} \end{array}$$

It is customary to rearrange the columns of the CMNAM in ascending order of their smallest labels. After exchanging columns 1 and 2, the final matrix becomes

$$\mathbf{Y}_{CM} = \begin{array}{c} \\ 1 \\ 2 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccc} 4,2,1 & 3 & 5 \end{array} \\ \begin{bmatrix} G_4 & 0 & -G_4 \\ G_1+G_2 & -G_2 & 0 \\ G_3+sC_1 & -G_3 & -sC_1 \end{bmatrix} \end{array}$$

2.5 Applications of Symbolic Analysis

There are many specific applications for which symbolic analysis algorithms have been developed over the years. The following is an attempt to categorize the uses of symbolic analysis methods. For a given application, some overlap of these categories might exist. The goal here is to give a general idea of the applications of symbolic analysis methods. It must be noted that most of these applications cover both *s*-domain and *z*-domain analyses.

It is now widely accepted that in all applications where symbolic formula is generated in order to provide means for subsequent repetitive exact numerical calculation, use of sequential formulas is most effective. For the purpose of interpretation, that is, gaining deeper understanding of circuit behavior, only a single formula can be used. Approximate symbolic analysis emerges here as a very promising approach.

2.5.1 Frequency Response Evaluation

This is an obvious application of having the symbolic transfer function stated in terms of the circuit variable parameters. The process of finding the frequency response curve over a frequency range for a given circuit involves the repetitive evaluation of Equation 2.1 with all the parameters numerically specified and sweeping the frequency over the desired range. A numerical simulator would require a simulation run for each frequency point.

2.5.2 Circuit Response Optimization

This process involves the repetitive evaluation of the symbolic function generated by a symbolic simulator [Lin91,Gie94]. The response of the circuit is repetitively evaluated by substituting different values for the circuit parameters in the equation until a desired numerical response is achieved. The concept, of course, requires a good deal of management in order to reduce the search space and the number of evaluations needed. Such a method for filter design by optimization is discussed in [Tem77] and [Bas72], and a method for solving piecewise resistive linear circuits is discussed in [Chu75]. The idea here is that for a given circuit topology, only one run through the symbolic circuit simulator is necessary.

2.5.3 Sensitivity Analysis

Sensitivity analysis is the process of finding the effect of the circuit performance due to changes in an element value [Lin92,Bal98,Bal01]. There are two types of sensitivities of interest to circuit designers: small-change (or differential) sensitivity and large-change sensitivity. The normalized differential sensitivity of a transfer function H with respect to a parameter x is given as

$$S_x^H = \frac{\partial(\ln H)}{\partial(\ln x)} = \frac{\partial H}{\partial x}\,\frac{x}{H}$$

The above expression can be found symbolically and then evaluated for the different circuit parameter values. For the transfer function H in a single expression form, the differentiation process is quite straightforward. When H is given in a sequence of expressions form, the most compact formula is obtained by the application of the two-port transimpedance concept and an efficient method of symbolic calculations of the elements of the inverse of the reduced modified node admittance matrix [Bal98]. Similar approach is used to obtain the large-change sensitivity [Bal01], defined as

$$\delta_x^H = \frac{\Delta H}{H}\,\frac{x}{\Delta x}$$

Once the sequence of expressions for the network function has been generated, only a few additional expressions are needed to determine the sensitivity (both differential and large-change). Additional effort to calculate sensitivities is practically independent of circuit size, making this approach attractive for large-scale circuits.

2.5.4 Circuit Sizing

Circuit sizing refers to the process of computing values for the elements of a given circuit topology such that some set of behavioral circuit specifications is satisfied [Hen00]. This process can be supported by symbolic techniques in two ways. Computer algebra can help to solve a system of design equations symbolically for the circuit parameters in order to obtain a set of generic analytic sizing formulas, or to set up and preprocess a system of design equations for subsequent numerical solution.

2.5.5 Parameter Extraction in Device Modeling

This process involves the repeated comparison of measured data from fabricated devices with the simulation results using the mathematical models for these devices [Kon93,Avi98]. The goal of the process is to update the device models to reflect the measured data. The model parameters are incrementally adjusted and the evaluation process is repeated until the difference between the measured and the simulated results are minimized. Such approaches are reported in [Kon89] and [Kon93] (active devices) and [Avi98] (passive devices).

2.5.6 Statistical Analysis

A widely used statistical analysis method is through Monte Carlo simulations [Sty95]. The circuit behavior has to be repetitively evaluated many times in order to evaluate the statistical variation of a circuit output in response to parameter mismatches due to, for instance, integrated circuits process variations.

2.5.7 Fault Diagnosis of Analog Circuits

The process reported in [Man93] takes measurements from the faulty fabricated circuit and compares it to simulation results. The process is continuously repeated with the parameter values in the simulations

changed until the faulty element is detected. Symbolic techniques have also been applied to multiple fault detection [Fed98].

2.5.8 Insight into Circuit Operation

The insight that can be provided by obtaining the symbolic transfer function versus its numerical counterpart is very evident. The simple example in Section 2.5 illustrates this powerful application. The effect of the different elements on the behavior of the circuit can be observed by inspecting the symbolic expression. This, of course, is possible if the number of symbolic parameters is small, that is, the circuit is small. Insight, however, can also be obtained by observing an approximate symbolic expression that reduces the number of symbolic terms to a manageable figure. Several approximation techniques are reported in [Wam92], [Fer92], [Sed92], [Hsu94], [Yu96], and [Hen00].

2.5.9 Education

Symbolic analysis is most helpful for students as a supplement to linear circuit analysis courses. These courses require the derivation of expressions for circuit impedances, gains, and transfer functions. A symbolic simulator can serve as a check for the correctness of the results in addition to aiding instructors in verifying solutions and making up exercises and examples [Hue89].

Symbolic methods in conjunction with mathematical software are used in teaching analog electronics in addition to numerical simulators. This adds another dimension to standard analysis and design process, gives students greater flexibility, and encourages creativeness [Bal00].

2.6 Symbolic Analysis Software Packages

Several stand-alone symbolic simulators available for public use exist nowadays. A comparison of these software packages based on their functionality is given in Table 2.1. The list is by no means exhaustive; it should be treated just as a representative sample. The software packages that were compared are

TABLE 2.1 Comparison between Some Symbolic Simulation Programs

	ISSAC	ASAP	SYNAP	SAPWIN	SSPICE	SCYMBAL	SCAPP	STAINS
Analysis domains	*s* and *z*	*s*	dc and *s*	*s*	*s*	*z*	*s*	s
Primitive elements	All	All	All	All	All	—	All	All
Small-signal linearization	Yes	Yes	Yes	No	Yes	No	Yes	No
Mismatching	Yes	Yes	Yes	No	No	No	No	No
Approximation	Yes	Yes	Yes	Yes	Yes	No	No	No
Weakly nonlinear analysis	Yes	No	No	No	No	No	No	No
Hierarchical analysis	No	No	No	No	Limited	No	Yes	Yes
Pole/zero extraction	No	Limited	No	Yes	Limited	No	No	No
Graphical interface	No	Yes	No	Yes	No	No	Yes	No
Equation formulation	CMNA	SFG	MNA	MNA	**Y**	SFG	RMNA SFG	RMNA
Language	LISP	C	C++	C++	C	FORTRAN	C	VBA

Source: Adapted from Gielen, G. and Sansen, W., *Symbolic Analysis for Automated Design of Analog Integrated Circuits*, Kluwer Academic, Boston, MA, 1991.

Note: SFG, signal flowgraph; **Y**, admittance matrix; VBA, Visual Basic for Applications™.

ISSAC [Gie89], SCAPP [Has89], ASAP [Fer91], SYNAP [Sed88], SCYMBAL [Kon88], STAINS [Pie01], SAPWIN [Lib95], and SSPICE [Wie89].

Add-on packages (toolboxes) that run in symbolic mathematics software environments, like Mathematica, belong to another class of symbolic analysis programs. Analog Insydes [Hen00] is a good example of such an add-on. The toolbox includes approximation algorithms, netlist management, linear and nonlinear device and behavioral modeling, setting up and solving circuit equations, numerical circuit analysis in the frequency and time domains, two- and three-dimensional graphing of analysis results, and data exchange with other circuit simulators. Some routines in the toolbox were written in C to overcome Mathematica's limitations in efficiently handling large sparse systems.

References

[Ald73] G.E. Alderson and P.M. Lin, Computer generation of symbolic network functions: A new theory and implementation, *IEEE Transactions on Circuit Theory*, 20(1): 48–56, January 1973.

[Als93] B. Alspaugh and M.M. Hassoun, A mixed symbolic and numeric method for closed-form transient analysis, *Proceedings of the 5th International Workshop on European Conference on Circuit Theory and Design*, Davos, Switzerland, pp. 1687–1692, September 1993.

[Avi98] G. Avitabile et al., Parameter extraction in electronic device modelling using symbolic techniques, *Proceedings of the 5th International Workshop on Symbolic Methods and Applications in Circuit Design*, Kaiserslautern, Germany, pp. 253–259, October 1998.

[Bal98] F. Balik and B. Rodanski, Calculation of first-order symbolic sensitivities in sequential form via the transimpedance method, *Proceedings of the 5th International Workshop on Symbolic Methods and Applications in Circuit Design*, Kaiserslautern, Germany, pp. 169–172, October 1998.

[Bal00] F. Balik and B. Rodanski, Symbolic analysis in the classroom, *Proceedings of the 6th International Workshop on Symbolic Methods and Applications in Circuit Design*, Lisbon, Portugal, pp. 17–20, October 2000.

[Bal01] F. Balik and B. Rodanski, Obtaining large-change symbolic sensitivities for large-scale circuits, *Proceedings of the European Conference on Circuit Theory and Design*, Espoo, Finland, pp. 205–208, August 2001.

[Bas72] S. Bass, The application of a fast symbolic analysis routine in a network optimization program, *Proceedings of the Midwest Symposium on Circuit Theory*, Rolla, MO, May 1972.

[Bra71] F.H. Branin et al., ECAP II: A new electronic circuit analysis program, *IEEE Journal of Solid-State Circuits*, 6(4): 146–165, August 1971.

[Bru80] L.T. Bruton, *RC Active Circuits: Theory and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1980.

[Cab98] R. Cabeza and A. Carlosena, The use of symbolic analysers in circuit synthesis, *Proceedings of the 5th International Workshop on Symbolic Methods and Applications in Circuit Design*, Kaiserslautern, Germany, pp. 146–149, October 1998.

[Chu75] L.O. Chua and P.M. Lin, *Computer-Aided Analysis of Electronic Circuits: Algorithms and Computational Techniques*. Englewood Cliffs, NJ: Prentice-Hall, 1975.

[Fed98] G. Fedi et al., On the application of symbolic techniques to the multiple fault location in low testability analog circuits, *IEEE Transactions on Circuits and Systems—II: Analog and Digital Signal Processing*, 45(10): 1383–1388, October 1998.

[Fer91] F.V. Fernandez, A. Rodriguez-Vazquez, and J.L. Huertas, An advanced symbolic analyzer for the automatic generation of analog circuit design equations, *Proceedings of the IEEE International Symposium on Circuits and Systems*, Singapore, pp. 810–813, June 1991.

[Fer92] F.V. Fernandez et al., On simplification techniques for symbolic analysis of analog integrated circuits, *Proceedings of the IEEE International Symposium on Circuits and Systems*, San Diego, CA, pp. 1149–1152, May 1992.

[Gha65] M.S. Ghausi, *Principles and Design of Linear Active Circuits*. New York: McGraw-Hill, 1965.

[Gie89] G. Gielen, H. Walscharts, and W. Sansen, ISSAC: A symbolic simulator for analog integrated circuits, *IEEE Journal of Solid-State Circuits*, 24(6): 1587–1597, December 1989.

[Gie91] G. Gielen and W. Sansen, *Symbolic Analysis for Automated Design of Analog Integrated Circuits*, Boston, MA: Kluwer Academic, 1991.

[Gie94] G. Gielen, P. Wambacq, and W. Sansen, Symbolic analysis methods and applications for analog circuits: A tutorial overview, *Proceeding of the IEEE*, 82(2): 287–301, February 1994.

[Gus70] F.G. Gustavson, W. Liniger, and R. Willoughby, Symbolic generation of an optimal Crout algorithm for sparse systems of linear equations, *Journal of ACM*, 17(1): 87–109, 1970.

[Hac71] G.D. Hachtel et al., The sparse tableau approach to network analysis and design, *IEEE Transactions on Circuit Theory*, 18(1): 101–113, January 1971.

[Has89] M.M. Hassoun and P.M. Lin, A new network approach to symbolic simulation of large-scale networks, *Proceedings of the IEEE International Symposium on Circuits and Systems*, Portland, OR, pp. 806–809, May 1989.

[Has91] M.M. Hassoun and J.E. Ackerman, Symbolic simulation of large-scale circuits in both frequency and time domains, *Proceedings of the IEEE Midwest Symposium on Circuits and Systems*, Calgary, Canada, pp. 707–710, August 1990.

[Has93a] M.M. Hassoun and K. McCarville, Symbolic analysis of large-scale networks using a hierarchical signal flowgraph approach, *International Journal on Analog Integrated Circuits and Signal Processing*, Kluwer Academic, 3(1): 31–42, January 1993.

[Has93b] M.M. Hassoun and P. Atawale, Hierarchical symbolic circuit analysis of large-scale networks on multi-processor systems, *Proceedings of the IEEE International Symposium on Circuits and Systems*, Chicago, IL, pp. 1651–1654, May 1993.

[Hen00] E. Henning, Symbolic approximation and modeling techniques for analysis and design of analog circuits, PhD dissertation, University of Kaiserslautern. Aachen, Germany: Shaker Verlag, 2000.

[Ho75] C. Ho, A.E. Ruehli, and P.A. Brennan, The modified nodal approach to network analysis, *IEEE Transactions on Circuits and Systems*, 25(6): 504–509, June 1975.

[Hsu94] J.J. Hsu and C. Sechen, DC small-signal symbolic analysis of large analog integrated circuits, *IEEE Transactions on Circuits and Systems—I: Fundamental Theory and Applications*, 41(12): 817–828, December 1994.

[Hue89] L.P. Huelsman, Personal computer symbolic analysis programs for undergraduate engineering courses, *Proceedings of the IEEE International Symposium on Circuits and Systems*, Portland, OR, pp. 798–801, May 1989.

[Idl71] T.E. Idleman et al., SLIC: A simulator for linear integrated circuits, *IEEE Journal of Solid-State Circuits*, 6(4): 188–204, August 1971.

[Kon88] A. Konczykowska and M. Bon, Automated design software for switched-capacitor ICs with symbolic simulator SCYMBAL, *Proceedings of the Design Automation Conference*, Anaheim, CA, pp. 363–368, June 1988.

[Kon89] A. Konczykowska and M. Bon, Symbolic simulation for efficient repetitive analysis and artificial intelligence techniques in CAD, *Proceedings of the IEEE International Symposium on Circuits and Systems*, Portland, OR, pp. 802–805, May 1989.

[Kon93] A. Konczykowska et al., Parameter extraction of semiconductor devices electrical models using symbolic approach, *Alta Frequenza Rivista Di Elettronica*, 5(6): 3–5, November 1993.

[Ley00] F. Leyn, G. Gielen, and W. Sansen, Towards full insight in small-signal behavior of analog circuits: Assessment of different symbolic analysis approaches, *Proceedings of the 7th International Workshop on Symbolic Methods and Applications in Circuit Design*, Lisbon, Portugal, pp. 89–93, October 2000.

[Lib93] A. Liberatore et al., Simulation of switching power converters using symbolic techniques, *Alta Frequenza Rivista Di Elettronica*, 5(6): 16–23, November 1993.

[Lib95] A. Liberatore et al., A new symbolic program package for the interactive design of analog circuits, *Proceedings of the IEEE International Symposium on Circuits and Systems*, Seattle, WA, pp. 2209–2212, May 1995.

[Lin91] P.M. Lin, *Symbolic Network Analysis*. Amsterdam, the Netherlands: Elsevier Science, 1991.

[Lin92] P.M. Lin, Sensitivity analysis of large linear networks using symbolic programs, *Proceedings of the IEEE International Symposium on Circuits and Systems*, San Diego, CA, pp. 1145–1148, May 1992.

[Man93] S. Manetti and M. Piccirilli, Symbolic simulators for the fault diagnosis of nonlinear analog circuits, *International Journal on Analog Integrated Circuits and Signal Processing*, Kluwer Academic, 3(1): 59–72, January 1993.

[Mar57] H.M. Markowitz, The elimination form of the inverse and its application to linear programming, *Management Science*, 3(3): 255–269, April 1957.

[Mat93] T. Matsumoto, T. Sakabe, and K. Tsuji, On parallel symbolic analysis of large networks and systems, *Proceedings of the IEEE International Symposium on Circuits and Systems*, Chicago, IL, pp. 1647–1650, May 1993.

[Nag75] L.W. Nagel, *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, Memo ERL-M520, Electronics Research Laboratory, University of California, Berkeley, CA, May 1975.

[Nob77] B. Noble and J. Daniel, *Applied Linear Algebra*, 2nd edn. Englewood Cliffs, NJ: Prentice-Hall, 1977.

[Pie01] M. Pierzchala and B. Rodanski, Generation of sequential symbolic network functions for large-scale networks by circuit reduction to a two-port, *IEEE Transactions on Circuits and Systems—I: Fundamental Theory and Applications*, 48(7): 906–909, July 2001.

[Rod08] B. Rodanski and M. Hassoun, Symbolic analysis, *Fundamentals of Circuits and Filters*, W.-K. Chen, Ed. CRC Press: Boca Raton, 2008.

[Sed88] S. Seda, M. Degrauwe, and W. Fichtner, A symbolic analysis tool for analog circuit design automation, *1988 International Conference on Computer-Aided Design*, Santa Clara, CA, pp. 488–491, 1988.

[Sed92] S. Seda, M. Degrauwe, and W. Fichtner, Lazy-expansion symbolic expression approximation in SYNAP, *1992 International Conference on Computer-Aided Design*, Santa Clara, CA, pp. 310–317, 1992.

[Som91] R. Sommer, EASY: An experimental analog design system framework, *Proceedings of the International Workshop on Symbolic Methods and Applications in Circuit Design*, Paris, France, October 1991.

[Som93] R. Sommer, D. Ammermann, and E. Hennig, More efficient algorithms for symbolic network analysis: Supernodes and reduced loop analysis, *International Journal on Analog Integrated Circuits and Signal Processing*, Kluwer Academic, 3(1), 73–83, January 1993.

[Sta86] J.A. Starzyk and A. Konczykowska, Flowgraph analysis of large electronic networks, *IEEE Transactions on Circuits and Systems*, 33(3): 302–315, March 1986.

[Sty95] M.A. Styblinski and M. Qu, Comparison of symbolic analysis, approximation and macro-modeling techniques for statistical design for quality of analog integrated circuits, *Proceedings of the IEEE International Symposium on Circuits and Systems*, Seattle, WA, pp. 2221–2224, May 1995.

[Tem77] G. Temes, *Introduction to Circuit Synthesis and Design*, New York: McGraw Hill, 1977.

[Vla94] J. Vlach and K. Singhal, *Computer Methods for Circuit Analysis and Design*, 2nd Ed. New York: Van Nostrand Reinhold, 1994.

[Wan77] P. Wang, On the expansion of sparse symbolic determinants, *Proceedings of the International Conference on System Sciences*, Honolulu, HI, 1977.

[Wee73] W.T. Weeks et al., Algorithms for ASTAP: A network analysis program, *IEEE Transactions on Circuit Theory*, 20(11): 628–634, November 1973.

[Weh93] E. Wehrhahn, Symbolic analysis on parallel computers, *Proceedings of the European Conference on Circuit Theory and Design*, Davos, Switzerland, pp. 1693–1698, September 1993.

[Wie89] G. Wiezba et al., SSPICE: A symbolic SPICE program for linear active circuits, *Proceedings of the Midwest Symposium on Circuits and Systems*, Urbana, IL, pp. 1197–1201, August 1989.

[Wam92] P. Wambacq, G. Gielen, and W. Sansen, A cancellation free algorithm for the symbolic simulation of large analog circuits, *Proceedings of the IEEE International Symposium on Circuits and Systems*, San Diego, CA, pp. 1157–1160, May 1992.

[Yu96] Q. Yu and C. Sechen, A unified approach to the approximate symbolic analysis of large analog integrated circuits, *IEEE Transactions on Circuits and Systems—I: Fundamental Theory and Applications*, 43(8): 656–669, August 1996.

3

Numerical Analysis Methods

3.1 Equation Formulation 3-1
Implications of KCL, KVL, and the Element Branch Characteristics • Sparse Tableau Formulation • Nodal Analysis • Modified Nodal Analysis • Nodal Formulation by Stamps • Modified Nodal Formulation by Stamps
3.2 Solution of Linear Algebraic Equations 3-11
Factorization • Forward Substitution • Backward Substitution • Pivoting • Computation Cost of *LU* Factorization
References 3-16

Andrew T. Yang
Apache Design Solutions, Inc.

3.1 Equation Formulation

The method by which circuit equations are formulated is essential to a computer-aided circuit analysis program. It affects significantly the setup time, the programming effort, the storage requirement, and the performance of the program.

A linear time-invariant circuit with n nodes and b branches is completely specified by its network topology and branch constraints. The fundamental equations that describe the equilibrium conditions of a circuit are the Kirchhoff's current law (KCL) equations, the Kirchhoff's voltage law (KVL) equations, and the equations which characterize the individual circuit elements. Two methods are popular: the sparse tableau approach and the modified nodal approach.

3.1.1 Implications of KCL, KVL, and the Element Branch Characteristics

Given an example with $n = 4$ and $b = 6$ (as shown in Figure 3.1) we can sum the branch currents leaving each node to zero and obtain

$$\begin{bmatrix} -1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & -1 & 0 & -1 & 0 & -1 \end{bmatrix} \begin{bmatrix} i_{b_1} \\ i_{b_2} \\ i_{b_3} \\ i_{b_4} \\ i_{b_5} \\ i_{b_6} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

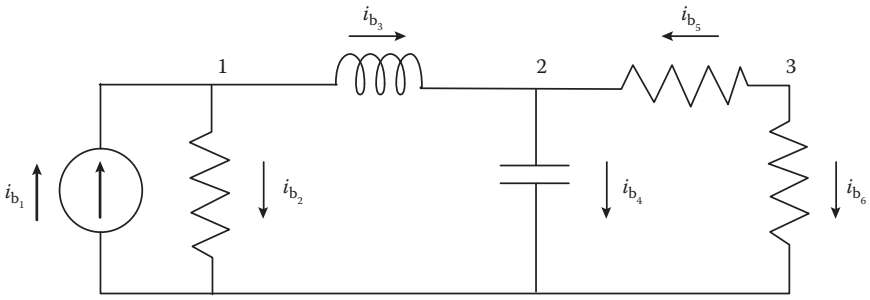


FIGURE 3.1 Network used to illustrate KCL, KVL, and the element branch characteristics.

or

$$A_a \times i_b = 0$$

where

A_a is an $n \times b$ incidence matrix contains $+1$, -1, and 0 entries
i_b is the branch current

Note that A_a is linearly dependent since the entries in each column add up to zero. A unique set of equations can be obtained by defining a datum node and eliminating its corresponding row of A_a. Hence, KCL results in $n-1$ equations and b unknowns. It implies

$$A \times i_b = 0 \tag{3.1}$$

where A is called the *reduced* incidence matrix.

KVL results in b equations with $b+n-1$ unknowns. It implies

$$v_b = A^T \times V_n \tag{3.2}$$

where

A^T is the transpose of the reduced incidence matrix
v_b is the branch voltage
V_n is the node-to-datum (nodal) voltage

Define the convention as follows (Figure 3.2):

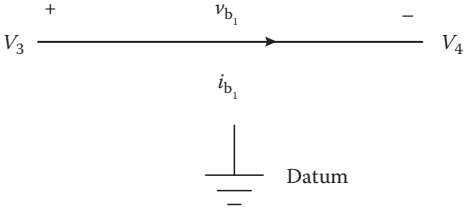


FIGURE 3.2 Reference convention for voltages and current.

One can sum the voltages around the loop using KVL (Figure 3.3):

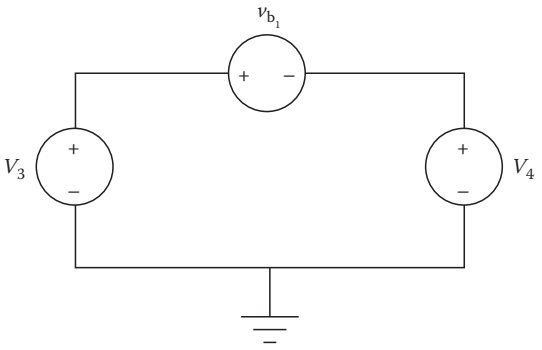


FIGURE 3.3 Voltage around the loop using KVL.

or

$$v_{b_1} = V_3 - V_4$$

Apply KVL to the example above and we obtain

$$\begin{bmatrix} v_{b_1} \\ v_{b_2} \\ v_{b_3} \\ v_{b_4} \\ v_{b_5} \\ v_{b_6} \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix}$$

or

$$v_b = A^T \times V_n$$

where v_b and V_n are unknowns.

The element characteristic results in generalized constraints equations in the form of

$$Y_b \cdot v_b + Z_b \cdot i_b = S_b \tag{3.3}$$

For example,
Resistor:

$$\frac{1}{R} v_b - i_b = 0$$

Voltage source:

$$v_b = E$$

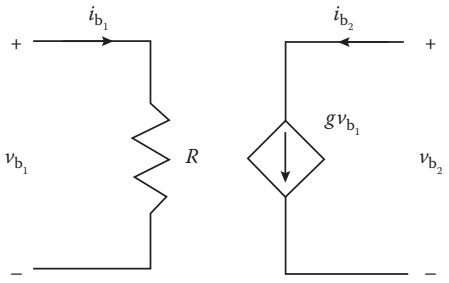


FIGURE 3.4 Voltage-controlled current source (VCCS).

VCCS (Figure 3.4):

$$\begin{bmatrix} 1/R & 0 \\ g & 0 \end{bmatrix} \begin{bmatrix} v_{b_1} \\ v_{b_2} \end{bmatrix} + \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_{b_1} \\ i_{b_2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

3.1.2 Sparse Tableau Formulation

The sparse tableau method simply combines Equations 3.1 through 3.3. The equation and unknown count can be summarized as follows:

	Number of Equations	Number of Unknowns
(3.1)	$n-1$	b
(3.2)	b	$b+n-1$
(3.3)	b	0
Total	$2b+\mathrm{n}-1$	$2b+n-1$

The sparse tableau in the matrix form is shown next:

$$\begin{bmatrix} 0 & A & 0 \\ A^{\mathrm{T}} & 0 & -1 \\ 0 & Z_{\mathrm{b}} & Y_{\mathrm{b}} \end{bmatrix} \begin{bmatrix} V_{\mathrm{n}} \\ i_{\mathrm{b}} \\ v_{\mathrm{b}} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ S_{\mathrm{b}} \end{bmatrix} \tag{3.4}$$

Equation 3.4 is a system of linear equations.

Note that the matrix formulated by the sparse tableau approach is typically sparse with many zero value entries. The main advantage is its generality (i.e., all circuit unknowns including branch current/voltage and node-to-datum voltage can be obtained in one pass). Since the number of equations is usually very large, an efficient sparse linear system solver is essential.

3.1.3 Nodal Analysis

The number of equations in Equation 3.4 can be reduced significantly by manipulating Equations 3.1 through 3.3. Motivated by the fact that the number of branches in a circuit is generally larger than the number of nodes, nodal analysis attempts to reduce the number of unknowns to V_{n}. As we will see, this is achieved by a loss of generality (i.e., not all types of linear elements can be processed).

We can eliminate the branch voltages v_b by substituting Equation 3.2 into Equation 3.3. This yields

$$Y_b A^T V_n + Z_b i_b = S_b \tag{3.5}$$

$$A \times i_b = 0 \tag{3.6}$$

Combining Equations 3.5 and 3.6 to eliminate the branch currents, i_b, we obtain a set of equations with V_n as unknowns:

$$A \cdot Z_b^{-1} \cdot \left(-Y_b A^T V_n + S_b\right) = 0 \tag{3.7}$$

Since the Z_b matrix may be singular, not all elements can be processed. For example, a voltage-controlled voltage source, as shown in Figure 3.5, can be cast in the form of Equation 3.3:

$$\begin{bmatrix} 0 & 0 \\ u & -1 \end{bmatrix} \begin{bmatrix} v_{b_1} \\ v_{b_2} \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_{b_1} \\ i_{b_2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Note that Z_b is singular and it cannot be inverted. Consider a special case where Equation 3.3 can be expressed as

$$Y_b v_b - i_b = 0 \tag{3.8}$$

or $-Z_b$ is a unit matrix and $S_b = 0$. This condition is true if the circuit consists of

- Resistor
- Capacitor
- Inductor
- Voltage-controlled current source

For example, Figure 3.6 is a VCCS.

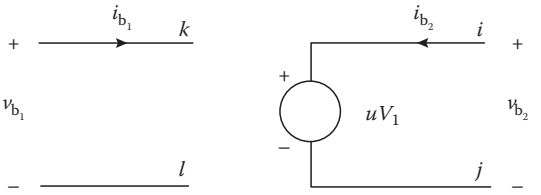


FIGURE 3.5 Voltage-controlled voltage source (VCVS).

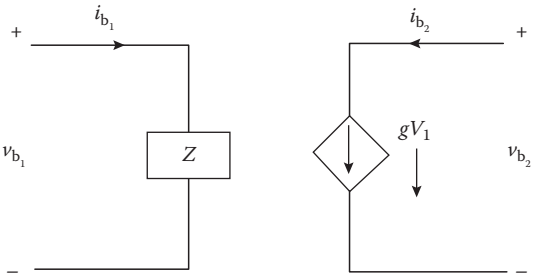


FIGURE 3.6 VCCS.

The branch constraints can be cast as

$$\begin{bmatrix} 1/z & 0 \\ g & 0 \end{bmatrix} \begin{bmatrix} v_{b_1} \\ v_{b_2} \end{bmatrix} + \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} i_{b_1} \\ i_{b_2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

and $z = 1/j\omega C$ for the capacitor, $z = j\omega L$ for an inductor, and $z = R$ for a resistor. For this type of circuit, the nodal analysis results in $n - 1$ equations with V_n as unknowns, or

$$A Y_b A^T \times V_n = 0$$

A circuit of this type contains no excitation (voltage/current source). The current sources can be included in the formulation by letting the corresponding edges of the current sources be numbered last. Hence, we can partition the reduced incidence matrix A into

$$A = [A_b | A_J] \tag{3.9}$$

where A_J corresponds to the subincidence matrix of the current source branches. Then, Equations 3.1, 3.2, and 3.8 can be expressed as

$$A_b i_b + A_J J = 0 \tag{3.10}$$

$$v_b = A_b^T V_n \tag{3.11}$$

$$i_b = Y_b v_b \tag{3.12}$$

Rearranging these equations yields

$$A_b Y_b A_b^T \times V_n = -A_J J \tag{3.13}$$

where J is a vector containing the current source values. Voltage sources can also be included in the formulation by simple source transformation which requires moderate preprocessing. The nodal analysis approach, therefore, can be applied to formulate equations for circuits consisting of

- Resistor
- Capacitor
- Inductor
- Voltage source (with preprocessing)
- Current source
- Voltage-controlled current source

Nodal analysis, however, when applied to an inductor, can cause numerical problems when the frequency is low:

$$\left(\frac{1}{j\omega L}\right)(v_{b_1}) + (-1)(i_{b_1}) = 0$$

or

$$\omega \to 0, \quad \frac{1}{j\omega L} \to \infty$$

To sum up, the nodal analysis must be extended to process the following linear elements (without proprocessing):

- Inductor
- Voltage source
- Current-controlled current source
- Current-controlled voltage source
- Voltage-controlled voltage source

3.1.4 Modified Nodal Analysis

A set of self-consistent modifications to the nodal analysis are proposed and the resultant formulation is called the modified nodal analysis (MNA). The MNA resolves the limitations of the nodal analysis method while preserving its advantages. In this section, we present the basic theory of MNA.

Divide all types of elements into three groups:

Group 1: Elements that satisfy

$$i_b = Y_b v_b$$

such as resistor, capacitor, and VCCS.

Group 2: Elements that satisfy

$$Y_b v_b + Z_b i_b = S_b$$

such as voltage source, inductor, VCVS, CCVS, and CCCS.

Group 3: Current source only.

Apply the partitioning technique to Group 1 and Group 2 elements. We can write

$$[A_1|A_2]\begin{bmatrix} i_1 \\ i_2 \end{bmatrix} = 0 \Rightarrow A_1 i_1 + A_2 i_2 = 0$$

$$\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} A_1^T \\ A_2^T \end{bmatrix} V_n \Rightarrow v_1 = A_1^T V_n, \quad v_2 = A_2^T V_n$$

$$i_1 = Y_1 v_1$$

$$Y_2 \cdot v_2 + Z_2 \cdot i_2 = S_2$$

Eliminating v_1, i_1, and v_2 from the preceding equations, we derive a system of linear equations with V_n and i_2 as unknowns:

$$A_1 Y_1 A_1^T V_n + A_2 i_2 = 0 \tag{3.14}$$

$$Y_2 A_2^T V_n + Z_2 i_2 = S_2 \tag{3.15}$$

Casting them in matrix form gives

$$\begin{bmatrix} A_1 Y_1 A_1^T & A_2 \\ Y_2 A_2^T & Z_2 \end{bmatrix} \begin{bmatrix} V_n \\ i_2 \end{bmatrix} = \begin{bmatrix} 0 \\ S_2 \end{bmatrix} \tag{3.16}$$

Finally, we apply the partitioning technique to include the current source (Group 3):

$$\begin{bmatrix} A_1 Y_1 A_1^{\mathrm{T}} & A_2 \\ Y_2 A_2^{\mathrm{T}} & Z_2 \end{bmatrix} \begin{bmatrix} V_{\mathrm{n}} \\ i_2 \end{bmatrix} = \begin{bmatrix} -A_J J \\ S_2 \end{bmatrix} \tag{3.17}$$

or

$$Y_{\mathrm{n}} \times x = J_{\mathrm{n}} \tag{3.18}$$

where

Y_{n} is the node admittance matrix
J_{n} is the source vector
x is the unknown vector

Implementing Equation 3.18 by matrix multiplication is difficult. Stamping methods have been developed to stamp in the entries of Y_{n}, J_{n} *element by element*. It is a very efficient way to implement Equation 3.18 into a network analysis program.

3.1.5 Nodal Formulation by Stamps

In this section, we developed the stamping rules for Group 1 and Group 3 elements only. Given a circuit consisting of Group 1 and Group 3 elements only:

We write the nodal equations at nodes 1, 2, and 3 (Figure 3.7):

$$-I + j\omega C(V_1 - V_3) + 1/R_1(V_1 - V_3) = 0 \qquad \text{(node 1)}$$

$$g(V_1 - V_3) + 1/R_2(V_2 - V_3) = 0 \qquad \text{(node 2)}$$

$$I + j\omega C(V_3 - V_1) + \frac{1}{R(V_3 - V_1)} - g(V_1 - V_3) + \frac{1}{R_2(V_3 - V_2)} = 0 \qquad \text{(node 3)}$$

Cast them in matrix form:

$$\begin{bmatrix} j\omega C + 1/R_1 & 0 & -j\omega C - 1/R_1 \\ g & 1/R_2 & -1/R_2 - g \\ -j\omega C - 1/R_1 - g & -1/R_2 & j\omega C + 1/R_1 + 1/R_2 + g \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix} = \begin{bmatrix} I \\ 0 \\ -I \end{bmatrix}$$

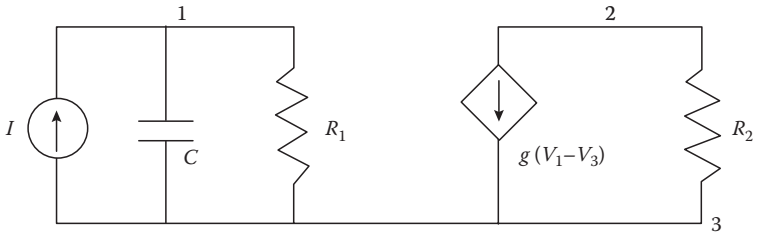


FIGURE 3.7 Network used to illustrate nodal formulation by stamps.

Note that for the left-hand side (LHS), we can write

$$\begin{bmatrix} 1/R_1 & 0 & -1/R_1 \\ 0 & 0 & 0 \\ -1/R_1 & 0 & 1/R_1 \end{bmatrix} + \begin{bmatrix} j\omega C & 0 & -j\omega C \\ 0 & 0 & 0 \\ -j\omega C & 0 & j\omega C \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ g & 0 & -g \\ -g & 0 & g \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1/R_2 & -1/R_2 \\ 0 & -1/R_2 & 1/R_2 \end{bmatrix}$$

Therefore, the sampling rule for a resistor is

$$Y_n(i,i) = Y_n(i,i) + \frac{1}{R}$$

$$Y_n(j,j) = Y_n(j,j) + \frac{1}{R}$$

$$Y_n(i,j) = Y_n(i,j) - \frac{1}{R}$$

$$Y_n(j,i) = Y_n(j,i) - \frac{1}{R}$$

If node i is grounded, the corresponding row and column can be eliminated from the node admittance matrix. We then obtained only

$$Y_n(j,j) = Y_n(j,j) + \frac{1}{R}$$

Similar stamping rules can be derived for a capacitor and a VCCS. The stamping rule for a current source J, flowing from i to j is

$$J_n(i) = J_n(i) - J$$

$$J_n(j) = J_n(j) + J$$

3.1.6 Modified Nodal Formulation by Stamps

Given a circuit including Group 2 elements only:

Let us first define auxiliary branch current unknowns i_v, i_L, and i_β for each type of element in Group 2. From nodal analysis, we obtain the following nodal equations (Figure 3.8):

$$i_v + i_L = 0 \qquad \text{(node 1)}$$

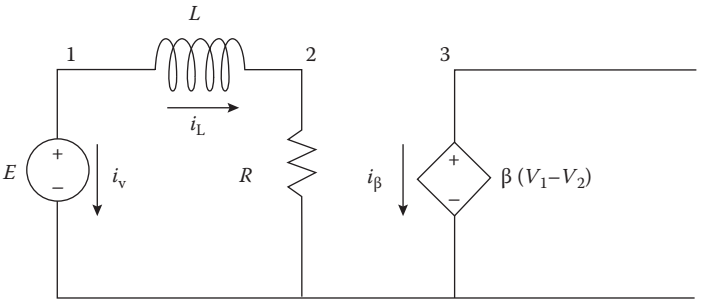


FIGURE 3.8 Network used to illustrate modified nodal formulation by stamps.

$$-i_{\mathrm{L}} + \frac{1}{R(V_2 - V_4)} = 0 \qquad \text{(node 2)}$$

$$i_{\beta} = 0 \qquad \text{(node 3)}$$

$$-i_{\mathrm{v}} + \frac{1}{R(V_4 - V_2)} - i_{\beta} = 0 \qquad \text{(node 4)}$$

For each auxiliary unknown, one auxiliary equation must be provided.

$$\begin{aligned} V_1 - V_4 &= E \\ V_1 - V_2 &= j\omega L i_{\mathrm{L}} \\ V_3 - V_4 &= \beta(V_1 - V_2) \end{aligned}$$

Cast these equations in matrix form:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1/R & 0 & -1/R & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & -1/R & 0 & 1/R & -1 & 0 & -1 \\ 1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & -j\omega L & 0 \\ -\beta & \beta & 1 & -1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \\ i_{\mathrm{v}} \\ i_{\mathrm{L}} \\ i_{\beta} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ E \\ 0 \\ 0 \end{bmatrix}$$

Hence, the following stamping rules are derived:
Voltage source:
LHS:

$$\begin{array}{c} \\ i \\ j \\ i_{\mathrm{v}} \end{array} \begin{array}{c} \begin{array}{ccc} i & j & i_{\mathrm{v}} \end{array} \\ \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ 1 & -1 & 0 \end{bmatrix} \end{array}$$

Right-hand side (RHS):

$$\begin{array}{c} i \\ j \\ i_{\mathrm{v}} \end{array} \begin{bmatrix} 0 \\ 0 \\ E \end{bmatrix}$$

Inductor:
LHS:

$$\begin{array}{c} \\ i \\ j \\ i_{\mathrm{L}} \end{array} \begin{array}{c} \begin{array}{ccc} i & j & i_{\mathrm{L}} \end{array} \\ \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & -1 & -j\omega L \end{bmatrix} \end{array}$$

Note that the numerical problem associated with an inductor is avoided when $\omega \to 0$.

VCVS (Figure 3.5):
LHS:

$$\begin{array}{c} \\ i \\ j \\ i_v \end{array} \begin{array}{c} \begin{array}{ccccc} i & j & k & l & i_v \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 \\ 1 & -1 & -\mu & \mu & 0 \end{bmatrix} \end{array}$$

The stamping rule for a CCVS and a CCCS can be developed following the same arguments. For DC analysis, set $\omega = 0$. In SPICE, the MNA is employed to formulate network equations. To probe a branch current, a user needs to insert a zero value voltage source between the adjacent nodes. The solution i_v is then the branch current. The implementation of MNA can be summarized as follows:

1. Implement an input parser to read in a circuit description file. A "free" input format does not restrict a user to follow column entry rules.
2. Build an internal node table which has a one-to-one relationship with user-defined nodes. Hence, a user needs not number the network nodes consecutively.
3. Number the auxiliary nodes for Group 2 elements last. For Group 2 element, one extra node is needed.
4. Solve the system of linear equations to find output voltages and currents.

3.2 Solution of Linear Algebraic Equations

The methods of solving a set of linear equations are basic to all computer-aided network analysis problems. If the network is linear, the equations are linear. Nonlinear networks lead to a system of nonlinear equations which can be linearized about some operating point. Transient analysis involves solving these linearized equations at many iteration points. Frequency domain analysis (small-signal AC analysis) requires the repeated solution of linear equations at specified frequencies.

The discussion in this section will be an introduction to the direct solution method based on *LU* decomposition, a variant of Gaussian elimination. This method is frequently used because of its efficiency, robustness, and ease of implementation. More advanced topics such as the general sparse matrix techniques are not discussed.

Consider a set of n linear algebraic equations of the form:

$$Ax = b \tag{3.19}$$

where
A is an $n \times n$ nonsingular real matrix
x and b are n-vectors

For the system to have a unique solution, A must be nonsingular (i.e., the determinant of A must not be zero). Equation 3.19 can be solved efficiently by first factorizing A into a product of two matrices L and U, which are respectively lower and upper triangular. This so-called *LU* decomposition method for solving a system of linear equations is similar to the Gaussian elimination except b is not required in advance. All operations performed on b using the Gaussian elimination are eliminated to save computation cost. The procedures are expressed as follows:

1. Step 1: Factorization/decomposition
2. Step 2: Forward substitution
3. Step 3: Backward substitution

3.2.1 Factorization

We factor A into a product LU, where L is a lower triangular matrix and U is an upper triangular matrix:

$$A = LU \tag{3.20}$$

Either L or U can have a diagonal of ones. The factors of A, being upper and lower triangular, can be stored in one matrix B, i.e., $B = L + U - I$. In practice, B is stored in place of A to save memory storage. There are two widely used algorithms for factorization: Crout's algorithm (setting the diagonal elements of U to 1) [1] and Doolittle's algorithm (setting the diagonal elements of L to 1) [3]. In the following, we will use a 4×4 matrix to illustrate the direct finding of L and U by the Crout's algorithm:

$$\begin{bmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix} \begin{bmatrix} 1 & u_{12} & u_{13} & u_{14} \\ 0 & 1 & u_{23} & u_{24} \\ 0 & 0 & 1 & u_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Multiplying the two matrices on the LHS of the preceding equations gives

$$\begin{bmatrix} l_{11} & l_{11}u_{12} & l_{11}u_{13} & l_{11}u_{14} \\ l_{21} & l_{21}u_{12} + l_{22} & l_{21}u_{13} + l_{22}u_{23} & l_{21}u_{14} + l_{22}u_{24} \\ l_{31} & l_{31}u_{12} + l_{32} & l_{31}u_{13} + l_{32}u_{23} + l_{33} & l_{31}u_{14} + l_{32}u_{24} + l_{33}u_{34} \\ l_{41} & l_{41}u_{12} + l_{42} & l_{41}u_{13} + l_{42}u_{23} + l_{43} & l_{41}u_{14} + l_{42}u_{24} + l_{43}u_{34} + l_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

The solution sequences, indicated by the superscripts, for solving the 16 unknowns ($n = 4$) are

$$\begin{matrix} l_{11}^{1} & u_{12}^{5} & u_{13}^{6} & u_{14}^{7} \\ l_{21}^{2} & l_{22}^{8} & u_{23}^{11} & u_{24}^{12} \\ l_{31}^{3} & l_{32}^{9} & l_{33}^{13} & u_{34}^{15} \\ l_{41}^{4} & l_{42}^{10} & l_{43}^{14} & l_{44}^{16} \end{matrix}$$

or

1st column: $l_{11} = a_{11}$, $l_{21} = a_{21}$, $l_{31} = a_{31}$, $l_{41} = a_{41}$
1st row: $u_{12} = \frac{a_{12}}{l_{11}}$, $u_{13} = \frac{a_{13}}{l_{11}}$, $u_{14} = \frac{a_{14}}{l_{11}}$

2nd column: $l_{22} = a_{22} - l_{21}u_{12}$, $l_{32} = a_{32} - l_{31}u_{12}$, $l_{42} = a_{42} - l_{41}u_{12}$
2nd row: $u_{23} = \frac{a_{23} - l_{21}u_{13}}{l_{22}}$, $u_{24} = \frac{a_{24} - l_{21}u_{14}}{l_{22}}$

3rd column: $l_{33} = a_{33} - l_{13}u_{13} - l_{32}u_{23}$, $l_{43} = a_{43} - l_{41}u_{13} - l_{42}u_{13}$
3rd row: $u_{34} = \frac{a_{34} - l_{31}u_{14} - l_{32}u_{24}}{l_{33}}$

4th column: $l_{44} = a_{44} - l_{41}u_{14} - l_{42}u_{24} - l_{43}u_{34}$

Note that l_{11}, l_{22}, and l_{33} are elements by which we divide and they are called *pivots*. Division by a zero pivot is not allowed. We now derive the Crout's algorithm for LU decomposition based on the solution procedures described previously:

1. $l_{ji} = a_{ji}$, $j = 1, 2, \ldots, n$ (1st column)
2. $u_{1j} = a_{1j}/l_{11}$, $j = 2, \ldots, n$ (1st row)
3. At the kth step, for column k,

$$l_{jk} = a_{jk} - l_{j1}u_{1k} - l_{j2}u_{2k} - \cdots = a_{jk} - \sum_{m=1}^{k-1} l_{jm}u_{mk}, \quad j = k, \ldots, n$$

4. At the kth step, for row k,

$$u_{kj} = \left(\frac{1}{l_{kk}}\right)(a_{kj} - l_{k1}u_{1j} - l_{k2}u_{2j} - \cdots) = \left(\frac{1}{l_{kk}}\right)\left(a_{kj} - \sum_{m=1}^{k-1} l_{km}u_{mj}\right), \quad j = k+1, \ldots, n$$

The algorithm can be summarized in a compact form:

1. Set $k = 1$
2. $l_{jk} = a_{jk} - \sum_{m=1}^{k-1} l_{jm}u_{mk}, \quad j = k, \ldots, n$
3. If $k = n$, stop
4. $u_{kj} = \left(\frac{1}{l_{kk}}\right)\left(a_{kj} - \sum_{m=1}^{k-1} l_{km}u_{mj}\right), \quad j = k+1, \ldots, n$
5. $k = k+1$, go to step 2

3.2.2 Forward Substitution

Once A has been factored into L and U, the system of equations is written as follows:

$$LUx = b$$

Define an auxiliary vector y, which can be solved by

$$L \cdot y = b \tag{3.21}$$

Due to the special form of L, the auxiliary vector γ can be solved very simply:

$$\begin{bmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

Starting from the first equation we write the solution as follows:

$$\begin{aligned} y_1 &= \frac{b_1}{l_{11}} \\ y_2 &= \frac{(b_2 - l_{21}y_1)}{l_{22}} \\ y_3 &= \frac{(b_3 - l_{31}y_1 - l_{32}y_2)}{l_{33}} \\ y_4 &= \frac{(b_4 - l_{41}y_1 - l_{42}y_2 - l_{43}y_3)}{l_{44}} \end{aligned}$$

and, in general

$$y_i = \frac{\left(b_i = \sum_{j=1}^{i-1} l_{ij}y_i\right)}{l_{ii}}, \quad i = 1, \ldots, n$$

This is called the *forward substitution* process.

3.2.3 Backward Substitution

Once y has been solved, we can proceed to solve for the unknown x by

$$Ux = y \tag{3.22}$$

Again, due to the special form of U, the unknown vector x can be solved very simply:

$$\begin{bmatrix} 1 & u_{12} & u_{13} & u_{14} \\ 0 & 1 & u_{23} & u_{24} \\ 0 & 0 & 1 & u_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

Starting from the first equation, we write the solution as follows:

$$\begin{aligned} x_4 &= y_4 \\ x_3 &= y_3 - u_{34}x_4 \\ x_2 &= y_2 - u_{23}x_3 - u_{24}x_4 \\ x_1 &= y_1 - u_{12}x_2 - u_{13}x_3 - u_{14}x_4 \end{aligned}$$

and, in general,

$$x_i = y_i - \sum_{j=i+1}^{n} u_{ij}x_j, \quad i = n, n-1, \ldots, 1$$

This is called the *backward substitution* process.

3.2.4 Pivoting

If in the process of factorization the pivot (l_{kk}) is zero, it is then necessary to interchange rows and possibly columns to put a nonzero entry in the pivot position so that the factorization can proceed. This is known as *pivoting*.

If $a_{ll} = 0$

1. Need to find another row i which has $a_{il} \neq 0$. This can always be done. Otherwise, all entries of column 1 are zero and hence, det $|A| = 0$. The solution process should then be aborted.
2. Interchange row i and row 1. Note that this must be done for both A and b.

If $l_{kk} = 0$

1. Find another row $r (r = k+1, \ldots, n)$, which has

$$l_{rk} = a_{rk} - \sum_{m=1}^{k-1} l_{rm}u_{mk} \neq 0$$

2. Interchange row r and row k in A and b.

Pivoting is also carried out for numerical stability (i.e., minimize round-off error). For example, one would search for the entry with the maximum absolute value of l_{rR} in columns below the diagonal and perform row interchange to put that element on the diagonal. This is called partial pivoting. Complete

pivoting involves searching for the element with the maximum absolute value in the unfactorized part of the matrix and moving that particular element to the diagonal position by performing both row and column interchanges. Complete pivoting is more complicated to program than partial pivoting. Partial pivoting is used more often.

3.2.5 Computation Cost of *LU* Factorization

In this section, we derive the multiplication/division count for the *LU* decomposition process. As a variation, we derive the computation cost for the Doolittle's algorithm (setting the diagonal elements of L to 1):

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

LHS after multiplication:

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ l_{21}u_{11} & l_{21}u_{12} + u_{22} & l_{21}u_{13} + u_{23} & l_{21}u_{14} + u_{24} \\ l_{31}u_{11} & l_{31}u_{12} + l_{32}u_{22} & l_{31}u_{13} + l_{32}u_{23} + u_{33} & l_{31}u_{14} + l_{32}u_{24} + u_{34} \\ l_{41}u_{11} & l_{41}u_{12} + l_{42}u_{22} & l_{41}u_{13} + l_{42}u_{23} + l_{43}u_{33} & l_{41}u_{14} + l_{42}u_{24} + l_{43}u_{34} + u_{44} \end{bmatrix}$$

$$\begin{aligned} &\text{Column 1: } (l_{21}, l_{31}, l_{41})u_{11} \\ &\text{Column 2: } (l_{21}, l_{31}, l_{41})u_{12} + (l_{32}, l_{42})u_{22} \\ &\text{Column 3: } (l_{21}, l_{31}, l_{41})u_{13} + (l_{32}, l_{42})u_{23} + l_{43}u_{33} \\ &\text{Column 4: } (l_{21}, l_{31}, l_{41})u_{14} + (l_{32}, l_{42})u_{24} + l_{43}u_{34} \end{aligned} \tag{3.23}$$

Let the symbol $\langle \cdot \rangle$ denote the number of nonzero elements of a matrix or vector. Or

- $\langle U \rangle$: number of nonzeros of U
- $\langle L \rangle$: number of nonzeros of L
- $\langle A_{i\cdot} \rangle$: number of nonzeros in row i of matrix A
- $\langle A_{\cdot j} \rangle$: number of nonzeros in column j of matrix A

From Equation 3.23, the total number of nonzero multiplications and divisions for *LU* factorization is given by

$$(\langle L_{\cdot 1} \rangle - 1)(\langle U_{1\cdot} \rangle) + (\langle L_{\cdot 2} \rangle - 1)(\langle U_{2\cdot} \rangle) + (\langle L_{\cdot 3} \rangle - 1)(\langle U_{3\cdot} \rangle) \tag{3.24}$$

Let α be the total number of multiplications and divisions. Express Equation 3.24 as a summation:

$$\alpha = \sum_{k=1}^{n} (\langle L_{\cdot k} \rangle - 1)(\langle U_{k\cdot} \rangle) \tag{3.25}$$

or for $n = 4$,

$$\alpha = 3 \times 4 + 2 \times 3 + 1 \times 2$$

If L and U are full, Equation 3.25 can be simplified as follows:

$$\begin{aligned}\alpha &= \sum_{k=1}^{n}(n-k)(n-k+1)\\ &= \sum_{k=1}^{n}(n^2+k^2-2kn+n-k)\\ &= n_3+\sum_{k=1}^{n}k^2-2n\times\frac{n(n+1)}{2}+n^2-\frac{n(n+1)}{2}\\ &= \frac{n(n+1)(2n+1)}{6}-\frac{n(n+1)}{2}\\ &= \frac{n^3-n}{3}\end{aligned} \tag{3.26}$$

The total number of mul/div for forward substitution is equal to the total number of nonzeros for L or $\langle L\rangle$. The total number of mul/div for backward substitution is equal to the total number of nonzeros for U for $\langle U\rangle$. Let β be the mul/div count for the forward and backward substitutions,

$$\beta = \langle L\rangle + \langle U\rangle$$

It follows that

$$\beta = n^2 \tag{3.27}$$

If L and U are full, combining Equations 3.26 and 3.27, we obtain the computation cost of solving a system of linear algebraic equations using direct LU decomposition:

$$\begin{aligned}\text{Total} &= \alpha+\beta\\ \text{Total} &= \alpha+\beta = \frac{n^3}{3}+n^2-\frac{n}{3}\end{aligned} \tag{3.28}$$

References

1. Hachtel et al., The sparse tableau approach to network analysis and design, *IEEE Trans. Circuit Theory*, CT-18: 101–113, January 1971.
2. L. W. Nagel, *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, Memo No. ERL-M520, Electronic Research Laboratory, University of California, Berkeley, CA, May 1975.
3. C. W. Ho et al., The modified nodal approach to network analysis, *IEEE Trans. Circuits Syst.*, CAS-22: 504–509, June 1975.
4. J. Vlach and K. Singhal, *Computer Methods for Circuit Analysis and Design*, Van Nostrand Reinhold, New York, 1983.
5. K. S. Kundert, *Advances in CAD for VLSI: Circuit Analysis, Simulation and Design*, Vol. 3, Part 1, Chapter 6, North-Holland, Amsterdam, the Netherlands, 1986.

4

Design by Optimization

4.1 Introduction 4-1
4.2 Optimization Algorithms 4-2
Nonlinear Optimization Problems • Basic Definitions • Constrained Optimization Methods • Multicriterion Optimization and Pareto Criticality
4.3 Transistor Sizing Problem for CMOS Digital Circuits ... 4-10
Problem Description • Delay Modeling • Area Model • Sensitivity-Based TILOS Algorithm • Transistor Sizing Using the Method of Feasible Directions • Lagrangian Multiplier Approaches • Two-Step Optimization • Convex Programming-Based Approach • Statistical Optimization
4.4 Analog Design Centering Problem 4-19
Problem Description • Simplicial Approximation Method • Ellipsoidal Method • Convexity-Based Approaches • Concluding Remarks
References 4-24

Sachin S. Sapatnekar
University of Minnesota

4.1 Introduction

In many integrated circuit (IC) design situations, a designer must make complex trade-offs between conflicting behavioral requirements, dealing with functions that are often nonlinear. The number of parameters involved in the design process may be large, necessitating the use of algorithms that provide qualitatively good solutions in a computationally efficient manner.

The theory and utilization of optimization algorithms in computer-aided design (CAD) of ICs are illustrated here. The form of a general nonlinear optimization problem is first presented, and some of the commonly used methods for optimization are overviewed. It is frequently said that setting up an optimization problem is an art, while (arguably) solving it is an exact science. To provide a flavor for both of these aspects, case studies on the following specific design problems are examined:

- *Transistor sizing*: The delay of a digital circuit can be tuned by adjusting the sizes of transistors within it. By increasing the sizes of a few selected transistors from the minimum size, significant improvements in performance are achievable. However, one must take care to ensure that the area overhead incurred in increasing these sizes is not excessive. The area-delay trade-off here is the transistor size optimization problem. We address the solution of this problem under deterministic and statistical delay models.
- *Analog design centering*: The values of design parameters of an analog circuit are liable to change due to manufacturing variations. This contributes to a deviation in the behavior of the circuit from the norm, and may lead to dysfunctional circuits that violate the performance parameters that they

were designed for. The problem of design centering attempts to ensure that under these variations, the probability of a circuit satisfying its performance specifications is maximized by placing the circuit in the center of its feasible region.

4.2 Optimization Algorithms

4.2.1 Nonlinear Optimization Problems

The standard form of a constrained nonlinear optimization problem is

$$\begin{array}{ll} \text{minimize} & f(\mathbf{x}): \mathbf{R}^n \rightarrow \mathbf{R} \\ \text{subject to} & \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \\ & \mathbf{g}: \mathbf{R}^n \rightarrow \mathbf{R}^m,\ \mathbf{x} \in \mathbf{R}^n \end{array} \tag{4.1}$$

representing the minimization of a function f of n variables under constraints specified by inequalities determined by functions $\mathbf{g} = [g_1 \cdots g_m]^{\mathrm{T}}$. f and g_i are, in general, nonlinear functions, so that the linear programming problem is a special case of the above. The parameters $\mathbf{x}$ may, for example, represent circuit parameters, and $f(\mathbf{x})$ and $g_i(\mathbf{x})$ may correspond to circuit performance functions. Note that "$\geq$" inequalities can be handled under this paradigm by multiplying each side by -1, and equalities by representing them as a pair of inequalities. The maximization of an objective function function $f(\mathbf{x})$ can be achieved by minimizing $-f(\mathbf{x})$.

The set $\mathcal{F} = \{\mathbf{x} \mid \mathbf{g}(\mathbf{x}) \leq 0\}$ that satisfies the constraints on the nonlinear optimization problem is known as the feasible set, or the feasible region. If $\mathcal{F}$ is empty (nonempty), then the optimization is said to be unconstrained (constrained).

Several mathematical programming techniques can be used to solve the optimization problem above; some of these are outlined here. For further details, the reader is referred to a standard text on optimization, such as [1]. Another excellent source for optimization techniques and their applications to IC design is a survey paper by Brayton et al. [2].

The above formulation may not directly be applicable to real-life design problems, where, often, multiple conflicting objectives must be optimized. In such a case, one frequently uses techniques that map on the problem to the form in Equation 4.1 (see Section 4.2.4).

4.2.2 Basic Definitions

In any discussion on optimization, it is virtually essential to understand the idea of a convex function and a convex set, since these have special properties, and it is desirable to formulate problems as convex programming problems, wherever it is possible to do so without an undue loss in modeling accuracy. (Unfortunately, it is not always possible to do so!)

Definition 4.1: A set C in $\mathbf{R}^n$ is said to be a convex set if, for every $\mathbf{x}_1, \mathbf{x}_2 \in C$, and every real number α, $0 \leq \alpha \leq 1$, the point $\alpha\mathbf{x}_1 + (1-\alpha)\mathbf{x}_2 \in C$.

This definition can be interpreted geometrically as stating that a set is convex if, given two points in the set, every point on the line segment joining the two points is also a member of the set. Examples of convex and nonconvex sets are shown in Figure 4.1.

Two examples of convex sets that are referred to later are the following geometric bodies:

1. An ellipsoid $E(\mathbf{x}, \mathcal{B}, r)$ centered at point $\mathbf{x}$ is given by the following equation:

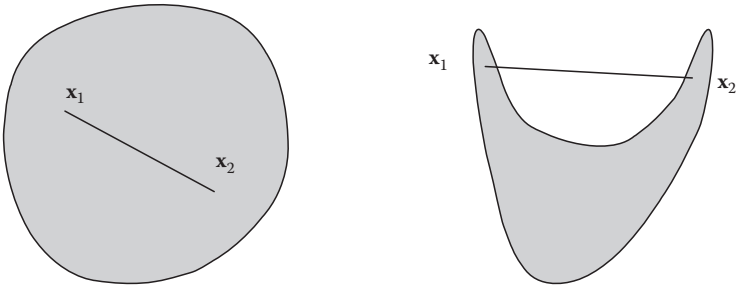


FIGURE 4.1 Convex sets.

$$\{\mathbf{y}|(\mathbf{y}-\mathbf{x})^{\mathrm{T}}\mathcal{B}(\mathbf{y}-\mathbf{x}) \leq r^2\}. \tag{4.2}$$

If $\mathcal{B}$ is a scalar multiple of the unit matrix, then the ellipsoid is called a hypersphere.

2. A (convex) polytope is defined as an intersection of half-spaces, and is given by the following equation:

$$\mathcal{P} \;=\; \{\mathbf{x}|A\mathbf{x} \geq \mathbf{b}\}, A \in \mathbf{R}^{m\times n}, \mathbf{b} \in \mathbf{R}^m, \tag{4.3}$$

corresponding to a set of m inequalities $\mathbf{a}_i^{\mathrm{T}}\,\mathbf{x} \geq b_i$, $\mathbf{a}_i \in \mathbf{R}^n$.

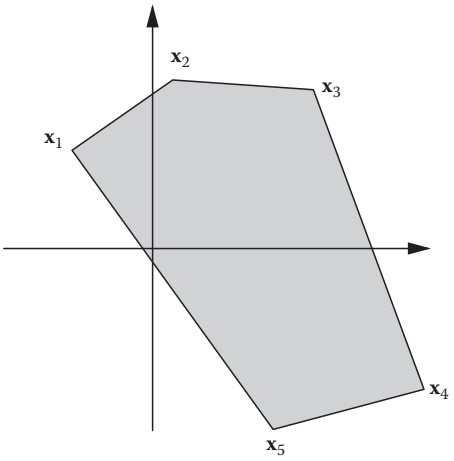


FIGURE 4.2 Convex hull of five points.

Definition 4.2: The convex hull of m points, $\mathbf{x}_1, \ldots, \mathbf{x}_m \in \mathbf{R}^n$, denoted co $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$, is defined as the set of points $\mathbf{y} \in \mathbf{R}^n$ such that

$$\mathbf{y} = \sum_{i=1}^{m} \alpha_i \mathbf{x}_i; \quad \alpha_i \geq 0 \;\forall\; i, \quad \sum_{i=0}^{m} \alpha_i = 1. \tag{4.4}$$

The convex hull is the smallest convex set that contains the m points. An example of the convex hull of five points in the plane is shown by the shaded region in Figure 4.2. If the set of points $\mathbf{x}_i$ is of finite cardinality (i.e., m is finite), then the convex hull is a polytope. Hence, a polytope is also often described as the convex hull of its vertices.

Definition 4.3: A function f defined on a convex set Ω is said to be a convex function if, for every $\mathbf{x}_1, \mathbf{x}_2 \in \Omega$, and every α, $0 \leq \alpha \leq 1$,

$$f[\alpha\mathbf{x}_1 + (1-\alpha)\mathbf{x}_2] \leq \alpha f(\mathbf{x}_1) + (1-\alpha) f(\mathbf{x}_2). \tag{4.5}$$

f is said to be strictly convex if the inequality in Equation 4.5 is strict for $0 < \alpha < 1$.

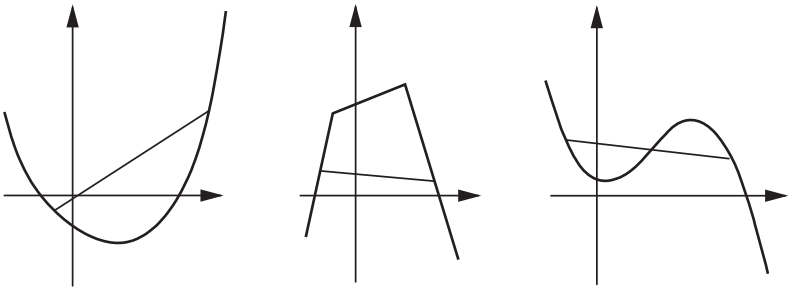


FIGURE 4.3 Convex functions.

Geometrically, a function is convex if the line joining two points on its graph is always above the graph. Examples of convex and nonconvex functions on $\mathbf{R}^n$ are shown in Figure 4.3.

Definition 4.4: A function g defined on a convex set Ω is said to be a concave function if the function $f = -g$ is convex. The function g is strictly concave if $-g$ is strictly convex.

Definition 4.5: The convex programming problem is stated as follows:

$$\text{minimize} \quad f(\mathbf{x}) \tag{4.6}$$

$$\text{such that} \quad \mathbf{x} \in S \tag{4.7}$$

where f is a convex function and S is a convex set. This problem has the property that any local minimum of f over S is a global minimum. An excellent reference on convex programming is [3].

Definition 4.6: A posynomial is a function h of a positive variable $\mathbf{x} \in \mathbf{R}^n$ that has the form

$$h(\mathbf{x}) = \sum_j \gamma_j \prod_{i=1}^{n} x_i^{\alpha_{ij}} \tag{4.8}$$

where the exponents $\alpha_{ij} \in \mathbf{R}$ and the coefficients $\gamma_j > 0$.

For example, the function $3.7x_1^{1.4}x_2^{\sqrt{3}} + 1.8x_1^{-1}x_3^{2.3}$ is a posynomial in the variables x_1, x_2, and x_3. Roughly speaking, a posynomial is a function that is similar to a polynomial, except that the coeffcients γ_j must be positive, and an exponent α_{ij} could be any real number, and not necessarily a positive integer, unlike a polynomial. A posynomial has the useful property that it can be mapped onto a convex function through an elementary variable transformation, $(x_i) = (e^{z_i})$. Such functional forms are useful since in the case of an optimization problem where the objective function and the constraints are posynomial, the problem can easily be mapped onto a convex programming problem.

The family of posynomials is expanded to the superset of generalized posynomials in Ref. [4], and these functions can also be mapped to convex functions under the above transform.

4.2.3 Constrained Optimization Methods

Most problems in IC design involve the minimization or maximization of a cost function subject to certain constraints. In this section, a few prominent techniques for constrained optimization are presented. The reader is referred to [1] for details on unconstrained optimization.

4.2.3.1 Linear Programming

Linear programming is a special case of nonlinear optimization, and is the convex programming problem where the objective and constraints are all linear functions. The problem is stated as

$$\begin{aligned} \text{minimize} \quad & \mathbf{c}^{\mathrm{T}}\,\mathbf{x} \\ \text{subject to} \quad & A\mathbf{x} \leq \mathbf{b} \\ \text{where} \quad & \mathbf{c}, \mathbf{x} \in \mathbf{R}^n,\ \mathbf{b} \in \mathbf{R}^m,\ A \in \mathbf{R}^{m \times n} \end{aligned} \tag{4.9}$$

It can be shown that any solution to a linear program must necessarily occur at a vertex of the constraining polytope. The most commonly used technique for solution of linear programs, the simplex method [1], is based on this principle. The computational complexity of this method can show an exponential behavior for pathological cases, but for most practical problems, it has been observed to grow linearly with the number of variables and sublinearly with the number of constraints. Algorithms with polynomial time worst-case complexity do exist; these include Karmarkar's method and the Shor–Khachiyan ellipsoidal method. The computational complexity of the latter, however, is often seen to be impractical from a practical standpoint.

Some examples of CAD problems that are posed as linear programs include placement, gate sizing, clock skew optimization, layout compaction, etc. In several cases, the structure of the problem can be exploited to arrive at graph-based solutions, for example, in layout compaction.

4.2.3.2 Lagrange Multiplier Methods

These methods are closely related to the first-order Kuhn–Tucker necessary conditions on optimality, which state that given an optimization problem of the form in Equation 4.1, if f and $\mathbf{g}$ are differentiable at $\mathbf{x}^\star$, then there is a vector $\boldsymbol{\lambda} \in \mathbf{R}^m$, $(\boldsymbol{\lambda})_i \geq 0$, such that

$$\nabla f(\mathbf{x}^\star) + \boldsymbol{\lambda}^{\mathrm{T}} \nabla \mathbf{g}(\mathbf{x}^\star) = 0 \tag{4.10}$$

$$\boldsymbol{\lambda}^{\mathrm{T}} \mathbf{g}(\mathbf{x}^\star) = 0 \tag{4.11}$$

These correspond to $m+1$ equations in $m+1$ variables; the solution to these provides the solution to the optimization problem. The variables $\boldsymbol{\lambda}$ are known as the Lagrange multipliers.

Note that since $g_i(\mathbf{x}^\star) \leq 0$, and because of the nonnegativity constraint on the Lagrange multipliers, $\boldsymbol{\lambda}$, it follows from Equation 4.11 that $(\boldsymbol{\lambda})_i = 0$ for inactive constraints (constraints with $g_i(\mathbf{x}) < 0$).

4.2.3.3 Penalty Function Methods

These methods convert the constrained optimization problem in Equation 4.1 into an equivalent unconstrained optimization problem, since such problems are easier to solve than constrained problems, as

$$\text{minimize } h(\mathbf{x}) = f(\mathbf{x}) + c \cdot P(\mathbf{x}) \tag{4.12}$$

where

$P(\mathbf{x})$: $\mathbf{R}^n \rightarrow \mathbf{R}$ is known as a penalty function

c is a constant

The value of $P(\mathbf{x})$ is zero within the feasible region, and positive outside the region, with the value becoming larger as one moves farther from the feasible region; one possible choice when the $g_i(\mathbf{x})$'s are continuous is given by

$$P(\mathbf{x}) = \sum_{i=1}^{m} \max(0, -g_i(\mathbf{x})) \tag{4.13}$$

For large c, it is clear that the minimum point of Equation 4.12 will be in a region where P is small. Thus, as c is increased, it is expected that the corresponding solution point will approach the feasible region and minimize f. As $c \to \infty$, the solution of the penalty function method converges to a solution of the constrained optimization problem.

In practice, if one were to begin with a high value of c, one may not have very good convergence properties. The value of c is increased in each iteration until c is high and the solution converges.

4.2.3.4 Method of Feasible Directions

The method of feasible directions is an optimization algorithm that improves the objective function without violating the constraints. Given a point $\mathbf{x}$, a direction $\mathbf{d}$ is feasible if there is a step size $\bar{\alpha} > 0$ such that $\mathbf{x} + \alpha \cdot \mathbf{d} \in \mathcal{F}\ \forall\ 0 \leq \alpha \leq \bar{\alpha}$, where $\mathcal{F}$ is the feasible region. More informally, this means that one can take a step of size up to $\bar{\alpha}$ along the direction $\mathbf{d}$ without leaving the feasible region. The method of feasible direction attempts to choose a value of α in a feasible direction $\mathbf{d}$ such that the objective function f is minimized along the direction, and α is such that $\mathbf{x} + \alpha \cdot \mathbf{d}$ is feasible.

One common technique that uses the method of feasible directions is as follows. A feasible direction at $\mathbf{x}$ is found by solving the following linear program:

$$\begin{aligned} &\text{minimize} && \varepsilon \\ &\text{subject to} && \langle \nabla f(\mathbf{x}) \cdot \mathbf{d} \rangle \leq \varepsilon && (4.14) \\ &&& \langle \nabla g_i(\mathbf{x}) \cdot \mathbf{d} \rangle \leq \varepsilon && (4.15) \\ &\text{and} && \text{normalization requirements on } \mathbf{d} \end{aligned}$$

where the second set of constraints are chosen for all $g_i \geq -b$, where b serves to incorporate the effects of near-active constraints to avoid the phenomenon of jamming (also known as zigzagging) [1]. The value of b is brought closer to 0 as the optimization progresses. One common method that is used as a normalization requirement is to set $\mathbf{d}^{\mathrm{T}}\mathbf{d} = 1$. This constraint is nonlinear and nonconvex, and is not added to the linear program as an additional constraint; rather, it is exercised by normalizing the direction $\mathbf{d}$ after the linear program has been solved. An appropriate step size in this direction is then chosen by solving a one-dimensional optimization problem.

Feasible direction methods are popular in finding engineering solutions because the value of $\mathbf{x}$ at each iteration is feasible, and the algorithm can be stopped at any time without waiting for the algorithm to converge, and the best solution found so far can be used.

4.2.3.5 Vaidya's Convex Programming Algorithm

As mentioned earlier, if the objective function and all constraints in Equation 4.1 are convex, the problem is a convex programming problem, and any local minimum is a global minimum. The first large-scale practical implementation of this algorithm is described in Ref. [5].

Initially, a polytope $\mathcal{P} \in \mathbf{R}^n$ that contains the optimal solution, $\mathbf{x}_{\text{opt}}$, is chosen. The objective of the algorithm is to start with a large polytope, and in each iteration, to shrink its volume while keeping the optimal solution, $\mathbf{x}_{\text{opt}}$, within the polytope, until the polytope becomes sufficiently small. The polytope $\mathcal{P}$ may, for example, be initially selected to be an n-dimensional box described by the set

$$\{\mathbf{x}|x_{\min} \leq x_i \leq x_{\max}\} \tag{4.16}$$

where $x_{\min}$ and $x_{\max}$ are the minimum and maximum values of each variable, respectively. The algorithm proceeds iteratively as follows:

Step 1. A center $\mathbf{x}_c$ deep in the interior of the polytope $\mathcal{P}$ is found.

Step 2. An oracle is invoked to determine whether the center $\mathbf{x}_c$ lies within the feasible region $\mathcal{F}$. This may be done by verifying that all of the constraints of the optimization problem are met at $\mathbf{x}_c$.

If the point $\mathbf{x}_c$ lies outside $\mathcal{F}$, it is possible to find a separating hyperplane passing through $\mathbf{x}_c$ that divides $\mathcal{P}$ into two parts, such that $\mathcal{F}$ lies entirely in the part satisfying the constraint

$$\mathbf{c}^T\mathbf{x} \geq \beta \tag{4.17}$$

where $\mathbf{c} = -[\nabla_{g_p}(\mathbf{x})]^T$ is the negative of the gradient of a violated constraint, g_p, and $\beta = \mathbf{c}^T\mathbf{x}_c$. The separating hyperplane above corresponds to the tangent plane to the violated constraint.

If the point $\mathbf{x}_c$ lies within the feasible region $\mathcal{F}$, then there exists a hyperplane (Equation 4.17) that divides the polytope into two parts such that $\mathbf{x}_{opt}$ is contained in one of them, with

$$\mathbf{c} = -[\nabla f(\mathbf{x})]^T \tag{4.18}$$

being the negative of the gradient of the objective function, and β being defined as $\beta = \mathbf{c}^T\mathbf{x}_c$ once again.

Step 3. In either case, Equation 4.17 is added to the current polytope to give a new polytope that has roughly half the original volume.

Step 4. Go to Step 1; the process is repeated until the polytope is sufficiently small.

Example 4.1

The algorithm is illustrated by using it to solve the following problem:

$$\text{minimize } f(x_1, x_2) \text{ such that } (x_1, x_2) \in S \tag{4.19}$$

where S is a convex set and f is a convex function. The shaded region in Figure 4.4a is the set S, and the dotted lines show the level curves of f. The point $\mathbf{x}_{opt}$ is the solution to this problem.

1. The expected solution region is bounded by a rectangle, as shown in Figure 4.4a.
2. The center, $\mathbf{x}_c$, of this rectangle is found.
3. The oracle is invoked to determine whether or not $\mathbf{x}_c$ lies within the feasible region. In this case, it can be seen that $\mathbf{x}_c$ lies outside the feasible region. Hence, the gradient of the constraint function is used to construct a hyperplane through $\mathbf{x}_c$, such that the polytope is divided into two parts of roughly equal volume, one of which contains the solution $\mathbf{x}_{opt}$. This is illustrated in Figure 4.4b, where the shaded region corresponds to the updated polytope.
4. The process is repeated on the new smaller polytope. Its center lies inside the feasible region; hence, the gradient of the objective function is used to generate a hyperplane that further shrinks the size of the polytope, as shown in Figure 4.4c.
5. The result of another iteration is illustrated in Figure 4.4d. The process continues until the polytope has been shrunk sufficiently.

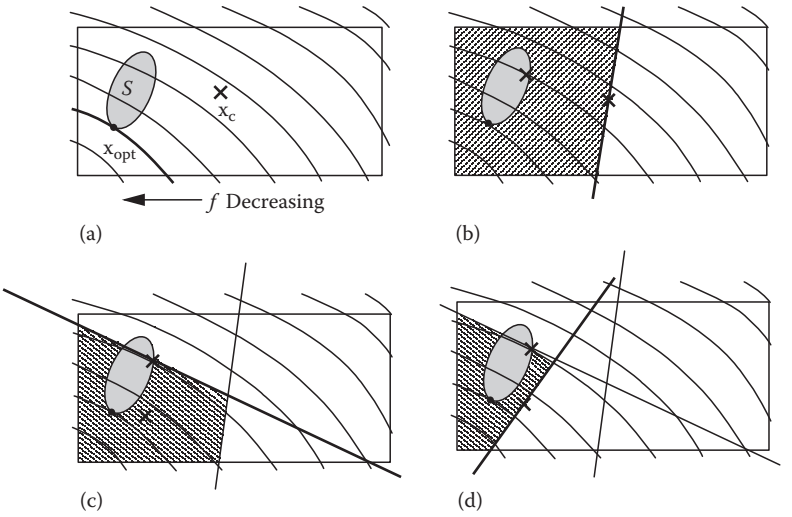


FIGURE 4.4 Example illustrating the convex programming algorithm. (From Sapatnekar, S. S., Rao, V. B., Vaidya, P. M., and Kang, S. M., *IEEE Trans. Comput. Aided Des.*, 12, 1621, 1993. With permission.)

4.2.3.6 Other Methods

The compact nature of this book makes it infeasible to enumerate or describe all of the methods that are used for nonlinear optimization. Several other methods, such as Newton's and modified Newton/quasi-Newton methods, conjugate gradient methods, etc., are often useful in engineering optimization. For these and more, the reader is referred to a standard text on optimization, such as [1].

4.2.4 Multicriterion Optimization and Pareto Criticality

Most IC design problems involve trade-offs between multiple objectives. In cases where one objective can be singled out as the most important one and a reasonable constraint set can be defined in terms of the other objectives, the optimization problem can be stated using the formulation in Equation 4.1. This is convenient since techniques for the solution of a problem in this form have been extensively studied, and a wide variety of optimization algorithms are available.

Let $\mathbf{f}$ be a vector of design objectives that is a function of the design variables $\mathbf{x}$, where

$$\mathbf{f}(\mathbf{x}):\ \mathbf{R}^n \rightarrow \mathbf{R}^m = (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_m(\mathbf{x})) \tag{4.20}$$

It is extremely unlikely in a real application that all of the f_i's will be optimal at the same point, and hence one must trade off the values of the f_i's in a search for the best design point.

In this context, we note that at a point $\mathbf{x}$, we are interested in taking a step δ in a direction $\mathbf{d}$, $\|\mathbf{d}\| = 1$, so that

$$f_i(\mathbf{x} + \delta \cdot \mathbf{d}) \leq f_i(\mathbf{x}) \ \forall\ 1 \leq i \leq m \tag{4.21}$$

A Pareto critical point is defined as a point $\mathbf{x}$ where no such small step of size less than δ exists in any direction. If a point is Pareto critical for any step size from the point $\mathbf{x}$, then $\mathbf{x}$ is a Pareto point. The notion of a Pareto critical point is, therefore, similar to that of a local minimum, and that of a Pareto point to a global minimum. In computational optimization, one is concerned with the problem of finding a local minimum since, except in special cases, it is the best that one can be guaranteed of finding without

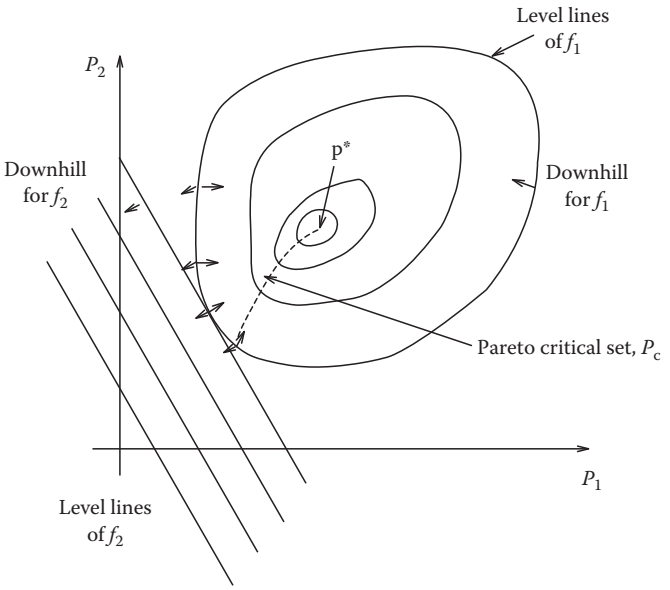


FIGURE 4.5 Exact conflict at a Pareto critical point. (From Brayton, R.K., Hachtel, G.D., and Sangiovanni-Vincentelli, A.L., *Proc. IEEE*, 69, 1334, 1981. With permission.)

an exhaustive search. If the set of all Pareto critical points is P_c, and the set of Pareto points is P, then clearly $P \subset P_c$. In general, there could be an infinite number of Pareto points, but the best circuit design must necessarily occur at a Pareto point $\mathbf{x} \in P$.

In Figure 4.5, the level curves of two objective functions are plotted in $\mathbf{R}^2$. f_1 is nonlinear and has a minimum at $\mathbf{x}^*$. f_2 is linear and decreases as both x_1 and x_2 decrease. The Pareto critical set, P_c, is given by the dashed curve. At few of the points, the unit normal to the level lines of f_1 and f_2, i.e., the negative gradients of f_1 and f_2, is shown. From the figure, it can be seen that if the unit normals at point $\mathbf{x}$ are not equal and opposite, then the unit normals will have a common downhill direction allowing a simultaneous decrease in f_1 and f_2, and hence, $\mathbf{x}$ would not be a Pareto critical point. Therefore, a Pareto critical point is one where the gradients of f_1 and f_2 are opposite in direction, i.e., $\lambda \nabla f_1 = -\nabla f_2$, where λ is some scale factor.

In higher dimensions, a Pareto critical point is characterized by the existence of a set of weights, $w_i > 0$ $\forall$ $1 \leq i \leq m$, such that

$$\sum_{i=1}^{m} w_i \nabla f_i = 0. \tag{4.22}$$

Some of the common methods that are used for multicriterion optimization are discussed below.

4.2.4.1 Weighted Sums Optimization

The multiple objectives $f_1(\mathbf{x}), \ldots, f_m(\mathbf{x})$ are combined as

$$F(\mathbf{x}) = \sum_{i=1}^{m} w_i f_i(\mathbf{x}) \tag{4.23}$$

where $w_i > 0$ $\forall$ $i = 1, \ldots, m$, and the function $F(\mathbf{x})$ is minimized.

At any local minimum point of $F(\mathbf{x})$, the relation in Equation 4.22 is seen to be valid, and hence, $\mathbf{x} \in P_c$. In general, $P \neq P_c$, but it can be shown that when each f_i is a convex function, then $P = P_c$; if so, it can also be shown that all Pareto points can be obtained by optimizing the function F in Equation 4.23. However, for nonconvex functions, there are points $\mathbf{x} \in P$ that cannot be obtained by the weighted sum optimization since Equation 4.22 is only a necessary condition for the minimum of F. A characterization of the Pareto points that can be obtained by this technique is provided in Ref. [2].

In practice, the w_i's must be chosen to reflect the magnitudes of the f_i's. For example, if one of the objectives is a voltage quantity whose typical value is a few volts, and another is a capacitor value that is typically a few picofarads, the weight corresponding to the capacitor value would be roughly 10^{12} times that for the voltage, in order to ensure that each objective has a reasonable contribution to the objective function value. The designer may further weigh the relative importance of each objective in choosing the w_i's. This objective may be combined with additional constraints to give a formulation of the type in Equation 4.1.

4.2.4.2 Minmax Optimization

The following objective function is used for Equation 4.1

$$\text{minimize } F(\mathbf{x}) = \max_{1 \leq i \leq m} w_i f_i(\mathbf{x}) \tag{4.24}$$

where the weights $w_i > 0$ are chosen as in the case of weighted sums optimization.

The above equation can equivalently be written as the following constrained optimization problem:

$$\begin{aligned} &\text{minimize } r \\ &\text{subject to } w_i f_i(\mathbf{x}) \leq r \end{aligned} \tag{4.25}$$

Minimizing the objective function described by Equation 4.24 with different sets of w_i values can be used to obtain all Pareto points [2].

Since this method can, unlike the weighted-sums optimization method, be used to find all Pareto critical points, it would seem to be a more natural setting for obtaining Pareto points than the weighted sum minimization. However, when the f_i's are convex, the weighted sums approach is preferred since it is an unconstrained minimization, and is computationally easier than a constrained optimization. It must be noted that when the f_i's are not all convex, the minmax objective function is nonconvex, and finding all local minima is a nontrivial process for any method.

4.3 Transistor Sizing Problem for CMOS Digital Circuits

4.3.1 Problem Description

Circuit delays in ICs often must be reduced to obtain faster response times. A typical CMOS digital IC consists of multiple stages of combinational logic blocks that lie between latches that are clocked by system clock signals. For such a circuit, delay reduction must ensure that valid signals are produced at each output latch of a combinational block, before any transition in the signal clocking the latch. In other words, the worst-case input–output delay of each combinational stage must be restricted to be below a certain specification.

Given the circuit topology, the delay of a combinational circuit can be controlled by varying the sizes of transistors in the circuit. Here, the size of a transistor is measured in terms of its channel width, since the channel lengths of MOS transistors in a digital circuit are generally uniform. In coarse terms, the circuit delay can usually be reduced by increasing the sizes of certain transistors in the circuit. Hence, making

the circuit faster usually entails the penalty of increased circuit area. The area-delay trade-off involved here is, in essence, the problem of transistor size optimization.

Three formal statements of the problem are stated below:

$$\text{Minimize } \textit{Area} \text{ subject to } \textit{Delay} \leq T_{\text{spec}} \tag{4.26}$$

$$\text{Minimize } \textit{Delay} \text{ subject to } \textit{Area} \leq A_{\text{spec}} \tag{4.27}$$

$$\text{Minimize } \textit{Area} \cdot [\textit{Delay}]^c \tag{4.28}$$

where c is a constant. Of all of these, the first form is perhaps the most useful practical form, since a designer's objective is typically to meet a timing constraint dictated by a system clock.

Depending on the type of analysis, the delay computation may be either deterministic or statistical [6]. In the former case, the problem reduces to a conventional nonlinear optimization problem. In the latter case, the optimization is statistical, though it can be transformed into a deterministic problem.

4.3.2 Delay Modeling

We examine the delay modeling procedure used in TILOS (Timed Logic Synthesizer) [7] at the transistor, gate, and circuit levels. Many transistor sizing algorithms use minor variations on this theme, although some use more exact models. The inherent trade-off here is between obtaining a provably optimal solution under inexact models, versus being uncertain about optimality under more correct models.

4.3.2.1 Transistor Level Model

A MOS transistor is modeled as a voltage-controlled switch with an on-resistance, R_{on}, between drain and source, and three grounded capacitances, C_{d}, C_{s}, and C_{g}, at the drain, source, and gate terminals, respectively, as shown earlier in Figure 4.6. The behaviors of the resistance and capacitances associated with a MOS transistor of channel width x are modeled as

$$R_{\text{on}} \propto 1/x \tag{4.29}$$

$$C_{\text{d}}, C_{\text{s}}, C_{\text{g}} \propto x. \tag{4.30}$$

4.3.2.2 Gate Level Model

At the gate level, delays are calculated in the following manner. For each transistor in a pull-up or pull-down network of a complex CMOS gate, the largest resistive path from the transistor to the gate output is computed, as well as the largest resistive path from the transistor to a supply rail. Thus, for each transistor, the network is transformed into an RC line, and its Elmore time constant [8] is computed and is taken to be the gate delay.

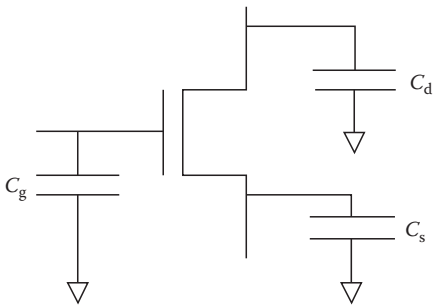


FIGURE 4.6 RC transistor model.

While finding the Elmore delay, the capacitances that lie between the switching transistor and the supply rail are assumed to be at the voltage level of the supply rail at the time of the switching transition, and do not contribute to the Elmore delay. For example, in Figure 4.7, the capacitance at node n_1 is ignored while computing the Elmore delay, the expression for which is

$$(R_1 + R_2)C_2 + (R_1 + R_2 + R_3)C_3. \tag{4.31}$$

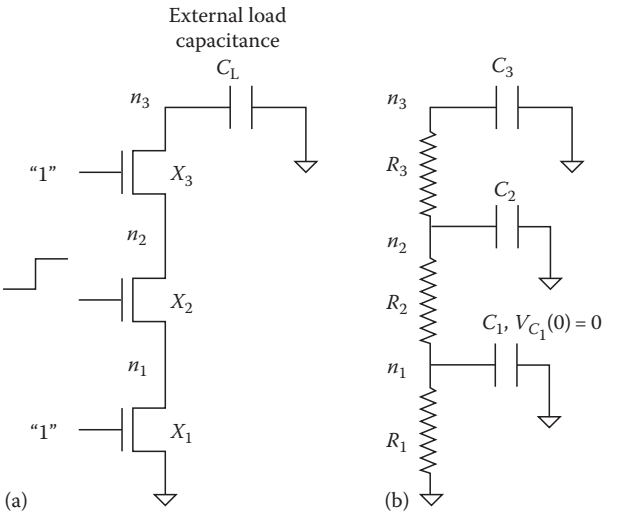


FIGURE 4.7 (a) Sample pull-down network and (b) its RC representation. (From Fishburn, J. and Dunlop, A., *Proceedings of the IEEE International Conference on Computer-Aided Design*, 1985, pp. 326–328. With permission.)

Each R_i is inversely proportional to the corresponding transistor size, x_i, and each C_i is some constant (for wire capacitance) plus a term proportional to the width of each transistor whose gate, drain, or source is connected to node i. Thus, Equation 4.31 can be rewritten as

$$(A/x_1 + A/x_2)(Bx_2 + Cx_3 + D) + (A/x_1 + A/x_2 + A/x_3)(Bx_3 + E),$$

which is a posynomial function (see Section 4.2.2) of x_1, x_2, and x_3. More accurate modeling techniques, using generalized posynomials, are described in Ref. [4].

4.3.2.3 Circuit Level Model

At the circuit level, the program evaluation and review technique (PERT), which will be described shortly, is used to find the circuit delay. The procedure is best illustrated by means of a simple example. Consider the circuit in Figure 4.8, where each box represents a gate, and the number within the box

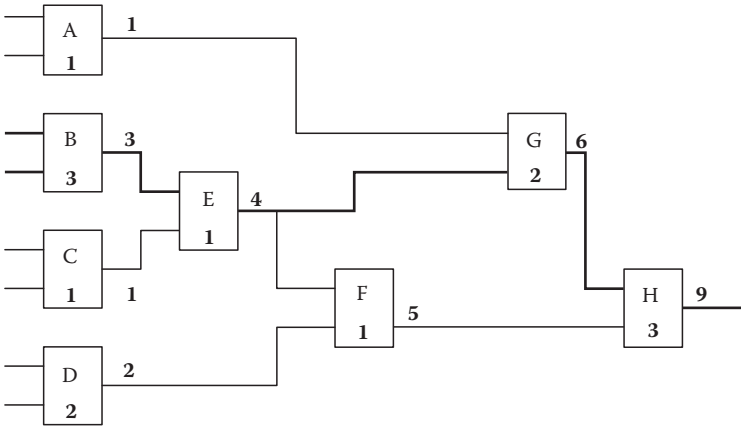


FIGURE 4.8 PERT.

represents its delay. We assume that the worst-case arrival time for a transition at any primary input, i.e., at the inputs to boxes A, B, C, and D, is 0.

A component is said to be ready for processing when the signal arrival time information is available for all of its inputs. Initially, since signal arrival times are known only at the primary inputs, only those components that are fed solely by primary inputs are ready for processing. These are placed in a queue and are scheduled for processing. In the iterative process, the component at the head of the queue is scheduled for processing. Each processing step consists of

- Finding the latest arriving input to the component, which triggers the output transition. This involves finding the maximum of all worst-case arrival times of inputs to the component.
- Adding the delay of the component to the latest arriving input time, to obtain the worst-case transition time at the output.
- Checking all of the components that the current component fans out to, to find out whether they are ready for processing. If so, the component is added to the tail of the queue.

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

Step 1. Initially, Queue = {A,B,C,D}.

Step 2. Schedule A; $Delay_A = 0 + 1 = 1$. Queue = {B,C,D}.

Step 3. Schedule B; $Delay_B = 0 + 3 = 3$. Queue = {C,D}.

Step 4. Schedule C; $Delay_C = 0 + 1 = 1$. Queue = {D,E}. (E is added to the queue.)

Step 5. Schedule D; $Delay_D = (0 + 2) = 2$. Queue = {E}.

Step 6. Schedule E; $Delay_E = (\max(3,1) + 1) = 4$. Queue = {F,G}.

Step 7. Schedule F; $Delay_F = (\max(4,2) + 1) = 5$. Queue = {G}.

Step 8. Schedule G; $Delay_G = (\max(4,1) + 2) = 6$. Queue = {H}.

Step 9. Schedule H; $Delay_H = (\max(6,5) + 3) = 9$. Queue = {}. The algorithm terminates.

The worst-case delays at the output of each component are shown in Figure 4.8. The critical path, defined as the path between an input and an output with the maximum delay, can now easily be found by successively tracing back, beginning from the primary output with the latest transition time, and walking back along the latest arriving fan-in of the current gate, until a primary input is reached. In the example, the critical path from the input to the output is B-E-G-H.

In the case of CMOS circuits, the rise and fall delay transitions are calculated separately. For inverting CMOS gates, the latest arriving input rise (fall) transition triggers off a fall (rise) transition at the output. This can easily be incorporated into the PERT method described above, by maintaining two numbers, t_r and t_f, for each gate, corresponding to the worst-case rise (high transition) and fall (low transition) delays from a primary input. To obtain the value of t_f at an output, the largest value of t_r at an input node is added to the worst-case fall transition time of the component; the computation of t_r is analogous. For noninverting gates, t_r and t_f are obtained by adding the rise (fall) transition time to the worst-case input rise (fall) transition time.

Since each gate delay is a posynomial, and the circuit delay found by the PERT is a sum of gate delays, the circuit delay is also a posynomial function of the transistor sizes. In general, the path delay can be written as

$$\sum_{i,j=1}^{n} a_{ij} \frac{x_i}{x_j} + \sum_{i=1}^{n} \frac{b_i}{x_i} + K. \tag{4.32}$$

In contrast to deterministic timing, statistical timing analysis techniques model gate delays as statistical functions of the underlying process parameter variations. Instead of propagating delays as fixed numbers, they propagate the probability density functions (PDFs) of delays. Techniques based on Gaussian or non-Gaussian delays, under linear or nonlinear models, with and without correlation, have been presented in the literature [9–12].

4.3.3 Area Model

The exact area of a circuit cannot easily be represented as a function of transistor sizes. This is unfortunate, since a closed functional form facilitates the application of optimization techniques. As an approximation, the following formula is used by many transistor sizing algorithms, to estimate the active circuit area:

$$Area = \sum_{i=1}^{n} x_i \tag{4.33}$$

where

x_i is the size of the ith transistor
n is the number of transistors in the circuit

In other words, the area is approximated as the sum of the sizes of transistors in the circuit that, from the definition Equation 4.8, is clearly a posynomial function of the x_i's.

4.3.4 Sensitivity-Based TILOS Algorithm

4.3.4.1 Steps in the Algorithm

The algorithm that was implemented in TILOS [7] was the first to recognize the fact that the area and delay can be represented as posynomial functions of the transistor sizes. The algorithm proceeds as follows.

An initial solution is assumed where all transistors are at the minimum allowable size. In each iteration, a static timing analysis is performed on the circuit, as explained in Section 4.3.2, to determine the critical path for the circuit.

Let N be the primary output node on the critical path. The algorithm then walks backward along the critical path, starting from N. Whenever an output node of a gate, $Gate_i$, is visited, TILOS examines the largest resistive path between V_{DD} (ground) and the output node if $Gate_i$'s t_r (t_f) causes the timing failure at N. This includes

- Critical transistor, i.e., the transistor whose gate terminal is on the critical path. In Figure 4.7, X_2 is the critical transistor.
- Supporting transistors, i.e., transistors along the largest resistive path from the critical transistor to the power supply (V_{DD} or ground). In Figure 4.7, X_1 is a supporting transistor.
- Blocking transistors, i.e., transistors along the highest resistance path from the critical transistor to the logic gate output. In Figure 4.7, X_3 is a blocking transistor.

TILOS finds the sensitivity, which is the reduction in circuit delay per increment of transistor size, for each critical, blocking, and supporting transistors. (The procedure of sensitivity computation is treated in greater detail shortly.) The size of the transistor with the greatest sensitivity is increased by multiplying it by a constant, BUMPSIZE, a user-settable parameter that defaults to 1.5. The above process is repeated until all constraints are met, implying that a solution is found, or the minimum delay state has been

passed, and any increase in transistor sizes would make it slower instead of faster, in which case TILOS cannot find a solution.

Note that since in each iteration, exactly one transistor size is changed, the timing analysis method can employ incremental simulation techniques to update delay information from the previous iteration. This substantially reduces the amount of time spent in critical path detection.

4.3.4.2 Sensitivity Computation

Figure 4.9 illustrates a configuration in which the critical path extends back along the gate of the upper transistor, which is the critical transistor. The sensitivity for this transistor is calculated as follows: set all transistor sizes, except x, to the size of the critical transistor. R is the total resistance of an RC chain driving the gate and C is the total capacitance of an RC chain being driven by the configuration. The total delay, $D(x)$, of the critical path is

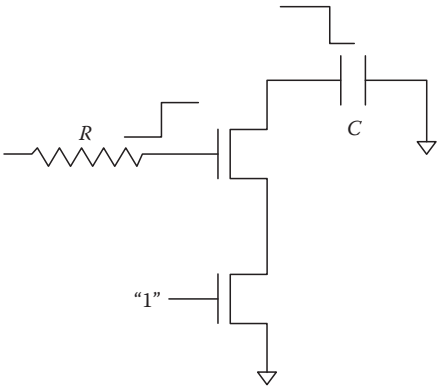


FIGURE 4.9 Sensitivity calculation in TILOS. (From Fishburn, J. and Dunlop, A., *Proceedings of the IEEE International Conference on Computer-Aided Design*, 1985, pp. 326–328. With permission.)

$$D(x) = K + RC_{\mathrm{u}}x + \frac{R_{\mathrm{u}}C}{x} \tag{4.34}$$

where

R_{u} and C_{u} are the resistance and capacitance of a unit-sized transistor, respectively

K is a constant that depends on the resistance of the bottom transistor, the capacitance in the driving RC chain, and the resistance in the driven RC chain

The sensitivity, $D'(x)$, is then

$$D'(x) = RC_{\mathrm{u}} - \frac{R_{\mathrm{u}}C}{x^2}. \tag{4.35}$$

The sensitivities of supporting transistors and blocking transistors can similarly be calculated.

Note that increasing the size of a transistor with negative sensitivity only means that the delay along the current critical path can be reduced by changing the size of this transistor, and does not necessarily mean that the circuit delay can be reduced; the circuit delay is the maximum of all path delays in the circuit, and a change in the size of this transistor could increase the delay along some other path, making a new path critical. This is the rationale behind increasing the size of the most sensitive transistor by only a small factor.

From an optimization viewpoint, the procedure of bumping up the size of the most sensitive transistor could be looked upon in the following way. Let the ith transistor (out of n total transistors) be the one with the maximum sensitivity. Define $\mathbf{e_i} \in \mathbf{R}^n$ as $(\mathbf{e_i})_j = 0$ if $i \neq j$ and $(\mathbf{e_i})_j = 1$ if $i = j$. In each iteration, the TILOS optimization procedure works in the n-dimensional space of the transistor sizes, chooses $\mathbf{e_i}$ as the search direction, and attempts to find the solution to the problem by taking a small step along that direction.

4.3.5 Transistor Sizing Using the Method of Feasible Directions

Shyu et al. proposed a two-stage optimization approach to solve the transistor sizing problem. The delay estimation algorithm is identical to that used in TILOS.

The algorithm can be summarized in the following pseudocode:

```
Use TILOS to size the entire circuit;
While (TRUE) {
    Select G_1, . . . , G_k, the k most critical paths,
    and X = {x_i}, the set of design parameters
    Solve the optimization problem
```

$$\begin{aligned} &\text{minimize} && \sum_{x_i \in X} x_i \\ &\text{such that} && G_i(X) \leq T \ \forall \ i \ = \ 1, \ \ldots, k \\ &\text{and} && x_i \ \geq \ \text{minsize} \ \forall \ x_i \ \in \ X. \end{aligned}$$

```
    If all constraints are satisfied, exit   }
```

In the first stage, the TILOS heuristic described in Section 4.3.4 is used to generate an initial solution. The heuristic finds a solution that satisfies the constraints, and only the sized-up transistors are used as design parameters. Although TILOS is not guaranteed to find an optimal solution, it can serve as an initial guess solution for an iterative technique.

In the second stage of the optimization process, the problem is converted into a mathematical optimization problem, and is solved by a method of feasible directions (MFD) algorithm described in Section 4.2.3, using the feasible solution generated in the first stage as an initial guess. To reduce the computation, a sequence of problems with a smaller number of design parameters is solved.

At first, the transistors on the worst-delay paths (usually more than one) are selected as design parameters. If, with the selected transistors, the optimizer fails to meet the delay constraints, and some new paths become the worst-delay paths, the algorithm augments the design parameters with the transistors on those paths, and restarts the process. However, while this procedure reduces the run time of the algorithm, one faces the risk of finding a suboptimal solution since only a subset of the design parameters is used in each step.

The MFD optimization method proceeds by finding a search direction **d**, a vector in the n-dimensional space of the design parameters, based on the gradients of the cost function and some of the constraint functions. Once the search direction has been computed, a step along this direction is taken, so that the decrease in the cost and constraint functions is large enough. The computation stops when the length of this step is sufficiently small.

Since this algorithm has the feature that once the feasible region (the set of transistor sizes where all delay constraints are satisfied) is entered, all subsequent improvements will remain feasible, and the algorithm can be terminated at any time with a feasible solution.

4.3.5.1 Practical Implementational Aspects

4.3.5.1.1 Generalized Gradient

For convergence, the MFD requires that the objective and constraint functions be continuously differentiable. However, since the circuit delay is defined as the maximum of all path delays, the delay constraint functions are usually not differentiable. To illustrate that the maximum of two continuously differentiable functions, $g_1(x)$ and $g_2(x)$, need not be differentiable, consider the example in Figure 4.10. The maximum function, shown by the bold lines, is nondifferentiable at x_0.

To cope with the nondifferentiability of the constraint functions, a modification of the MFD is used, which employs the concept of the generalized gradient. The idea is to use a convex combination of the gradients of the active, or nearly active, constraints near a discontinuity.

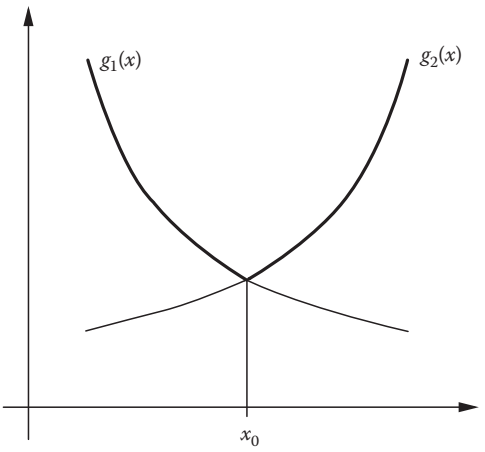


FIGURE 4.10 Nondifferentiability of the max function.

4.3.5.1.2 Scaling

It is important to scale the gradients of the cost and path delay functions since the use of their unscaled values may produce poor descent search directions due to the large difference in magnitude of the gradients of the constraint and objective functions. When a gradient has a magnitude that is much smaller than other gradients, it dominates the search direction. In such a case, the descent direction is unjustly biased away from the other constraints or the cost function.

4.3.6 Lagrangian Multiplier Approaches

As can be seen from the approaches studied so far, the problem of transistor sizing can be formulated as a constrained nonlinear programming problem. Hence, the method of Lagrangian multipliers, described in Section 4.2.3, is applicable.

An early approach by Marple [13] begins with a prespecified layout, and performs the optimization using an area model for that layout. While such an approach has the disadvantage that it may not result in the minimal area over all layouts, it still maintains the feature that the area and delay constraints are posynomials. Apart from the delay constraints, there also exist some area constraints, modeled by constraint graphs that are commonly used in layout compaction. These constraints maintain the minimum spacing between objects in the final layout, as specified by design rules.

The delay of the circuit is modeled by a delay graph, D (V, E) where V is the set of nodes (gates) in D, and E is the set of arcs (connections among gates) in D. This is the same graph on which the PERT analysis is to be carried out. Let m_i represent the worst-case delay at the output of gate i, from the primary inputs. Then for each gate, the delay constraint is expressed as

$$m_i + d_j \leq m_j, \tag{4.36}$$

where gate $i \in$ fan-in (gate j), and d_j is the delay of gate j. Thus, the number of delay constraints is reduced from a number that could, in the worst case, be exponential in $|V|$, to one that is linear in $|E|$, using $|V|$ additional variables.

More recent work [14] recognizes that many of the Lagrange multipliers can be removed to build a relaxed problem, and uses Lagrangian relaxation on the dual to obtain a very fast optimal solution.

4.3.7 Two-Step Optimization

Since the number of variables in the transistor sizing problem, which equals the number of transistors in a combinational segment, is typically too large for most optimization algorithms to handle efficiently, many algorithms choose a simpler route by performing the optimization in two steps. The most successful of these methods is given in Ref. [15], which goes alternately through a D-phase and a W-phase; one of these creates delay budgets for each gate by solving a flow problem, and the other assigns transistor sizes based on these budgets.

4.3.8 Convex Programming-Based Approach

The chief shortcoming of most of the approaches above is that the simplifying assumptions that are made to make the optimization problem more tractable may lead to a suboptimal solution. The algorithm in iCONTRAST solves the underlying optimization problem exactly.

The objective of the algorithm is to solve the transistor sizing problem in Equation 4.28, where both the area and the delay are posynomial functions of the vector **x** of transistor sizes. The procedure described below may easily be extended to solve the formulations in Equations 4.29 and 4.30 as well; however, these formulations are not as useful to the designer. The variable transformation, $(x_i) = (e^{z_i})$, maps the problem in Equation 4.28 to

$$\begin{aligned} &\text{minimize} \quad Area(\mathbf{z}) = \sum_{i=1}^{n} e^{z_i} \\ &\text{subject to} \quad D(\mathbf{z}) \leq T_{\text{spec}}. \end{aligned} \tag{4.37}$$

The delay of a circuit is defined to be the maximum of the delays of all paths in the circuit. Hence, it can be formulated as the maximum of posynomial functions of **x**. This is mapped by the above transformation onto a function $D(\mathbf{z})$ that is a maximum of convex functions; a maximum of convex functions is also a convex function. The area function is also a posynomial in **x**, and is transformed into a convex function by the same mapping. Therefore, the optimization problem defined in Equation 4.28 is mapped to a convex programming problem, i.e., a problem of minimizing a convex function over a convex constraint set. Due to the unimodal property of convex functions over convex sets, any local minimum of Equation 4.28 is also a global minimum.

Vaidya's convex programming method described in Section 4.2.3 is then used to find the unique global minimum of the optimization problem.

4.3.8.1 Gradient Calculations

In an iteration of Vaidya's convex programming algorithm, when the center $\mathbf{z}_c$ of a polytope lies within the feasible region S, the gradient of the area function is required to generate the new hyperplane passing through the center. The gradient of the area function (Equation 4.37) is given by

$$\nabla Area(\mathbf{z}) = [e^{z_1}, e^{z_2}, \ldots, e^{z_n}]. \tag{4.38}$$

In the case when the center $\mathbf{z}_c$ lies outside the feasible region S, the gradient of the critical path delay function $D_{\text{critpath}}(\mathbf{z}_c)$ is required to generate the new hyperplane that is to be added. Note that transistors in the circuit can contribute to the kth component of the gradient of the delay function in either of two ways:

1. If the kth transistor is a critical, supporting, or blocking transistor (as defined in Section 4.3.4)
2. If the kth transistor is a capacitive load for some critical transistor

Transistors that satisfy neither of these two requirements have no contribution to the gradient of the delay function.

4.3.9 Statistical Optimization

In the presence of statistical variations, it is important to perform timing optimization based on statistical static timing analysis. An obvious way to increase the timing yield of a digital circuit is to pad the specifications to make the circuit robust to variations, i.e., choose a delay specification of the circuit that is tighter than the required delay. This new specification must be appropriately selected to avoid large area or power overheads due to excessively conservative padding.

Early work in Ref. [9] proposes formulation of statistical objective and timing constraints, and solves the resulting nonlinear optimization formulation, but this is highly computational. Other approaches have extended TILOS heuristic to optimize $\mu + 3\sigma$, where μ is the mean of the delay PDF and σ its standard deviation, but while these methods work reasonably in practice, they have no theoretical backing.

More formal optimization approaches have also been used. The work in Ref. [16] presents approaches to optimize the statistical power of the circuit, subject to timing yield constraints under convex formulation of the problem as a second-order conic program. The work in Ref. [17] proposes a gate sizing technique based on robust optimization theory: robust constraints are added to the original constraints set by modeling the intrachip random process parameter variations as Gaussian variables, contained in a constant probability density uncertainty ellipsoid, centered at the nominal values.

4.4 Analog Design Centering Problem

4.4.1 Problem Description

While manufacturing an analog circuit, it is inevitable that process variations will cause design parameters, such as component values, to waver from their nominal values. As a result, the manufactured analog circuit may no longer meet some behavioral specifications, such as requirements on the delay, gain, and bandwidth, that it has been designed to satisfy. The procedure of design centering attempts to select the nominal values of design parameters to move the circuit to the center of the feasible region.

The values of n design parameters may be ordered as an n-tuple that represents a point in $\mathbf{R}^n$. A point is feasible if the corresponding values for the design parameters satisfy the behavioral specifications on the circuit. The feasible region (or the region of acceptability), $R_f \subset \mathbf{R}^n$, is defined as the set of all design points for which the circuit satisfies all behavioral specifications.

The random variations in the values of the design parameters are modeled by a PDF, $\Phi(\mathbf{z})$: $\mathbf{R}^n \rightarrow [0, 1]$, with a mean corresponding to the nominal value of the design parameters. The yield of the circuit, Y, as a function of the mean, $\mathbf{x}$, is given by

$$Y(\mathbf{x}) = \int_{R_f} \Phi_{\mathbf{x}}(\mathbf{z}) d\mathbf{z}. \tag{4.39}$$

The design center is the point $\mathbf{x}$ at which the yield, $Y(\mathbf{x})$, is maximized. There have traditionally been two approaches to solving this problem: one based on geometrical methods and another based on statistical sampling. In addition, several methods that hybridize these approaches also exist. We now provide an exposition of geometrical approaches to the optimization problem of design centering.

A common assumption made by geometrical design centering algorithms is that R_f is a convex-bounded body. Geometrical algorithms recognize that the evaluation of the integral (Equation 4.39) is computationally difficult, and generally proceed as follows: the feasible region in the space of design parameters, i.e., the region where the behavioral specifications are satisfied, is approximated by a known geometrical body, such as a polytope or an ellipsoid. The center of this body is then approximated, and is taken to be the design center.

4.4.2 Simplicial Approximation Method

4.4.2.1 Outline of the Method

The simplicial approximation method [18] is a method for approximating a feasible region by a polytope and finding its center. This method proceeds in the following steps:

1. Determine a set of $m \geq n + 1$ points on the boundary of R_f.
2. Find the convex hull (see Section 4.2.2) of these points and use this polyhedron as the initial approximation to R_f. In the two-dimensional example in Figure 4.11a, the points 1, 2, and 3 are chosen in Step 1, and their convex hull is the triangle with vertices 1, 2, and 3. Set $k = 0$.

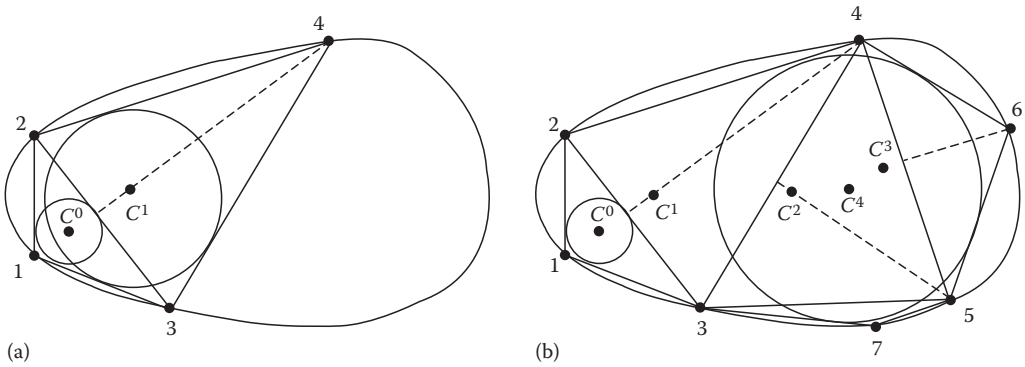


FIGURE 4.11 Simplicial approximation method. (From Director, S.W. and Hachtel, G.D., *IEEE Trans. Circuits Syst.*, CAS-24, 363, 1977. With permission.)

3. Inscribe the largest n-dimensional hypersphere in this approximating polyhedron and take its center as the first estimate of the design center. This process involves the solution of a linear program. In Figure 4.11a, this is the hypersphere C^0.
4. Find the midpoint of the largest face of the polyhedron, i.e., the face in which the largest $n-1$-dimensional hypersphere can be inscribed. In Figure 4.11a, the largest face is 2–3, the face in which the largest one-dimensional hypersphere can be inscribed.
5. Find a new boundary point on R_f by searching along the outward normal of the largest face found in Step 4 extending from the midpoint of this face. This is carried out by performing a line search. In Figure 4.11a, point 4 is thus identified.
6. Inflate the polyhedron by forming the convex hull of all previous points, plus the new point generated in Step 5. This corresponds to the quadrilateral 1, 2, 3, and 4 in Figure 4.11a.
7. Find the center of the largest hypersphere inscribed in the new polyhedron found in Step 6. This involves the solution of a linear program. Set $k = k + 1$, and go to Step 4. In Figure 4.11a, this is the circle C^1.

Further iterations are shown in Figure 4.11b. The process is terminated when the sequence of radii of the inscribed hypersphere converges.

4.4.2.2 Inscribing the Largest Hypersphere in a Polytope

Given a polytope specified by Equation 4.3, if the $\mathbf{a}_i$'s are chosen to be unit vectors, then the distance of a point $\mathbf{x}$ from each hyperplane of the polytope is given by $r = \mathbf{a}_i^{\mathrm{T}}\mathbf{x} - b_i$.

The center $\mathbf{x}$ and radius r of the largest hypersphere that can be inscribed within the polytope $\mathcal{P}$ are then given by the solution of the following linear program:

$$\begin{aligned} &\text{minimize} && r \\ &\text{subject to} && \mathbf{a}_i^{\mathrm{T}}\mathbf{x} - r \geq b_i \end{aligned} \tag{4.40}$$

Since the number of unknowns of this linear program is typically less than the number of constraints, it is more desirable to solve its dual [1]. A similar technique can be used to inscribe the largest hypersphere in a face of the polytope.

This method has also been generalized for the inscription of maximal norm bodies, to handle joint PDFs with (nearly) convex level contours.

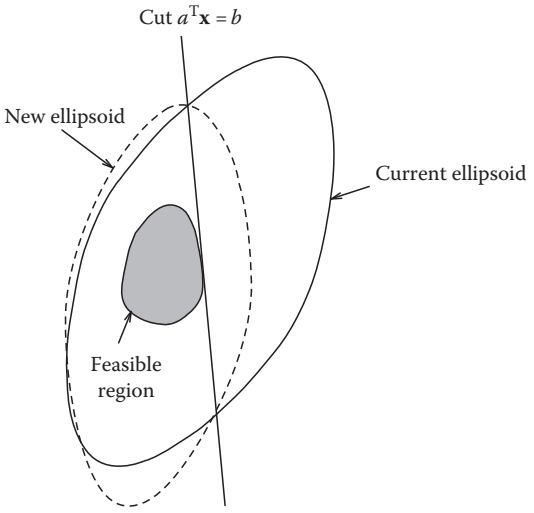


FIGURE 4.12 Ellipsoidal method. (From Abdel-Malek, H.L. and Hassan, A.-K.S.O., *IEEE Trans. Comput. Aided Des.*, 10, 1006, 1991. With permission.)

4.4.2.3 Elongated Feasible Regions

If the design centering procedure outlined earlier is applied to a rectangular feasible region, the best possible results may not be obtained by inscribing a hypersphere. For elongated feasible regions, it is more appropriate to determine the design center by inscribing an ellipsoid rather than a hypersphere. Simplicial approximation handles this problem by scaling the axes so that the lower and upper bounds for each parameter differ by the same magnitude, and one may inscribe the largest ellipsoid by inscribing the largest hypersphere in a transformed polytope. This procedure succeeds in factoring in reasonably the fact that feasible regions may be elongated; however, it considers only a limited set of ellipsoids, which have their axes aligned with the coordinate axis, as candidates for inscription within the polytope.

4.4.3 Ellipsoidal Method

This method [19] is based on principles similar to those used by the Shor–Khachiyan ellipsoidal algorithm for linear programming. This algorithm attempts to approximate the feasible region by an ellipsoid, and takes the center of the approximating ellipsoid as the design center. It proceeds by generating a sequence of ellipsoids, each smaller than the last, until the procedure converges. Like other methods, this procedure assumes that an initial feasible point is provided by the designer. The steps involved in the procedure are as follows (Figure 4.12):

1. Begin with an ellipsoid, E_0, that is large enough to contain the desired solution. Set $j = 0$.
2. From the center of the current ellipsoid, choose a search direction, and perform a binary search to identify a boundary point along that direction. One convenient set of search directions are the parameter directions, searching along the ith, $i = 1, 2, \ldots, n$ in a cycle, and repeating the cycle, provided the current ellipsoid center is feasible. If not, a linear search is conducted along a line from the current center to the given feasible point.
3. A supporting hyperplane [1] at the boundary point can be used to generate a smaller ellipsoid, E_{j+1}, that is guaranteed to contain the feasible region R_f, if R_f is convex. The equation of E_{j+1} is provided by an update procedure.
4. Increment j, and go to Step 1 unless the convergence criterion is met. The convergence criterion is triggered when the volume is reduced by a given factor, ε. Upon convergence, the center of the ellipsoid is taken to be the design center.

4.4.4 Convexity-Based Approaches

4.4.4.1 Introduction

This section describes the approach in Ref. [20]. Here, the feasible region is first approximated by a polytope in the first phase. Next, two geometrical approaches to find the design center are proposed. In the first, the properties of polytopes are utilized to inscribe the largest ellipsoid within the approximating polytope. The second method proceeds by formulating the design centering problem as a convex

programming problem, assuming that the variations in the design parameters are modeled by Gaussian probability distributions, and uses Vaidya's convex programming algorithm described in Section 4.2.3 to find the solution.

4.4.4.2 Feasible Region Approximation

The feasible region, $R_f \subset \mathbf{R}^n$, is approximated by a polytope given by Equation 4.3 in this step. The algorithm begins with an initial feasible point, $\mathbf{z}_0 \in R_f$. An n-dimensional box, namely, $\{\mathbf{z} \in \mathbf{R}^n \mid z_{\min} \leq z_i \leq z_{\max}\}$, containing R_f is chosen as the initial polytope $\mathcal{P}_0$. In each iteration, n orthogonal search directions, $\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_n$ are chosen (possible search directions include the n coordinate directions). A binary search is conducted from $\mathbf{z}_0$ to identify a boundary point $\mathbf{z}_{\mathbf{b}i}$ of R_f, for each direction $\mathbf{d}_i$. If $\mathbf{z}_{\mathbf{b}i}$ is relatively deep in the interior of $\mathcal{P}$, then the tangent plane to R_f at $\mathbf{z}_{\mathbf{b}i}$ is added to the set of constraining hyperplanes in Equation 4.3. A similar procedure is carried out along the direction $-\mathbf{d}_i$. Once all of the hyperplanes have been generated, the approximate center of the new polytope is calculated. Then $\mathbf{z}_0$ is reset to be this center, and the above process is repeated.

Therefore, unlike simplicial approximation that tries to expand the polytope outward, this method starts with a large polytope and attempts to add constraints to shrink it inward. The result of polytope approximation on an ellipsoidal feasible region is illustrated in Figure 4.13.

4.4.4.3 Algorithm I: Inscribing the Largest Hessian Ellipsoid

For a polytope given by Equation 4.3, the log-barrier function is defined as

$$F(\mathbf{z}) = -\sum_{i=1}^{m} \log_e(\mathbf{a_i}^T\mathbf{z} - b_i). \tag{4.41}$$

The Hessian ellipsoid centered at a point $\mathbf{x}$ in the polytope $\mathcal{P}$ is defined as the ellipsoid E($\mathbf{x}$,$\mathcal{H}(\mathbf{x})$,1) (see Equation 4.2), where $\mathcal{H}(\mathbf{x}_c)$ is the Hessian [1] of the log-barrier function above, and is given by

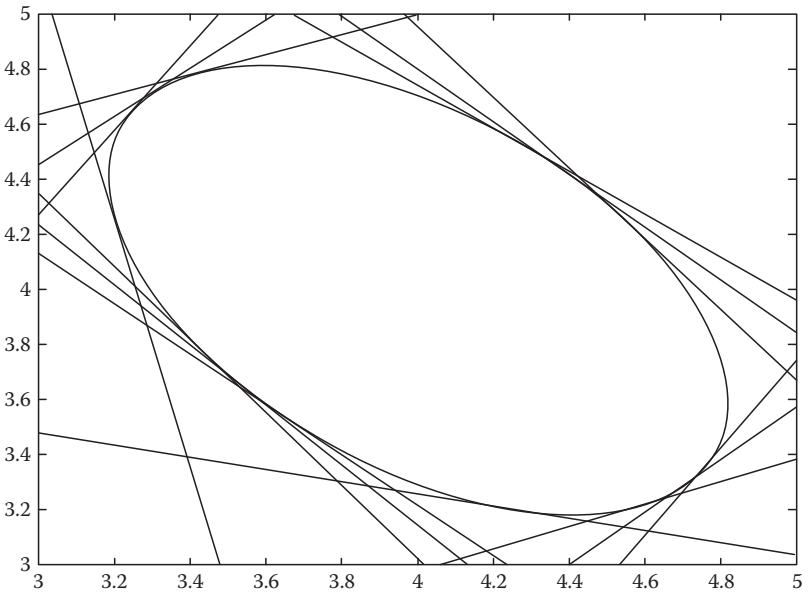


FIGURE 4.13 Polytope approximation for the convexity-based methods. (From Saptnekar, S.S., Vaidya, P.M. and Kang, S.M., *IEEE Trans. Comput. Aided Des.*, 13, 1536, 1994. With permission.)

$$\mathcal{H}(\mathbf{z}) = \nabla^2 F(\mathbf{z}) = \sum_{i=1}^{m} \frac{\mathbf{a_i}\mathbf{a_i}^{\mathrm{T}}}{(\mathbf{a_i}^{\mathrm{T}}\mathbf{z} - b_i)^2}. \tag{4.42}$$

This is known to be a good approximation to the polytope locally around $\mathbf{x}$.

Hence, the goal is to find the largest ellipsoid in the class $\mathbf{E}(\mathbf{x}, \mathcal{H}(\mathbf{x}),\mathrm{r})$ that can be inscribed in the polytope, and its center $\mathbf{x}_c$. The point $\mathbf{x}_c$ will be taken to be the computed design center. An iterative process is used to find this ellipsoid. In each iteration, the Hessian at the current point $\mathbf{x}_k$ is calculated, and the largest ellipsoid $\mathbf{E}(\mathbf{x}, \mathcal{H}(\mathbf{x}_k),\mathrm{r})$ is inscribed in the polytope. The inscription of this ellipsoid is equivalent to inscribing a hypersphere in a transformed polytope. The process of inscribing a hypersphere in a polytope is explained in Section 4.4.2.

4.4.4.4 Algorithm II: Convex Programming Approach

When the PDFs that represent variations in the design parameters are Gaussian in nature, the design centering problem can be posed as a convex programming problem.

The joint Gaussian PDF of n independent random variables $\mathbf{z} = (z_1, \ldots, z_n)$ with mean $\mathbf{x} = (x_1, \ldots, x_n)$ and variance $\sigma = (\sigma_1, \ldots, \sigma_n)$ is given by

$$\Phi_{\mathbf{x}}(\mathbf{z}) = \frac{1}{(2\pi)^{n/2}\sigma_1\sigma_2\cdots\sigma_n} \exp\left[\sum_{i=0}^{i=n} -\frac{(z_i - x_i)^2}{2\sigma_i^2}\right] \tag{4.43}$$

This is known to be a log-concave function of $\mathbf{x}$ and $\mathbf{z}$. Also, note that arbitrary covariance matrices can be handled, since a symmetric matrix may be converted into a diagonal form by a simple linear (orthogonal) transformation. The design centering problem is now formulated as

$$\begin{aligned} &\text{maximize} \quad \mathrm{Y}(\mathbf{x}) = \int_{\mathcal{P}} \Phi_{\mathbf{x}}(\mathbf{z})\mathrm{d}\mathbf{z} \\ &\text{such that} \quad \mathbf{x} \in \mathcal{P}. \end{aligned} \tag{4.44}$$

where $\mathcal{P}$ is the polytope approximation to the feasible region R_f. It is a known fact that the integral of a log-concave function over a convex region is also a log-concave function. Thus, the yield function $\mathrm{Y}(\mathbf{x})$ is log-concave, and the above problem reduces to a problem of maximizing a log-concave function over a convex set. Hence, this can be transformed into a convex programming problem. The convex programming algorithm in Section 4.2.3 is then applied to solve the optimization problem.

4.4.5 Concluding Remarks

The above list of algorithms is by no means exhaustive, but provides a general flavor for how optimization methods are used in geometrical design centering. The reader is referred to [21–24] for further information about statistical design. In conclusion, it is appropriate to list a few drawbacks associated with geometrical methods:

Limitations of the approximating bodies: In the case of ellipsoidal approximation, certain convex bodies cannot be approximated accurately, because an ellipsoid is symmetric about any hyperplane passing through its center, and is inherently incapable of producing a good approximation to a body that has a less symmetric structure. A polytope can provide a better approximation to a convex body than an ellipsoid since any convex body can be thought of as a polytope with an infinite number of faces. However, unlike the ellipsoidal case, calculating the exact center of a polytope is computationally difficult and one must resort to approximations.

Statistical effects are ignored: Methods such as simplicial approximation, ellipsoidal approximation, and Algorithm I of the convexity-based methods essentially approximate the feasible region by means of an ellipsoid, and take the center of that ellipsoid to be the design center, regardless of the probability distributions that define variations in the design parameters. However, the design center could be highly dependent on the exact probability distributions of the variables, and would change according to these distributions.

Nonconvexities: Real feasible regions are seldom convex. While in many cases, they are nearly convex, there are documented cases where the feasible region is not very well behaved. In a large number of cases of good designs, since the joint PDF of the statistical variables decays quite rapidly from the design center, a convex approximation does not adversely affect the result. However, if the nominal design has a very poor yield, a convex approximation will prove to be inadequate.

The curse of dimensionality: Geometrical methods suffer from the so-called curse of dimensionality, whereby the computational complexity of the algorithm increases greatly with the number of variables. However, as noted in Ref. [25], for local circuit blocks within a die, performance variations in digital MOS circuits depend on only four independent statistical variables. Moreover, for this class of circuits, the circuit performance can be modeled with reasonable accuracy by linear functions. For such circuits, the deterministic (i.e., nonstatistical) algorithm in Ref. [25] is of manageable complexity.

References

1. D. G. Luenberger, *Linear and Nonlinear Programming*, 2nd edn. Reading, MA: Addison-Wesley, 1984.
2. R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, A survey of optimization techniques for integrated-circuit design, *Proceedings of the IEEE*, 69: 1334–1362, October 1981.
3. S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge, U.K.: Cambridge University Press, 2004.
4. K. Kasamsetty, M. Ketkar, and S. S. Sapatnekar, A new class of convex functions for delay modeling and their application to the transistor sizing problem, *IEEE Transactions on Computer-Aided Design*, 19: 779–788, July 2000.
5. S. S. Sapatnekar, V. B. Rao, P. M. Vaidya, and S. M. Kang, An exact solution to the transistor sizing problem for CMOS circuits using convex optimization, *IEEE Transactions on Computer-Aided Design*, 12: 1621–1634, November 1993.
6. S. S. Sapatnekar, *Timing*, Boston, MA: Springer, 2004.
7. J. Fishburn and A. Dunlop, TILOS: A posynomial programming approach to transistor sizing, in *Proceedings of the IEEE International Conference on Computer-Aided Design*, Santa Clara, CA, pp. 326–328, 1985.
8. J. Rubenstein, P. Penfield, and M. A. Horowitz, Signal delay in RC tree networks, *IEEE Transactions on Computer-Aided Design*, CAD-2: 202–211, July 1983.
9. E. Jacobs and M. Berkelaar, Gate sizing using a statistical delay model, in *Design, Automation, and Test in Europe*, Paris, France, pp. 283–290, 2000.
10. H. Chang and S. S. Sapatnekar, Statistical timing analysis considering spatial correlations using a single PERT-like traversal, in *Proceedings of the IEEE International Conference on Computer-Aided Design*, San Jose, CA, pp. 621–625, 2003.
11. Y. Zhan, A. J. Strojwas, X. Li, L. T. Pileggi, D. Newmark, and M. Sharma, Correlation-aware statistical timing analysis with non-Gaussian delay distributions, in *Proceedings of the ACM/IEEE Design Automation Conference*, Anaheim, CA, pp. 77–82, 2005.
12. J. Singh and S. S. Sapatnekar, Statistical timing analysis with correlated non-Gaussian parameters using independent component analysis, in *Proceedings of the ACM/IEEE Design Automation Conference*, San Francisco, CA, pp. 155–160, 2006.

13. D. Marple, Performance optimization of digital VLSI circuits, Technical Report No. CSL-TR-86-308, Stanford University, Stansford, CA, October 1986.
14. C.-P. Chen, C. C.-N. Chu, and M. D.-F. Wong, Fast and exact simultaneous gate and wire sizing by Lagrangian relaxation, *IEEE Transactions on Computer-Aided Design*, 18: 1014–1025, July 1999.
15. V. Sundararajan, S. S. Sapatnekar, and K. K. Parhi, Fast and exact transistor sizing based on iterative relaxation, *IEEE Transactions on Computer-Aided Design*, 21: 568–581, May 2002.
16. M. Mani, A. Devgan, and M. Orshansky, An efficient algorithm for statistical power under timing yield constraints, in *Proceedings of the ACM/IEEE Design Automation Conference*, Anaheim, CA, pp. 309–314, 2005.
17. J. Singh, V. Nookala, Z.-Q. Luo, and S. Sapatnekar, Robust gate sizing by geometric programming, in *Proceedings of the ACM/IEEE Design Automation Conference*, Anaheim, CA, pp. 315–320, 2005.
18. S. W. Director and G. D. Hachtel, The simplicial approximation approach to design centering, *IEEE Transactions on Circuits and Systems*, CAS-24 (7): 363–372, 1977.
19. H. L. Abdel-Malek and A.-K. S. O. Hassan, The ellipsoidal technique for design centering and region approximation, *IEEE Transactions on Computer-Aided Design*, 10: 1006–1014, August 1991.
20. S. S. Sapatnekar, P. M. Vaidya, and S. M. Kang, Convexity-based algorithms for design centering, *IEEE Transactions on Computer-Aided Design*, 13: 1536–1549, December 1994.
21. S. W. Director, P. Feldmann, and K. Krishna, Statistical integrated circuit design, *IEEE Journal of Solid-State Circuits*, 28: 193–202, March 1993.
22. M. D. Meehan and J. Purviance, *Yield and Reliability in Microwave Circuit and System Design*, Boston, MA: Artech House, 1993.
23. S. W. Director, W. Maly, and A. J. Strojwas, *VLSI Design for Manufacturing: Yield Enhancement*, Boston, MA: Kluwer Academic, 1990.
24. R. Spence and R. S. Soin, *Tolerance Design of Integrated Circuits*, Reading, MA: Addison-Wesley, 1988.
25. D. E. Hocevar, P. F. Cox, and P. Yang, Parametric yield optimization for MOS circuit blocks, *IEEE Transactions on Computer-Aided Design*, 7: 645–658, June 1988.

5

Statistical Design Optimization

5.1 Introduction 5-1
5.2 Problems and Methodologies of Statistical Circuit Design 5-1
5.3 Underlying Concepts and Techniques 5-2
Circuit Variables, Parameters, and Performances • Statistical Modeling of Circuit (Simulator) Variables • Acceptability Regions • Methods of Acceptability Region Approximation • Manufacturing (Parametric) Yield
5.4 Statistical Methods of Yield Optimization 5-12
Large-Sample vs. Small-Sample Methods • Methods Using Standard Deterministic Optimization Algorithms • Large-Sample Heuristic Methods for Discrete Circuits • Large-Sample, Derivative-Based Methods for Discrete Circuits • Large-Sample, Derivative-Based Method for Integrated Circuits • Small-Sample Stochastic Approximation-Based Methods • Small-Sample Stochastic Approximation Methods for Integrated Circuits • Case Study: Process Optimization for Manufacturing Yield Enhancement • Generalized Formulation of Yield, Variability, and Taguchi Circuit Optimization Problems
5.5 Conclusion 5-30
References 5-31

Maciej A. Styblinski
Texas A&M University

5.1 Introduction

Manufacturing process variations and environmental effects (such as temperature) result in the variations of the values of circuit elements and parameters. Statistical methods of circuit design optimization take those variations into account and apply statistical (or statistical/deterministic) optimization techniques to obtain an ''optimal'' design. Statistical design optimization belongs to a general area of statistical circuit design.

5.2 Problems and Methodologies of Statistical Circuit Design

A broad class of problems exists in this area: statistical analysis involves studying the effects of element variations on circuit performance. It applies statistical techniques, such as Monte Carlo simulation [34] and the variance propagation method [39], to estimate variability of performances. Design centering attempts to find a center of the acceptability region [12] such that manufacturing yield is maximized.

Direct methods of yield optimization use yield as the objective function and utilize various statistical (or mixed statistical/deterministic) algorithms to find the yield maximum in the space of designable circuit/process parameters. Design centering and tolerance assignment (used mostly for discrete circuits) attempt to find the design center, with simultaneous optimal assignment of circuit element tolerances, minimizing some suitable cost function and providing 100% yield (worst-case design) [4,6]. To solve this problem, mostly deterministic algorithms of nonlinear programming are used. Worst-case design is often too pessimistic and too conservative, leading to substantial overdesign. This fact motivates the use of statistical techniques, which provide a much more realistic estimation of the actual performance variations and lead to superior designs. Stringent requirements of the contemporary very large scale integration (VLSI) design prompted a renewed interest in the practical application of these techniques. The most significant philosophy introduced recently in this area is statistical Design for Quality (DFQ). It was stimulated by the practical appeal of the DFQ methodologies introduced by Taguchi [30], oriented toward ''on-target'' design with performance variability minimization. In what follows, mostly the techniques of manufacturing yield optimization and their design for quality generalization are discussed.

5.3 Underlying Concepts and Techniques

5.3.1 Circuit Variables, Parameters, and Performances

5.3.1.1 Designable Parameters

Designable parameters, represented by the n-dimensional vector* $x = (x_1, \ldots, x_n)$, are used by circuit designers as ''decision'' variables during circuit design and optimization. Typical examples are nominal values of passive elements, nominal MOS transistor mask dimensions, process control parameters, etc.

5.3.1.2 Random Variables

The t-dimensional vector of random variables (or ''noise'' parameters in Taguchi's terminology [30]) is denoted as $\theta = (\theta_1, \ldots, \theta_t)$. It represents statistical R, L, C element variations, disturbances or variations of manufacturing process parameters, variations of device model parameters such as t_{ox} (oxide thickness), V_{TH} (threshold voltage), and environmental effects such as temperature, supply voltage, etc. Usually, θ represents principal random variables, selected to be statistically independent and such that all other random parameters can be related to them through some statistical models. Probability density function (p.d.f.) of θ parameters will be denoted as $f_\theta(\theta)$.

5.3.1.3 Circuit (Simulator) Variables

These variables represent parameters and variables used in circuit, process, or system simulators such as SPICE. They are represented as the c-dimensional vector $e = (e_1, \ldots, e_c)$. Specific examples of e variables are: R, L, C elements, gate widths W_j and lengths L_j of MOS transistors, device model parameters, or, if a process simulator is used, process-related control, and physical and random parameters available to the user. The e vector contains only those variables that are directly related to the x and θ vectors.† This relationship is, in general, expressed as

$$e = e(x, \theta) \tag{5.1}$$

The p.d.f. of θ is transformed into $f_e(e)$ the p.d.f. of e. This p.d.f. can be singular, i.e., defined in a certain subspace of the e-space (see examples below). Moreover, it can be very complicated, with highly nonlinear statistical dependencies between different parameters, so it is very difficult to represent it directly as

* Vectors are denoted by lowercase letters subscripts or superscripts.
† This means that, e.g., some SPICE parameters will always be *fixed*.

a p.d.f. of e. In the majority of cases, the analytic form of $f_e(e)$ is not known. For that reason, techniques of statistical modeling are used (see the next section).

5.3.1.4 Circuit Performances

The vector of circuit performance function (or simply performances) is defined as the m-dimensional vector $y = (y_1, \ldots, y_m)$. Its elements can be gain, bandwidth, slew rate, signal delay, and circuit response for a single frequency or time, etc. Each of the performances y_j is a function of the vector of circuit elements e: $y_j = y_j(e) = y_j(e(x, \theta))$. These transformations are most often not directly known in analytical form and a circuit simulator (such as SPICE) must be used to find the values of y_j's corresponding to the given values of x and θ. The overall simulator time required for the determination of all the performances y can be substantial for large circuits. This is the major limiting factor for the practical application of statistical circuit design techniques. To circumvent this problem, new statistical macromodeling techniques are being introduced [31] (see example in Section 5.4.9).

5.3.2 Statistical Modeling of Circuit (Simulator) Variables

Statistical modeling is the process of finding a suitable transformation $e = e(x, \theta)$, such that given the distribution of θ, the distribution of e can be generated. The transformation $e = e(x, \theta)$ can be described by closed form analytical formulas or by a computer algorithm.

For discrete active RLC circuits (e.g., such as a common emitter amplifier), vector e is composed of two parts: the first part contains the actual values of statistically perturbed RLC elements: i.e., $e_i = x_i + \theta_i$ for those elements. In this formula, θ represents absolute element spreads and its expected (average) value, $E\{\theta\} = 0$ and x_i is the nominal value of e_i, often selected as the expected value of e_i. This implies that the variance of e_i, $\text{var}\{e_i\} = \text{var}\{\theta_i\}$; $E\{e_i\} = x_i$ and the distribution of e_i is the same as that of θ_i, with the expected value shifted by x_i. Alternatively, if θ_i represents relative element spreads, $e_i = x_i (1 + \theta_i)$, where $E\{\theta_i\} = 0$. Therefore, $E\{e_i\} = x_i$; $\text{var}\{e_i\} = x_i^2\ \text{var}\{\theta_i\}$, i.e., the standard deviations σ_{ei} and $\sigma_{\theta i}$ are related: $\sigma_{ei} = x_i\ \sigma_{\theta i}$, or $\sigma_{ei}/E\{e_i\} = \sigma_{ei}/x_i = \sigma_{\theta i}$. This means that with fixed $\sigma_{\theta i}$, the relative standard deviation of e_i is constant, as it is often the case in practice, where standard deviations of RLC elements are described in percents of the element nominal values. Both forms of e_i indicate that each e_i is directly associated with its corresponding θ_i and x_i, and that there is one-to-one mapping between e_i and θ_i. These dependencies are important, since many of the yield optimization algorithms were developed assuming that $e_i = x_i + \theta_i$. A typical p.d.f. for discrete elements is shown in Figure 5.1, before "binning" into different categories (the whole curve) and after binning into ±1%, ±5%, and ±10% resistors (the shaded and white areas: e.g., the ±10% resistors will have the distribution characterized by the external shaded areas, with a ±5% "hole" in the middle).

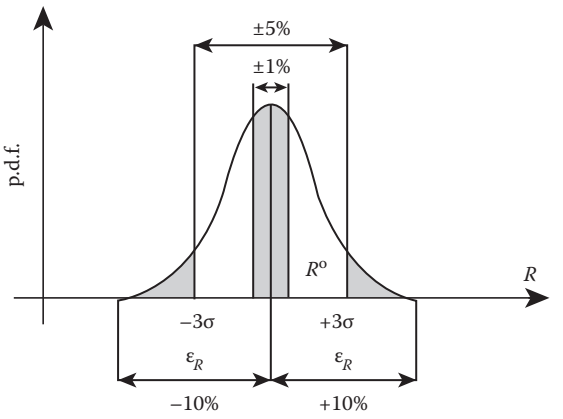


FIGURE 5.1 Typical probability density function (p.d.f.) of a discrete resistor before and after "binning."

Usually, passive discrete elements are statistically independent* as shown in Figure 5.2.

* But, for instance, if R_L (R_C) is a loss resistance of an inductor L (capacitor C) then L and R_L (C and R_C) are correlated.

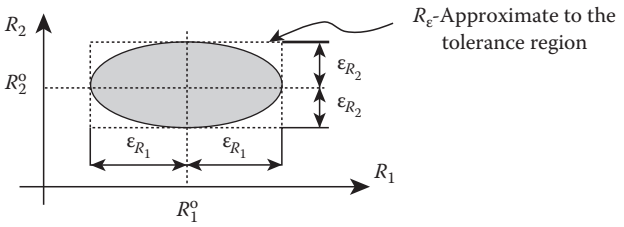


FIGURE 5.2 Level set (cross section) of a p.d.f. function for two discrete resistors after manufacturing.

The cross-section shown is often called a level set, a norm body [12], or a tolerance body. It is defined as a set of element values for which the element p.d.f. is larger than a prescribed value.

This value is selected such that the probability of the element values falling into the tolerance body is equal to e.g., 95%, (i.e., the tolerance body represents 95% of the entire element population). Alternatively, the tolerance body can be represented as a (hyper-) box shown in the figure, with the sides equal to $2\varepsilon_i$ ($2\varepsilon_{R_i}$ in the figure), called the tolerance region and denoted by R_ε. Figure 5.2 also shows that the dependence $e_i = x_i + \theta_i$ is equivalent in this case to $R_i = R_{\mathrm{NOM},i} + \Delta R_i$, where $x_i = R_{\mathrm{NOM},i}$; $\theta = \Delta R_i$).

The second part of e is composed of the parameters representing active device (e.g., BJT, MOS) model parameters. They are usually strongly correlated within each device model (same applies to ICs), but typically, no correlations occur between device model parameters of different devices. Each of the device model parameters e_d is related through a specific model $e_\mathrm{d} = e_\mathrm{d}(x, \theta)$ to the vector θ parameters,* representing principal random variables, which are themselves often some device model parameters (such as oxide thickness t_{ox} of MOS transistors), and/or are some dummy random variables. For example, in the BJT empirical statistical model introduced in [3], the base transient time T_n is modeled as follows:

$$e_\mathrm{d} = e_\mathrm{d}(x_\mathrm{d}, \theta_1, \theta_2) = T_n(\beta, X_{r5}) = \left(a + \frac{b}{\sqrt{\beta}}\right)(1 + cX_{r5}) \tag{5.2}$$

i.e., it is the function of the current gain $\theta_1 = \beta$ (the principal random variable, affecting the majority of the BJT model parameters) and $\theta_2 = X_{r5}$ (a dummy random variable, uniformly distributed in the interval $[-1, 1]$ independent of β and having no physical meaning); a, b, and c are empirically selected constants (Figure 5.3b).

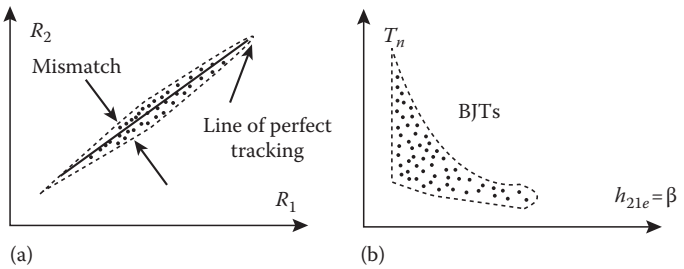


FIGURE 5.3 Dependencies between passive IC elements and between device model parameters: (a) linear, between two resistors R_1 and R_2 and (b) nonlinear, between the base transit time T_n and the current gain $h_{21e} = \beta$ for bipolar junction transistors.

* Observe that these models are parameterized by x: e.g., the MOS transistor model parameters e_i will also depend on the device length L and width W.

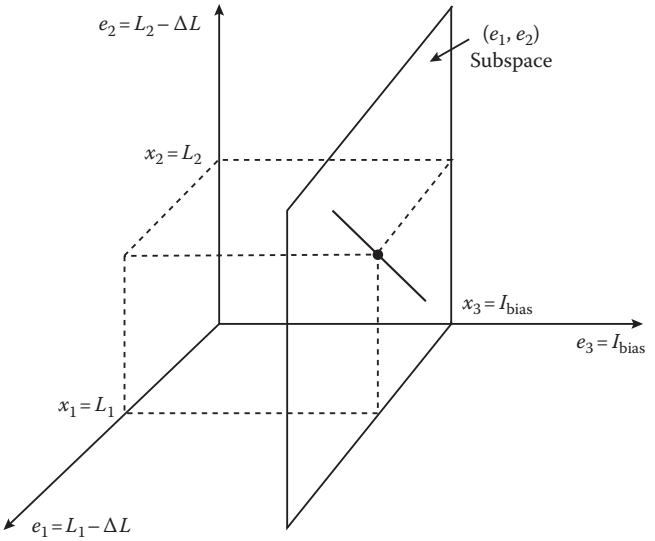


FIGURE 5.4 Singular distribution $f_e(e_1, e_2, e_3)$ in three-dimensional e-space, represented by the thick line in the (e_1, e_2)-subspace (plane). This is due to the perfect matching of $\Delta L \equiv \Delta L_1 \equiv \Delta L_2$ values.

In the discrete circuit case, each designable parameter x_i has a random variable θ_i added to it (i.e., x and θ in the same space in which the p.d.f. $f_e(e) = f_\theta(x + \theta)$ is defined.

For integrated circuits, the passive elements (R's, C's), active dimensions (such as Ws and Ls for MOS transistors), and other transistor model parameters are strongly correlated (see Figure 5.3a). Because of this, the distribution of θ parameters is limited to a certain subspace of the entire e-space, i.e., $f_e(e(x, 0))$ is singular, and the formula $e_i = x_i + \theta_i$ does not hold. As an example, consider a subset of all e parameters representing the gate lengths $x_1 = L_1$, $x_2 = L_2$ (designable parameters) of two MOS transistors T_1, T_2 and an independent designable parameter $x_3 + I_{bias}$, representing the transistor bias current. Assume also (as is frequently the case) that $\theta \equiv \Delta L$, the technological gate length reduction (a common random parameter) changes the same way for T_1 and T_2, i.e., there is ideal matching of $\Delta L \equiv \Delta L_1 = \Delta L_2$,* i.e., $e_1 + x_1 + \theta + L_1 - \Delta L$, $e_2 + x_2 + \theta + L_2 - \Delta L$, $e_3 + x_3 + I_{bias}$. The only random variable in this model is ΔL, as shown in Figure 5.4. The p.d.f. $f_e(e)$ is in this case defined only in the one-dimensional subspace of the e-space (i.e., all the realization of the vector e are located on the thick line in Figure 5.4). The major consequence of this difference is that, in general, many yield optimization algorithms developed for discrete circuits (i.e., using the explicit assumption that $(e_i = e_i + \Delta_i)$, cannot be used for IC design.

Statistical modeling for ICs is concerned with representing global variations and parameter mismatches† occurring between different devices on the same chip. In the last example, mismatch can be modeled by two additional independent local mismatch variables ε_{L_1}, ε_{L_2}, representing small random local deviations of ΔL_1 and ΔL_2 from the ideal value ΔL, i.e., $e_1 + x_1 + \theta + \varepsilon_i + L_1 - \Delta L - \varepsilon_{L_1}$, $e_2 + x_2 + \theta + \varepsilon_i + L_2 - \Delta L - \varepsilon_{L_2}$. With this model, ΔL represents a global or common factor, affecting both channel lengths, and ε_{L_1}, ε_{L_2}, represented *specific* (local) variations. The model just presented is a simple (simple, one-factor) case of the Factor Analysis (FA) [20] model of correlations between e_1 and e_2.

In [8], FA analysis, together with Principal Component Analysis (PCA)‡ and nonlinear regression were used for the determination of linear and nonlinear statistical models for CMOS transistor parameters. The following common factors $F_1, \ldots, F_8$ were identified: t_{ox} (oxide thickness common to n- and p-type

* Such a model is commonly used for digital ICs [9], since it is of sufficient accuracy for digital applications.
† Element mismatches are very important for analog ICs.
‡ Principal component analysis involves coordinate rotation and leads to *uncorrelated principal components*.

transistors) $N_{SUB,n}$, $N_{SUB,p}$ (n- and p-type substrate doping), ΔL_n, ΔL_p (length reduction), ΔW_n, ΔW_p (width reduction—for narrow transistors only), and XJ_p (p-type junction depth). These variables were causing about 96% of the total variability of all parameters. All the other CMOS transistor model parameters were related to the $F_1, \ldots, F_8$ factors through quadratic (or linear in simplified models) regression formulas. The resulting models were able to represent—with a high level of accuracy—the strong nonlinear statistical dependencies existing between some model parameters.

The major theoretical and empirical findings in mismatch modeling are the device area (or length) and distance dependencies of the mismatch level (see [23], where references to other authors can be found). For example, the variance of the difference $e_1 - e_2$ between two MOS transistor model parameters e_1, e_2 is modeled as [3,9]

$$\text{var}(e_1 - e_2) = \frac{a_p}{2W_1L_1} + \frac{a_p}{2W_2L_2} + s_p^2 d_{12}^2 \tag{5.3}$$

where

W_1, L_1, W_2, and L_2 are widths and lengths of the two transistors
d_{12} is the distance between the transistor centers
a_p and s_p are empirical coefficients adjusted individually for each model parameter

Using this model together with PCA, and introducing other concepts, two quite sophisticated linear statistical models in the form $e_i = e(x_i, \theta, W_i, L_i, d)$ were proposed in [23]. They include the transistor separation distance information, collected into the vector d in two different forms. The models, constructed from on-chip measured data, were used for practical yield optimization. θ parameters were divided into two groups: a group of correlated random variables responsible for the common part of each parameter variance and correlations between model parameters of each *individual* transistor, and the second group of local (mismatched related) random variables, responsible for mismatches between different transistors. Additional dependencies, related to transistor spacing and device area related coefficients, maintain proper mismatch dependencies.

5.3.3 Acceptability Regions

The acceptability region A_y is defined as a region in the space of performance parameters y (y-space), for which all inequality and equality constraints imposed on y are fulfilled. In the majority of cases, A_y is a hyperbox, i.e., all the constraints are of the form: $S_j^L \le y_j \le S_j^U$; $j = 1, \ldots, m$, where S_j^L and S_j^U are the (designer defined) lower and upper bounds imposed on y_j, called also designer's specifications. More complicated specifications, involving some relations between y_i parameters, or S_j^L and S_j^U bounds can also be defined.

For the simplest case of the y-space box-constraints, the acceptability region A in the e-space is defined as such a set of e vectors in the c-dimensional space, for which all inequalities $S_j^L \le y_j(e) \le S_j^U, j = 1, \ldots, m$, are fulfilled. Illustration of this definition is shown in Figure 5.5. It can be interpreted as the mapping of the acceptability region A_y from the y-space into the e-space. Acceptability regions A can be very complicated: they can be nonconvex and can contain internal infeasible regions (or "holes"), as shown in Figure 5.6 for a simple active RC filter [25].

For discrete circuits, A is normally represented in the e-space, due to the simple relationship $e_i = x_i + \theta_i$ (or $e_i = x_i(1 + \theta_i)$), between e_i, x_i, and θ_i. For integrated circuits, e is related to x and θ through the statistical model, x and θ are in different spaces (or subspaces), the dimension of θ is lower than the dimension e, and the p.d.f. $f_e(e)$ is singular and usually unknown in analytic form. For these reasons, it is more convenient to represent A in the joint (x, θ)-space, as shown in Figure 5.7. For a fixed x, A can be defined in the θ-space and labeled as $A\theta\ (x)$, since it is parameterized by the actual values of x, as shown in Figure 5.7. The shape and location of $A\theta\ (x)$ change with x, as shown. For a fixed x, $A\theta\ (x)$ is defined as

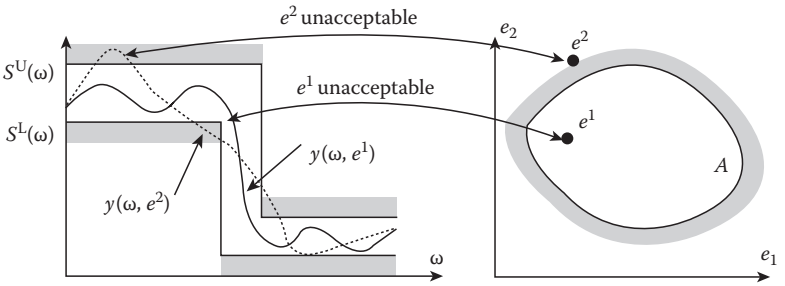


FIGURE 5.5 Illustration of the mapping of the $S^L(\omega)$, $S^U(\omega)$ constraints imposed on $y(\omega, e)$ into the e-space of circuit parameters.

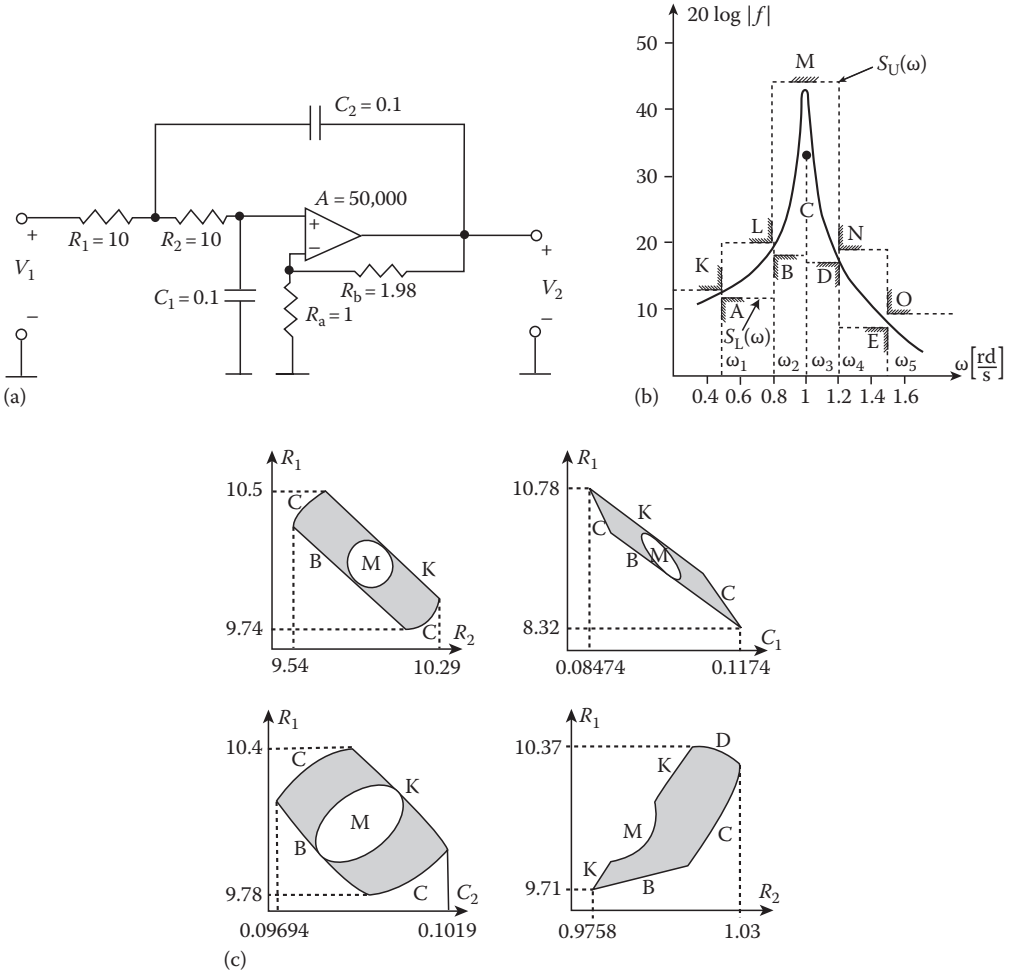


FIGURE 5.6 (a) Sallen–Key active filter, (b) lower and upper bounds imposed in the filter frequency response, and (c) two-dimensional cross sections of the acceptability region A. Capital letters in (b) and (c) indicate the correspondence of constraints to the boundaries of A.

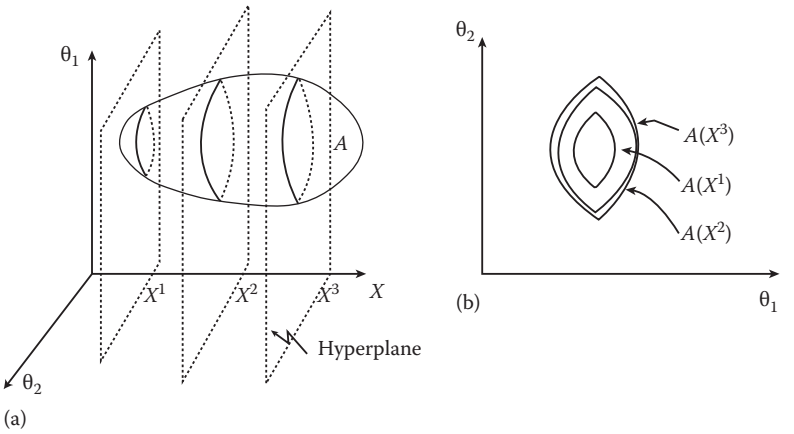


FIGURE 5.7 Acceptability region for integrate circuits: (a) in the joint (x, θ)-space and (b) in the θ-space, parameterized by different values of x. The hyperplanes shown represent t-dimensional subspaces of θ parameters.

such a region in the t-dimensional θ space, for which all the inequalities $S_j^L \leq y_j(e(x, \theta)) \leq S_j^U; j = 1, \ldots, m$, are fulfilled.

In order to recognize if a given point $e(x, \theta)$ in the circuit parameter space belongs to A [or $A\theta$ (x)], an indicator function $\phi(\cdot)$ is introduced:

$$\phi(e(x,\theta)) = \begin{cases} 1 & \text{if } e(x,\theta) \text{ belongs to } A \text{ (a successful, or ''pass point'')} \\ 0 & \text{otherwise (a ''fall point'')} \end{cases} \tag{5.4}$$

A complementary indicator function $\phi_F(e(x, \theta)) = \phi(e(x, \theta)) - 1$ is equal to 1 if $e(x, \theta)$ does not belong to A and 0 otherwise. Both indicator functions will be used in what follows.

5.3.4 Methods of Acceptability Region Approximation

Except for some simple cases, the acceptability region A in the e (or the joint (x, θ))-space is unknown and it is impossible to fully define it. For yield optimization and other statistical design tasks and implicit or explicit knowledge of A and/or its boundary is required. If only the points belonging to A are stored, this can be considered a point-based ''approximation'' to A. Some of the point-based methods are Monte Carlo-based design centering, centers of gravity method, point-based simplicial approximation and yield evaluation (see below), ''parametric sampling-based'' yield optimization [36], yield optimization with ''reusable'' points [39], and others.

The acceptability segment-based method of the A-region approximation was called in [27] a one-dimensional orthogonal search (ODOS) technique leading to several yield optimization methods [25,46]. Its basic principle is shown in Figure 5.8b, where line segments passing through the points e^i randomly sampled in the e-space and parallel to the coordinate axes, are used for the approximation of A. ODOS is very efficient for large linear circuits, since the intersections with A can be directly found from analytical formulas. The two-dimensional cross-sections of A, shown in Figure 5.6, were obtained using this approach. The surface integral-based yield and yield gradient estimation and optimization method proposed in [13] also use the segment approximation to the boundary of A. A variant of this method is segment approximation in one direction, as shown in Figure 5.8c. This method was then extended to plane and hyperplane approximation to A in [42] (Figure 5.8d). In another approach, called ''radial exploration of space'' in [54], the segments approximating A are in radial directions, as shown in Figure 5.8e.

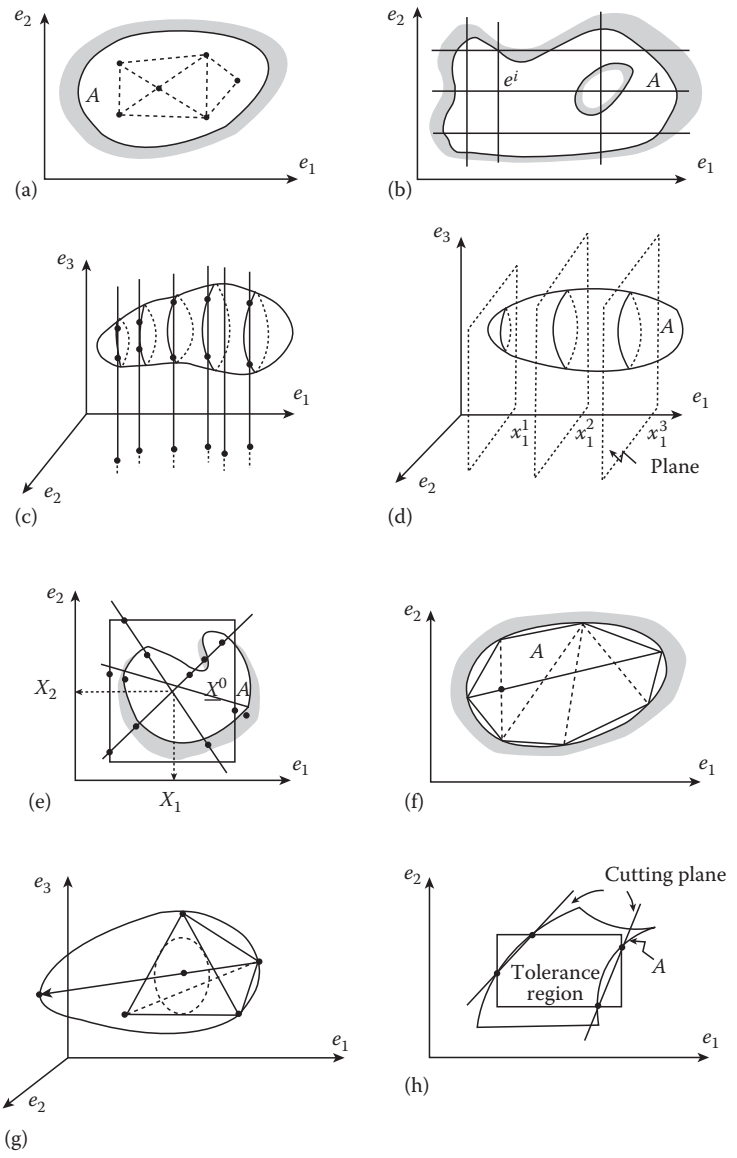


FIGURE 5.8 Various methods of acceptability region approximation: (a) "point-based" simplicial approximation to the A-region, (b) segment approximation to A in all directions along e_i axes, (c) segment approximation to A in one direction, (d) (hyper)-plane approximation A, (e) segment approximation to A in radial directions, (f) simplicial approximation to A in two dimensions, (g) simplicial approximation to A in three dimensions, and (h) cutting-plane approximation to A. *Note:* All A-regions are shown in e-space, for the discrete case, i.e., for $e = x + \theta$.

The techniques just described rely on the fact that "segment" yields (calculated in the e subspace) can be, for some special cases [27,54], calculated more efficiently than using a standard Monte Carlo method. This leads to higher efficiency and accuracy of yield estimation, in comparison to the point-based yield estimation.

The simplicial approximation proposed in [10] (described in more detail in Section 4.4), is based on approximating the boundary of A in the e-space, by a polyhedron, i.e., by the union of those partitions of a set of c-dimensional hyperplanes which lie inside of the boundary of A or on it. The boundary of

A-region is assumed to be convex (see Figure 5.8f and g). The approximating polyhedron is a convex hull of points. Simplicial approximation is obtained by locating points on the boundary of A, by a systematic expansion of the polyhedron. The search for next vertex is always performed in the direction passing through the center of the largest face of the polyhedron already existing. In the MC-oriented "point-based" version of the method [11] (see Figure 5.8a), subsequent simplicial approximations $\tilde{A}_i$ to A are not based on the points located on the boundary of A (which is computationally expensive) but on the points e^i belonging to A already generated during the MC simulation; after each new point is generated, it is checked if e^i belongs to $\tilde{A}^{i-1}$, (where $\tilde{A}^{i-1}$ is the latest simplicial approximation to A); if yes, the sampled point is considered successful *without* performing the circuit analysis; if e^i does not belong to $\tilde{A}^{i-1}$ and the circuit analysis reveals that e^i belongs to A, the polyhedron is expanded to include this point. Several versions of this general approach, leading to the substantial reduction of the computational effort of yield estimation, are described in the original article.

The method of "cuts" proposed in [5], creates a "cutting-plane" approximation to A in the corners of the tolerance region R_ε assumed to be a hypercube, as shown in Figure 5.8h. The method was combined with discretization of the p.d.f. $f_e(e)$ and multidimensional quadratic approximation to circuit constraints, leading to simplified yield and yield derivative formulas and yield optimization for arbitrary statistical distributors [1].

5.3.5 Manufacturing (Parametric) Yield

Manufacturing yield is defined as the percentage of the total number of products manufactured that fulfill both functional and parametric performance requirements.* Functional circuit performance is the circuit ability to perform desired functions. Catastrophic (or "hard") circuit failures (such as shorts or open circuit faults caused, e.g., by particular wafer contamination) will completely eliminate some of the circuit functions, thus decreasing the part of the overall yield called functional yield. Parametric circuit performance is a measure of circuit quality and is represented by measurable performance functions such as gain, delay, bandwidth, etc., constituting the y-parameter vector. The part of yield related to parametric circuit performance is called parametric yield and is the only type of yield considered in what follows. The actual manufacturing yield is smaller than the parametric yield, since it is equal to the product of the functional or parametric yield [12]. In general, parametric yield is used during circuit electrical design, while functional yield is used during circuit layout design.† Both are used to predict and optimize yield during circuit design.

Parametric yield, therefore, is equal to the percentage of circuits that fulfill all parametric requirements, i.e., it is equal to the probability that e belongs to the acceptability region A. So, it can be calculated as the integral of the p.d.f. of e, $f_e(e)$ over A, for a given vector of designable parameters x. Since $e = e(x, \theta)$ is a function of x, then e, $f_e(e) = f_e(e, x)$ (e.g., $E\{e\}$ and $\mathrm{var}\{e\}$ can be both functions of x). Therefore,‡

$$Y(x) = P\{e \in A\} = \int_A f_e(e, x)\mathrm{d}e = \int_{R^c} \phi(e) f_e(e, x)\mathrm{d}e = E_e\{\phi(e)\} \tag{5.5}$$

where

$P\{\cdot\}$ denotes probability

$\phi(e)$ is the indicator function (Equation 5.4)

$E_e\{\cdot\}$ is expectation with respect to the random variable e

* For more detailed yield definitions involving different types of yield, e.g., design yield, wafer yield, probe yield, processing yield, etc., see [12].

† Layout design (i.e., transistor spacing, location, size) has also influence on parameter variations and mismatch, as discussed in Section 5.3.

‡ Multiple integration performed below is over the acceptability region A, or over the entire c-dimensional space R^c of real numbers. ε means "belongs to."

The above formula is useful if $f_e(e, x)$ is a nonsingular p.d.f., which is usually the case for discrete circuits, for which $e_i = x_i + \theta_i$, (or $e_i = x_i\ (1 + \theta_i)$). In a general case, however (e.g., for integrated circuits), the p.d.f. $f_e(e, x)$ is not known, since it has to be obtained from a complicated transformation $e = e\ (x, \theta)$, given the p.d.f. $f_\theta(\theta)$ of θ. Therefore, it is more convenient to integrate directly in the θ-space. Since parametric yield is also the probability that θ belongs to $A\theta(x)$ (the acceptability region in the θ-space for any fixed x), yield becomes

$$Y(x) = P\{\theta \in A_\theta(x)\} = \int_{A_{\theta(x)}} f_\theta(\theta)\mathrm{d}\theta$$
$$= \int_{R^t} \phi(e(x,\theta))f_\theta(\theta)\mathrm{d}\theta = E_\theta\{\phi(e(x,\theta))\} \tag{5.6}$$

Equation 5.6 is general, and is valid for both discrete and integrated circuits. An unbiased estimator of $E_\theta\{\rho(e(x,\ \theta))\} + E_\theta\{\rho(\theta)\}$ (for fixed x), is the arithmetic mean, based on N points θ^i, sample in θ-space with the p.d.f. $f_\theta(\theta)$, for which the function $\phi(\theta^i)$ is calculated (this involves circuit analyses). Thus, the yield estimator $\hat{Y}$ is expressed as

$$\hat{Y} = \frac{1}{N}\sum_{i=1}^{N} \phi\left(\theta^i\right) = \frac{N_s}{N} \tag{5.7}$$

where N_S is the number of successful trials, i.e., the number of circuits for which $\theta \in A\theta\ (x)$ (all circuit constraints are fulfilled). Integral of Equation 5.6 is normally calculated using Monte Carlo (MC) simulations [34] and Equation 5.7. The MC method is also used to determine statistical parameters of the p.d.f. $f_y(y)$ of $y = y(\mathrm{x}, \theta)$. In order to sample the θ parameters with p.d.f. $f_\theta(\theta)$, special numerical procedures, called random number generators, are used. The basic MC algorithm is as follows:

1. Set $i = 0$, $N_S = 0$ (i is the current index of a sampled point and N_S is the total number of successful trials).
2. Substitute $i = i + 1$, generate the ith realization of $\theta : \theta^i = (\theta_1^i, \ldots, \theta_t^i)$, with the p.d.f. $f_\theta(\theta)$.
3. Calculate the ith realization of $y^i = (y_1^i, \ldots, y_m^i) = y(x, \theta^i)$, with the aid of an appropriate circuit analysis program, and store the results.
4. Check if all circuit constraints are fulfilled, i.e., if $S^L \le y^i \le S^U$; if yes, set $N_S = N_S + 1$.
5. If $i \neq N$, go to (2); otherwise, find the yield estimator $\hat{Y} = N_S/N$. If needed, find also some statistical characteristics of y-parameters (e.g., create histograms of y, find statistical moments of y, etc.).

To generate θ^i's with the p.d.f. $f_\theta(\theta)$, the uniformly distributed random numbers are generated first and then transformed to $f_\theta(\theta)$. The most typical random number generator (r.n.g.), generating a sequence of pseudorandom, uniformly distributed integers θ_k in the interval $[0, M]$ (M is an integer), is a multiplicative r.n.g., using the formula [34]: $\theta_{k+1} = c\theta_k \pmod M$ where c is an integer constant and $\theta_k \pmod M$ denotes a remainder from dividing θ_k by M. The initial value θ_0 of θ_k is called the ''seed'' of the r.n.g., and, together with c, should usually be chosen very carefully, to provide good quality of the random sequence generated. Several other r.n.g.s are used in practice [34]. The r_k numbers in the [0, 1) interval are obtained from $r_k = \theta_k/M$. Distributions other than uniform are obtained through different transformations of the uniformly distributed random numbers, such as the inverse of the cumulative distribution function [34]. To generate correlated normal variables $\theta = (\theta_1, \ldots, \theta_n)$, with a given covariance matrix K^θ, the transformation $\theta = CZ$ is used, where $Z = (z_1, z_2, \ldots, z_n)$ is the vector of independent normal variables with $E\{z_i\} = 0$, $\mathrm{var}\{z_i\} = 1$, $i = 1, \ldots, n$, and C is a matrix obtained from the so-called

Cholesky decomposition of the covariance matrix K^{θ} such that $K^{\theta} = CC^{t}$ where t denotes transposition. C is usually lower or upper triangular and can be easily constructed from a given matrix K^{θ} [34].

The yield estimator $\hat{Y}$ is a random variable, since performing different, independent MC simulations, one can expect different values of $\hat{Y} = N_S/N$. As a measure of $\hat{Y}$ variations, variance or standard deviation of $\hat{Y}$ can be used. It can be shown [7] that the standard deviation of $\hat{Y}$, is equal to $\sigma_{\hat{Y}} = Y(1-Y)/N$, i.e., it is proportional to $1/\sqrt{N}$ Hence, to decrease the error of $\hat{Y}$ 10 times, the number of samples has to be increased 100 times. This is a major drawback of the MC method. However, the accuracy of the MC method (measured by $\sigma_{\hat{Y}}$) is independent of the dimensionality of the θ-space, which is usually a drawback of other methods of yield estimation. One of the methods of variance reduction of the $\hat{Y}$ estimator is importance sampling [7,34,39]. Assume that instead of sampling θ with the p.d.f. $f_\theta(\theta)$, some other p.d.f. $g_\theta(\theta)$ is used. Then,

$$Y = \int_{R'} \phi[e(\theta)] \frac{f_\theta(\theta)}{g_\theta(\theta)} g_\theta(\theta)\, d\theta \equiv E\left\{ \phi[e(\theta)] \frac{f_\theta(\theta)}{g_\theta(\theta)} \right\} \tag{5.8}$$

where $g_\theta(\theta) \neq 0$ if $\phi(\theta) = 1$. Yield Y can now be estimated as

$$\tilde{Y} = \frac{1}{N} \sum_{i=1}^{N} \phi\left[e(\theta^i)\right] \frac{f_\theta(\theta^i)}{g_\theta(\theta^i)} \tag{5.9}$$

sampling N points θ^i with the p.d.f. $g_\theta(\theta)$. The variance of this estimator is var $\{\tilde{Y}\} = E\{[\phi(\theta)/g_\theta(\theta) - Y]^2\}/N$. If it is possible to choose $g_\theta(\theta)$ such that it mimics (or is similar to) $\phi(\theta)f_\theta(\theta)/Y$, the variability of $[\phi(\theta)f_\theta(\theta)/g_\theta(\theta) - Y]$ is reduced, and thus the variance of $\hat{Y}$. This can be accomplished if some approximation to $\phi(\theta)$ i.e., to the acceptability region A is known. Some possibilities of using importance sampling techniques were studied, e.g., in [16]. One of such methods, called parametric sampling was used in [36], and other variants of important sampling were used in [2,40] for yield optimization. There are several other methods of variance reduction, such as the method of control variates, correlated sampling, stratified sampling, antithetic variates, and others [7,34,39]. Some of them have been used for statistical circuit design [7,39].

5.4 Statistical Methods of Yield Optimization

The objective of the yield optimization is to find a vector of designable parameters $x = x_{opt}$, such that $Y(x_{opt})$ is maximized. This is illustrated in Figure 5.9 for the case of discrete circuits where $e_1 = x_1 + \theta_1$, $e_2 = x_2 + \theta_2$.* Figure 5.9a corresponds to low initial yield proportional to the weighted (by the p.d.f. $f_e(e)$) area (hypervolume, in general), represented by the dark shaded part of the tolerance body shown. Figure 5.9b corresponds to optimized yield, obtained by shifting the nominal point (x_1, x_2) to the vicinity of the geometric center of the acceptability region A. Because of this geometric property, approximate yield maximization can often be accomplished using methods called deterministic or geometrical design centering. They solve the yield maximization problem indirectly (since yield is not the objective function optimized), using a geometrical concept of maximizing the distance from x_{opt} to the boundary of A. The best known method of this class [10,12] inscribes the largest hypersphere (or other norm-body) into the simplicial approximation to the boundary of A, shown in Figure 5.8g. This approach is described in more detail in Section 4.4, together with other geometrical design centering methods.

A class of methods known as "performance-space oriented design centering methods" is also available. These methods attempt to maximize the scaled distances of the performances y_j from the lower S_j^L

* Alternatively, the model $e_i = x_i(1 + \theta_i)$ can be used, in which case the size of the tolerance body (see Section 5.3) will increase proportionally to x_i.

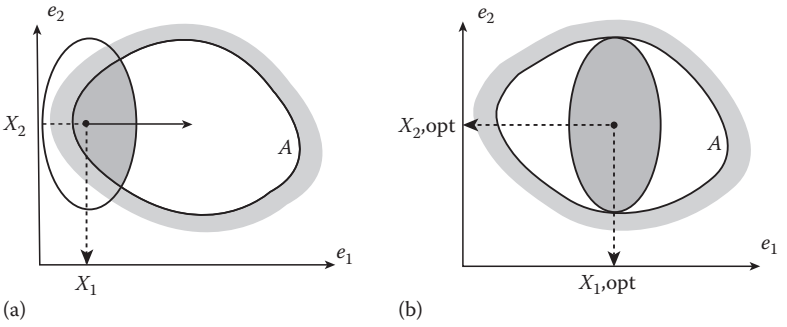


FIGURE 5.9 Interpretation of yield maximization for discrete circuits: (a) initial (low) yield and (b) optimized yield.

and/or upper S_j^U specifications, leading to approximate design centering. Some of these methods are also used for variability minimization and performance tuning, which are the most important tenets of design for quality [30], to be briefly discussed later in this section.

The major feature of statistical yield optimization methods, referred to as statistical design centering, is statistical sampling in either θ-space only or in both θ and x spaces. Sampling can be also combined with some geometrical approximation to the A-region, such as the segment, or radial-segment approximation.

In dealing with various statistical methods of yield optimization, the type of the transformation $e = e(x, \theta)$ from the θ-space to the circuit-parameter space e, has to be considered. The early, mostly heuristic yield optimization methods were based on the simple additive model $e_i = x_i + \theta_i$, valid for discrete circuits. Because of that, the majority of these methods cannot be used for IC or manufacturing process optimization, where θ and x in different spaces, or subspaces, and the distribution of θ is defined over some subspace of the e-space, i.e., it is singular.

The type of the statistical yield optimization algorithm to be used in practice strongly depends on the type of the circuit to be considered and the information available, namely: whether the transformation $e = e(x, \theta)$ is a simple one: $e_i = x_i + \theta_i$ (or $e_i = x_i(1 + \theta_i)$)(for discrete circuits) or general: $e = e(x, \theta)$ (for ICs); whether the values of $\gamma_j(x, \theta)$ for given x and θ only or also the derivatives of y_j with respect to x_k and/or θ_s are available from the circuit simulator; whether some approximation $\tilde{y}(x, \theta)$ to $y(x, \theta)$ is available—either with respect to (w.r.t.) θ only (for a fixed x), or w.r.t. both x and θ; whether the analytical form of $f_\theta(\theta)$ is known and $f_\theta(\theta)$ is differentiable w.r.t. θ, or only samples θ^i of θ are given (obtained from a numerical algorithm or from measurements), so, analytical forms of $f_\theta(\theta)$ and its derivatives w.r.t. θ are not known.

Different combinations of the cases listed above require different optimization algorithms. The more general a given algorithm is, the largest number of cases it is able to cover, but simultaneously it can be less efficient than specialized algorithms covering only selected cases. An ideal algorithm would be the one that is least restrictive and could use the minimum necessary information, i.e., it could handle the most difficult case characterized by the general transformation $e = e(x, \theta)$, with the values of $y = y(x, \theta)$ only available (generated from a circuit simulator) but without derivatives, no approximation to $y(x, \theta)$ available, and unknown analytic form of $f_\theta(\theta)$ (only the samples of θ given). Moreover, under these circumstances such an algorithm should be reasonably efficient even for large problems, and with the presence of some additional information, should become more efficient. The selected yield optimization algorithms discussed below fulfill the criteria of the algorithm ''optimality'' to a quite different level of satisfaction. It has to be stressed that due to different assumptions made during the development of different algorithms and the statistical nature of the results, an entirely fair evaluation of the actual algorithm efficiency is very difficult, and is limited to some specific cases only. Therefore, in what follows, no algorithm comparison is attempted.

5.4.1 Large-Sample vs. Small-Sample Methods

Yield optimization is concerned with the maximization of the regression function $Y(x) = E_\theta\{\phi(x, \theta)\}$ with respect to (w.r.t.) x (see Equation 5.6). In solving general problems of this type, $\phi(\cdot)$ is replaced by an arbitrary function $w(\cdot)$. Large-sample methods of optimizing $E_\theta\{w(x, \theta)\}$ calculate the expectation (average) of w (and/or its gradient) w.r.t. θ for each $x^0, x^1, x^2, \ldots$ from a large number of θ^i samples. Therefore, the averages used in a specific optimization procedure are relatively accurate, following to take relatively large steps $x^{k+1} - x^k$. On the other hand, small-sample methods use just a few (very often just one) samples of $w(x, \theta^i)$ for any given point x, and make relatively small steps in the x-space, but they utilize also a special averaging procedure, which calculates the average of w or its gradient over a certain number of steps. So, in this case, the averaging in θ-space and progression in the x-space are combined, while in the large-sample methods they are separated. Both techniques have proven convergence under certain (different) conditions. The majority of yield optimization methods belong to the large-sample category (but some can be modified to use a small number of samples per iteration). *A* class of small-sample yield optimization methods was proposed in [50] and is based on the well-known techniques of stochastic approximation [34], to be discussed later in this section.

5.4.2 Methods Using Standard Deterministic Optimization Algorithms

The most natural method of yield optimization would be to estimate the yields $Y(x^0), Y(x^1), \ldots$ from a large number of samples (as described in the previous section) for each $x^0, x^1, \ldots$ of the sequence $\{x^k\}$ generated by a standard, nonderivative deterministic search algorithm, such as the simplex method of Nelder and Mead, Powell, or other algorithms discussed in [14]. This is very appealing, since most of the conditions for algorithm's "optimality" are fulfilled: it would work for any $e = e(x, \theta)$ and only the values of $y = y(x, \theta)$ and the samples of θ would be required. However, if no approximation to $y = y(x, \theta)$ was available, the method would require tens of thousands of circuit analyses, which would be prohibitively expensive. Moreover, if the number of samples per iteration was reduced to increase efficiency, the optimizer would be receiving a highly noise-corrupted information leading to poor algorithm convergence or divergence, since standard optimization algorithms work poorly with noisy data (special algorithms, able to work under uncertainly—such as stochastic approximation algorithms—have to be used).

If some approximating functions $\hat{y} = \hat{y}(x^k, \theta)$ are available separately for each x^k, a large number of MC analyses can be cheaply performed, reducing the statistical error. In practice, such an approach is most often too expensive, due to the high cost of obtaining the approximating formulas, if the number of important θ parameters is large. The approximating functions $\hat{y}_{j(x,\theta)}$ for each y_j can be also created in the *joint* (x, θ) space [56,61]. In [45], an efficient new approximating methodology was created, highly accurate for a relatively large range of the x_i values. However, also in this case the dimension of the joint space (x, θ) cannot be too large, since the cost of obtaining the approximating functions $\hat{y}_{j(x,\theta)}$ for becomes itself prohibitively high. Because of these difficulties, several dedicated yield optimization methods have been developed, for which the use of function approximation is not required. Some of these methods are described in what follows.

5.4.3 Large-Sample Heuristic Methods for Discrete Circuits

These methods have been developed mostly for discrete circuits, for which $e_i = x_i + \theta_i$. Only $y^i = y(x + \theta^i)$ function values are required, for the sample θ^i obtained in an arbitrary way. Approximation in the e-space can be constructed to increase efficiency, but for discrete circuits the number of θ parameters can be large (proportional to the number of active devices, since no correlations between different devices exist), so the use of approximation is most often not practical. The most typical representative of this class is the centers of gravity method [39]. The method is based on a simple observation that if $\bar{x}_S$ is the center of gravity of "pass" points (as shown in Figure 5.10), defined as $\bar{x}_S = (x_A^1 + x_A^2 + \cdots + x_A^{N_A})/N_A$,

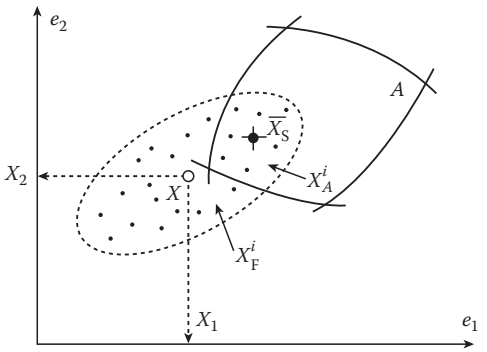


FIGURE 5.10 Interpretation of the original centers of gravity method.

where N_A is the number of points x_A^i falling into the A-region, then a step from x to $\bar{x}_S$ will improve yield. In [37], also the center of gravity of the "fail" points $\bar{x}_F = (x_F^1 + x_F^2 + \cdots + x_F^{N_F})/N_F$ was defined, and the direction of yield increase taken as going from $\bar{x}_F$ through $\bar{x}_A$, as shown in Figure 5.11. Moving in this direction with the step-size equal to $\mu(\bar{x}_S - \bar{x}_F)$, where $\mu \approx 0.2 - 2$ (often taken as $\mu = 1$) leads to a sequence of optimization steps, which is stopped if $\|\bar{x}_S - \bar{x}_F\|$ is less than a predefined small constant. This is based on the property (proved for a class of p.d.f.s in [43] that, under some conditions, at the yield maximum $\|\bar{x}_S - \bar{x}_F\| = 0$ (and $\bar{x}_S - \bar{x}_F = \hat{x}$ where $\hat{x}$ is the point of the yield maximum). It was also shown in [43] that for the normal p.d.f. $f_\theta(\theta)$ with zero correlations and all standard deviations $\sigma_{\theta_i} = \sigma_\theta$, $i = 1, \ldots, t$, equal, the "centers of gravity" direction coincides with the yield gradient direction. However, with correlations and σ_{θ_i}s not equal, the two directions can be quite different. Various schemes, aimed at the reduction of the total required number of analyses were developed based on the concepts of "reusable" points [39].

In [18,19] the original centers of gravity method was significantly improved, introducing a concept of "Gaussian adaptation" of the covariance matrix of the sampled points of θ^i, such that they (temporarily) adopt to the shape of the acceptability region, leading to higher (optimal) efficiently of the algorithm. The method was successfully used on large industrial design examples, involving as many as 130 designable parameters, not only for yield optimization, but also for standard function minimization.

The methods of "radial exploration of space" [54] (see Figure 5.8e) and one-dimensional orthogonal searches (ODOS) [26,27] (see Figure 5.8b and c) discussed in Section 5.3 in the context of Acceptability Region approximation, have also been developed into yield optimization methods: in the "radial exploration" case the asymmetry vectors were introduced, generating a direction of yield increase, and in the ODOS case using a Gauss–Seidel optimization method. Both techniques were especially efficient for linear circuits due to a high efficiency of performing circuit analyses in radial and orthogonal directions.

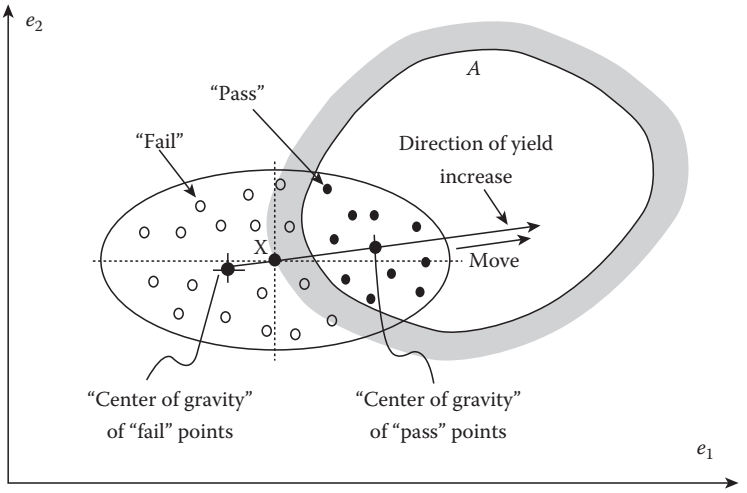


FIGURE 5.11 Interpretation of the modified centers of gravity method.

5.4.4 Large-Sample, Derivative-Based Methods for Discrete Circuits

For discrete circuits, the relation $e_i = x_i + \theta_i$ holds for a part of the entire e vector, so, x_i and θ_i are in the same space. Moreover, the p.d.f. $f_\theta(\theta)$ is most often known. Let x denote the vector of expectations $x \equiv E_\theta\{e(x,\theta)\}$, and $f_e(e, x)$ is the p.d.f. $f_\theta(\theta)$ transformed to the e-space (i.e., of the same shape as $f_\theta(\theta)$, but shifted by x). Then, from Equation 5.5, differentiating w.r.t. x_i, one obtains

$$\frac{\partial Y(x)}{\partial x_i} = \int_{R^n} \phi(e) \frac{\partial f_e(e,x)}{\partial x_i} \frac{f_e(e,x)}{f_e(e,x)} \mathrm{d}e = E_e\left\{\phi(e) \frac{\partial \ln f_e(e,x)}{\partial x_i}\right\} \tag{5.10}$$

where the equivalence $[\partial f_e(e,x)/f_e(e,x)]/\partial x_i \equiv \partial \ln f_e(e,x)/\partial x_i$ was used. Therefore, yield derivatives w.r.t. x_i can be calculated as the average of the expression in the braces of Equation 5.10, calculated from the same θ^i samples as those used for yield estimation, provided that the p.d.f. $f_e(e, x)$ is differentiable w.r.t x (e.g., the normal or log-normal p.d.f.s are differentiable, but the uniform p.d.f is not). Notice that instead of sampling with the p.d.f. $f_e(e, x)$, some other (better) p.d.f. $g_e(e, x)$ can be used as in the importance sampling yield estimation (see Equation 5.8). Then,

$$\frac{\partial Y(x)}{\partial x_i} = E_e\left\{\phi(e) \frac{\partial f_e(e,x)}{\partial x_i} \left[\frac{f_e(e,x)}{g_e(e,x)}\right]\right\} \tag{5.11}$$

where sampling is performed with the p.d.f. $g_e(e, x) \neq 0$. This technique was used in [2,36,51] (to be discussed next). Consider the multivariate normal p.d.f., with the positive definite covariance matrix K:

$$f_e(e) = \frac{1}{(2\pi)^{t/2}\sqrt{\det K}} \exp\left[-\frac{1}{2}(e-x)^{\mathrm{t}} K^{-1}(e-x)\right] \tag{5.12}$$

where $e - x \equiv \theta$ (discrete circuits), and det K is the determinant of K. Then, it can be shown that the yield gradient $\nabla_x Y(x)$ is expressed by

$$\nabla_x Y(x) = E\{\phi(e) K^{-1}(e-x)\} = Y(x) K^{-1}(\bar{x}_S - x) \tag{5.13}$$

where $\bar{x}_S$ is the center of gravity of "pass" points. If yield $Y(x)$ is a continuously differentiable function of x, then the necessary condition for the yield maximum is $\nabla_x y(\hat{x}) = 0$, which combined with Equation 5.13 means that the stationery point $\hat{x}$ for the yield function (the yield maximum if $Y(x)$ is also concave) is $\hat{x} = \bar{x}_S$, the center of gravity of the pass points. This result justifies (under the assumptions stated previously) the centers of gravity method of yield optimization (since its objective is to make $\hat{x} = \bar{x}_S \equiv \bar{x}_F$).

For $K = \mathrm{diag}\{\sigma_{e_1}^2, \ldots, \sigma_{e_t}^2\}$, i.e., with zero correlations, the yield gradient w.r.t. x is expressed as

$$\nabla_x Y(x) = E_e\left\{\phi(e)\left[\frac{\theta_1}{\sigma_{\theta_1}^2}, \cdots, \frac{\theta_t}{\sigma_{\theta_t}^2}\right]^{\mathrm{t}}\right\} \tag{5.14}$$

where $\sigma_{e_\theta} \equiv \sigma_{e_i}$ was used instead of σ_{e_i}. It can be readily shown that for all $\sigma_{\theta_i} = \cdots = \sigma_{\theta_t} = \sigma_\theta$ equal, the yield gradient direction coincides with the center-of-gravity direction [43] (Figure 5.12).

Since all higher-order derivatives of the normal p.d.f. exist, all higher order yield derivatives can also be estimated from the same sampled points θ^i, as those used for yield estimation. The yield gradient can also be calculated from the "fail" points simply using the $\phi_F(\cdot) = \phi(\cdot) - 1$ indicator function in all the expressions above. Then, the two resulting estimators can be combined as one joint average, as it was done in the $\bar{x}_S$, $\bar{x}_F$-based centers of gravity method. Actually, there exists an optimal weighted combination of the two estimators for any given problem, resulting in the minimum variability of the

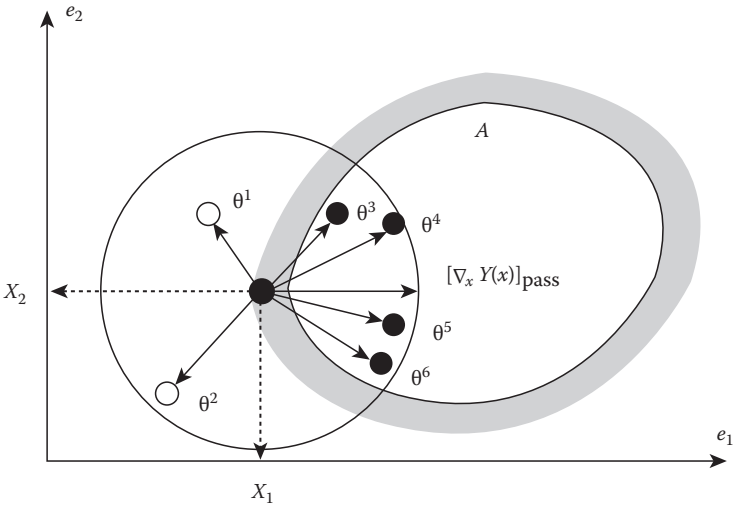


FIGURE 5.12 Interpretation of the yield gradient formula for normal p.d.f., with no correlations and $\sigma_{\theta 1} = \sigma_{\theta 2} = 1$. For the "pass" points (black dots) $[\nabla_x Y(x)]_{\text{pass}} \approx (\theta^3 + \theta^4 + \theta^5 + \theta^6)/4$; for the "fail" points (white dots) $[\nabla_x Y(x)]_{\text{fail}}$ $(-\theta^1 - \theta^2)/2$. The two estimators can be combined together (equal weighting assumed): $\nabla_x Y(x) = -\theta^1 - \theta^2 + \theta^3 + \theta^4 + \theta^5 + \theta^6)/6$. It is clearly seen that the two yield gradient estimators coincide with the center of gravity of the "pass" and "fail" points, respectively.

gradient, but it is difficult to precisely determine in practice. A general rule is that at the beginning of optimization, when x is far away from $\hat{x}$ (the optimal point), the yield gradient estimator based on the "pass" points should be more heavily weighted; the opposite is true at the end of optimization, when the "fail" points carry more precise gradient information. An interpretation of the yield gradient formula Equation 5.14 is shown in Figure 5.12, for the case where $\sigma_{\theta_1} = \sigma_{\theta_2} = 1$.

In the majority of practical applications of large-sample derivative methods, it was assumed that $f_e(e)$ was normal. A typical iteration step is made in the gradient direction:

$$x^{k+1} = x^k + \alpha_k \nabla_x Y(x^k) \tag{5.15}$$

where α_k is most often selected empirically, since yield maximization along the gradient direction is too expensive, unless some approximating functions $\hat{y} = \hat{y}(x + \theta)$ are used (normally, this is not the case for the class of methods discussed). Since the number of points θ^i sampled for each x^k, is large, the main difference between various published algorithms is how to most efficiently use the information already available. The three methods to be discussed introduced almost at the same time [2,36,51]), utilize for that purpose some form of importance sampling, discussed in Section 5.3.

In [36], a "parametric sampling" technique was proposed, in which the θ^i points were sampled with the p.d.f. $g_e(e, x)$ in a broader range than for the original p.d.f. (i.e., all the σ_{θ_i}'s were artificially increased). All points sampled were stored in a database, and the gradient-direction steps made according to Equation 5.15. The importance sampling-based gradient formula (Equation 5.11) was used in subsequent iterations within the currently available database. Then, a new set of points was generated and the whole process repeated.

The methods developed in [2,51] were also based on the importance sampling concept, but instead of using the gradient steps as in Equation 5.15, the yield gradient and Hessian* matrices were calculated

* Matrix of second derivatives.

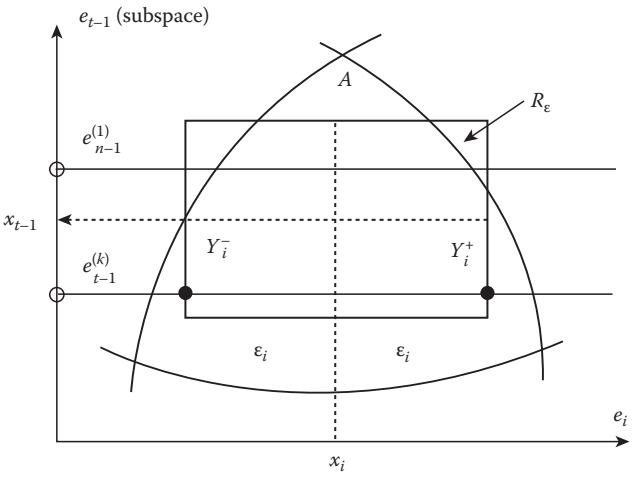


FIGURE 5.13 Yield and yield derivative estimation for uniform distribution. Y^+ and Y^- denote symbolically the "yields" calculated on the $(t-1)$-dimensional faces of the tolerance hypercube R_ε. The $e_{t-1}{}^{(k)}$ points are sampled in the $(t-1)$-dimensional subspaces of e_{t-1} parameters, with the uniform p.d.f. $f_{et-1}(e_{t-1})$.

and updated within a given database. Then, a more efficient Newton's direction was taken in [51] or a specially derived and efficient "yield prediction formula" used in [2]. In order to deal with the singularity or nonpositive definiteness of the Hessian matrix (which is quite possible due to the randomness of data and the behavior of $Y(x)$ itself), suitable Hessian corrections were implemented using different kinds of the Hessian matrix decomposition (Cholesky-type in [51] and eigenvalue decomposition in [2]).

As it was the case for the heuristic methods, the methods just discussed are relatively intensive to the dimensionality of x and θ spaces.

For the uniform p.d.f., centered at $e = x$ (Figure 5.13) and defined within a hyperbox $x_i - \varepsilon_i \le e_i \le x_i + \varepsilon$, $i = 1, \ldots, t$, where ε_i are element tolerances (see Figure 5.2), the yield gradient formula (Equation 5.10) cannot be used, since the uniform p.d.f. is nondifferentiable w.r.t. x_i. It can be shown [26,42,43] that yield can be calculated by sampling in the $(t-1)$-dimensional subspace of the e-space, represented by $e_{t-1} + (e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_t)$ and analytical integration in the one-dimensional subspace e_i as shown in Figure 5.13. Using this approach, it can be further proved [43] that the yield derivatives w.r.t. x_i are expressed by

$$\frac{\partial Y(x)}{\partial x_i} = \frac{1}{2\varepsilon_i}\left(Y_i^+ - Y_i^-\right) \tag{5.16}$$

where $Y_i{}^+$, $Y_i{}^-$ are "yields" calculated on the faces of the $t-1$ tolerance hyperbox R_ε, corresponding to $x_i + \varepsilon_i$ and $x_i - \varepsilon_i$, respectively. Calculation of these "yields" is very expensive, so in [43]* different algorithms improving efficiency were proposed. In [53], an approximate method using efficient three-level orthogonal array (OA) sampling on the faces of R_ε was proposed, in which (due to specific properties of OAs) the same sample points were utilized on different faces (actually one-third of all sampled points were available for a single face). This has lead to substantial computational savings and faster convergence.

* In [43], general formulas for gradient calculation for *truncated* p.d.f.s were derived.

5.4.5 Large-Sample, Derivative-Based Method for Integrated Circuits

In this case, which is typical of IC yield optimization, yield gradient calculations cannot be performed in the *e*-space, as was the case for discrete circuits. In general, yield gradient could be calculated by differentiating the $\phi(e(x,\theta)) \equiv \phi(x,\theta)$ term in the θ-space-based yield formula (Equation 5.6), derived for the general case, where the general transformation $e = e(x,\theta)$ is used. Differentiation of $\phi(x, \theta)$ is, however, not possible in the traditional sense, since $\phi(x,\theta)$ is a nondifferentiable unit step function determined over the acceptability region *A*. One possible solution was proposed in [15]. In what follows, a related, but more general method proposed in [13] is discussed.

It was first shown in [13] that yield can be evaluated as a surface-integral rather than the volume-integral (as it is normally done using the MC method). To understand this, observe that yield can be evaluated by sampling in the $t-1$ subspace of the *t*-dimensional θ-subspace, as shown in Figure 5.14, evaluating "local yields" along the lines parallel to the θ_i axis, and averaging all the local yields.* Each "local yield" can be evaluated using the values of the cumulative (conditional) distribution function† along each parallel, at the points of its intersection with the boundary of the acceptability region *A*. This process is equivalent to calculating the surface integral over the value of the cumulative density function calculated (with appropriate signs) on the boundary of the acceptability region.

The next step in [13] was to differentiate the "surface-integral" based yield formula w.r.t. x_i, leading to the following yield gradient formula:

$$\nabla Y(x) = E_{\theta_{t-1}}\left\{\sum_k f_{\theta_i}\left[\theta_i^{(k)}\right] \cdot \frac{\nabla_x y_a(x,\theta)}{\left|\partial y_a(x,\theta)/\partial\theta_i\right|}\bigg|_{x,\theta_{t-1},\theta_i^{(k)}}\right\} \tag{5.17}$$

Where summation is over all intersection points $\theta_i^{(k)}$ of the parallel shown in Figure 5.14 with the boundary of $A_\theta(x)$; y_a is that specific performance function $y_j(x,\theta)$ which, out of all the other performances, actually determines the boundary of the acceptability region at the $\theta_i^{(k)}$ intersection point.‡ The gradient $\nabla_x y_a(x,\theta)$, and the derivative $\partial y_a(x,\theta)/\partial\theta_i$ have to be calculated for every fixed sampled point

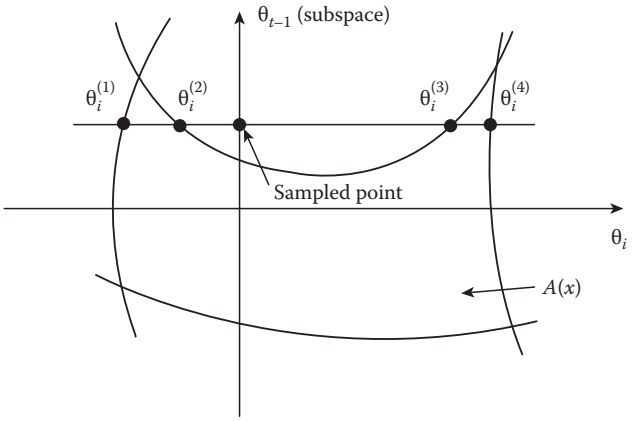


FIGURE 5.14 Interpretation of "local yield" calculation along a parallel.

* Identical technique was used in the previous section [42,43] to derive the yield derivative formula (Equation 5.16) (see Figure 5.13).

† If θ_i's are independent, it is the *marginal* p.d.f. of θ_i, as it was assumed in [13].

‡ At this point, a specific equation $y_a(x, \theta_{t-1}, 1_i^{(k)} = S_a$ (where $S_a + S_a^L$ or $S_a + S_a^U$ or are lower and upper bounds on y_a, respectively) must be *solved* to find $\theta_i^{(k)}$.

(x, θ_{t-1}), at each intersecting point $\theta_i^{(k)}$ shown in Figure 5.14. Observe that the derivative calculations in Equation 5.17 can (and often will have to) be performed in two steps: since $y_a(x, \theta) = y_a(e(x, \theta))$ so, $\partial y_a/\partial x_p = \Sigma_s(\partial y_a/\partial e_s)(\partial e_s/\partial x_p)$, and $\partial y_a/\partial \theta_i = \Sigma_s(\partial y_a/\partial e_s)(\partial e_s/\partial \theta_i)$, where the derivatives appearing in the first parentheses of both formulas are calculated using a circuit simulator and those in the second parentheses from a given statistical model $e = e(x, \theta)$.

In the practical implementation presented in [13], random points θ^r are sampled in the θ-space, in the same way as shown in Figure 5.8b but replacing e by θ, then searches for the intersections $\theta_i^{(k)}$ are performed in all axis directions, the formula in the braces of Equation 5.17 is calculated for each intersection, and averaged out over all outcomes. Searches along the parallel lines in all directions are performed to increase the accuracy of the yield gradient estimator. Observe that this technique requires tens of thousands of circuit analyses to iteratively find the intersection points $\theta_i^{(k)}$, plus additional analyses (if needed) for the calculation of the gradient and the derivatives in Equation 5.17. This problem has been circumvented in [13] by constructing approximating functions $\hat{y} = \hat{y}(x, \theta)$ w.r.t. θ for each x, together with approximating functions for all the derivatives.* Due to a high level of statistical accuracy obtained in evaluating both yield and its gradients, an efficient, gradient-based deterministic optimization algorithm, based on sequential quadratic programming was used, requiring a small number of iterations (from 5 to 11 for the examples discussed in [13]). The gradient $\nabla_x y_a(x, \theta)$ was either directly obtained from circuit simulator, or (if not available) using (improved) finite difference estimators. The method showed to be quit efficient for a moderate size of the θ-space (10–12 parameters).

The resulting yield optimization method is independent of the form of $e = e(x, \theta)$ the derivatives of y_j w.r.t. both x_k and θ_s are required and the analytical form of $f_\theta(\theta)$ and its cumulative function distribution must be known. The method cannot practically work without constructing the approximating functions $\tilde{y} = y(x, \theta)$ (approximating function in the joint space (x, θ) could also be used, if available).

5.4.6 Small-Sample Stochastic Approximation-Based Methods

Standard methods of nonlinear programming perform poorly in solving problems with statistical errors in calculating the objective functions (i.e., yield) and its derivatives.† One of the methods dedicated to the solution of such problems is the stochastic approximation (SA) approach [33] developed for solving the regression equations, and then adopted to the unconstrained and constrained optimization by several authors. These methods are aimed to the unconstrained and (maximum) of a function corrupted by noise (a regression function). The SA methods were first applied to yield optimization and statistical design centering in [49,50]. The theory of SA methods is well established, so its application to yield optimization offers the theoretical background missing, e.g., in the heuristic methods of Section 5.3. As compared to the large-sample methods, the SA algorithms to be discussed use a few (or just one) randomly sampled points per iteration, which is compensated for by a large number of iterations exhibiting a trend toward the solution. The method tends to bring large initial improvements with a small number of circuit analyses, efficiently utilizing the high content of the deterministic information present at the beginning of optimization.

In 1951, in their pioneering work, Robins and Monro [33] proposed a scheme for finding a root of a regression function, which they named the stochastic approximation procedure. The problem was to find a zero of a function, whose ''noise corrupted'' values could be observed only, namely $G(x) = g(x) + \theta(x)$ where $g(x)$ (unknown) function of x, θ is a random variable, such that $E\{\theta\} = 0$ and $\text{var}\{\theta\} \leq L < \infty$. Therefore, a zero of the regression function $g(x) + E\{G(x)\} = 0$ was to be found. The SA algorithm proposed in [33] works as follows: given a point $x^{(k)}$, set the next point as $x^{(k+1)} = x^{(k)} - a^k G[x^{(k)}]$.

* Low-degree polynomials were used with very few terms, generated from a *stepwise regression* algorithm.

† This was the reason a high level of accuracy was required while calculating both yield and its derivatives using the method described in the previous section (and *consistency* between the yield and gradient estimators), otherwise, the deterministic optimization algorithm would often diverge, as observed in [13].

The sequence of $\{x^{(k)}\}$ points converges to the solution $\hat{x}$ under some conditions (one of them is, for instance, that a_k can change according to the harmonic sequence $\left\{\mathbf{1}, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots\right\}$). Assuming that $G(x) \equiv \xi^k$ is a "noise corrupted" observation of the gradient of a regression function $f(x) = E_\theta\{w(x, \theta)\}$, the algorithm can be used to find a stationary point (e.g., a maximum) of $f(x)$, since this is equivalent to finding $\hat{x}$ such the $E\{G(\hat{x})\} = E\{\xi(\hat{x})\} = \nabla_x Y(\hat{x}) = 0$. For yield optimization $f(x) \equiv Y(x) = E\{\phi(x, \theta)\}$. This simplest scheme that can be used, if the yield gradient estimator is available is

$$x^{k+1} = x^k + \tau_k \xi^k \tag{5.18}$$

where $\xi^k + \widehat{\nabla_x Y}(x^k)$ is an estimate of the yield gradient, based on one (or more) point θ^i sampled with the p.d.f. $f_\theta(\theta)$ and $\tau_k > 0$ is the step length coefficient selected such that the sequence: $\{\tau_k\} \to 0$, and $\sum_{k=0}^{\infty} \tau_k = \infty, \sum_{k=0}^{\infty} \tau_k^2 < \infty$ (e.g., the harmonic series $\left\{\mathbf{1}, \frac{1}{2}, \frac{1}{3}, \ldots\right\}$ fulfills these conditions). For the convergence with probability 1, it is also required that the conditional expectation $E\{\xi^k | x^1, x^2, \ldots, x^k\} = \nabla Y(x^k)$. The algorithm of Equation 5.18 is similar to the steepest ascent algorithms of nonlinear programming, so, it will be slowly convergent for ill-conditioned problems. A faster algorithm was introduced in [35] and used for yield optimization in [49,50]. It is based on the following iterations:

$$x^{k+1} = x^k + \tau_k d^k \tag{5.19}$$

$$d^k = (1 - \rho_k) d^{k-1} + \rho_k \xi^k, \quad 0 < \rho_k < 1 \tag{5.20}$$

where ξ_k is a (one- or more-point) estimator of $\nabla_x Y(x^k)$ and $\{\tau_k\} \to 0$, $\{\rho_k\} \to 0$ are nonnegative coefficients. $d^{(k)}$ is a convex combination of the previous (old) direction d^{k-1} and the new gradient estimate ξ_k, so the algorithm is an analog of a more efficient, conjugate gradient method. Equation 5.2 provides gradient averaging. The ρ_k coefficient controls the "memory" or "inertia" of the search direction d^k, as shown in Figure 5.15. If ρ_k is small, the "inertia" of the algorithm is large, i.e., the algorithm tends to follow the previous gradient directions. For convergence with probability 1, the same conditions must hold as those for Equation 5.18.

The coefficients τ_k and ρ_k automatically determined based on some heuristic statistical algorithms proposed in [35]. Several other enhancements were used to speed up the convergence of the algorithm, especially in its much refined version proposed in [35]. For solving the optimization problems, the one-or two-point yield gradient estimator is found from Equation 5.10 in general, and from Equation 5.13 or 5.14 for the normal p.d.f.

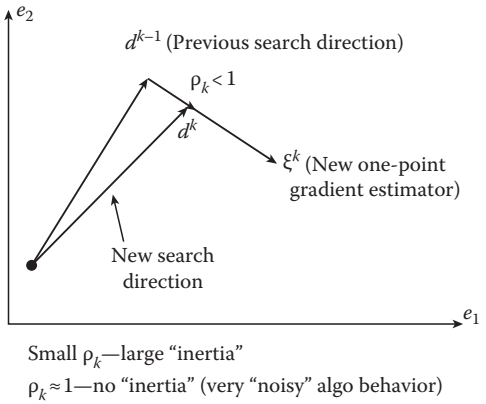


FIGURE 5.15 Illustration of the gradient averaging equation.

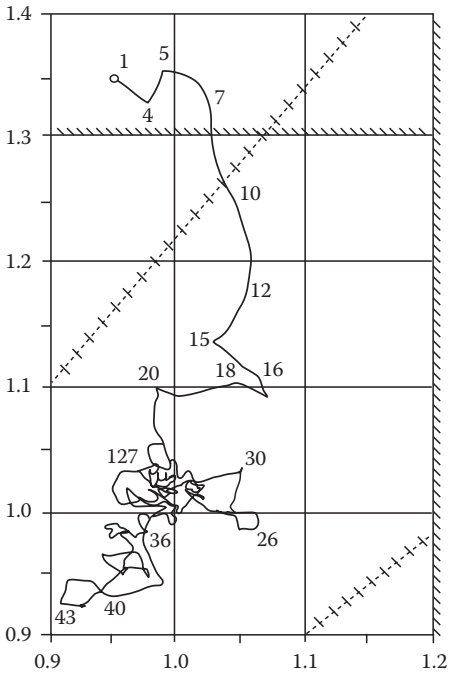


FIGURE 5.16 Typical trajectory of the stochastic approximation algorithm (Example 5.1).

Example 5.1

A simple two-dimensional case is considered in order to illustrate the algorithm properties. The acceptability region (for a voltage divider) [49] is defined in the two-dimensional space (e_1, e_2) by the inequalities:

$$0.45 \leq \frac{e_2}{e_1 + e_2} \leq 0.55 \quad 0 \leq e_1 \leq 1.2 \quad 0 \leq e_2 \leq 1.3 \tag{5.21}$$

where $e_i = x_i + \theta_i$ and the p.d.f. of θ_i is normal with $E\{\theta_1\} = E\{\theta_2\} = 0$ and $\text{Cov}\{\theta_1, \theta_2\} = 0$ (no correlations).

A typical algorithm trajectory is shown in Figure 5.16. The one-sample yield gradient formula (Equation 5.14) was used. The initial yield was low (7.4%). Very few (25–30) iterations (equal to the number of circuit analyses) were required to bring the minimal point close to the final solution. This is due to the fact that at the beginning of optimization, there is a high content of deterministic gradient information available even from a few sampled points, so the algorithm progress is fast. Close to the solution, however, the yield gradient estimator is very noisy, and the algorithm has to filter out the directional information out of noise, which takes the majority of the remaining iterations. After the total of 168 circuit analyses, the yield increases to 85.6%. Other starting points resulted in a similar algorithm behavior. It is also observed that the yield optimum is "flat" in the diagonal direction, reflected in random nominal point movement in this direction (iterations 30–43).

Example 5.2

Parametric yield for the Sallen–Key active filter of Figure 5.6a (often used in practice as a test circuit) with the specifications on its frequency response shown in Figure 5.6b was optimized (recall that the shape of the acceptability region for this case was very complicated, containing internal nonfeasible regions

["holes"]). All R, C variables were assumed designable. The relative standard deviations were assumed equal to 1% for each element, the p.d.f. was normal with 0.7 correlation coefficient between the like elements (R's or C's) and zero correlations between different elements. The initial yield was 6.61%. Using the SA algorithm as in the previous example, 50 iterations (equal to the number of circuit analyses) brought yield to 46.2%; the next 50 iterations to 60%, while the remaining 132 iterations increased yield to only 60.4% (again the initial convergence was very fast). The algorithm compared favorably with the Hessian matrix-based, large-sample method discussed previously [51], which required about 400 circuit analyses to obtain the same yield level, and whose initial convergence was also much slower. It has to be stressed, however, that the results obtained are statistical in nature and it is difficult to draw strong general conclusions. One observation (confirmed also by other authors) is that the SA-type algorithms provide, in general, fast initial convergence into the neighborhood of the optimal solution, as it was shown in the examples just investigated.

5.4.7 Small-Sample Stochastic Approximation Methods for Integrated Circuits

The methods of yield gradient estimation for discrete circuits cannot be used for ICs because of the form of the $e = e(x, \theta)$ transformation, as discussed at the beginning of Section 5.4. Previously in this section, a complicated algorithm was described for gradient estimation and yield optimization in such situations. In this section, a simple method based on random perturbations in x-space, proposed in [47,48], is described. It is useful in its own merit, but especially in those cases, where the conditions for the application of the yield gradient formula (Equation 5.17) are hard to meet, namely: the cost of constructing the approximating functions $\hat{y} = \hat{y}(x, \theta)$ for fixed x is high and calculating the gradient of y w.r.t. x is also expensive (which is the case, e.g., if the number of important x and θ parameters is large), and the analytical form of $f_\theta(\theta)$ is not available (as it is required in Equation 5.17).

The method applications go beyond yield optimization: a general problem is to find a minimum of a general regression function $f(x) = E_\theta\{w(x, \theta)\}$, using the SA method, in the case where the gradient estimator of $f(x)$ is not directly available. Several methods have been proposed to estimate the gradient indirectly, all based on adding some extra perturbations to x parameters. Depending on their nature, size, and the way the perturbations are changed during the optimization process, different interpretations result, and different problems can be solved. In the simplest case, some extra deterministic perturbations (usually double-sided) are added individually to each x_k (one-at-a-time), while random sampling is performed in the θ-space, and the estimator of the derivative of $f(x)$ w.r.t. x_k is estimated from the difference formula $\hat{\xi}_k = \widehat{\partial f(x)}/\partial x_k = (1/N)\sum_{i=1}^{N}\left\{\left[w\left(x + a_k e_k, \theta_1^i\right) - w\left(x - a_k e_k, \theta_2^i\right)\right]/(2a_k)\right\}$, where $a_k > 0$ is the size of the perturbation step and e_k is the unit vector along the e_k coordinates axis. Usually, $\theta_1^i \equiv \theta_2^1$ to reduce the variance of the estimator. Normally, $N = 1$. Other approaches use random direction derivative estimation by sampling points randomly on: a unit sphere of radius a [34], randomly at the vertices of a hypercube in the x-space [38], or at the points generated by orthogonal arrays [53], commonly used in the design of experiments. Yet another approach, dealing with nondifferentiable p.d.f.s was proposed in [55]. In what follows, the random perturbation approach resulting in convolution function smoothing is described for a more general case, where a *global* rather than a local minimum of $f(x)$ is to be found.

The multi-extremal regression function $f(x)$ defined previously, can be considered a superposition of an uni-extremal function (i.e., having just one minimum) and other multi-extremal functions that add some deterministic "noise" to the uni-extremal function (which itself has also some "statistical noise"—due to θ—superimposed on it). The objective of convolution smoothing can be visualized as "filtering out" but types of noise and performing minimization on the "smoothed" uni-extremal function (or on a family of these functions), in order to reach the global minimum. Since the minimum of the smoothed uni-extremal function does not, in general, coincide with the global function minimum, a sequence of minimization runs is required with the amount of smoothing eventually reduced to zero in the

neighborhood of the global minimum. The smoothing process is performed by averaging $f(x)$ over some region of the n-*dimensional* parameter space x using a proper weighting (or smoothing) function $\hat{h}(x)$, defined below. Let the n-dimensional vector η denote a vector of random perturbations; it is added to x to create the convolution function [34]:

$$\begin{aligned}\tilde{f}(x,\beta) &= \int_{R^n} \hat{h}(\eta,\beta) f(x-\eta)\mathrm{d}\eta \\ &= \int_{R^n} \hat{h}(x-\eta,\beta) f(\eta)\mathrm{d}\eta = E_\eta\{f(x-\eta)\}\end{aligned} \tag{5.22}$$

where $\hat{f}(x,\beta)$ is the smoothed approximation to the original multi-extremal function $f(x)$, and the kernel function $\hat{h}(\eta, \beta)$ is the p.d.f. used to sample η. Note that $f(x, \beta)$ can be interpreted as an averaged version of $f(x)$ weighted by $\hat{h}(\eta, \beta)$. Parameter β controls the dispersion of $\hat{h}$, i.e., the degree of $f(x)$ smoothing (e.g., β can control the standard deviations of $\eta_1 \ldots \eta_n$). $E_\eta\{f(x-\eta)\}$ is the expectation with respect to the random variable η. Therefore, an unbiased estimator $\hat{f}(x,\beta)$ of $\tilde{f}(x,\beta)$ is the average: $\hat{f}(x,\beta) = (1/N)\sum_{i-1}^{N} f(x-\eta^i)$, where η is sampled with the p.d.f. $\hat{h}$ $(\eta-\beta)$. The kernel function $\hat{h}(\eta, \beta)$ should have certain properties discussed in [34], fulfilled by several p.d.f.s, e.g., the Gaussian and uniform. For the function $f(x) = x^4 - 16x^2 + 5x$, which has two distinct minima, smoothed functionals, obtained using Equation 5.22, are plotted in Figure 5.17 for different values of $\beta \to 0$, for (a) Gaussian and (b) uniform kernels. As seen, smoothing is able to eliminate the local minima of $\tilde{f}(x, \beta)$ if β is sufficiently large. If $\beta \to 0$, then $\tilde{f}(x, \beta) \to f(x)$.

The objective now is to solve the following optimization problem: minimize the smoothed functional $\tilde{f}(x, \beta)$ with $\beta \to 0$ as $x \to \hat{x}$, where $\hat{x}$ is the global minimum of the original function $f(x)$. The modified optimization problem can be written as minimize $\tilde{f}(x, \beta)$ w.r.t. x, with $\beta \to 0$ as $x \to \hat{x}$. Differentiating Equation 5.22 and using variable substitution, the gradient formula is obtained:

$$\nabla_x \tilde{f}(x,\beta) = \int_{R^n} \nabla_\eta \hat{h}(\eta,\beta) f(x-\eta)\mathrm{d}\eta = \frac{1}{\beta}\int_{R^n} \nabla_\eta h(\eta) f(x-\beta\eta)\mathrm{d}\eta, \tag{5.23}$$

where $\hat{h}(\cdot)$ is as defined previously, and $h(y)$ is a normalized version of $\hat{h}(y)$ (obtained if $\beta = 1$). For normalized multinormal p.d.f. with zero correlations, the gradient of $\tilde{f}(x, \beta)$ is

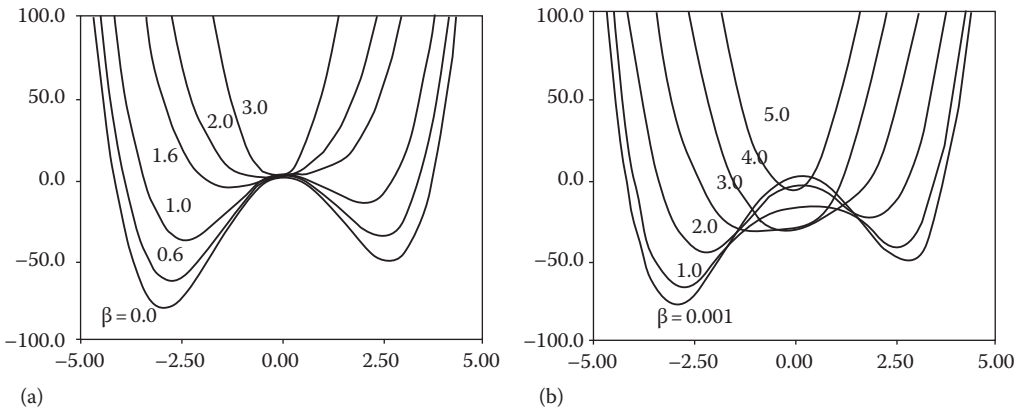


FIGURE 5.17 Smoothed functional $f(x, \beta)$ for different β's using (a) Gaussian kernel and (b) uniform kernel.

$$\nabla_x \tilde{f}(x,\beta) = \frac{-1}{\beta}\int_{R^n} \eta f(x-\beta\eta)h(\eta)d\eta = \frac{-1}{\beta}E_\eta\{\eta f(x-\beta\eta)\}$$
$$= \frac{-1}{\beta}E_{\eta,\theta}\{\eta w(x-\beta\eta,\theta)\} \tag{5.24}$$

where sampling is performed in x-space with the p.d.f. $h(\eta)$, and in θ-space with the p.d.f. $f_\theta(\theta)$; $E_{\eta,\theta}$ denotes expectation w.r.t. *both* η and θ, and it was taken into account for $f(x)$ is a noise corrupted version of $w(x, \theta)$ i.e., $f(x) = E_\theta\{w(x, \theta)\}$. The unbiased single-sided gradient estimator is therefore

$$\hat{\nabla}_x \tilde{f}(x,\beta) = \frac{-1}{\beta}\frac{1}{N}\sum_{i=1}^{N}\eta^i w(x-\beta\eta^i,\theta^i) \tag{5.25}$$

In practice, a double-sided estimator [34] of smaller variance is used

$$\hat{\nabla}_x \tilde{f}(x,\beta) = \frac{1}{2\beta}\frac{1}{N}\sum_{i=1}^{N}\eta^i\left[w(x+\beta\eta^i,\theta_2^i)\right]. \tag{5.26}$$

Normally, $N = 1$ for best overall efficiency. Statistical properties of these two estimators (such as their variability) were studied in [52]. To reduce variability, the same $\theta_1^i = \theta_2^i$ are usually used in Equation 5.26 for positive and negative $\beta\eta^i$ perturbations. For yield optimization, $w(\cdot)$ is simply replaced by the indicator function $\phi(\cdot)$. For multi-extremal problems, β values should be originally relatively large and then systematically reduced to some small number rather than to zero. For single-extremal problems (this might be the case for yield optimization) it is often sufficient to perform just a *single* optimization with a relatively small value of β, as it was done in the examples to follow.

5.4.8 Case Study: Process Optimization for Manufacturing Yield Enhancement

The object of the work presented in [48] was to investigate how to modify the MOS control process parameters together with a simultaneous adjustment of transistor widths and lengths to maximize parametric yield.* To make it possible, both process/device simulators (such as FABRICS, SUPREM, PISCES, etc.) and a circuit simulator must be used. In what follows FABRICS [12,22,24] is used as a process/device simulator. IC technological process parameters are statistical in nature, but variations of some of the parameters (e.g., times of operations, implant doses, etc.) might have small relative variations, and some parameters are common to several transistors on the chip. Because of that, the transformation $e = e\ (x, \theta)$ (where θ are now process related random variables), is such that standard methods of yield optimization developed for discrete circuits cannot be used. Let $Z_1 = z_1 + \xi_1$ be the process control parameters (doses, times, temperatures), where ξ_1 are random parameter variations, and z_1 are deterministic designable parameters. Let $Z_2 = z_2 + \xi_2$ be the designable layout dimensions, where z_2 are designable and ξ_2 are random variations (common to several transistors on a chip). $P = p + \psi$ are process physical parameters (random, nondesignable) such as diffusivities, impurity concentrations, etc. (as above, p is the nominal value and ψ is random). All random perturbations are collected into the vector of random parameters θ, called also the vector of process disturbances: $\theta = (\xi_1, \xi_2, \psi)$. The vector of designable parameters $x = (z_1, z_2)$ is composed of the process z_1 and layout z_2 designable parameters. Therefore, x and θ are in different spaces (subspaces). There are also other difficulties: the analytical form of the p.d.f. of θ is most often not known, since θ parameters are hierarchically generated from a

* This might be important, e.g., for the process refinement and IC cell *redesign*.

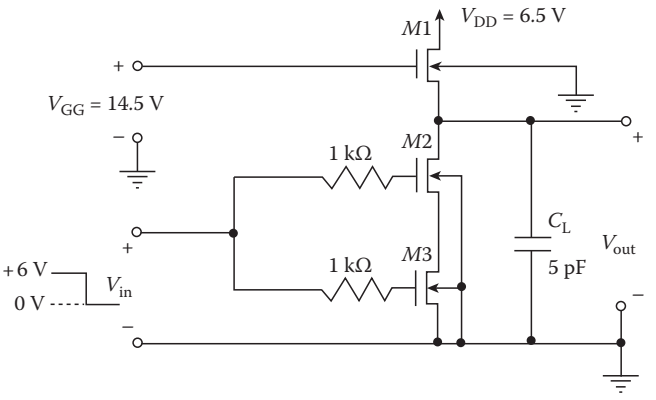


FIGURE 5.18 NMOS NAND gate of Example 5.3.

numerical procedure [12], the derivatives of the performances y w.r.t. x and θ are not known from FABRICS and can only be estimated by finite differences, and the θ and x spaces are very large (see below). So, creating approximation and/or finding derivatives using finite differences is expensive. Because of these difficulties, the smoothed-functional approach discussed in this section will be used, as shown in the following example.

Example 5.3

The objective is to maximize parametric yield for the NMOS NAND gate shown in Figure 5.18 [48], by automated adjustment of process and layout parameters. Specifications are: $V_0 = V_{out}$ $(t = 0) \leq 0.7$ V, V_{out} $(t_1 = 50$ ns$) > 6.14$ V, circuit area ≤ 2500 μm^2. There are 45 designable parameters: all 39 technological process parameters, 6 transistor dimensions, and about 40 noise parameters, so it is a large problem suitable for the use of the SA-based random perturbation method described above.

The initial yield was $Y = 20\%$. After the first optimization using the method of random perturbations with 2% relative perturbations of each of the designable parameters involving the total of 110 FABRICS/SPICE analyses, yield increased to 100% and the nominal area decreased to 2138 μm^2. Then, specs were tightened to V_{out} $(t_2 = 28$ ns$) > 6.14$ V with the same constraints on V_0 and area, causing yield to drop to 10.1%. After 60 FABRICS/SPICE analyses, using the perturbation method, yield increased to: $Y = 92\%$ and area $= 2188$ μm^2. These much-improved results produced the nominal circuit responses shown in Figure 5.19. Several technological process parameters were changed during optimization in the range between 0.1% and 17%: times of oxidation, annealing, drive-in, partial pressure of oxygen, and others, while the transistor dimensions changed in the range between 0.8% and 6.3%. The cost of obtaining these results was quite reasonable: the total of 170 FABRICS/SPICE analyses, despite the large number of optimized and noise parameters. Other examples are discussed in [48].

5.4.9 Generalized Formulation of Yield, Variability, and Taguchi Circuit Optimization Problems

Parametric yield is not the only criterion that should be considered during statistical circuit design. Equally, and often more important is minimization of performance variability caused by various manufacturing and environmental disturbances. Variability minimization has been an important issue in circuit design for many years [17] (see [17] for earlier reference). It leads (indirectly) to parametric yield improvement. Circuits characterized by low performance variability are regarded as high quality products. Most recently, variability minimization has been reintroduced into practical industrial design

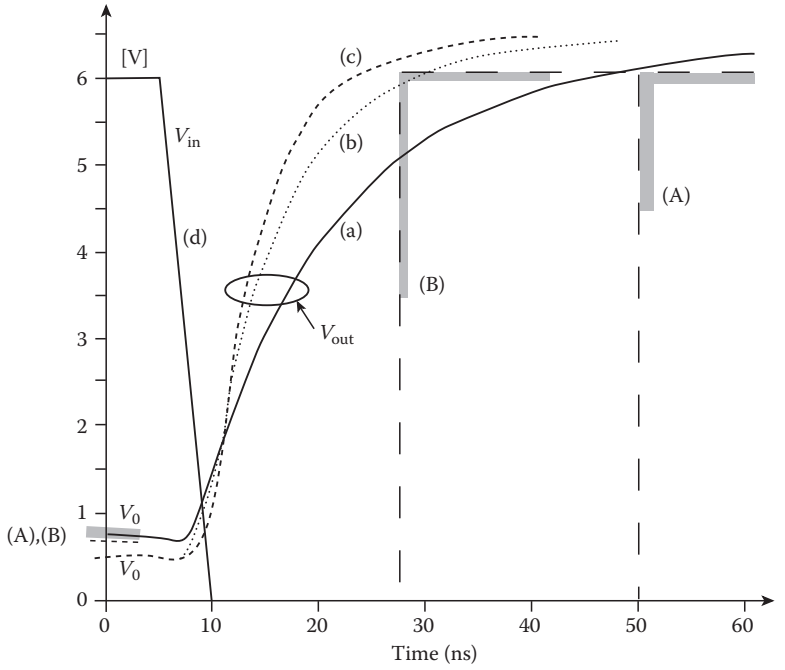


FIGURE 5.19 Nominal transient responses for Example 5.3: (a) initial, (b) after first optimization, and (c) after tightening the specs and second optimization.

due to the work of Taguchi [30]. He has successfully popularized the notion of ''Off-Line Quality Control'' through an intensive practical implementation of his strategy of designing products with low performance variability, tuned to the ''target'' values S_j^T of the performance functions y_j.

In what follows, a generalized approach proposed in [44] is discussed, in which a broad range of various problems, including yield optimization and Taguchi's variability minimization as special cases can be solved, using the SA approach and the general gradient formulas developed previously in this section.

It is convenient to introduce the M-dimensional vector $g(e)$ of scaled constraints, composed of the vectors $g^L(e)$, and $g^U(e)$, defined as

$$g_k^L(e) = \frac{S_k^T - y_k(e)}{S_k^T - S_k^L}, \quad k = 1, \ldots, M_L \tag{5.27}$$

$$g_t^U(e) = \frac{y_t(e) - S_t^T}{S_t^U - S_t^T}, \quad t = 1, \ldots, M_U \tag{5.28}$$

where $e = e(x, \theta)$, $M = M_L + M_U \leq 2$ m (note that in general $M \leq 2$ m, since some of the lower or upper specs might not be defined). These constraints are linear functions of $y_i(\cdot)$; they have important properties: $g^L = g^U = 0$ if $y = S^T$, $g^L = 1$ if $y = S^L$, and $g^U = 1$ if $y = S^U$. For $S^L < y < S^U$ and $y \neq S^T$, either g^L or g^U is greater than zero, but never both.

For any ''good'' design, we would like to make each of the normalized constraints introduced previously equal to zero, which might not be possible due to various existing design trade-offs. Therefore, this is a typical example of a multiobjective optimization problem. The proposed scaling helps to compare various trade-off situations. The Taguchi ''on-target'' design with variability minimization is formulated as follows (for a single performance function $y(e)$)

$$\text{minimize}\{M_{\text{TAG}}(x) = E_\theta\left\{\left[y(e(x,\theta)) - S^{\text{T}}\right]^2\right\} = var\{y(x)\} + \left[\bar{y}(x) - S^{\text{T}}\right]^2\right\} \tag{5.29}$$

where

$\bar{y}(x)$ is the mean (average) value of y (w.r.t θ)

M_{TAG} is called the "loss" function (in actual implementations Taguchi uses related but different performance statistics)

Generalization of Equation 5.29 to several objectives can be done in many different ways. One possible approach is to introduce $u(e) = \max_{s=1,\ldots,M}\{g_s(e)\}$ and define the generalized loss function (GLF) as

$$M_{\text{GLF}}(x) = E_\theta\left\{[u(e(x,\theta))]^2\right\} \tag{5.30}$$

This generalization is meaningful, since $u(e) = \max_{s=1,\text{K},M}\{g_s(\text{e})\}$ is a scalar, and for the proposed scaling either $u(\cdot) > 0$, or it is equal to zero if all the performances assume their target values S_i^{T}. Since Equations 5.27 and 5.28 are less or equal to 1 any time $S_i^{\text{T}} \le y_i\ (\text{e}) \le S_i^{\text{U}}$, then $e = e(x,\ \theta)$ belongs to A (the acceptability region in the e-space), if $u(e) < 1$. Let us use the complementary indicator function $\phi_{\text{F}}[u\ (\text{e})] = 1$, if $u(e) \ge 1$, i.e., $e \notin A$ (or $\theta \notin A_\theta(\text{x})$) (failure) and equal to zero otherwise. Maximization of the parametric yield $Y(x)$ is equivalent to the minimization of the probability of failures $F(x) = 1 - Y(x)$, which can be formulated as the following minimization process, w.r.t. x:

$$\text{minimize}\{F(x) = P\{\theta \notin A_\theta(x)\} = \int_{R^t} \phi_{\text{F}}(u(x,\theta)) f_\theta(\theta)\text{d}\theta = E_\theta[\phi_{\text{F}}(u(x,\theta))]\Big\} \tag{5.31}$$

The scalar "step" function $\phi_{\text{F}}(\cdot)$ (where the argument is also scalar function $u(e) = \max_{s=1,\text{K},M}\{g_s(\text{e})\}$) can now be generalized into a scalar weight function $w(\cdot)$, in the same spirit as in the Zadeh fuzzy set theory [62,63]. For further generalization, the original p.d.f. $f_\theta(\theta)$ used in Equation 5.31 is parameterized, multiplying θ by the smoothing parameters β to control the dispersion of e, which leads to the following optimization problem, utilizing a generalized measure $M_w(\text{x}, \beta)$

$$\text{minimize}\{M_w(x,\beta) = E_\theta\{w[u(e(x,\beta\theta))]\}\} \tag{5.32}$$

where $u(e(x,\ \beta\theta)) = \max_{s=1,\ldots,M}\{g_s(e(x,\ \beta\theta))\}$. If $0 \le w(\cdot) \le 1$, $M_w(\cdot)$ corresponds to the probability measure of fuzzy event introduced by Zadeh [63]. The choice of $w(\cdot)$ and β leads to different optimization problems and different algorithms for yield/variability optimization. The standard yield optimization problem results if $w(\alpha) = \phi_{\text{F}}(\alpha)$ and $\beta = 1$ [Equation 5.31] (Figure 5.20a). Yield can be also approximately optimized indirectly, using a "smoothed" (e.g., sigmoidal) membership function $w(\alpha)$ (Figure 5.20e). For variability minimization, we have a whole family of possible approaches: (1) The generalized Taguchi approach with $w(\alpha) = \alpha^2$ (see Equation 5.30) and $\beta = 1$ (see Figure 5.20c). (2) If $w(\alpha) = \alpha$ and $\beta = 1$ is kept constant, we obtain a statistical mini–max problem, since the expected value of the max function is used; this formulation will also lead to performance variability reduction. (3) If $w(\alpha)$ is piecewise constant (Figure 5.20d), the approach is equivalent to the income index maximization with separate quality classes, introduced in [28], and successfully used there for the increase of the percentage of circuits belonging to the best classes (i.e., those characterized by small values of $u(e)$); this approach also reduces variability of circuit performance functions. For all the cases, optimization is performed using the proposed SA approach. The smoothed functional gradient formulas (Equation 5.25 and 5.26)

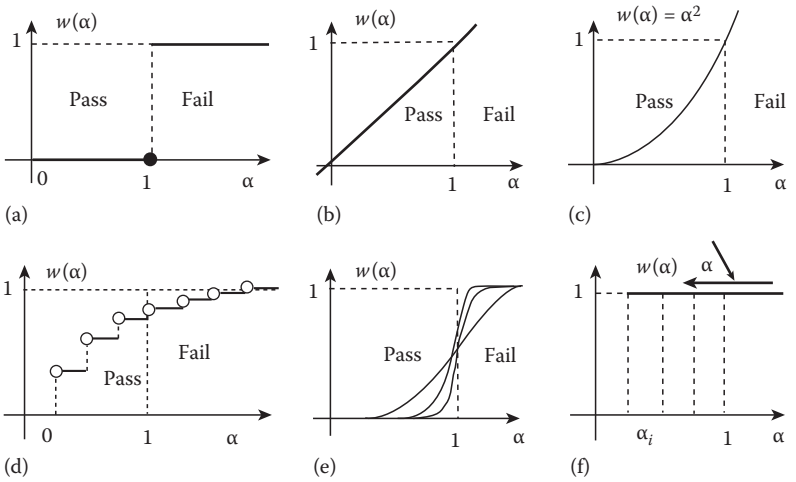


FIGURE 5.20 Various cases of the weight function $w(\alpha)$.

have to be used, in general, since the ''discrete-circuit'' type gradient formulas are valid only for a limited number of situations.

Example 5.4: Performance Variability Minimization for a MOSFET-C Filter

The objective of this example [31] was to reduce performance variability and to tune to target values the performances of the MOSFET-C filter in Figure 5.21. The MOSFETS are used in groups composed of four

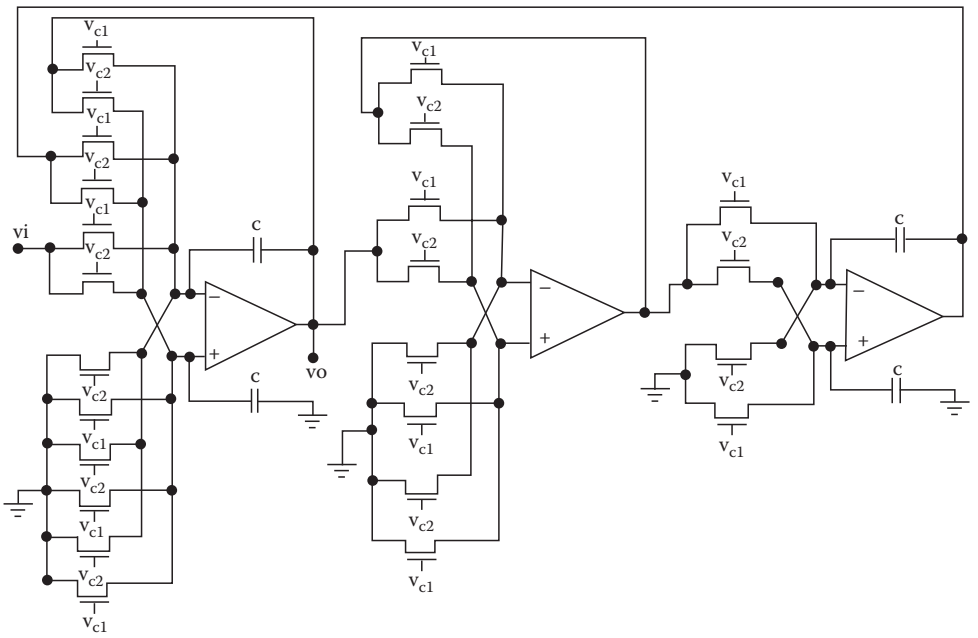


FIGURE 5.21 MOSFET-C bandpass filter optimized in Example 5.4.

transistors implementing equivalent resistors with improved linearity. Design specifications are f_0, the center frequency, with the S^L, S^T, S^U values specified as {45, 50, 55} kHz; H_0, the voltage gain at f_0, with the specs {15, 20, 25} dB; and Q, the pole quality factor, with the specs {8, 10, 12}. The Taguchi-like "on-target" design and tuning was performed by minimizing the generalized measure Equation 5.32 with $w(x) = x^2$, using the SA approach with convolution smoothing, since the designable and random parameters are in different (sub)spaces. The circuit has the total number of 90 transistors so its direct optimization using transistor-level simulation would be too costly. Therefore, in [31] the operational amplifiers (op-amps) shown in Figure 5.21 were modeled by a statistical macromodel, representing the most relevant op-amp characteristics: DC gain A_0, output resistance R_0, and the −20 dB/dec frequency roll-off. Transistor model parameters were characterized by a statistical model developed in [8], based on six common factors (see Section 5.3): t_{ox} (oxide thickness), ΔL_n, ΔL_p (length reduction), $N_{SUB,n}$, $N_{SUB,p}$ (substrate doping), and x_{jp} (junction depth). All other transistor model parameters were calculated from the six common factors using second-order regression formulas developed in [8]. The major difficulty was to create statistical models of the op-amp macromodel parameters A_0, R_0, and f_{3dB} as the functions of the six common factors listed previously. A special extraction procedure (similar in principle to the model of factor analysis [20]) was developed and the relevant models created. Perfect matching between the transistor model parameter of the 22 transistors of each of the op-amps was assumed for simplicity* (correlational dependencies between individual parameters for individual transistors were maintained using the statistical mode of [8], described earlier). Moreover, perfect matching between the three macromodels (for the three op-amps) was also assumed. Mismatches between the threshold voltages and K_p (gain) coefficients of all the transistors in Figure 5.21 were taken into account, introducing 48 additional noise parameters. So, including the 6 global noise parameters discussed previously, the overall number of $\boldsymbol{\theta}$ parameters was 54. Moreover, it was found that even if a large number of noise parameters had small individual effect, their total contributions was significant, so, no meaningful reduction of the dimension of the $\boldsymbol{\theta}$-space was justified. Because of that, it was difficult to use approximating models in the $\boldsymbol{\theta}$-space, and even more difficult in the joint $(x, \boldsymbol{\theta})$ space of $7 + 54 = 61$ parameters. Therefore, no approximation was used.

The Monte Carlo studies showed that the proposed statistical macromodel provided quite reasonably statistical accuracy for f_0 (<1% errors for both the mean and the standard deviation), and <9.5% errors for both, H_0 and Q. The macromodel-based analysis was about 30 times faster than using the full device-level SPICE analysis. Twenty-five designable parameters were selected, including transistor channel length of the "resistor-simulating transistors," four capacitors, and one of the gate voltages. Due to the designable parameter tracking for transistor quadruples, the actual vector of optimized parameters was reduced to 7: $x = (V_{Gs}, C_1, C_2, L_2, L_3, L_4, L_5)$, where V_{Gs} denotes the gate voltage, and C_i, L_j are capacitor and channel length values, respectively. The original yield was 43.5% and relative standard deviations for H_0, f_0, and Q were reduced to 3.98%, 7.24%, and 22.06%, respectively, and all nominal values were close to their target values. After SA optimization with convolution smoothing involving 5% perturbations for each of the x parameters, and the total of about 240 circuit analyses, yield increased to 85.0%, and H_0, f_0, and Q relative standard deviations were reduced to 3.87%, 6.35%, and 13.91%, respectively. Therefore, the largest variability reduction (for Q) was about 37%, with simultaneous significant yield increase.

The preceding example demonstrates a typical approach that has to be taken in the case of large analog circuits: the circuit has to be (hierarchically) macromodeled first† and suitable *statistical* macromodels have to be created, including mismatch modeling.

5.5 Conclusion

Several different techniques of statistical yield optimization were presented. Other approaches to yield estimation and optimization can be found, e.g., in [21] (process optimization), and in [15,57,59,60,61]. It was also shown that yield, variability minimization, Toguchi design, and other approaches can be generalized into one methodology called design for quality‡ (DFQ). DFQ is a quickly growing area [30],

* A more sophisticated model, taking the mismatches into account, was later proposed in [32].

† Behavioral models can also be used.

‡ Other approaches to yield generalization/Design for Quality were presented in [13,28,58].

where the major objective is not to maximize the parametric yield as the only design criterion (yield is often only vaguely defined), but to minimize the performance variability around the designer specified target performance values. This has been a subject of research for many years using a sensitivity-based approach, and most recently, using the Taguchi methodology, based on some of the techniques of Design of Experiments rather than on the use of sensitivities. To increase the efficiency of the (mostly manual) Taguchi techniques [30], some automated methods started to appear, such as the generalized methodology described previously, or the automated approach based on capability indices C_p/C_{pk} (used extensively in process quality control) proposed in [41].

Statistical design optimization is still an active research area, but several mature techniques have been already developed and practically applied to sophisticated industrial IC design. This is of great importance to the overall manufacturing cost reduction, circuit quality improvement, and shortening of the overall IC design cycle.

References

1. H. L. Abdel-Malek and J. W. Bandler, Yield optimization for arbitrary statistical distributions: Part I—Theory, *IEEE Trans. Circuits Syst.*, CAS-27(4): 245–253, April 1980.
2. K. J. Antreich and R. K. Koblitz, Design centering by yield prediction, *IEEE Trans. Circuits Syst.*, CAS-29: 88–95, February 1982.
3. P. Balaban and J. J. Golembeski, Statistical analysis for practical circuit design, *IEEE Trans. Circuits Syst.*, CAS-22(2): 100–108, February 1975.
4. J. W. Bandler, Optimization of design tolerance using nonlinear programming, *J. Optimization Theory Appl.*, 14: 99, 1974; also in *Proceedings of the Princeton Conference on Information Science and Systems*, p. 655, Princeton, NJ, February 1972.
5. J. W. Bandler and H. L. Abdel-Malek, Optical centering, tolerancing and yield determination via updated approximations and cuts, *IEEE Trans. Circuits Syst.*, CAS-25: 853–871, 1978.
6. J. W. Bandler, P. C. Liu, and H. Tromp, A nonlinear programming approach to optimal design centering, tolerancing and tuning, *IEEE Trans. Circuits Syst.*, CAS-23: 155, March 1976.
7. P. W. Becker and F. Jensen, *Design of Systems and Circuits for Maximum Reliability or Maximum Production Yield*, New York: McGraw-Hill, 1977.
8. J. Chen and M. A. Styblinski, A systematic approach of statistical modeling and its application to CMOS circuits, in *Proceedings of the IEEE International Symposium on Circuits and Systems '93*, pp. 1805–1808, Chicago, IL, May 1993.
9. P. Cox, P. Yang, S. S. Mahant-Shetti, and P. Chatterjee, Statistical modeling for efficient parametric yield estimation of MOS VLSI circuits, *IEEE Trans. Electron Devices*, ED-32: 471–478, February 1985.
10. S. W. Director and G. D. Hatchel, The simplicial approximation to design centering, *IEEE Trans. Circuits Syst.*, CAS-24(7): 363–372, July 1977.
11. S. W. Director and G. D. Hatchel, A point basis for statistical design, in *Proceedings of the IEEE International Symposium on Circuits and Systems*, New York, 1978.
12. S. W. Director, W. Maly, and A. J. Strojwas, *VLSI Design for Manufacturing: Yield Enhancement,* Boston, MA: Kluwer Academic, 1990.
13. P. Feldman and S. W. Director, Integrated circuit quality optimization using surface integrals, *IEEE Trans. Comput. Aided Des.*, 12(12): 1868–1879, December 1993.
14. P. H. Gill, W. Murray, and M. H. Wright, *Practical Optimizations*, San Diego, CA: Academic Press, 1981.
15. D. E. Hocevar, P. F. Cox, and P. Yang, Parametric yield optimization for MOS circuit blocks, *IEEE Trans. Comput. Aided Des.*, 7(6): 645–658, June 1988.
16. D. E. Hocevar, M. R. Lightner, and T. N. Trick, A study of variance reduction techniques of estimating circuit yields, *IEEE Trans. Comput. Aided Des.*, CAD-2(3): 180–192, July 1983.

17. A. Ilumoka, N. Maratos, and R. Spence, Variability reduction: Statistically based algorithms for reduction of performance variability of electrical circuits, *IEEE Proc.*, 129, Part G(4): 169–180, August 1982.
18. G. Kjellstrom and L. Taxen, Stochastic optimization in system design, *IEEE Trans. Circuits Syst.*, CAS-28: 702–715, July 1981.
19. G. Kjellstrom, L. Taxen, and P. O. Lindberg, Discrete optimization of digital filter using Gaussian adaptation and quadratic function minimization, *IEEE Trans. Circuits Syst.*, CAS-34(10): 1238–1242, October 1987.
20. D. N. Lawley and A. E. Maxwell, *Factor Analysis as a Statistical Method*, New York: Elsevier, 1971.
21. K. K. Low and S. W. Director, An efficient methodology for building macromodels of IC fabrication processes, *IEEE Trans. Comput. Aided Des.*, 8(12): 1299–1313, December 1989.
22. W. Maly and A. J. Strojwas, Statistical simulation of IC manufacturing process, *IEEE Trans. Comput. Aided Des.*, Vol. CAD-1: 120–131, July 1982.
23. C. Michael and M. Ismail, *Statistical Modeling for Computer-Aided Design of MOS VLSI Circuits*, Boston, MA: Kluwer Academic, 1993.
24. S. R. Nassif, A. J. Strojwas, and S.W. Director, FABRICS II, *IEEE Trans. Comput. Aided Des.*, CAD-3: 40–46, January 1984.
25. J. Ogrodzki, L. Opalski, and M. A. Styblinksi, Acceptability regions for a class of linear networks, in *Proceedings of the IEEE International Symposium on Circuits and Systems*, Houston, TX, pp. 187–190, May 1980.
26. J. Ogrodzki and M. A. Styblinski, Optimal tolerancing, centering and yield optimization by one-dimensional orthogonal search (ODOS) technique, in *Proceedings of the European Conference on Circuit Theory and Design (ECCTD)*, Vol. 2, pp. 480–485, Warsaw, Poland, September 1980.
27. L. Opalski, M. A. Styblinski, and J. Ogrodzki, An orthogonal search approximation to acceptability regions and its application to tolerance problems, in *Proceedings of the Conference on SPACECAD*, Bologna, Italy, September 1979.
28. L. J. Opalski and M. A. Styblinski, Generalization of yield optimization problem: Maximum income approach, *IEEE Trans. Comput. Aided Des. ICAS*, CAD-5(2): 346–360, April 1986.
29. M. J. M. Pelgrom, A. C. J. Duinmaijer, and A. P. G. Welbers, Matching properties of MOS transistors, *IEEE J. Solid State Circuits*, 24: 1334–1362, October 1989.
30. M. S. Phadke, *Quality Engineering Using Robust Design*, Englewood Cliff, NJ: Prentice Hall, 1989.
31. M. Qu and M. A. Styblinski, Hierarchical approach to statistical performance improvement of CMOS analog circuits, in *SRC TECHCON'93*, Atlanta, GA, pp. 1809–1812, September 1993.
32. M. Qu and M. A. Styblinski, Statistical characterization and modeling of analog functional blocks, in *Proceedings of the IEEE International Symposium on Circuits and Systems*, London, May–June 1994.
33. H. Robins and S. Monro, A stochastic approximation method, *Annal. Math. Stat.*, 22: 400–407, 1951.
34. R. Y. Rubinstein, *Simulation and the Monte Carlo Method*, New York: John Wiley & Sons, 1981.
35. A. Ruszczynski and W. Syski, Stochastic approximation algorithm with gradient averaging for constrained problems, *IEEE Trans. Autom. Control*, AC28: 1097–1105, December 1983.
36. K. Singhal and J. F. Pinel, Statistical design centering and tolerancing using parametric sampling, *IEEE Trans. Circuits Syst.*, CAS-28: 692–702, July 1981.
37. R. S. Soin and R. Spence, Statistical exploration approach to design centering, in *Proc. Inst. Elect. Eng.*, 127, Part G: 260–263, 1980.
38. J. C. Spall, Multivariate stochastic approximation using a simultaneous perturbation gradient approximation, *IEEE Trans. Autom. Control*, 37(3): 332–341, 1992.
39. R. Spence and R. S. Soin, *Tolerance Design of Electronic Circuits*, Electronic Systems Engineering Series, Reading, MA: Addison-Wesley, 1988.
40. W. Strasz and M. A. Styblinski, A second derivative Monte Carlo optimization of the production yield, in *Proceedings of the European Conference on Circuit Theory and Design*, Vol. 2, pp. 121–131, Warsaw, Poland, September 1980.

41. M. A. Styblinski and S. A. Aftab, IC variability minimization using a new C_p and C_{pk} based variability/performance measure, in *Proceedings of the IEEE International Symposium on Circuits and Systems*, London, May–June 149–152, 1994.
42. M. A. Styblinski, Estimation of yield and its derivatives by Monte Carlo sampling and numerical integration in orthogonal subspaces, in *Proceedings of the European Conference on Circuit Theory and Design (ECCTD)*, Vol. 2, pp. 474–479, Warsaw, Poland, September 1980.
43. M. A. Styblinski, Problems of yield gradient estimation for truncated probability density functions, *IEEE Trans. Comput. Aided Des. ICAS*, CAD-5(1): 30–38, January 1986 (special issue on statistical design of VLSI circuits).
44. M. A. Styblinski, Generalized formulation of yield, variability, minimax and Taguchi circuit optimization problems, *Microelectron Reliability*, 34(1): 31–37, 1994.
45. M. A. Styblinski and S. A. Aftab, Combination of interpolation and self organizing approximation techniques—A new approach to circuit performance modeling, *IEEE Trans. Comput. Aided Des.*, 12(11): 1775–1785, November 1993.
46. M. A. Styblinski, J. Ogrodzki, L. Opalski, and W. Strasz, New methods of yield estimation and optimization and their application to practical problems (invited paper), in *Proceedings of the International Symposium on Circuits and Systems*, Chicago, IL, 1981.
47. M. A. Styblinski and L. J. Opalski, A random perturbation method for IC yield optimization with deterministic process parameters, in *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 977–980, Montreal, Canada, May 7–10, 1984.
48. M. A. Styblinski and L. J. Opalski, Algorithms and software tools for IC yield optimization based on fundamental fabrication parameters, *IEEE Trans. Comput. Aided Des. ICAS*, CAD-5(1): 79–89, January 1986 (special issue on statistical design of VLSI circuits).
49. M. A. Styblinski and A. Ruszczynski, Stochastic approximation approach to production yield optimizaion, in *Proceedings of the 25th Midwest Symposium on Circuits and Systems*, Houghton, MI, August 30–31, 1982.
50. M. A. Styblinski and A. Ruszczynski, Stochastic approximation approach to statistical circuit design, *Electron Lett.*, 19(8): 300–302, April 14, 1983.
51. M. A. Styblinski and W. Strasz, A second derivative Monte Carlo optimization of the production yield, in *Proceedings of the European Conference on Circuit Theory and Design (ECCTD'80)*, Vol. 2, pp. 121–131, Warsaw, Poland, September 1980.
52. M. A. Styblinski and T.-S. Tang, Experiments in nonconvex optimization: Stochastic approximation with function smoothing and simulated annealing, *Neural Networks J.*, 3(4): 467–483, 1990.
53. M. A. Styblinski and J. C. Zhang, Orthogonal array approach to gradient based yield optimization, in *Proceedings of the International Symposium on Circuits and Systems*, pp. 424–427, New Orleans, LA, May 1990.
54. K. S. Tahim and R. Spence, A radial exploration algorithm for the statistical analysis of linear circuits, *IEEE Trans. Circuits Syst.*, CAS-27(5): 421–425, May 1980.
55. T.-S. Tang and M. A. Styblinski, Yield optimization for non-differentiable density functions using convolution techniques, *IEEE Trans. Comput. Aided Des. IC Syst.*, 7(10): 1053–1067, 1988.
56. W. J. Welch, T.-K. Yu, S. M. Kang, and J. Sacks, Computer experiments for quality control by parameter design, J. *Qual. Technol.*, 22(1): 15–22, January 1990.
57. P. Yang, D. E. Hocevar, P. F. Cox, C. Machala, and P. K. Chatterjee, An integrated and efficient approach for MOS VLSI statistical circuit design, *IEEE Trans. Comput. Aided Des. VLSI Circuits Syst.*, CAD-5: 5–14, January 1986.
58. D. L. Young, J. Teplik, H. D. Weed, N. T. Tracht, and A. R. Alvarez, Application of statistical design and response surface methods to computer-aided VLSI device design II: Desirability functions and Taguchi methods, *IEEE Trans. Comput. Aided Des.*, 10(1): 103–115, January 1991.

59. T. K. Yu, S. M. Kang, I. N. Hajj, and T. N. Trick, Statistical performance modeling and parametric yield estimation of MOS VLSI, *IEEE Trans. Comput. Aided Des. VLSI Circuits Syst.*, CAD-6(6): 1013–1022, November 1987.
60. T. K. Yu, S. M. Kang, I. N. Hajj, and T. N. Trick, iEDISON: An interactive statistical design tool for MOS VLSI circuits, in *IEEE International Conference on Computer-Aided Design (ICCAD-88)*, pp. 20–23, Santa Clara, CA, November 7–10, 1988.
61. T. K. Yu, S. M. Kang, J. Sacks, and W. J. Welch, Parametric yield optimization of MOS integrated circuits by statistical modeling of circuit performances, Technical Report No. 27, Department of Statistics, University of Illinois, Champaign, IL, July 1989.
62. L. A. Zadeh, Fuzzy sets, *Info. Control*, 8: 338–353, 1965.
63. L. A. Zadeh, Probability measures of fuzzy events, *J. Math. Anal. Appl.*, 23: 421–427, 1968.

6

Physical Design Automation*

6.1 Introduction 6-1
6.2 Very Large-Scale Integration Design Cycle 6-2
6.3 Physical Design Cycle 6-4
6.4 Design Styles 6-6
6.5 Partitioning 6-9
Classification of Partitioning Algorithms • Kernighan–Lin Partitioning Algorithm
6.6 Other Partitioning Algorithms 6-12
6.7 Placement 6-12
Classification of Placement Algorithms • Simulated Annealing Placement Algorithm • Other Placement Algorithms
6.8 Routing 6-15
6.9 Classification of Global Routing Algorithms 6-16
6.10 Classification of Detailed Routing Algorithms 6-16
Lee's Algorithm for Global Routing • Greedy Channel Router for Detailed Routing • Other Routing Algorithms
6.11 Compaction 6-21
Classification of Compaction Algorithms • Shadow-Propagation Algorithm for Compaction • Other Compaction Algorithms
6.12 Summary 6-23
References 6-24

Naveed A. Sherwani
Intel Corporation

6.1 Introduction

In the last three decades integrated circuit (IC) fabrication technology has evolved from integration of a few transistors in small-scale integration (SSI) to integration of several million transistors in very large-scale integration (VLSI). This phenomenal progress has been made possible by automating the process of design and fabrication of VLSI chips.

Integrated circuits consist of a number of electronic components, built by layering several different materials in a well-defined fashion on a silicon base called a wafer. The designer of an IC transforms a circuit description into a geometric description, which is known as a layout. A layout consists of a set of planar geometric shapes in several layers. The layout is then checked to ensure that it meets all the design

* The material presented in this chapter was adapted from the author's book *Algorithms for VLSI Physical Design Automation*, Boston: Kluwer Academic, 1993, with editorial changes for clarity and continuity. Copyright permission has been obtained from Kluwer Academic Publishers.

requirements. The result is a set of design files in a particular unambiguous representation, known as an intermediate form, which describes the layout. The design files are then converted into pattern generator files, which are used to produce patterns called masks by an optical pattern generator. During fabrication, these masks are used to pattern a silicon wafer using a sequence of photolithographic steps. The component formation requires very exacting details about geometric patterns and separation between them. The process of converting the specifications of an electrical circuit into a layout is called the physical design. It is an extremely tedious and error-prone process because of the tight tolerance requirements and the minuteness of the individual components. Currently, the smallest geometric feature of a component can be as small as 0.35 μm (1 $\mu m = 1.0 \times 10^{-6}$ m). However, it is expected that the feature size can be reduced to 0.1 μm within 5 years. This small feature size allows fabrication of as many as 4.5 million transistors on a 25×25 mm maximum size chip. Due to the large number of components and the exacting details required by the fabrication process, the physical design is not practical without the help of computers. As a result, almost all phases of physical design extensively use computer-aided design (CAD) tools and many phases have already been partially or fully automated. This automation of the physical design process has increased the level of integration, reduced the turnaround time, and enhanced chip performance.

VLSI physical design automation is essentially the study of algorithms related to the physical design process. The objective is to study optimal arrangements of devices on a plane (or in a three-dimensional space) and efficient interconnection schemes between these devices to obtain the desired functionality. Because space on a wafer is very expensive real estate, algorithms must use the space very efficiently to lower costs and improve yield. In addition, the arrangement of devices plays a key role in determining the performance of a chip. Algorithms for physical design must also ensure that all the rules required by the fabrication are observed and that the layout is within the tolerance limits of the fabrication process. Finally, algorithms must be efficient and should be able to handle very large designs. Efficient algorithms not only lead to fast turnaround time, but also permit designers to iteratively improve the layouts.

With the reduction in the smallest feature size and increase in the clock frequency, the effect of electrical parameters on physical design will play a more dominant role in the design and development of new algorithms.

In this section, we present an overview of the fundamental concepts of VLSI physical design automation. Different design styles are discussed in Section 6.4. Section 6.2 discusses the design cycle of a VLSI circuit. In Section 6.3 different steps of the physical design cycle are discussed. The rest of the sections discuss each step of the physical design cycle.

6.2 Very Large-Scale Integration Design Cycle

Starting with a formal specification, the VLSI design cycle follows a series of steps and eventually produces a packaged chip. A flow chart representing a typical design cycle is shown in Figure 6.1.

1. *System specification*: The specifications of the system to be designed are exactly specified here. This necessitates creating a high-level representation of the system. The factors to be considered in this process include performance, functionality, and the physical dimensions. The choice of fabrication technology and design techniques are also considered. The end results are specifications for the size, speed, power, and functionality of the VLSI system to be designed.
2. *Functional design*: In this step, behavioral aspects of the system are considered. The outcome is usually a timing diagram or other relationships between subunits. This information is used to improve the overall design process and to reduce the complexity of the subsequent phases.
3. *Logic design*: In this step, the functional design is converted into a logical design, typically represented by Boolean expressions. These expressions are minimized to achieve the smallest logic design which conforms to the functional design. This logic design of the system is simulated and tested to verify its correctness.

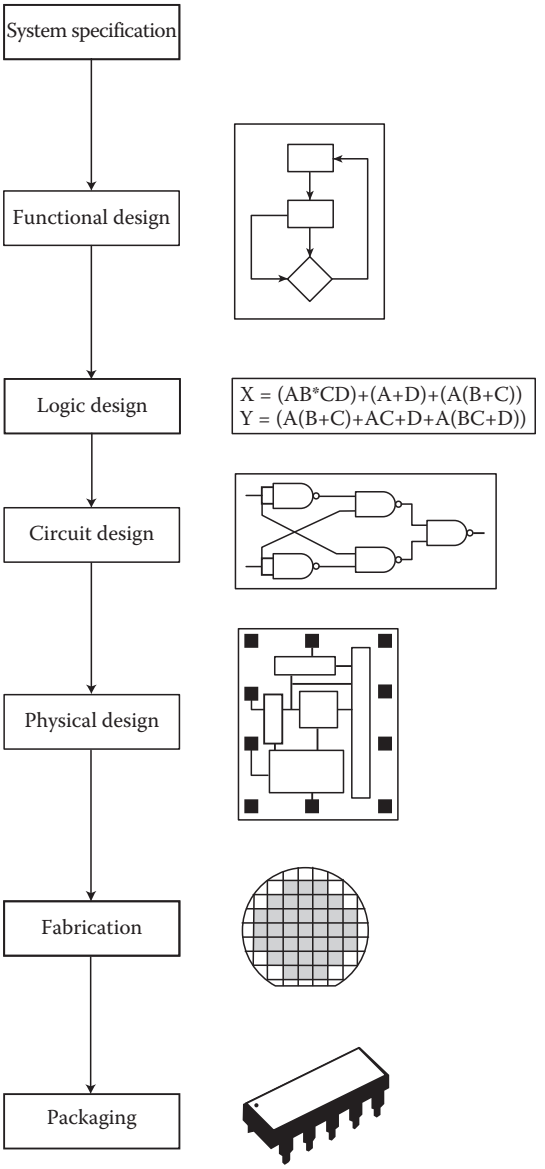


FIGURE 6.1 Design process steps.

4. *Circuit design*: Here, the Boolean expressions are converted into a circuit representation by taking into consideration the speed and power requirements of the original design. The electrical behavior of the various components are also considered in this phase. The circuit design is usually expressed in a detailed circuit diagram.
5. *Physical design*: In this step, the circuit representation of each component is converted into a geometric representation. This representation is in fact a set of geometric patterns which perform the intended logic function of the corresponding component. Connections between different components are also expressed as geometric patterns. As stated earlier, this geometric representation of a circuit is called a layout. The exact details of the layout also depend on design rules, which are guidelines based on the limitations of the fabrication process and the

electrical properties of the fabrication materials. Physical design is a very complex process; therefore, it is usually broken down into various substeps in order to handle the complexity of the problem. In fact, physical design is arguably the most time-consuming step in the VLSI design cycle.

6. *Design verification*: In this step, the layout is verified to ensure that the layout meets the system specifications and the fabrication requirements. Design verification consists of design rule checking (DRC) and circuit extraction. DRC is a process which verifies that all geometric patterns meet the design rules imposed by the fabrication process. After checking the layout for design rule violations and removing them, the functionality of the layout is verified by circuit extraction. This is a reverse engineering process and generates the circuit representation from the layout. This reverse engineered circuit representation can then be compared to the original circuit representation to verify the correctness of the layout.
7. *Fabrication*: After verification, the layout is ready for fabrication. The fabrication process consists of several steps: preparation of wafer, deposition, and diffusion of various materials on the wafer according to the layout description. A typical wafer is 10 cm in diameter and can be used to produce between 12 and 30 chips. Before the chip is mass produced, a prototype is made and tested.
8. *Packaging, testing, and debugging*: In this step, the chip is fabricated and diced in a fabrication facility. Each chip is then packaged and tested to ensure that it meets all the design specifications and that it functions properly. Chips used in printed circuit boards (PCBs) are packaged in a dual-in-line package (DIP) or pin grid array (PGA). Chips which are to be used in a multichip module (MCM) are not packaged because MCMs use bare or naked chips.

The VLSI design cycle is iterative in nature, both within a step and between steps. The representation is iteratively improved to meet system specifications. For example, a layout is iteratively improved so that it meets the timing specifications of the system. Another example may be detection of design rule violations during design verification. If such violations are detected, the physical design step needs to be repeated to correct the error. The objective of VLSI CAD tools is to minimize the number of iterations and thus reduce the time-to-market.

6.3 Physical Design Cycle

Physical design cycle converts a circuit diagram into a layout. This is accomplished in several steps such as partitioning, floorplanning, placement, routing, and compaction, as shown in Figure 6.2.

1. *Partitioning*: The complex task of chip layout is divided into several smaller tasks. A chip may contain several million transistors. Layout of the entire circuit cannot be handled due to the limitation of memory space as well as computation power available. Therefore, it is normally partitioned by grouping the components into blocks (subcircuits/modules). The actual partitioning process considers many factors such as size of the blocks, number of blocks, and number of interconnections between the blocks. The output of partitioning is a set of blocks along with the interconnections required between blocks. The set of interconnections required is referred to as a netlist. Figure 6.2a shows that the input circuit has been partitioned into three blocks. In large circuits the partitioning process is hierarchical and at the topmost level a chip may have between 5 and 25 blocks. Each module is then partitioned recursively into smaller blocks.
2. *Placement*: In this step, good layout alternatives are selected for each block, as well as the entire chip. The area of each block can be calculated after partitioning and is based approximately on the number and the type of components in that block. The actual rectangular shape of the block, which is determined by the aspect ratio may, however, be varied within a prespecified range. Floorplanning is a critical step, as it sets up the groundwork for a good layout. However, it is

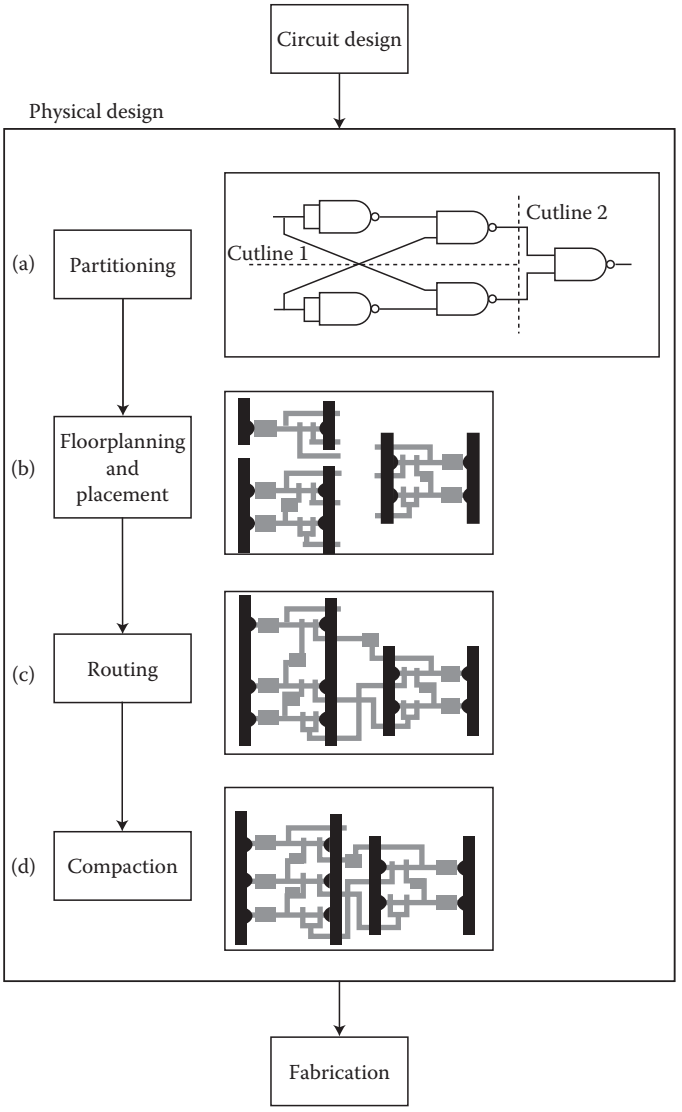


FIGURE 6.2 Physical design cycle.

computationally quite hard. Very often the task of floorplan layout is done by a design engineer, rather than by a CAD tool. This is sometimes necessary as the major components of an IC are often intended for specific locations on the chip. The placement process determines the exact positions of the blocks on the chip, so as to find a minimum area arrangement for the blocks that allows completion of interconnections between the blocks. Placement is typically done in two phases. In the first phase an initial placement is created. In the second phase the initial placement is evaluated and iterative improvements are made until the layout has minimum area and conforms to design specifications. Figure 6.2b shows that three blocks have been placed. It should be noted that some space between the blocks is intentionally left empty to allow interconnections between blocks. Placement may lead to unroutable design, i.e., routing may not be possible in the space provided. Thus, another iteration of placement is necessary. To limit the number of iterations of the placement algorithm, an estimate of the required routing space is used during the placement

phase. A good routing and circuit performance heavily depend on a good placement algorithm. This is due to the fact that once the position of each block is fixed, very little can be done to improve the routing and the overall circuit performance.

3. *Routing*: In this step, the objective is to complete the interconnections between blocks according to the specified netlist. First, the space not occupied by the blocks (called the routing space) is partitioned into rectangular regions called channels and switchboxes. The goal of a router is to complete all circuit connections using the shortest possible wire length and using only the channels and switchboxes. This is usually done in two phases, referred to as the global routing and detailed routing phases. In global routing, connections are completed between the proper blocks of the circuit disregarding the exact geometric details of each wire and pin. For each wire, the global router finds a list of channels which are to be used as a passageway for that wire. In other words, global routing specifies the ''loose route'' of a wire through different regions in the routing space. Global routing is followed by detailed routing, which completes point-to-point connections between pins on the blocks. Loose routing is converted into exact routing by specifying geometric information such as width of wires and their layer assignments. Detailed routing includes channel routing and switchbox routing. Routing is a very well-studied problem and several hundred articles have been published about all its aspects. Because almost all problems in routing are computationally hard, the researchers have focused on heuristic algorithms. As a result, experimental evaluation has become an integral part of all algorithms and several benchmarks have been standardized. Due to the nature of the routing algorithms, complete routing of all the connections cannot be guaranteed in many cases. As a result, a technique called rip-up and reroute is used, which basically removes troublesome connections and reroutes them in a different order. The routing phase of Figure 6.2c shows that all the interconnections between three blocks have been routed.
4. *Compaction*: In this step, the layout is compressed in all directions such that the total area is reduced. By making the chip smaller, wire lengths are reduced, which in turn reduces the signal delay between components of the circuit. At the same time, a smaller area may imply more chips can be produced on a wafer, which in turn reduces the cost of manufacturing. However, the expense of computing time mandates the extensive compaction is used only when large quantities of ICs are produced. Compaction must ensure that no rules regarding the design and fabrication process are violated during the process. The final diagram in Figure 6.2d shows the compacted layout.

Physical design, like VLSI design, is iterative in nature, and many steps such as global routing and channel routing are repeated several times to obtain a better layout. In addition, the quality of results obtained in a step depends on the quality of solutions obtained in earlier steps. For example, a poor quality placement cannot be ''cured'' by high quality routing. As a result, earlier steps have more influence on the overall quality of the solution. In this sense partitioning, floorplanning, and placement problems play a more important role in determining the area and chip performance, as compared to routing and compaction. Because placement may produce an ''unroutable'' layout, the chip might need to be replaced or repartitioned before another routing is attempted. In general, the whole design cycle may be repeated several times to accomplish the design objectives. The complexity of each step varies depending on the design constraints as well as the design style used.

6.4 Design Styles

Even after decomposing physical design into several conceptually easier steps, it has been shown that each step is computationally very hard. Market requirements demand a quick time-to-market and high yield. As a result, restricted models and design styles are used in order to reduce the complexity of

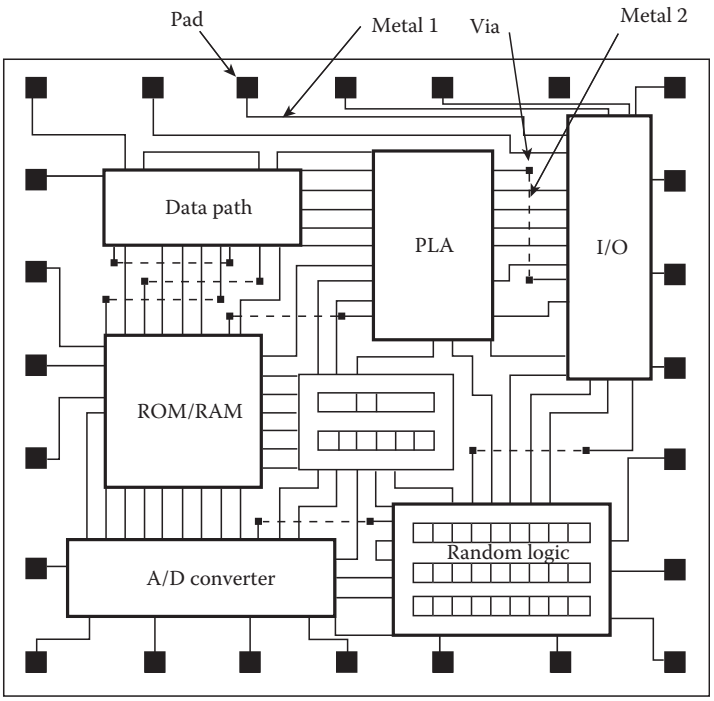


FIGURE 6.3 Full-custom structure.

physical design. This practice began in the late 1960s and led to the development of several restricted design styles [8].

The most general form of layout is called the full-custom design style, in which the circuit is partitioned into a collection of subcircuits according to some criteria such as functionality of each subcircuit. In this design style, each subcircuit is called a functional block or simply a block. The full-custom design style allows functional blocks to be of any size. Figure 6.3 shows an example of very simple circuit with few blocks. Internal routing in each block is not shown for the sake of clarity. Blocks can be placed at any location on the chip surface without restriction. In other words, this style is characterized by the absence of any constraints on the physical design process. This design style allows for very compact designs. However, the process of automating a full-custom design style has a much higher complexity than other restricted models. For this reason, it is used only when final design must have a minimum area and designing time is less of a factor. The automation process for a full-custom layout is still a topic of intensive research. Some phases of physical design of a full-custom chip may be done manually to optimize the layout. Layout compaction is a very important aspect in full-custom. The rectangular solid boxes around the boundary of the circuit are called I–O pads. Pads are used to complete interconnections between chips or interconnections between chip and the board. The space not occupied by blocks is used for routing of interconnecting wires. Initially all the blocks are placed within the chip area, with the objective of minimizing the total area. However, enough space must be left between the blocks to complete the routing. Usually several metal layers are used for routing interconnections. Currently, two metal layers are common for routing and the three-metal layer process is gaining acceptance, as the fabrication costs become more feasible. The routing area needed between the blocks becomes increasingly smaller as more routing layers are used. This is because some routing is done on top of the transistors in the additional metal layers. If all the routing can be done on top of the transistors, the total chip area is determined by the area of the transistors.

In a hierarchical design of circuit each block in full-custom design may be very complex and may consist of several subblocks, which in turn may be designed using the full-custom design style or other design styles. It is easy to see that because any block is allowed to be placed anywhere on the chip, the problem of optimizing area and interconnection of wires becomes difficult. Full-custom design is very time consuming; thus, the method is inappropriate for very large circuits, unless performance is of utmost importance. Full-custom is usually used for the layout of microprocessors.

A more restricted design style is called the standard cell design style. The design process in standard cell design style is simpler than a full-custom design style. Standard cell methodology considers the layout to consist of rectangular cells of the same height. Initially, a circuit is partitioned into several smaller blocks, each of which is equivalent to some predefined subcircuit (cell). The functionality and the electrical characteristics of each predefined cell are tested, analyzed, and specified. A collection of these cells is called a cell library, usually consisting of 200–400 cells. Terminals on cells may be located either on the boundary or in the center of the cells. Cells are placed in rows and the space between two rows is called a channel. These channels are used to perform interconnections between cells. If two cells to be interconnected lie in the same row or in adjacent rows, then the channel between the rows is used for interconnection. However, if two cells to be connected lie in two nonadjacent rows, then their interconnection wire passes through the empty space between any two cells, or feedthrough.

Standard cell design is well suited for moderate-size circuits and medium production volumes. Physical design using standard cells is simpler as compared to full-custom and efficient using modern design tools. The standard cell design style is also widely used to implement the ''random logic'' of the full-custom design, as shown in Figure 6.3. While standard cell designs are developed more quickly, a substantial initial investment is needed in the development of the cell library, which may consist of several hundred cells. Each cell in the cell library is ''handcrafted'' and requires a highly skilled design engineer. Each type of cell must be created with several transistor sizes. Each cell must then be tested by simulation and its performance must be characterized. A standard cell design usually takes more area than a full-custom or a handcrafted design. However, as more metal layers become available for routing, the difference in area between the two design styles will gradually be reduced.

The gate array design style is a simplified version of the standard cell design style. Unlike the cells in standard cell designs, all the cells in gate array are identical. The entire wafer is prefabricated with an array of identical gates or cells. These cells are separated by both vertical and horizontal spaces called vertical and horizontal channels. The circuit design is modified such that it can be partitioned into a number of identical blocks. Each block must be logically equivalent to a cell on the gate array. The name ''gate array'' signifies the fact that each cell may simply be a gate, such as a three-input NAND gate. Each block in the design is mapped or placed onto a prefabricated cell on the wafer during the partitioning/placement phase, which is reduced to a block-to-cell assignment problem. The number of partitioned blocks must be less than or equal to that of the total number of cells on the wafer. Once the circuit is partitioned into identical blocks, the task is to make the interconnections between the prefabricated cells on the wafer using horizontal and vertical channels to form the actual circuit. The uncommitted gate array is taken into the fabrication facility and routing layers are fabricated on top of the wafer. The completed wafer is also called a customized wafer.

This simplicity of gate array design is gained at the cost of rigidity imposed upon the circuit both by the technology and the prefabricated wafers. The advantage of gate arrays is that the steps involved for creating any prefabricated wafer are the same, and only the last few steps in the fabrication process actually depend on the application for which the design will be used. Hence, gate arrays are cheaper and easier to produce than full-custom or standard cell. Similar to standard cell design, gate array is also a nonhierarchical structure. The gate array architecture is the most restricted form of layout. This also means it is the simplest for algorithms to work with. For example, the task of routing in gate array is to determine if a given placement is routable. The routability problem is conceptually simpler as compared to the routing problem in standard cell and full-custom design styles.

The choice of design style for a particular circuit depends on many factors such as functionality of the chip, time-to-market, and the volume of chips to be manufactured. Full-custom is typically reserved for high-performance, high-volume chips, while standard cells are used for moderate performance, where the cost of full-custom cannot be justified. Gate arrays are typically used for low-performance, low-cost applications. A design style may be applicable to the entire chip or a block of the chip.

Irrespective of the choice of the design style, all steps of the physical design cycle need to be carried out. However, the complexity, the effectiveness, and the algorithms used differ considerably depending on the design style. The following sections discuss algorithms for different phases of the physical design cycle.

6.5 Partitioning

As stated earlier, the basic purpose of partitioning is to simplify the overall design process. The circuit is decomposed into several subcircuits to make the design process manageable. This section considers the partitioning phase of the physical design cycle, study constraints, and objective functions for this problem and presents efficient algorithms.

Given a circuit, the partitioning problem is to decompose it into several subcircuits subject to constraints while optimizing a given objective function. The constraints for the partitioning problem include area constraints and terminal constraints. Area constraints are user specified for design optimization and terminal count depends on the area and aspect ratio of the block. In particular, the terminal count for a partition is given by the ratio of the perimeter of the partition to the terminal pitch. The minimum spacing between two adjacent terminals is called terminal pitch and is determined by the design rules. The objective functions for a partitioning problem include the minimization of the number of nets that cross the partition boundaries, and the minimization of the maximum number of times a path crosses the partition boundaries. The constraints and the objective functions used in the partitioning problem vary depending upon the partitioning level and the design style used. The actual objective function and constraints chosen for the partitioning problem may also depend on the specific problem. The number of nets that connect a partition to other partitions cannot be greater than the terminal count of the partition. In addition, the number of nets cut by partitioning should be minimized to simplify the routing task. The minimization of the number of nets cut by partitioning is one of the most important objectives in partitioning.

A disadvantage of the partitioning process is that it may degrade the performance of the final design. During partitioning, critical components should be assigned to the same partition. If such an assignment is not possible, then appropriate timing constraints must be generated to keep the two critical components close together. Usually, several components, forming a critical path, determine the chip performance. If each component is assigned to a different partition, the critical path may be too long. Minimizing the length of critical paths improves system performance.

After a chip has been partitioned, each of the subcircuits must be placed on a fixed plane and the nets between all the partitions must be interconnected. The placement of the subcircuits is done by the placement algorithms and the nets are routed by using routing algorithms.

6.5.1 Classification of Partitioning Algorithms

The partitioning algorithms also may be classified based on the nature of the algorithms, of which two types exist: deterministic algorithms and probabilistic algorithms. Deterministic algorithms produce repeatable or deterministic solutions. For example, an algorithm which makes use of deterministic functions will always generate the same solution for a given problem. On the other hand, the probabilistic algorithms are capable for producing a different solution for the same problem each time they are used, as they make use of some random functions.

The partitioning algorithms also may be classified on the basis of the process used for partitioning. Thus, we have group migration algorithms, simulated annealing and evolution-based algorithms, and other partitioning algorithms.

The group migration algorithms [9,17] start with some partitions, usually generated randomly, and then move components between partitions to improve the partitioning. The group migration algorithms are quite efficient. However, the number of partitions must be specified, which is usually not known when the partitioning process starts. In addition, the partitioning of an entire system is a multilevel operation, and the evaluation of the partitions obtained by the partitioning depends on the final integration of partitions at all levels, from the basic subcircuits to the whole system. An algorithm used to find a minimum cut at one level may sacrifice the quality of cuts for the following levels. The group migration method is a deterministic method that is often trapped at a local optimum and cannot proceed further.

The simulated annealing/evolution algorithms [3,10,18,26] carry out the partitioning process by using a cost function, which classifies any feasible solution, and a set of moves, which allows movement from solution to solution. Unlike deterministic algorithms, these algorithms accept moves which may adversely affect the solution. The algorithm starts with a random solution and as it progresses, the proportion of adverse moves decreases. These degenerate moves act as a safeguard against entrapment in local minima. These algorithms are computationally intensive as compared to group migration and other methods.

Among all the partitioning algorithms, the group migration and simulated annealing/evolution have been the most successful heuristics for partitioning problems. The group migration and simulated annealing/evolution methods are most widely used, and extensive research has been carried out in these two types of algorithms.

6.5.2 Kernighan–Lin Partitioning Algorithm

The Kernighan–Lin (K–L) algorithm is a bisectioning algorithm. It starts by initially partitioning the graph $G = (V, E)$ into two subsets of equal size. Vertex pairs are exchanged across the bisection if the exchange improves the cutsize. The preceding procedure is carried out iteratively until no further improvement can be achieved.

The basic idea of the K–L algorithm is illustrated with the help of an example before presenting the algorithm formally. Consider the example given in Figure 6.4a. The initial partitions are

$$A = \{1,2,3,4\}$$
$$B = \{5,6,7,8\}$$

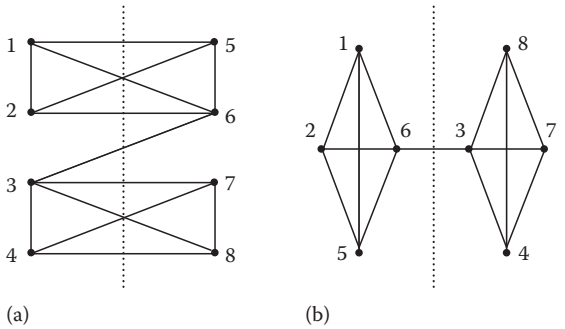


FIGURE 6.4 Graph bisected by K–L algorithm: (a) initial bisections and (b) final bisections.

TABLE 6.1 Log of the Vertex Exchanges

i	Vertex Pair	$g(i)$	$\sum_{j=1}^{i} g(i)$	Cutsize
0	—	—	—	9
1	(3, 5)	3	3	6
2	(4, 6)	5	8	1
3	(1, 7)	−6	2	7
4	(2, 8)	−2	0	9

Notice that the initial cutsize is 9. The next step of the K–L algorithm is to choose a pair of vertices whose exchange results in the largest decrease of the cutsize or results in the smallest increase, if no decrease is possible. The decrease of the cutsize is computed using gain values $D(i)$ of vertices v_i. The gain of a vertex v_i is defined as

$$D(i) = \text{inedge}(i) - \text{outedge}(i)$$

where

inedge(i) is the number of edges of vertex i that do not cross the bisection boundary
outedge(i) is the number of edges that cross the boundary

The amount by which the cutsize decreases, if vertex v_i changes over to the other partition, is represented by $D(i)$. If v_i and v_j are exchanged, the decrease of the cutsize is $D(i) + D(j)$. In the example given in Figure 6.4, a suitable vertex pair is (3, 5), which decreases the cutsize by 3. A tentative exchange of this pair is made. These two vertices are then locked. This lock on the vertices prohibits them from taking part in any further tentative exchanges. The preceding procedure is applied to the new partitions, which gives a second vertex pair of (4, 6). This procedure is continued until all the vertices are locked. During this process, a log of all tentative exchanges and the resulting cutsizes is stored. Table 6.1 shows the log of vertex exchanges for the given example. Note that the partial sum of cutsize decrease $g(i)$ over the exchanges of first i vertex pairs is given in the table; e.g., $g(1) = 3$ and $g(2) = 8$. The value of k for which $g(k)$ gives the maximum value of all $g(i)$ is determined from the table. In this example $k = 2$ and $g(2) = 8$ is the maximum partial sum. The first k pairs of vertices are actually exchanged. In the example the first two vertex pairs (3, 5) and (4, 6) are actually exchanged, resulting in the bisection shown in Figure 6.4b. This completes an iteration and a new iteration starts. However, if no decrease of cutsize is possible during an iteration, the algorithm stops. Figure 6.5 presents the formal description of the K–L algorithm.

The procedure INITIALIZE finds initial bisections, and initializes the parameters in the algorithm. The procedure IMPROVE tests if any improvement has been made during the last iteration, while the procedure UNLOCK checks if any vertex is unlocked. Each vertex has a state of either *locked* or *unlocked*. Only those vertices whose status is *unlocked* are candidates for the next tentative exchanges. The procedure TENT-EXCHGE tentatively exchanges a pair of vertices. The procedure LOCK locks the vertex pair, while the procedure LOG stores the log table. The procedure ACTUAL-EXCHGE determines the maximum partial sum of $g(i)$, selects the vertex pairs to be exchanged, and fulfills the actual exchange of these vertex pairs.

The time complexity of the K–L algorithm is $O(n^3)$. The K–L algorithm is, however, quite robust. It can accommodate additional constraints, such as a group of vertices requiring to be in a specified partition. This feature is very important in layout because some blocks of the circuit are to be kept together due to the functionality. For example, it is important to keep all components of an adder together. However, the K–L algorithm has several disadvantages. For example, the algorithm is not applicable for hypergraphs, it cannot handle arbitrarily weighted graphs and the partition sizes must be specified before partitioning. Finally, the complexity of the algorithm is considered too high even for moderate-size problems.

Algorithm *KL*

```
Begin
    INITIALIZE();
    while (IMPROVE (table) = TRUE) do
    (* if an improvement has been made during last iteration,
    the process is carried out again.*)
        while (UNLOCK(A) = TRUE) do
        (* if there exists any unlocked vertex in A,
        more tentative exchanges are carried out.*)
            for (each a∈A) do
                if (a = unlocked) then
                    for (each b ∈ B) do
                        if (b = unlocked) then
                            if (D_max < D(a) + D(b)) then
                                D_max = D(a) + D(b);
                                a_max = a;
                                b_max = b;
            TENT-EXCHGE(a_max, b_max);
            LOCK(a_max, b_max);
            LOG(table);
            D_max = −∞;
        ACTUAL-EXCHGE(table);
End;
```

FIGURE 6.5 Algorithm K–L.

6.6 Other Partitioning Algorithms

In order to overcome the disadvantages of the K–L algorithm, several extensions of the K–L algorithm such as the Fiduccia–Mattheyses algorithm, the Goldberg and Burstein algorithm, the component replication algorithm, and the ratio cut algorithm were developed. In the class of probabilistic and iterative algorithms simulated annealing and evolution-based algorithms have been developed for partitioning. For details on these partitioning algorithms, refer to Chapter 4 of [29].

6.7 Placement

The placement phase follows the partitioning phase of the physical design cycle. After the circuit has been partitioned, the area occupied by each block (subcircuit) can be calculated and the number of terminals (pins) required by each block is known. Partitioning also generates the netlist which specifies the connections between the blocks. The layout is completed by arranging the blocks on the layout surface and interconnecting their pins according to the netlist. The arrangement of blocks is done in the placement phase, while interconnection is completed in the routing phase. In the placement phase, blocks are assigned a specific shape and are positioned on a layout surface in such a fashion that no two blocks overlap and enough space is left on the layout surface to complete the interconnections between the blocks. 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 placement phase is a set of blocks, 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 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, during the placement phase, we need to determine an appropriate shape for each block (if shape is not known), the 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 the fixed blocks on a layout surface is called the placement problem. If some or all of the blocks are flexible, the problem is called the floorplanning problem. Hence, the placement problem is a restricted version of the floorplanning 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 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.

The placement phase is crucial in the overall physical design cycle because an ill-placed 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 placement phase.

6.7.1 Classification of Placement Algorithms

The placement algorithms can be classified on the basis of the input to the algorithms, the nature of output generated by the algorithms, and the process used by the algorithms.

Depending on the input, the placement algorithms can be classified into two major groups: constructive placement and iterative improvement methods. The input to the constructive placement algorithms consists of a set of blocks along with the netlist. The algorithm finds the locations of blocks. On the other hand, iterative improvement algorithms start with an initial placement. These algorithms modify the initial placement in search of a better placement. These algorithms are typically used in an iterative manner until no improvement is possible.

The nature of output produced by an algorithm is another way of classifying the placement algorithms. Some algorithms generate the same solution when presented with the same problem, i.e., the solution produced is repeatable. These algorithms are called deterministic placement algorithms. Algorithms that function on the basis of fixed connectivity rules (or formulas) or determine the placement by solving simultaneous equations are deterministic and always produce the same result for a particular placement problem. Some algorithms, on the other hand, work by randomly examining configurations and may produce a different result each time they are presented with the same problem. Such algorithms are called probabilistic placement algorithms.

The classification based on the process used by the placement algorithms is perhaps the best way of classifying these algorithms. Two important classes of algorithms come under this classification: simulation-based algorithms and partitioning-based algorithms. Simulation-based algorithms simulate some natural phenomenon while partitioning-based algorithms use partitioning for generating the placement. The algorithms which use clustering and other approaches are classified under ''other'' placement algorithms.

6.7.2 Simulated Annealing Placement Algorithm

Simulated annealing is one of the most well-developed methods available [2,10–12,16,19,23,25–28]. The simulated annealing technique has been successfully used in many phases of VLSI physical design, e.g., circuit partitioning. A detailed description of the application of this method to partitioning may be found in Chapter 4 of [29]. Simulated annealing is used in placement as an iterative improvement algorithm. Given a placement configuration, a change to that configuration is made by moving a component or interchanging locations of two components. In the case of the simple pairwise interchange algorithm, it is possible that an achieved configuration has a cost higher than that of the optimum, but no interchange can cause a further cost reduction. In such a situation, the algorithm is trapped at a local optimum, and cannot proceed further. Actually, this happens quite often when this algorithm is used in real-life examples. Simulated annealing avoids getting stuck at a local optimum by occasionally accepting moves that result in a cost increase.

Algorithm *Simulated Annealing*

```
Begin
    temp = INIT-TEMP;
    place = INIT-PLACEMENT;
    while (temp > FINAL-TEMP) do
      while (inner_loop_criterion = FALSE) do
        new_place = PERTURB (place);
        ΔC = COST(new_place) - COST (place);
        if (ΔC < 0) then
          place = new_place;
        else if (RANDOM(0,1) > e^(Δc/T)) then
          place = new_place;
      temp = SCHEDULE(temp);
End;
```

FIGURE 6.6 Algorithm simulated annealing.

In simulated annealing, all moves that result in a decrease in cost are accepted. Moves that result in an increase in cost are accepted with a probability that decreases over the iterations. The analogy to the actual annealing process is heightened with the use of a parameter called temperature T. This parameter controls the probability of accepting moves that result in an increased cost. Additional moves are accepted at higher values of temperature than at lower values. The acceptance probability can be given by $e^{\Delta C/T}$, where ΔC is the increase in cost. The algorithm starts with a very high value of temperature, which gradually decreases so that moves that increase cost have a lower probability of being accepted. Finally, the temperature reduces to a very low value which causes only moves that reduce cost to be accepted. In this way the algorithm converges to an optimal or near-optimal configuration.

In each stage, the configuration is shuffled randomly to obtain a new configuration. This random shuffling could be achieved by displacing a block to a random location, an interchange of two blocks, or any other move that can change the wire length. After the shuffle, the change in cost is evaluated. If a decrease in cost occurs, the configuration is accepted; otherwise, the new configuration is accepted with a probability that depends on the temperature. The temperature is then lowered using some function which, for example, could be exponential in nature. The process is stopped when the temperature has dropped to a certain level. The outline of the simulated annealing algorithm is shown in Figure 6.6.

The parameters and functions used in a simulated annealing algorithm determine the quality of the placement produced. These parameters and functions include the cooling schedule consisting of initial temperature (*init_temp*), final temperature (*final_temp*), and the function used for changing the temperature (SCHEDULE), *inner_loop_criterion*, which is the number of trials at each temperature, the process used for shuffling a configuration (PERTURB), acceptance probability (F), and the cost function (COST). A good choice of these parameters and functions can result in a good placement in a relatively short time.

6.7.3 Other Placement Algorithms

Several other algorithms which simulate naturally occurring processes have been developed for routing. The simulated evolution algorithm is analogous to the natural process of mutation of species as they evolve to better adapt to their environment. The force directed placement explores the similarity between the placement problem and the classical mechanics problem of a system of bodies attached to springs. The partitioning-based placement techniques include Breuer's algorithm and the terminal propagation algorithm. Several other algorithms, such as the cluster growth algorithm, the quadratic assignment algorithm, the resistive network optimization algorithm, and the branch and bound algorithm, also exist. For more details on these algorithms, refer to Chapter 5 of [29].

6.8 Routing

The exact locations of circuit blocks and pins are determined in the placement phase. A netlist is also generated which specifies the required interconnections. Space not occupied by the blocks can be viewed as a collection of regions. These regions are used for routing and are called routing regions. The process of finding the geometric layouts of all the nets is called routing. 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. Nets must be routed within the routing regions and must not violate the capacity of any routing region. In addition, nets must not short circuit; that is, nets must not intersect each other. The objective of routing is dependent on the nature of the chip. For general-purpose chips, it is sufficient to minimize the total wire length. For high-performance chips, total wire length may not be a major concern. Instead, we may want to minimize the longest wire to minimize the delay in the wire and therefore maximize its performance. 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.

Channels and switchboxes are the two types of routing regions. A switchbox is a rectangular area bounded on all sides. A channel is a rectangular area bounded on two opposite sides by the blocks. The 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., $capacity = (l \times h)/(w + s)$. For example, if for channel C (Figure 6.7), $l = 2$, $h = 18\lambda$, $w = 3\lambda$, $s = 3\lambda$, then the capacity is $(2 \times 18)/(3 + 3) = 6$.

A VLSI chip may contain several million transistors. As a result, tens of thousands of nets must be routed to complete the layout. In addition, several hundred routes are possible for each net. This makes the routing problem computationally hard. In fact, even when the routing problem is restricted to channels, it cannot be solved in polynomial time; i.e., the channel routing problem is NP-complete [30]. Therefore, routing traditionally is divided 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. The second phase, which is called detailed routing, finds the actual geometric layout of each net within the assigned routing regions (Figure 6.8b). 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. Detailed routing includes channel routing and switchbox routing. Another approach to routing is called area routing, which is a single-phase routing technique. However, this technique is computationally infeasible for general VLSI circuits and is typically used for specialized problems.

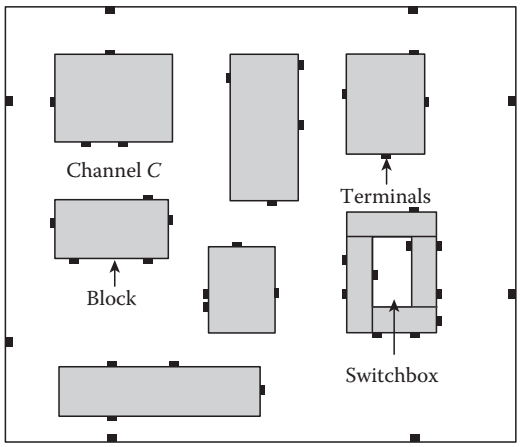


FIGURE 6.7 Layout of circuit blocks and pins after placement.

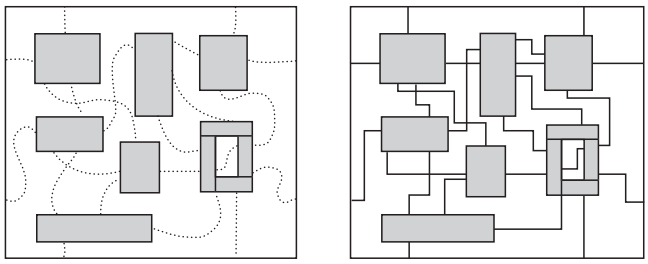


FIGURE 6.8 (a) Global routing and (b) detailed routing.

6.9 Classification of Global Routing Algorithms

Two approaches solve the global routing problem: the sequential and the concurrent:

1. *Sequential 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, perimeter of the bounding rectangle, and the number of terminals. The criticality of a net is determined by the importance of the net.
2. *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 from the top down are employed to partition the problem into smaller subproblems, which can be solved by integer programming.

6.10 Classification of Detailed Routing Algorithms

Many ways are possible for classifying the detailed routing algorithms. The algorithms could be classified on the basis of the routing models used. Some routing algorithms use grid-based models, while some other algorithms use the gridless model. The gridless model is more practical as all the wires in a design do not have the same width. Another possible classification scheme could be to classify the algorithms based on the strategy they use. Thus, we could have greedy routers or hierarchical routers to name two. We classify the algorithms based on the number of layers used for routing. Single-layer routing problems frequently appear as subproblems in other routing problems which deal with more than one layer. Two-layer routing problems have been thoroughly investigated because until recently, due to limitations of the fabrications process, only two metal layers were allowed for routing. A third metal layer is now allowed, thanks to improvements in the fabrication process, but it is expensive compared to the two-layer metal process. Several multilayer routing algorithms also were developed recently, which can be used for routing MCMs, which have up to 32 layers.

6.10.1 Lee's Algorithm for Global Routing

This algorithm, which was developed by Lee in 1961 [20], 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 on 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 wavefront

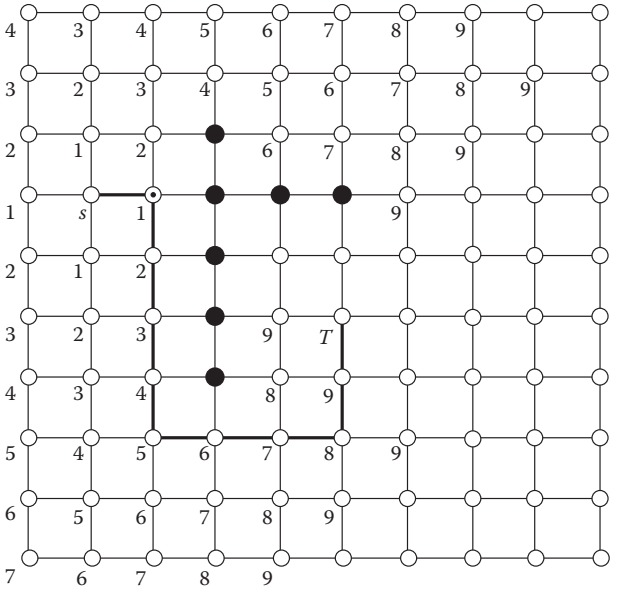


FIGURE 6.9 Net routed by Lee's algorithm.

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 6.9. Due to the breadth-first nature of the search, Lee's maze router is guaranteed to find a path between the source and the target, if one exists. In addition, it is guaranteed to be the shortest path between the vertices.

The input to 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 Lee's algorithm. Two linked lists, *plist* (propagation list) and *nlist* (neighbor list), are used to keep track of the vertices on the wavefront 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 counterclockwise order, that is, top, left, bottom, and then right.

The formal description of Lee's algorithm appears in Figure 6.10. The time and space complexity of Lee's algorithm is $O(h \times w)$ for a grid of dimension $h \times w$.

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

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 $h \times w$ grid, $k = \log_2(h \times w)$. Aker [1] 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 wavefront, L, are always adjacent to the vertices in wavefront $L-1$ and $L+1$. Thus, during wave propagation, instead of using a sequence 1, 2, 3, . . . , the wavefronts are labeled by a sequence such as 0, 0, 1, 1, 0, 0, ... The predecessor of any wavefront 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 2 bits, regardless of the problem size.

```
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 6.10 Lee's routing algorithm.

Compared with 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 $O(h \times w)$ in the worst case.

6.10.2 Greedy Channel Router for Detailed Routing

Assigning the complete trunk of a net or a two-terminal net segment of a multiterminal net severely restricts LEA and dogleg routers. Optimal channel routing can be obtained if for each column it can be guaranteed that only one horizontal track per net exists. Based on this observation, one approach to reduce channel height could be to route nets column by column trying to join split horizontal tracks (if any) that belong to the same net as much as possible.

Based on the preceding observation and approach, Rivest and Fiduccia [24] developed the greedy channel router. This makes fewer assumptions than LEA and dogleg routers. The algorithm starts from the leftmost column and places all the net segments of a column before proceeding to the next right column. In each column the router assigns net segments to tracks in a greedy manner. However, unlike the dogleg router, the greedy router allows the doglegs in any column of the channel, not necessarily where the terminal of the doglegged net occurs.

Given a channel routing problem with m columns, the algorithm uses several steps while routing a column. In the first step the algorithm connects any terminal to the trunk segment of the corresponding net. This connection is completed by using the first empty track, or the track that already contains the net. In other words, minimum vertical segment is used to connect a trunk to a terminal. For example, net 1 in Figure 6.11a in column 3 is connected to the same net. The second step attempts to collapse any split nets (horizontal segments of the same net present on two different tracks) using a vertical segment as shown in Figure 6.11b. A split net will occur when two terminals of the same net are located on different

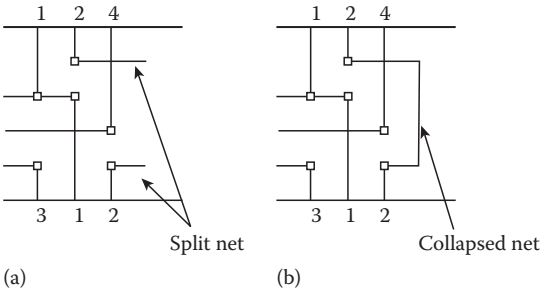


FIGURE 6.11 (a) Split net and (b) collapsed split net.

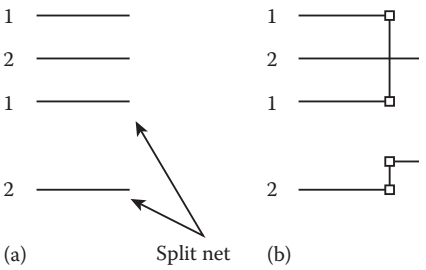


FIGURE 6.12 (a) Reducing the distance between split nets and (b) its result.

sides of the channel and cannot be connected immediately because of existing vertical constraints. This step also brings a terminal connection to the correct track if it has stopped on an earlier track. If two sets of split nets overlap, the second step will only be able to collapse one of them.

In the third step, the algorithm tries to reduce the range or the distance between two tracks of the same net. This direction is accomplished by using a dogleg, as shown in Figure 6.12a and b. The fourth step attempts to move the nets closer to the boundary which contains the next terminal of that net. If the next terminal of a net being considered is on the top (bottom) boundary of the channel, then the algorithm tries to move the net to the upper (lower) track. In case no track is available, the algorithm adds extra tracks and the terminal is connected to this new track. After all five steps have been completed, the trunks of each net are extended to the next column and the steps are repeated. A detailed description of the greedy channel routing algorithm is shown in Figure 6.13.

The greedy router sometimes gives solutions which contain an excessive number of vias and doglegs. It does, however, have the capability of providing a solution even in the presence of cyclic vertical constraints. The greedy router is more flexible in the placement of doglegs due to fewer assumptions about the topology of the connections. An example routed by the greedy channel router is shown in Figure 6.14.

6.10.3 Other Routing Algorithms

Soukup proposed an algorithm that basically cuts down the search time of Lee's algorithm by exploring in the direction toward the target, without changing the direction until it reaches the target or an obstacle. An alternative approach to improve upon the speed was suggested by Hadlock. One class of algorithm is called the line-probe algorithms. The basic idea of the line-probe algorithms is to reduce the memory requirement by using line segments instead of grid notes in the search. Several algorithms based on Steiner trees have been developed. The main advantage of these algorithms is that they can be used for routing multiterminal nets. For further details on these global routing algorithms, refer to Chapter 6 of [29].

Extensive research has been carried out in the area of detailed routing and several algorithms exist for channel and switchbox routing. There are LEA-based algorithms which use a reserved layer model and do not allow any vertical constraints or doglegs. The YACR2 algorithm can handle vertical constraint

```
Algorithm Greedy-Channel-Router (N)

Input: N

Begin
    d = DENSITY (N);
  (* calculate the lower bound of channel density *)
    insert d tracks to channel;
    for i = 1 to m do
      T1 =  GET-EMPTY-TRACK;
      if T1 = 0 then
        ADD-TRACK(T1);
        ADD-TRACK(T2);
      else
        T2 =  GET-EMPTY-TRACK;
        if T2 = 0 then
          ADD-TRACK(T2);
      CONNECT(T_i,T1);
      CONNECT(B_i,T2);
      join split nets as much as possible;
      bring split nets closer by jogging;
      bring nets closer to either top or bottom boundary;
    while split nets exists do
      increase number of column by 1;
      join split nets as much as possible;
End;
```

FIGURE 6.13 Algorithm greedy channel router.

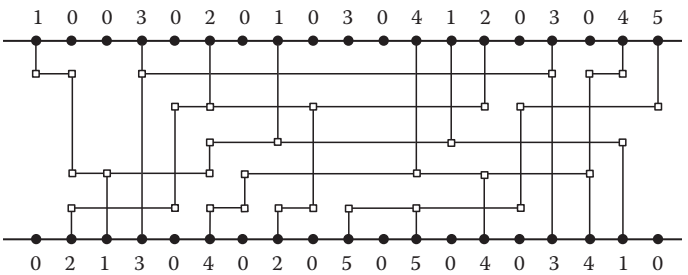


FIGURE 6.14 Channel routed using a greedy router.

violations. The net merge channel router works for two-layer channel routing problems and it exploits the graph theoretic structure of channel routing problems. Glitter is an algorithm for gridless channel routing and can handle nets of varying widths. The hierarchical channel router divides the channel routing problem into smaller problems, each of which is solved in order to generate the complete routing for the nets in the channel. Several algorithms such as the extended net merge channel routing algorithm, HVH routing from the HV solution, and the hybrid HVH–VHV channel routing algorithm exist for the three-layer channel routing problem. For further details on these detailed routing algorithms, refer to Chapter 7 of [29].

6.11 Compaction

After completion of detailed routing, the layout is functionally complete. At this stage, the layout is ready to be used to fabricate a chip. However, due to nonoptimality of placement and routing algorithms, some vacant space is present in the layout. In order to minimize the cost, improve performance, and yield, layouts are reduced in size by removing the vacant space without altering the functionality of the layout. This operation of layout area minimization is called layout compaction.

The compaction problem is simplified by using symbols to represent primitive circuit features, such as transistors and wires. The representation of layout using symbols is called a symbolic layout. Special languages [4,21,22] and special graphic editors [13,14] are available to describe symbolic layouts. To produce the actual masks, the symbolic layouts are translated into actual geometric features. Although a feature can have any geometric shape, in practice only rectangular shapes are considered.

The goal of compaction is to minimize the total layout area without violating any design rules. The area can be minimized in three ways:

1. *By reducing the space between features*: This can be performed by bringing the features as close to each other as possible. However, the spacing design rules must be met while moving features closer to each other.
2. *By reducing the size of each feature*: The size rule must be met while resizing the features.
3. *By reshaping the features*: Electrical characteristics must be preserved while reshaping the feature.

Compaction is a very complex phase in the physical design cycle. It requires understanding many details of the fabrication process such as the design rules. Compaction is critical for full-custom layouts, especially for high-performance designs.

6.11.1 Classification of Compaction Algorithms

Compaction algorithms can be classified in two ways. The first classification scheme is based on the direction of movements of the components (features): one-dimensional (1-D) and two-dimensional (2-D). In 1-D compaction components are moved only in x- or the y-direction. As a result, either the x- or the y-coordinate of the components is changed due to the compaction. If the compaction is done along the x-direction, then it is called x-compaction. Similarly, if the compaction is done along the y-direction, then it is called y-compaction. In 2-D compaction the components can be moved in both the x- and y-directions simultaneously. As a result, in 2-D compaction both x- and y-coordinates of the components are changed at the same time in order to minimize the layout area.

The second approach to classify the compaction algorithms is based on the technique for computing the minimum distance between features. In this approach, we have two methods: constraint graph-based compaction and virtual grid-based compaction. In the constraint graph method, the connections and separations rules are described using linear inequalities which can be modeled using a weighted directed graph (constraint graph). This constraint graph is used to compute the new positions for the components. On the other hand, the virtual grid method assumes the layout is to be drawn on a grid. Each component is considered attached to a grid line. The compaction operation compresses the grid along with all components placed on it, keeping the grid lines straight along the way. The minimum distance between two adjacent grid lines depends on the components on these grid lines. The advantage of the virtual grid method is that the algorithms are simple and can be easily implemented. However, the virtual grid method does not produce compact layouts as does the constraint graph method.

In addition, compaction algorithms can be classified on the basis of the hierarchy of the circuit. If compaction is applied to different levels of the layout, it is called hierarchical compaction. Any of the above-mentioned methods can be extended to hierarchical compaction. A variety of hierarchical compaction algorithms have been proposed for both constraint graph and virtual grid methods. Some compaction algorithms actually ''flatten the layout'' by removing all hierarchy and then

perform compaction. In this case it may not be possible to reconstruct the hierarchy, which may be undesirable.

6.11.2 Shadow-Propagation Algorithm for Compaction

A widely used and one of the best known techniques for generating a constraint graph is the shadow-propagation used in the CABBAGE system [15]. The ''shadow'' of a feature is propagated along the direction of compaction. The shadow is caused by shining an imaginary light from behind the feature under consideration (see Figure 6.15). Usually the shadow of the feature is extended on both sides of the features in order to account for diagonal constraints. This leads to greater than minimal Euclidean spacings because an enlarged rectangle is used to account for corner interactions. (See shadow of feature in Figure 6.15.)

When the shadow is obstructed by another feature, an edge is added to the graph between the vertices corresponding to the propagating feature and the obstructing feature. The obstructed part of the shadow is then removed from the front and no longer propagated. The process is continued until all of the shadow has been obstructed. This process is repeated for each feature in the layout. The algorithm shadow-propagation, given in Figure 6.16, presents an overview of the algorithm for *x*-compaction of a single feature from left to right.

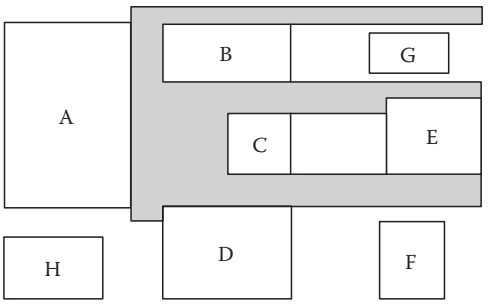


FIGURE 6.15 Example of shadow propagation.

The shadow-propagation routine accepts the list of components (*Comp_list*), which is sorted on the *x*-coordinates of the left corner of the components and the component (*component*) for which the constraints are to be generated. The procedure, INITIALIZE-SCANLINE, computes the total length of the interval in which the shadow is to be generated. This length includes the design rule separation distance. The *y*-coordinate of the top and the bottom of this interval are stored in the global variables, *top* and *bottom*, respectively. The procedure, GET-NXT-COMP, returns the next component (*curr_comp*) from *Comp_list*. This component is then removed

```
Algorithm Shadow-Propagation (Comp_list, component)
Input: Comp_list, component
Begin
    INITIALIZE-SCANLINE (component);
  I = ϕ;
   while ( (LENGTH-SCANLINE(I) < (top – bottom))
         and (Comp_list ≠ ϕ))
      curr_comp = GET-NXT-COMP (Comp_list);
      I_i = LEFT-EDGE (curr_comp);
      if (IN-RANGE (I_i, top, bottom))
         I' = UNION (I_i, I);
         if(I' ≠ I)
            ADD-CONSTRAINT (component, curr_comp);
            I = UNION(I_i, I);
End;
```

FIGURE 6.16 Shadow-propagation algorithm.

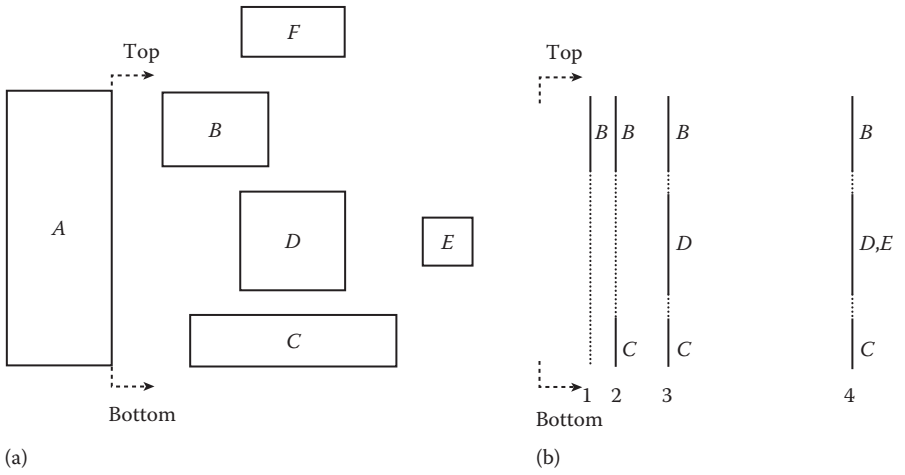


FIGURE 6.17 Interval generation for shadow propagation. (a) The layout of components and (b) the intervals in the interval set.

from the list. Procedure LEFT-EDGE returns the vertical interval of component, *curr_comp*. If this interval is within the *top* and *bottom* then *curr_comp* may have a constraint with *component*. This check is performed by the procedure IN-RANGE. If the interval for *curr_component* lies within *top* and *bottom* and if this interval is not already contained within one of the intervals in the interval set, $\mathcal{I}$, then the component lies in the shadow of *component*, and hence a constraint must be generated. Each interval represents the edge at which the shadow is blocked by a component. The constraint is added to the constraint graph by the procedure ADD-CONSTRAINT. The procedure UNION inserts the interval corresponding to *curr_comp* in the interval set at the appropriate position. This process is carried out until the interval set completely covers the interval from *top* to *bottom* or no more components are in *Comp_list*. Figure 6.17a shows the layout of components. The constraint for component *A* with other components is being generated. Figure 6.17b shows the intervals in the interval set as the shadow is propagated. From Figure 6.17b it is clear that the constraints will be generated between components *A* and components *B*, *C*, and *D* in that order. As component *F* lies outside the interval defined by *top* and *bottom* it is not considered for constraint generation. The interval generated by component *E* lies within one of the intervals in the interval set. Hence, no constraint is generated between components *A* and *E*.

6.11.3 Other Compaction Algorithms

Several algorithms such as constraint graph-based compaction algorithms, scanline algorithm, and grid-based compaction algorithms exist for the 1-D compaction problem. An algorithm based on a simulation of the zone-refining process also was developed. This compactor is considered a $1\frac{1}{2}$-D compactor, as the key idea is to provide enough lateral movements to blocks during compaction to resolve interferences. For further details on these algorithms, refer to Chapter 10 of [29].

6.12 Summary

The sheer size of the VLSI circuit, the complexity of the overall design process, the desired performance of the circuit, and the cost of designing a chip dictate that the whole design process must be automated. Also, the design process must be divided into different stages because of the complexity of the entire process. Physical design is one of the steps in the VLSI design cycle. In this step each component of a circuit is converted into a set of geometric patterns which achieves the functionality of the component.

The physical design step can be divided further into several substeps. All the substeps of the physical design step are interrelated. Efficient and effective algorithms are required to solve different problems in each of the substeps. Despite significant research efforts in this field, the CAD tools still lag behind the technological advances in fabrication. This calls for the development of efficient algorithms for physical design automation.

References

1. S. B. Aker, A modification of Lee's path connection algorithm, *IEEE Trans. Electron. Comput*., EC-16(1): 97–98, February 1967.
2. P. Bannerjee and M. Jones, A parallel simulated annealing algorithm for standard cell placement on a hypercube computer, *Proceedings of the IEEE International Conference on Computer Design*, IEEE Computer Society Press, Los Alamitos, CA, 1986, p. 34.
3. A. Chatterjee and R. Hartley, A new simultaneous circuit partitioning and chip placement approach based on simulated annealing, *Proceedings of the 27th ACM/IEEE Design Automation Conference*, IEEE Computer Society Press, Orlando, FL, June 24–28, 1990, pp. 36–39.
4. P. A. Eichenberger, Fast symbolic layout translation for custom VLSI integrated circuits, PhD thesis, Stanford University, Stanford, CA, 1986.
5. A. El Gamal et al., An architecture for electrically configurable gate arrays, *IEEE J. Solid-State Circuits*, 24(2): 394–398, April 1989.
6. H. Hseih et al., A 9000-gate user-programmable gate array, *Proceedings of the IEEE 1988 Custom Integrated Circuits Conference*, May 1988, pp. 15.3.1–15.3.7.
7. S. C. Wong et al., A 5000-gate CMOS EPLD with multiple logic and interconnect arrays, *Proceedings of the IEEE 1988 Custom Integrated Circuits Conference*, May 1989, pp. 5.8.1–5.8.4.
8. M. Feuer, VLSI design automation: An introduction, *Proc. IEEE*, 71(1): 1–9, January 1983.
9. C. M. Fiduccia and R. M. Mattheyses, A linear-time heuristics for improving network partitions, *Proceedings of the 19th Design Automation Conference*, Las Vegas, NV, 1982, pp. 175–181.
10. J. Greene and K. Supowit, Simulated annealing without rejected moves, *Proceedings of the International Conference on Computer Design*, IEEE Computer Society Press, Los Alamitos, CA, October 1984, pp. 658–663.
11. L. K. Grover, Standard cell placement using simulated sintering, *Proceedings of the 24th ACM/IEEE Design Automation Conference*, IEEE Computer Society Press, Miami Beach, FL, June 28–July 1, 1987, pp. 56–59.
12. B. Hajek, Cooling schedules for optimal annealing, Math. *Oper. Res*., 13(2): 311–329, May 1988.
13. D. D. Hill, ICON: A toll for design at schematic, virtual-grid and layout levels, *IEEE Des. Test*, 1(4): 53–61, 1984.
14. M. Y. Hsueh, Symbolic layout and compaction of integrated circuits, Technical Report No. UCB/ERL M79/80, Electronics Research Laboratory, University of California, Berkeley, CA, 1979.
15. M. Y. Hsueh and D. O. Pederson, Computer-aided layout of LSI circuit building-blocks, PhD thesis, University of California, Berkeley, CA, December 1979.
16. M. D. Huang, F. Romeo, and A. Sangiovanni-Vincentelli, An efficient general cooling schedule for simulated annealing, *Proceedings of the IEEE International Conference on Computer-Aided Design*, IEEE Computer Society Press, Los Alamitos, CA, 1986, pp. 381–384.
17. W. Kernighan and S. Lin, An efficient heuristic procedure for partitioning graphs, *Bell Syst. Tech. J*., 49: 291–307, 1970.
18. S. Kirkpatrick, G. D. Gellat, and M. P. Vecchi, Optimization by simulated annealing, *Science*, 220: 671–680, May 1983.
19. J. Lam and J. Delosme, Performance of a new annealing schedule, *Proceedings of the* 25*th ACM/IEEE Conference on Design Automation*, DAC'88, Anaheim, CA, June 12–15, 1988, pp. 306–311.

20. C. Y. Lee, An algorithm for path connections and its applications, *IRE Trans. Electron. Comput*., EC-10: 346–365, 1961.
21. T. M. Lin and C. A. Mead, Signal delay in general RC networks, *IEEE Trans. CAD*, CAD-3(4): 331–349, October 1984.
22. J. M. Da Mata, Allenda: A procedural language for the hierarchical specification of VLSI layout, *Proceedings of the 22nd ACM/IEEE Conference on Design Automation*, DAC'85, ACM Press, Las Vegas, NV, 1985, pp. 183–189.
23. T. Ohtsuke, *Partitioning, Assignment and Placement*, Amsterdam, the Netherlands: North-Holland, 1986.
24. R. Rivest and C. Fiduccia, A greedy channel router, *Proceedings of the 19th ACM/IEEE Design Automation Conference*, Las Vegas, NV, 1982, pp. 418–424.
25. F. Romeo and A. Sangiovanni-Vincentelli, Convergence and finite time behavior of simulated annealing, *Proceedings of the 24th Conference on Decision Control*, Ft. Lauderdale, FL, December 1985, pp. 761–767.
26. F. Romeo, A. S. Vincentelli, and C. Sechen, Research on simulated annealing at Berkeley, *Proceedings of the IEEE International Conference on Computer-Aided Design*, IEEE Computer Society Press, Los Alamitos, CA, 1984, pp. 652–657.
27. C. Sechen and K. W. Lee, An improved simulated annealing algorithm for row-based placement, *Proceedings of the IEEE International Conference on Computer-Aided Design*, IEEE Computer Society Press, Los Alamitos, CA, 1987, pp. 478–481.
28. C. Sechen and A. Sagiovanni-Vincentelli, The timber wolf placement and routing package, *IEEE J. Solid-State Circuits*, SC-20: 510–522, 1985.
29. N. Sherwani, *Algorithms for VLSI Physical Design Automation*, Boston: Kluwer Academic, 1993.
30. T. G. Szymanski, Dogleg channel routing is NP-Complete, *IEEE Trans. Comput. Aided Des*., CAD-4: 31–41, January 1985.

7

Design Automation Technology

7.1 Introduction 7-1
7.2 Design Entry 7-3
7.3 Conceptual Design 7-6
Design Planning • Decision Assistance • Exploring Alternative Designs • Applications
7.4 Synthesis 7-14
7.5 Verification 7-16
Timing Analysis • Simulation • Emulation
7.6 Test 7-21
Fault Modeling • Fault Testing
7.7 Frameworks 7-27
7.8 Summary 7-29
References 7-29

Allen M. Dewey*
Duke University

7.1 Introduction

The field of *design automation* technology, also commonly called computer-aided design (CAD) or computer-aided engineering (CAE), involves developing computer programs to conduct portions of product design and manufacturing on behalf of the designer. Competitive pressures to produce in less time and use fewer resources, new generations of products having improved function and performance are motivating the growing importance of design automation. The increasing complexities of microelectronic technology, shown in Figure 7.1, illustrate the importance of relegating portions of product development to computer automation [1,3]. Advances in microelectronic technology enable over 1 million devices to be manufactured on an integrated circuit substrate smaller in size than a postage stamp, yet the ability to exploit this capability remains a challenge. Manual design techniques are unable to keep pace with product design cycle demands and are being replaced by automated design techniques [2,4].

Figure 7.2 summarizes the historical development of design automation technology and computer programs. Design automation programs are also called ''applications'' or ''tools.'' Design automation efforts started in the early 1960s as academic research projects and captive industrial programs, focusing on individual tools for physical and logical design. Later developments extended logic simulation to more detailed *circuit* and *device* simulation and more abstract *functional* simulation. Starting in the mid to the

* Allen M. Dewey passed away before the publication of this edition.

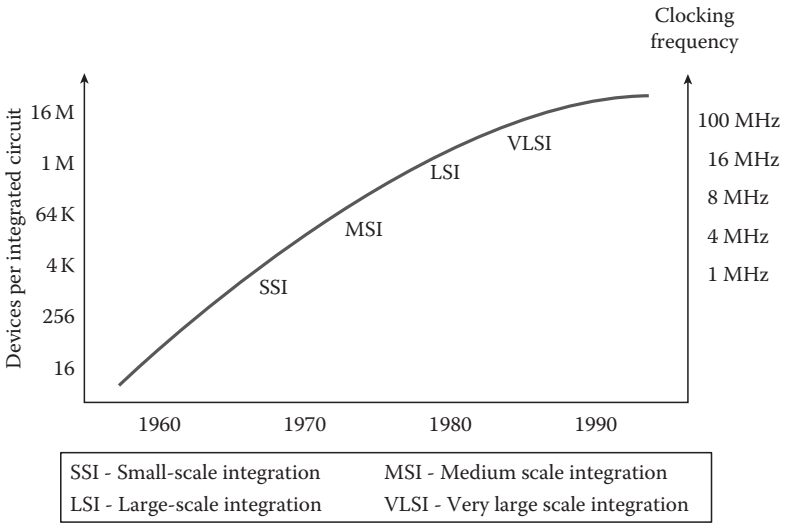


FIGURE 7.1 Complexity of microelectronic technology.

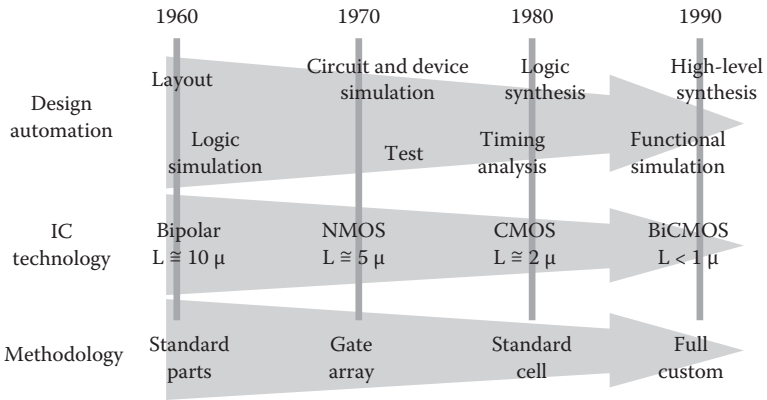


FIGURE 7.2 Development of design automation technology.

late 1970s, new areas of test and synthesis emerged and commercial design automation products appeared. Today, the electronic design automation industry is an international business with a well-established and expanding technical base [5]. The electronic design automation technology base will be examined by presenting an overview of the following topical areas:

- Design entry
- Conceptual design
- Synthesis
- Verification
- Testing
- Frameworks

7.2 Design Entry

Design entry, also called design capture, is the process of communicating with a design automation system. Design entry involves describing a system design to a design automation system, invoking applications to analyze and/or modify the system design, and querying the results. In short, design entry is how an engineer ''talks'' to a design automation application and/or system.

Any sort of communication is composed of two elements: language and mechanism. Language provides common semantics; mechanism provides a means by which to convey the common semantics. For example, two people communicate via a language such as English, French, or German, and a mechanism, such as a telephone, electronic mail, or facsimile transmission. For design, a digital system can be described in many ways, involving different perspectives or abstractions. An abstraction is a model for defining the behavior or semantics of a digital system, i.e., how the outputs respond to the inputs. Figure 7.3 illustrates several popular levels of abstractions and the trends toward higher levels of design entry abstraction to address greater levels of complexity [10,12].

The physical level of abstraction involves geometric information that defines electrical devices and their interconnection. Geometric information includes the shaped objects and where objects are placed relative to each other. For example, Figure 7.4 shows the geometric shapes defining a simple complementary metal–oxide semiconductor (CMOS) inverter. The shapes denote different materials, such as aluminum and polysilicon, and connections, called *contacts* or *vias*.

The design entry mechanism for physical information involves textual and graphic techniques. With textual techniques, geometric shape and placement are described via an artwork description language, such as Caltech Intermediate Form or Electronic Design Intermediate Form. With graphical techniques, geometric shape and placement are described by drawing the objects on a display terminal [13].

The electrical level abstracts geometric information into corresponding electrical devices, such as capacitors, transistors, and resistors. For example, Figure 7.5 shows the electrical symbols denoting a CMOS inverter. Electrical information includes the device behavior in terms of terminal or pin current and voltage relationships. Device behavior also may be defined in terms of manufacturing parameters, such as resistances or chemical compositions.

The logical level abstracts electrical information into corresponding logical elements, such as `and` gates, `or` gates, and inverters. Logical information includes truth table and/or characteristic switching algebra equations and active level designations. For example, Figure 7.6 shows the logical symbol for a CMOS inverter. Notice how the amount of information decreases as the level of abstraction increases.

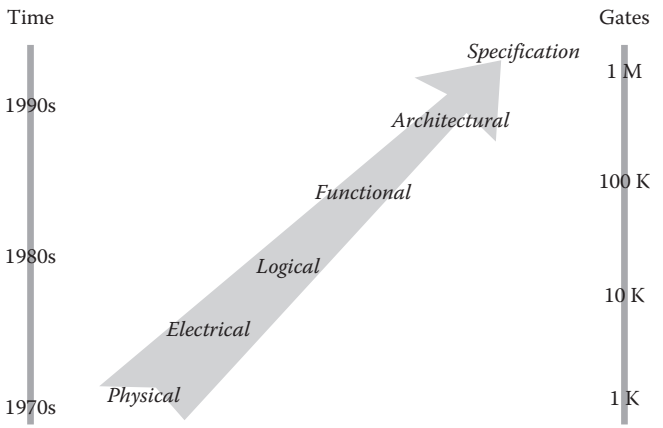


FIGURE 7.3 Design automation abstractions.

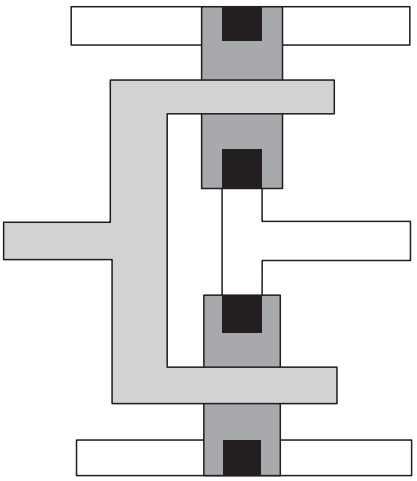


FIGURE 7.4 Physical information.

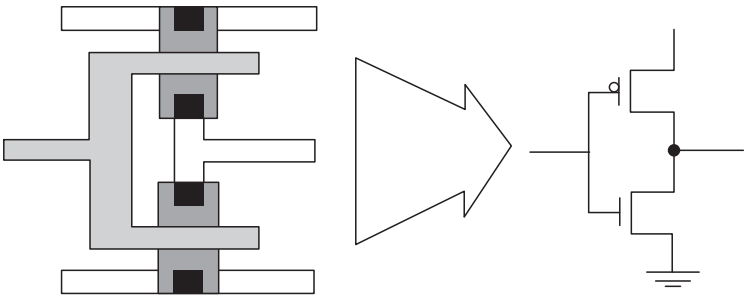


FIGURE 7.5 Electrical information.

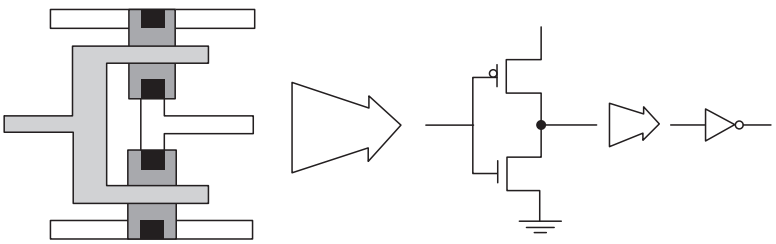


FIGURE 7.6 Logical information.

Design entry mechanisms for electrical and logical abstractions are commonly collectively called *schematic* capture techniques. Schematic capture defines hierarchical structures, commonly called *netlists*, of components. A designer creates instances of components supplied from a library of predefined components and connects component pins or ports via wires [9,11].

The functional level abstracts logical elements into corresponding computational units, such as registers, multiplexers, and arithmetic logic units (ALUs). The architectural level abstracts functional

information into computational algorithms or paradigms. Some examples of common computational paradigms are

- State diagrams
- Petri nets
- Control/data flow graphs
- Function tables
- Spreadsheets
- Binary decision diagrams

These higher levels of abstraction support a more expressive ''higher bandwidth'' communication interface between an engineer and design automation programs. An engineer can focus his or her creative, cognitive skills on concept and behavior, rather than the complexities of detailed implementation. Associated design entry mechanisms typically use hardware description languages with a combination of textual and graphic techniques [6].

Figure 7.7 shows a simple state diagram. The state diagram defines three states, denoted by circles. State-to-state transitions are denoted by labeled arcs; state transitions depend on the present state and the input *X*. The output, *Z*, per state is given within each state. Because the output is dependent on only the present state, the digital system is classified as a *Moore finite state machine*. If the output is dependent on the present state and input, then the digital system is classified as a *Mealy finite state machine*. The state table corresponding to the state diagram in Figure 7.7 is given in Table 7.1. A hardware description language model written in VHDL of the Moore finite state machine is given in Figure 7.8. The VHDL model, called a *design entity*, uses a ''date flow'' description style to describe the state machine [7,8]. The

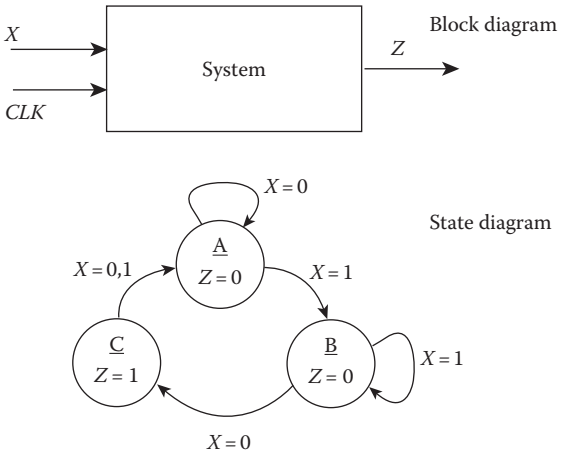


FIGURE 7.7 State diagram.

TABLE 7.1 State Table

	Next State		
Present State	$X = 0$	$X = 1$	Output
A	A	B	0
B	C	B	0
C	A	A	1

```
entity MOORE_MACHINE is
  port (X, CLK: in BIT; Z: out BIT);
end MOORE_MACINE;
architecture FSM of MOORE_MACHINE is
  type STATE_TYPE is (A, B, C);
  signal STATE: STATE_TYPE: = A;
begin
  NEXT_STATE:
  block (CLK = '1' and not CLK'STABLE)
  begin
     STATE <= guarded B when (STATE = A and X = '1') else
                          C when (STATE = B and X = '0') else
                          A when (STATE = C) else
                          STATE;
  end block NEXT_STATE;

  with STATE select
     Z <= '0' when A,
          '0' when B,
          '1' when C;
end FSM;
```

FIGURE 7.8 VHDL model.

entity statement defines the interface, i.e., the ports. The ports include two input signals, *X* and *CLK*, and an output signal, *Z*. The ports are of type BIT, which specifies that the signals may only carry the values "0" or "1." The architecture statement defines the I–O transform via two concurrent signal assignment statements. The internal signal STATE holds the finite state information and is driven by a guarded, conditional concurrent signal assignment statement that executes when the associated block expression (*CLK* = "1" and not CLK'STABLE) is true, which is only on the rising edge of the signal *CLK*. STABLE is a predefined attribute of the signal *CLK*; CLK'STABLE is true if *CLK* has not changed value. Thus, if not CLK'STABLE is true, meaning that *CLK* has just changed value, and *CLK* = "1", then a rising transition has occurred on *CLK*. The output signal *Z* is driven by a nonguarded, selected concurrent signal assignment statement that executes anytime STATE changes value.

7.3 Conceptual Design

Figure 7.9 shows that the conceptual design task generally follows the design entry task. Conceptual design addresses the initial decision-making process by assisting the designer to conduct initial feasibility studies, determine promising design alternatives, and obtain preliminary indication of estimated performance [16].

During early stages of the electronic design process, one typically makes a number of high-level decisions that can have a significant impact on the outcome of a design. For example, consider the early decisions usually made when designing a memory subsystem. The designer must choose a configuration (e.g., ×1, ×4, ×8, etc.), memory technology (e.g., static RAM, dynamic RAM, mask programmable ROM, etc.), package style (e.g., dual-in-line, chip carrier, pin grid array, etc.), and so on. Taken collectively, these decisions may be referred to as the design plan. The design plan then guides the ensuing, more detailed design process, acting as a high-level road map.

In general, many different plans may result in designs that satisfy the same set of functional specifications, but have varying trade-offs in terms of performances, such as area or power dissipation. For example, Figure 7.10 shows relative area and speed performance trade-offs for six combinational multiplier designs. Figure 7.10 shows the typical trend that increasing speed implies increasing area.

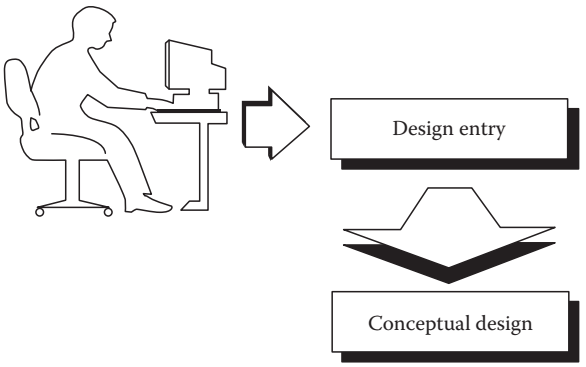


FIGURE 7.9 Design process.

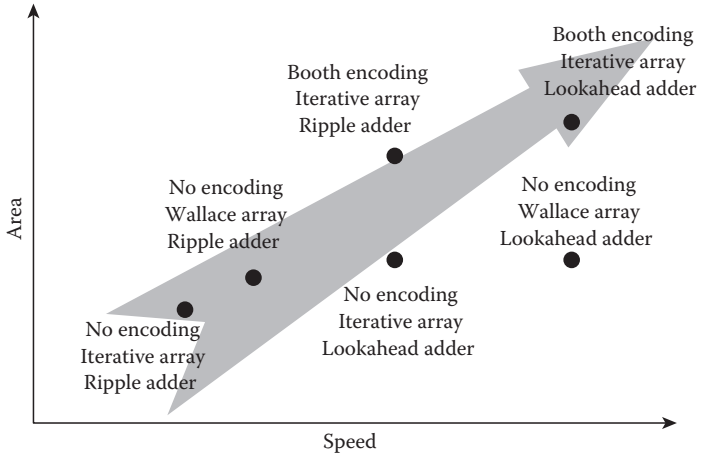


FIGURE 7.10 Performance trade-offs.

More importantly, the chart indicates how various multiplier designs relate to each other and which designs present the best area/speed trade-offs for a particular application.

Given this large set of possible design plans to consider, it is easy to imagine getting hopelessly lost in analyzing the various design plans and keeping track of the trade-offs and decision interdependencies. Choosing the most appropriate design plan is important because it is usually not economically feasible to choose an alternative plan and repeat the detailed design process if the final design does not meet the desired specifications. To aid the designer during the early stages of the design process in maximizing his or her chances that certain high-level decisions will result in a design that meets the desired specifications, conceptual design tools assist the designer in analyzing possible ramifications of high-level design decisions before actually undertaking the detailed design and fabrication process.

During conceptual design, the engineer has a global view of the design process in considering the key decisions associated with each lower level of abstraction. Thus, the engineer can undertake a breadth-first design methodology. Costly errors can be avoided by investing time at the outset in thoroughly thinking through a particular plan, the result being a more complete and credible specification that better guides the ensuing actions. In addition, the unnecessary use of detailed design CAD tools is minimized because conceptual design acts as a ''coarse filter'' eliminating infeasible design plans.

7.3.1 Design Planning

To discuss conceptual design, it is convenient to introduce the following terms:

- *Design issue*—Pertinent aspect of a design for which a decision must be made.
- *Design option*—Possible choice that can be made for a design issue.
- *Design decision*—Particular option chosen for a design issue.
- *Design plan*—Set of design decisions that identifies a high-level approach for implementing the design.

Conceptual design begins with the identification of the set of relevant design issues that must be considered before detailed design begins. For example, for the design of an analog amplifier, the design issues include selection of a type of amplifier (e.g., current, voltage, etc.), selection of a class of operation (e.g., A, B, AB, etc.), selection of a biasing scheme, etc. For the design of a digital signal processing filter, the design issues include selection of an algorithm (e.g., infinite impulse response or finite impulse response), selection of an architecture (e.g., direct, cascade, parallel, etc.), selection of a multiplier logic design, etc. Having identified the relevant design issues, conceptual design proceeds to develop a design plan in the step-by-step fashion outlined in Figure 7.11. For each design issue, the various associated options are examined, and the most appropriate option is selected. To make this selection, the designer chooses some method of discrimination and may need expert advice about the possible options or may desire a prediction about the effect a particular option has on performance. Such a conceptual design approach implies several important features, which are considered in more detail next.

Designers often structure design issues in a hierarchical fashion to control the complexity of the conceptual design process. Complex design issues are decomposed into sets of design subissues or subplans [22]. For example, an expanded, hierarchical listing of candidate design issues for digital signal processing filters is given next [14,20]:

1. Selection of a filter algorithm
2. Selection of a polynominal approximation technique
3. Selection of a filter architecture
   a. Selection of a filter structure
   b. Selection of a pipelining strategy
4. Selection of a logic design
   a. Selection of a multiplier logic design
   b. Selection of an encoding technique

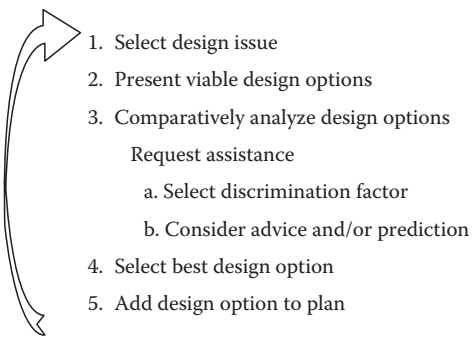


FIGURE 7.11 Constructing a design plan.

c. Selection of an array design
d. Selection of a final adder design
e. Selection of an adder design style
5. Selection of a fabrication technology
6. Selection of a layout design style

The selection of a multiplier logic design can be decomposed into three design subissues: selection of an encoding technique, selection of a combinational array design, and selection of a final adder design. Notice that many details about digital filter design are not listed because conceptual design is typically not concerned with every design issue required to eventually fully design and fabricate the initial specification. Hierarchical design issues provide for a more intuitive representation of the conceptual design process, enabling the designer to better comprehend the overall magnitude and composition of the design domain.

Each design issue has associated commonly used options or selections. For example, the possible digital filter polynomial approximation techniques are

1. Equiripple
2. Frequency sampling
3. Windowing
4. Beta
5. Elliptic
6. Butterworth
7. Chebyshev

During conceptual design, the order in which the designer examines the various issues may not be completely arbitrary because the options available for a design issue may depend on which options were chosen for other design issues. Hence, a partial ordering may be imposed on addressing design issues. As an example, equiripple, frequency sampling, windowing, and beta digital filter approximations apply only to the finite impulse response algorithm, while elliptic, Butterworth, and Chebyshev digital filter approximations apply only to the infinite impulse response algorithm. Hence, the polynomial approximation technique depends on the choice of the filter algorithm and thus, the choice of a filter algorithm must precede the choice of a polynomial approximation technique. Similarly, if the designer chooses a bipolar logic family, then the Manchester carry adder logic design is not a viable option because this option requires pass transistor technology that exists only in MOS-related logic families. Hence, the adder's logic design depends on the choice of the logic family and thus, the choice of a logic family must precede the choice of a logic design style.

Conceptual design supports such ordering of design issues via ordering constraints, enforcing that the more pervasive design issues be addressed first [23]. In specifying the interdependencies between design issues, as many or as few ordering constraints may be imposed as is warranted by the particular design domain. At one extreme, a strict sequential ordering of design issues may be required. At the other extreme, no ordering may occur, signifying that the design issues have no interdependencies and that each issue may be examined independently of the others. Partial orderings may also occur in which some design issues typically arise that may be examined and some design issues that may not be examined, owing to interdependencies. Hence, ordering constraints provide a general mechanism within the conceptual design methodology for encoding various design strategies, such as top-down, bottom-up, or meet-in-the-middle.

In addition to interdependencies between design issues (ordering constraints), there may be interdependencies between options. For the digital adder design example, in which the designer investigates the logic design style issue, only the options that are consistent with the selected logic family option are viable. Interdependencies between design options are supported via consistency constraints. Using

consistency constraints, a system planner helps to determine, for a particular design issue, which options are consistent with earlier decisions. The designer is not presented with an option (step 2 in Figure 7.11) that is inconsistent with his or her earlier decisions. As such, the designer is prevented from developing an incorrect or inconsistent plan.

7.3.2 Decision Assistance

It may be the situation for complex conceptual designs that the designer consults an expert on a particular issue. The expert offers assistance in understanding the relative advantages and disadvantages of the available options in order to select the most appropriate option(s). Two general forms of assistance, advice and prediction, are available.

Advice describes the qualitative advantages and/or disadvantages that should be considered before accepting or rejecting a design option. On the other hand, prediction describes the quantitative advantages and/or disadvantages. Prediction is a valuable tool for reaching down through several levels of the design hierarchy to obtain quantitative insights into the implications of high-level design decisions. Such advice is given in lieu of the more expensive route of actually executing the detailed design process in order to determine the effects of a decision on design performance. While the predicted performance is only an approximation, it is often sufficient to allow the designer to choose a particular option, or at least to rule out an option.

Different types of prediction techniques are summarized in the taxonomy presented in Figure 7.12. Heuristic models use general guidelines or empirical studies to model the design process [15]. Analytical models employ a more rigorous, mathematical approach to model the design process. Some such models may be based on probability theory, e.g., wirability theory, and capture the particulars of a design activity in a set of random variables [17,18], while other analytical models may be based on graph and information theory, e.g., very large scale integration (VLSI) complexity theory [19,24,25]. Finally, simulation models include the judicious partial execution of a design activity on critical portions of the design. The use of previous designs serves to capture the experience or ''corporate history'' factor used by an expert designer in attempting to predict the performance of a new function based on previous similar designs.

As an example, Figure 7.13 shows how performance estimates for digital filters can be obtained in a hierarchical, incremental fashion, employing a variety of prediction models that cooperate to yield the

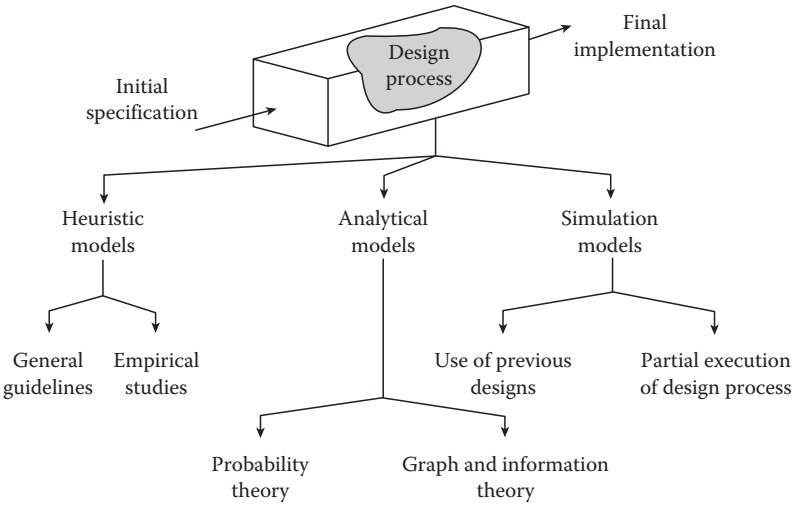


FIGURE 7.12 Types of prediction techniques.

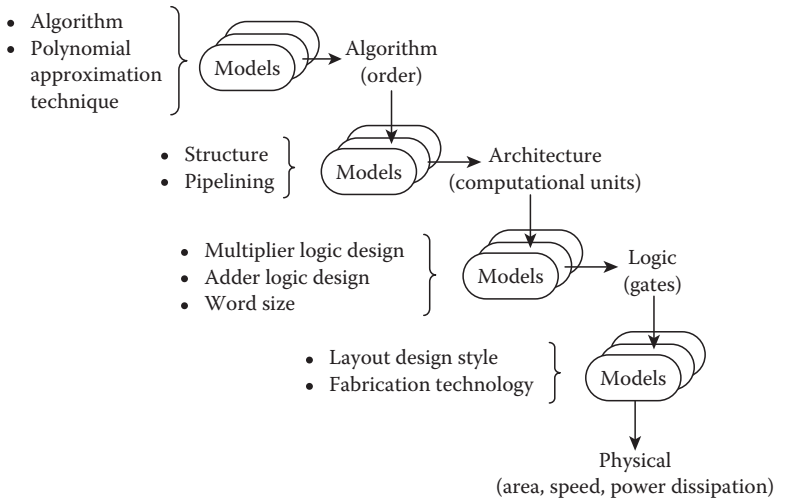


FIGURE 7.13 Hierarchical structure of filter performance prediction.

desired results. The prediction task is performed by incrementally translating the initial specifications into a series of complexity measures. Each succeeding complexity measure is a refinement of the performance estimate, starting with a performance estimate in terms of the filter order and ending with the performance estimate in terms of the physical units of area, speed, and power dissipation. Each prediction model draws its required input information from higher-level prediction models, the initial functional specifications, and the applicable decisions from the conceptual plan. Incrementally generating a prediction allows the flexibility to use different types of prediction techniques as the situation warrants to model the implications of different parts of the design plan.

The starting point in the prediction methodology is the initial functional specifications. For digital filters, functional specifications are usually given in terms of the desired frequency response, involving the following:

- Frequency bands of importance
- Widths of the transition bands between the desired frequency bands
- Levels of amplification or attenuation per frequency band
- Allowable tolerance or deviation from the specified levels of amplification and/or attenuation

An extensive set of prediction models has been developed for predicting the performance of linear, shift-invariant digital filters having the general I–O behavior given by the difference equation

$$y(n) = \sum_{k=1}^{N} a_k(n-k) + \sum_{k=0}^{M} b_k x(n-k) \tag{7.1}$$

The variables $x(n)$ and $y(n)$ denote the values of signals $x(t)$ and $y(t)$ at time t_n. As an example of an algorithm-level prediction model, the order of a low-pass filter using the equiripple polynomial approximation technique is given by the empirical model shown in Equation 7.2 [21].

$$N \approx \frac{\left\{ \begin{array}{c} \left[5.309 \times 10^{-3}(\log_{10}\delta_1)^2 + 0.07114\log_{10}\delta_1 - 0.4761\right]\log_{10}\delta_2 \\ + \left[-2.66 \times 10^{-3}\left(\log_{10}\delta_1^2\right) - 0.5941\log_{10}\delta_1 - 0.4278\right] \end{array} \right\}}{\Delta F} - \left[0.51244(\log_{10}\delta_1 - \log_{10}\delta_2) + 11.01217\right]\Delta F + 1 \tag{7.2}$$

where N is the filter order, δ_1 is the passband tolerance, δ_2 is the stopband tolerance, and ΔF is the width of the transition band. As another example of an algorithm prediction model, the order of a low-pass filter using the elliptic polynomial approximation technique is given by

$$N = \frac{\Psi(\alpha)\Psi\left(\sqrt{1-\beta^2}\right)}{\Psi(\beta)\Psi\left(\sqrt{1-\alpha^2}\right)}$$
$$\alpha = \frac{\omega_p}{\omega_s} \tag{7.3}$$
$$\beta = \frac{\sqrt{\delta_1}\sqrt{2-\delta_1\delta_2}}{\sqrt{1-\delta_2^2(1-\delta_1)}}$$

where ω_p is the cutoff frequency or the high end of the passband, ω_s is the start or low end of the stopband, and $\Psi(\cdot)$ denotes the complete elliptic integral of the first kind.

As an example of a logic level prediction model, filter word size can be estimated to be the maximum of the required data word size and coefficient word size. Equation 7.4 estimates the data word size ($data_{ws}$), i.e., number of bits required to represent the filter input data samples to ensure an adequate signal:noise ratio:

$$data_{ws} = \left\lceil \frac{SNR_q - 4}{6} \right\rceil \tag{7.4}$$

where $[\cdot]$ denotes the ''smallest integer not less than'' and SNR_q denotes the root mean square signal: quantization noise ratio in decibels. Equation 7.5 estimates the coefficient word size ($coeff_{ws}$), i.e., number of bits required to represent the filter coefficients to ensure that finite precision coefficient quantization and arithmetic round-off errors do not appreciably deteriorate filter frequency response:

$$coeff_{ws} = \left\lceil \log_2\left(\frac{\omega_s + \omega_p}{\omega_s - \omega_p}\right) + \log_2\left(\frac{1}{\min(\delta_1, \delta_2)}\right) - \frac{1}{2}\log_2\left(\frac{\omega_S}{\omega_s - \omega_p}\right) + 3 \right\rceil \tag{7.5}$$

where ω_S denotes the sampling frequency.

7.3.3 Exploring Alternative Designs

After analyzing the viable options, the final steps in Figure 7.11 involve making the design decision. Sometimes designers want to delay making a decision during a conceptual design session until other aspects of the design can be explored. If more than one option for a particular design issue looks equally viable, then the designer may elect to consider all of the viable options. For example, returning to digital filters, the designer may wish to investigate the implications of using both the lookahead carry and block lookahead carry logic design styles for the adder. Later in the conceptual design session, when the competing adder design plans have been more fully developed and their relative merits more clearly understood, the designer may have the information necessary to select one logic design style over the other. Such an approach is often referred to as ''nonlinear planning'' or adopting a ''least-commitment strategy.''

Multiple, alternative plans support delaying design decisions. New plans are created each time more than one option is chosen for a design issue. For example, if the designer wishes to investigate both adder logic designs, then the planner records the decision by generating two plans. One design plan uses the lookahead carry adder logic design, and the other plan uses the block lookahead carry logic design. Having generated alternative plans, the designer can then further develop each plan independently by requesting that the planner switch the focus of the conceptual design session between the competing

plans. It should be noted that a consequence of creating alternative plans is that the conceptual design process may not always yield one plan. Rather, the conceptual design process may yield multiple plans that appear roughly equally promising. The conceptual design methodology is not intended to always possess the ability to definitively differentiate between alternative plans due to the general nature of the plans or the approximate nature of the predictions. In cases in which the conceptual design process yields more than one candidate plan, the designer could invoke more detailed synthesis and analysis design automation tools to further investigate and resolve the plans.

In a lengthy conceptual design session, a designer may want to reconsider and possibly change an earlier decision; this situation is supported via backtracking. When the designer requests a reconsideration of an earlier decision for a particular design issue, the planner must back up the state of the conceptual design session and reactivate the associated viable options. The designer may then reexamine the options and possibly select a different option. The backtracking process also involves checking the rest of the plan to see if any of the designer's other, earlier decisions might be affected by the new decision. If any other design decisions depend on the decision that the designer wants to change, then these decisions must be similarly undone. In this manner only the appropriate part of the conceptual design session is backed up; the decisions that are not dependent on the desired change are left unaffected.

7.3.4 Applications

Conceptual design involves well-defined, algorithmic-based knowledge and ill-defined, heuristic-based knowledge. For instance, maintaining alternative plans involves primarily algorithmic-based knowledge. For each plan, the available design issues are determined in accordance with the ordering constraints. For each available design issue, the viable options are determined in accordance with the decisions made thus far and the consistency constraints. As the conceptual design session progresses, new design decisions are recorded, or possibly, previous design decisions are changed.

In contrast, providing advice involves primarily heuristic-based knowledge. Based on the state of the plan, the appropriate piece of advice or prediction model is identified, guidance is formulated, and the results are supplied to the designer. Advice and predictions serve as knowledge sources working to solve the conceptual design ''problem.'' The knowledge sources opportunistically invoke themselves to address an applicable portion of the advice or prediction request, taking their input from the system plan(s), and posting their results back to the system plan(s) in an attempt to refine and develop a conceptual design.

Domain-dependent information concerning design issues, options, trade-offs, and interdependencies is typically specified as part of the knowledge acquisition process involved in initially constructing a system planner, and is separate from domain-independent information to facilitate creating planners for different design tasks and moving existing planners to new design tasks. If alternative plans are developed during conceptual design, multiple versions of the domain-dependent information are correspondingly created. Each version of the domain-dependent information, also called contexts, worlds, or belief spaces, contains a plan's state information, such as which design issues have been addressed, which options have been chosen, and which advice has been given. When a designer changes the focus of the conceptual design session between alternative plans, a context switch is performed between the associated versions of the domain-dependent information. An analogous situation is an operating system that allows multiple jobs. Each time the computer switches to a new job, the state of the current job must be saved, and the state of the new job must be restored.

Clearly, for complex design domains, the amount of domain-dependent information can be sizable. To limit the search time required to find applicable options or appropriate advice, the domain-dependent information can be segmented or partitioned by design issue. Such an organization allows for dynamically limiting the portion of information searched. For instance, in searching for appropriate advice, only the advice that is associated with the current design issue is made accessible. Thus, any search is confined to the advice associated with the current design issue; a search is never executed over the entire domain-dependent information.

Conceptual design is an emerging area of design automation that demonstrates the potential for significantly improving our design capability. Conceptual design involves studying a design process, modeling the process as a hierarchy of design issues, delineating the options, and identifying the associated interdependencies and discriminating characteristics. A conceptual design plan realizes the initial specifications by delineating the important design decisions. Such a methodology emphasizes spending more time at the initial stage of the design process in obtaining a more credible system specification. In addition to beginning an expensive design process with a careful definition of what is required of the design, conceptual design emphasizes the need for a general plan for how to realize the design.

Associated applications play a supporting role of managing and presenting the large and complex amount of information typically involved in conceptual design to assist the designer in making the decisions that constitute the conceptual design plan. Plans are developed incrementally and interactively. In a breadth-first fashion, conceptual design guides the designer through a myriad of alternative plans, allowing the designer to explore differences and seek optimal solutions. The advantage of conceptual design is that expensive design resources are not wasted in trying plans that eventually prove unsatisfactory. Applications also provide performance estimation, which can be useful in coordinating between groups involved with a design. For example, an early estimate of chip area can be used to check with the manufacturing group that the design will be within acceptable fabrication yield margins or will adhere to any packaging or printed circuit board size restrictions. An early estimate of chip performance also can be used to coordinate with marketing and management in projecting potential applications, probable costs, and future business.

7.4 Synthesis

Figure 7.14 shows that the synthesis task generally follows the conceptual design task. The designer describes the desired system via design entry, generates a conceptual plan of major design decisions, and then invokes synthesis design automation programs to assist in generating the required implementation [32].

Synthesis translates or transforms a design at one level of abstraction to another, more detailed level of abstraction. The more detailed level of abstraction may be only an intermediate step in the entire design process or it may be the final implementation. Synthesis programs that yield a final implementation are sometimes called ''silicon compliers'' because the programs generate sufficient detail to proceed directly to silicon fabrication [26,29].

Similar to design abstractions, synthesis techniques can be hierarchically categorized, as shown in Figure 7.15. The higher levels of synthesis offer the advantage of less complexity, but also the disadvantage of less control over the final implementation.

Algorithmic synthesis also called ''behavioral synthesis,'' addresses ''multicycle'' behavior, which means behavior that spans more than one control step. A control step equates to a clock cycle of a synchronous, sequential digital system, i.e., a state in a finite state machine controller or a microprogram step in a microprogrammed controller. Algorithmic synthesis typically accepts sequential design descriptions using a procedural or imperative modeling style. Such descriptions define the I–O transform, but provide little information about the parallelism of the implementation [28,30].

Partitioning decomposes the design description into smaller behaviors. Partitioning is an example of a

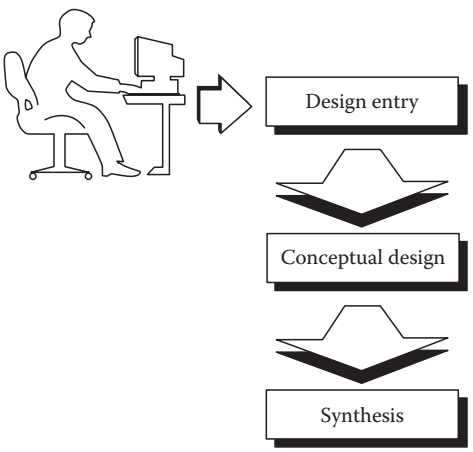


FIGURE 7.14 The design process.

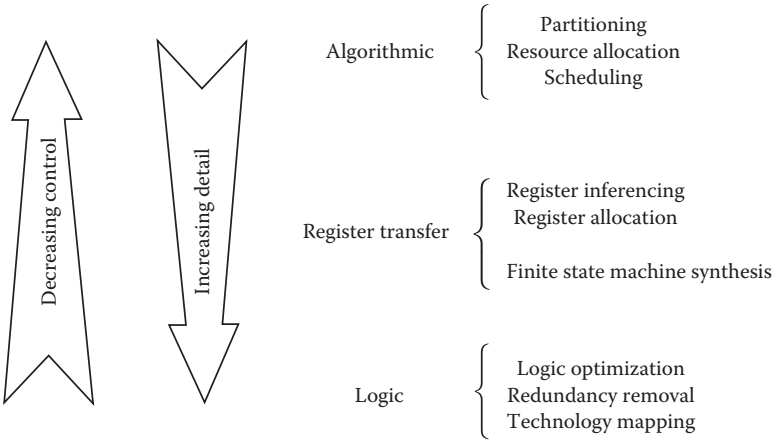


FIGURE 7.15 Taxonomy of synthesis techniques.

high-level transformation that modifies the initial sequential design description to optimize a hardware implementation. High-level transformations include several common software programming complier optimizations, such as loop unrolling, subprogram in-line expansion, constant propagation, and common subexpression elimination.

Resource allocation associates behaviors with hardware computational units and scheduling determines the order in which behaviors execute. Behaviors that are mutually exclusive can potentially share computational resources. Allocation is performed using a variety of graphic clique covering or node coloring algorithms. Allocation and scheduling are interdependent and different synthesis strategies perform allocation and scheduling in different ways. Sometimes scheduling is performed first, followed by allocation; sometimes allocation is performed first, followed by scheduling; and sometimes allocation and scheduling are interleaved.

Scheduling assigns computational units to control steps, thereby determining which behaviors execute in which clock cycles. At one extreme, all computational units can be assigned to a single control step, exploiting maximum concurrency. At the other extreme, computational units can be assigned to individual control steps, exploiting maximum sequentiality. Figure 7.16 illustrates two possible schedules

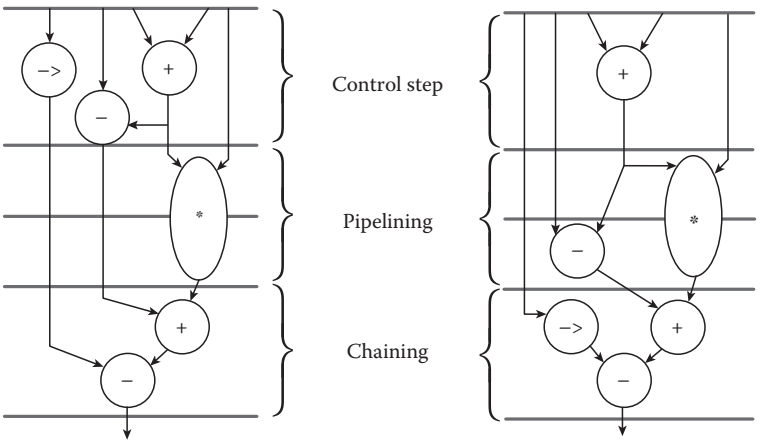


FIGURE 7.16 Scheduling.

of the same behavior. *Pipelining* schedules an operation across multiple control steps, whereas *chaining* combines multiple operations within a single control step.

Several popular scheduling algorithms are

- As soon as possible (ASAP)
- As late as possible (ALAP)
- List scheduling
- Force directed scheduling
- Control step splitting/merging

ASAP and ALAP scheduling algorithms order the computational units based on data dependencies. List scheduling is based on ASAP and ALAP scheduling, but can consider additional global constraints, such as a maximum number of control steps. Force-directed scheduling computes the probabilities of computational units being assigned to control steps and attempts to evenly distribute computation activity among all control steps. Control step splitting scheduling starts with all computational units assigned to one control step and generates a schedule by splitting the computational units into multiple control steps. Control step merging scheduling starts with all computational units assigned to individual control steps and generates a schedule by merging or combining units and steps [28,31].

Register transfer synthesis takes as input the results of algorithmic behavior and addresses ''per-cycle'' behavior, which means the behavior during one clock cycle. Register transfer synthesis selects logic to realize the hardware computational units generated during algorithmic synthesis, such as realizing an addition operation with a carry-save adder or realizing addition and subtraction operations with an ALU. Data that must be retained across multiple clock cycles are identified and registers are allocated to hold these data. Finally, state machine synthesis involves classical state minimization and state assignment techniques. State minimization seeks to eliminate redundant or equivalent states and state assignment assigns binary encodings for states to minimize combinational logic [27,33].

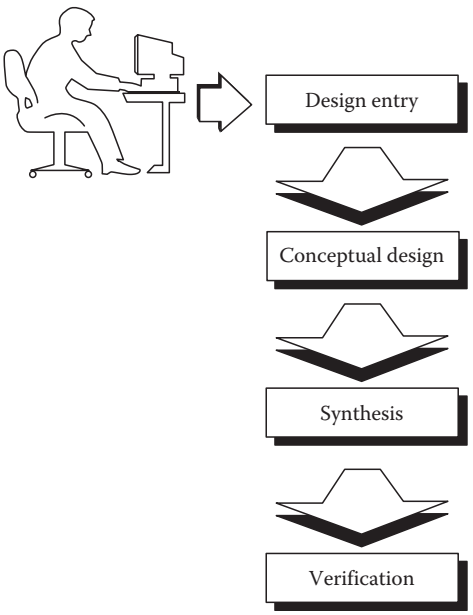


FIGURE 7.17 Design process.

Logic synthesis optimizes the logic generated by register transfer synthesis and maps the minimized logic operations onto physical gates supported by the target fabrication technology. Technology mapping considers the foundry cell library and associated electrical restrictions, such as fanin/fanout limitations.

7.5 Verification

Figure 7.17 shows that the verification task generally follows the synthesis task. The verification task checks the correctness of the function and performance of a design to ensure that an intermediate or final implementation faithfully realizes the initial, desired specification.

Several common types of verification are

- Timing analysis
- Simulation
- Emulation
- Formal verification

These types of verification are examined in more detail in the following sections [42].

7.5.1 Timing Analysis

As the name implies, timing analysis checks that the overall design satisfies operating speed requirements and individual signals within design satisfy transition requirements. Common signal transition requirements, also called timing hazards, include rise and fall times, propagation delays, clock times, race conditions, glitch detection, and setup and hold times. Set and hold timing checks are illustrated in Figure 7.18. For synchronous sequential digital systems, memory devices (level-sensitive latches or edge-sensitive flip-flops) require that the data and control signals obey setup and hold timings to ensure correct operation, i.e., that the memory device correctly and reliably stores the desired data. The control signal is typically a clock signal and Figure 7.18 assumes that a control signal transition, rising or falling, triggers activation of the memory device. The data signal carrying the information to be stored in the memory device must be stable for a period equal to the setup time prior to the control signal transition to ensure the correct value is sensed by the memory device. Also, the data signal must be stable for a period equal to the hold time after the control signal transition to ensure the memory device has enough time to store the sensed value.

Another class of timing transition requirements, commonly called ''signal integrity checks,'' include reflections, cross talk, ground bounce, and electromagnetic interference. Signal integrity checks are typically required for high-speed designs operating at clock frequencies above 75 MHz. At such high frequencies, the transmission line behavior of wires must be analyzed. A wire must be properly terminated, i.e., connected, to a port having an impedance matching the wire characteristic impedance to prevent signal reflections. Signal reflections are portions of an emanating signal that ''bound back'' from the destination to the source. Signal reflections reduce the power of the emanating signal and can damage the source. Cross talk refers to unwanted reactive coupling between physically adjacent signals, providing a connection between signals that are supposed to be electrically isolated. Cross talk causes information carried on a signal to interfere or corrupt information carried on a neighboring signal. Ground bounce is another signal integrity problem. Because all conductive material has finite impedance, a ground signal network does not in practice offer the same electrical potential throughout an entire design. These potential differences are usually negligible because the distributive impedance of the ground signal network is small compared to other finite component impedances. However, when many signals switch value simultaneously, a substantial current can flow through the ground signal network. High intermittent currents yield proportionately high intermittent potential drops, i.e., ground bounces, which can cause unwanted circuit behavior. Finally, electromagnetic interference refers to signal harmonics radiating from design components and interconnects. This harmonic radiation may interfere with other electronic equipment or may exceed applicable environmental safety regulatory limits [38].

Timing analysis can be done dynamically or statically. Dynamic timing analysis exercises the design via simulation or emulation for a period of time with a set of input stimuli and records the timing

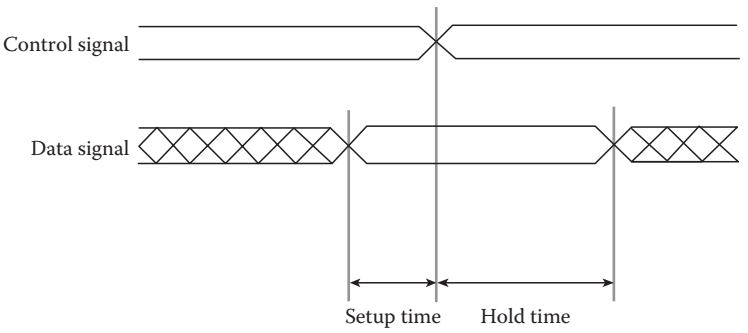


FIGURE 7.18 Signal transition requirements.

behavior. Dynamic timing analysis generally requires simulating many input test vectors to extensively exercise the design to ensure that signal paths are sufficiently identified and characterized. Static timing analysis does not exercise the design via simulation or emulation. Rather, static analysis records timing behavior based on the timing behavior (e.g., propagation delay) of the design components and their interconnection. With static timing analysis, the complexity of the verification task is partitioned into separate functional and performance checks.

Static timing analysis techniques are primarily *block oriented* or *path oriented*. Block-oriented timing analysis generates design input, also called primary input, to design output, also called primary output, propagation delays by analyzing the design, ''stage-by-stage'' and summing up the individual stage delays. All devices driven by primary inputs constitute stage 1, all devices driven by the outputs of stage 1 constitute stage 2, and so on. Starting with the first stage, all devices associated with a stage are annotated with worst-case delays. A worst-case delay is the propagation delay of the device plus the delay of the last input to arrive at the device, i.e., the signal path with the longest delay leading up to the device inputs. For example, the device labeled ''H'' in stage 3 in Figure 7.19 is annotated with the worst-case delay of 13, representing the device propagation delay of 4 and the delay of the last input to arrive through devices ''B'' and ''C'' of 9 [39]. When the devices associated with the last stage, i.e., the devices driving the primary outputs, are processed, the accumulated worst-case delays record the longest delay from primary inputs to primary outputs, also called the critical paths. The critical path for each primary output is highlighted in Figure 7.19.

Path-oriented timing analysis generates primary input to primary output propagation delays by traversing all possible signal paths one at a time. Thus, finding the critical path via path-oriented timing analysis is equivalent to finding the longest path through a directed acyclic graph, where devices are graph vertices and interconnections are graph edges [41].

A limitation of static timing analysis concerns detecting *false violations* or *false paths*. False violations are signal timing conditions that may occur due to the static structure of the design, but do not occur due to the dynamic nature of the design's response to actual input stimuli.

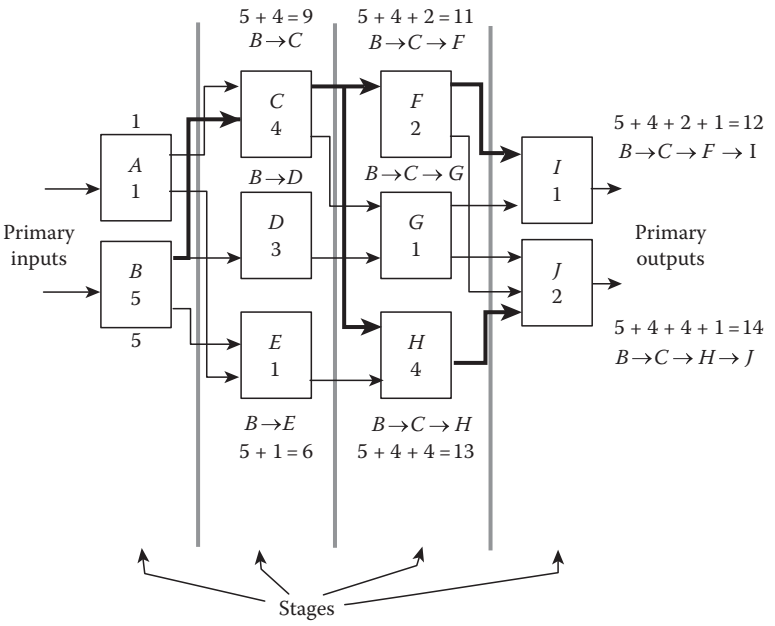


FIGURE 7.19 Block-oriented static timing analysis.

To account for realistic variances in component timing due to manufacturing tolerances, aging, or environmental effects, timing analysis often provides stochastic or statistical checking capabilities. Statistical timing analysis uses random number generators based on empirically observed probabilistic distributions to determine component timing behavior. Thus, statistical timing analysis describes design performance and the likelihood of the design performance.

7.5.2 Simulation

Simulation exercises a design over a period of time by applying a series of input stimuli and generating the associated output responses. The general event-driven, or schedule-driven, stimulation algorithm is diagrammed in Figure 7.20. An event is a change in signal value. Simulation starts by initializing the design; initial values are assigned to all signals. Initial values include starting values and pending values which constitute future events. Simulation time is advanced to the next pending event(s), signals are updated and sensitized models are evaluated. Sensitized models refer to models having outputs dependent on the updated signals [35,40]. The process of evaluating the sensitized model yields new, potentially different values for signals, i.e., a new set of pending events. These new events are added to the list of pending events, time is advanced to the next pending event(s), and the simulation algorithm repeats. Each pass through the loop of evaluating sensitized models at a particular time step is called a simulation cycle (see Figure 7.20). Simulation ends when the design yields no further activity, i.e., no more pending events exist to process.

Logic simulation is a computational intensive task for large, complex designs. Prior to committing to manufacturing, processor designers often simulate bringing up or ''booting'' the operating system. Such simulation tasks require sizable simulation computational resources. As an example, consider simulating 1s of a 200K gate, 20 MHz processor design. Assuming that on average only 10% of the total 200K gates are active or sensitized on each processor clock cycle, Equation 7.6 shows that simulating 1 s of actual processor time equates to 400 billion events.

$$\begin{aligned} 400\text{ B events} &= (20\text{ M clock cycles})(200\text{K gates})(10\%\text{ activity}) \\ 140\text{ h} &= (400\text{ B events})\left(\frac{50\text{ instructions}}{\text{event}}\right)\left(\frac{50\text{ M instruction}}{\text{s}}\right) \end{aligned} \tag{7.6}$$

Assuming that on average a simulation program executes 50 computer instructions per event on a computer capable of processing 50 million instructions per second, Equation 7.6 also shows that

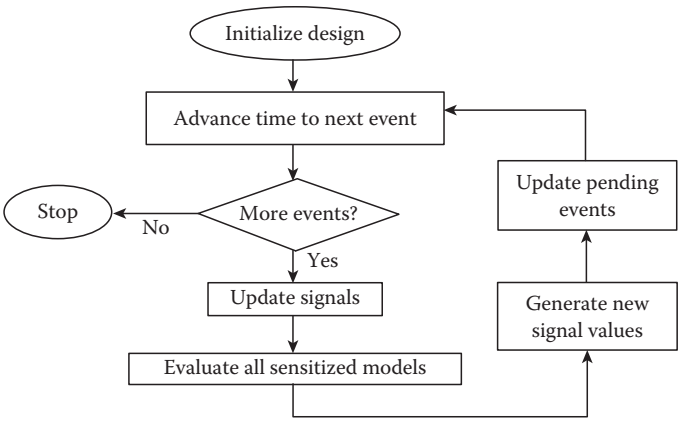


FIGURE 7.20 General event-driven simulation algorithm.

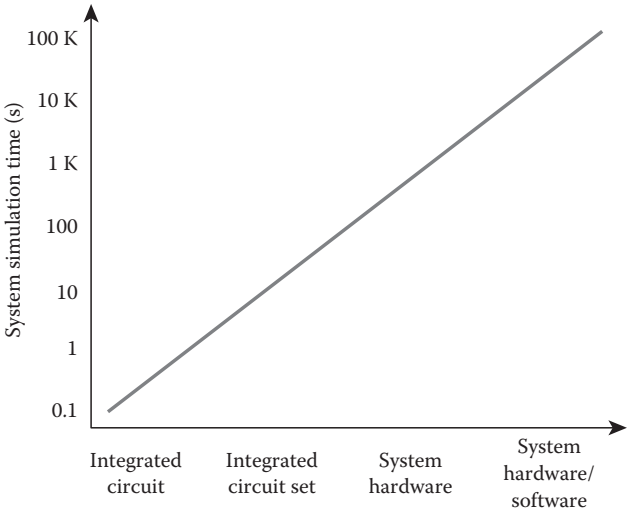


FIGURE 7.21 Simulation requirements.

processing 400 billion events requires 140 h or just short of 6 days. This simple example demonstrates the sizable computational properties of simulation. Figure 7.21 shows how simulation computation scales with design complexity.

To address the growing computational demands of simulation, several simulation acceleration techniques have been introduced. Schedule or event-driven simulation (explained previously) can be accelerated by removing layers of interpretation and running simulation as a native executable image; such an approach is called complied, scheduled-driven simulation. Schedule-driven simulation can be accelerated also by using more efficient event management schemes. In a conventional, central event management scheme all events are logged into a time-ordered list. As simulation time advances, pending events become actual events and the corresponding sensitized devices are executed to compute the response events. One the other hand, in a dataflow event management scheme events are ''self-empowered,'' active agents that flow through networks and trigger device evaluations without registering with a central time-order list and dispatcher [37].

Instead of evaluating a device in a stimulus–response manner, *cycle-driven* simulation avoids the overhead of event queue processing by evaluating all devices at regular intervals of time. Cycle-driven simulation is efficient when a design exhibits a high degree of concurrency, i.e., a large percentage of the devices are active per simulation cycle. Based on the staging of devices, the devices are *ranked-ordered* to determine the order in which they are evaluated at each time step to ensure the correct causal behavior yielding the proper ordering of events. For functional verification, logic devices are often assigned zero-delay and memory devices are assigned unit-delay. Thus, any number of stages of logic devices may execute between system clock periods.

Another simulation acceleration technique is *message-driven* simulation, also called parallel or distributed simulation. Device execution is divided among several processors, and the device simulations communicate event activity via messages. Messages are communicated using a conservative or an optimistic strategy. Optimistic message passing strategies, such as *time warp* and *lazy cancellation*, make assumptions about future event activity to advance local device simulation. If the assumptions are correct, the processors operate more independently and better exploit parallel computation. However, if the assumptions are incorrect, then local device simulations may be forced to ''roll back'' to synchronize local device simulations [34,36].

Schedule-driven, cycle-driven, and message-driven simulation are software-based simulation acceleration techniques. Simulation also can be accelerated by relegating certain simulation activities to dedicated hardware. For example, *hardware modelers* can be attached to simulators to accelerate the activity of device evaluation. As the name implies, hardware modeling uses actual hardware devices instead of software models to obtain stimulus–response information. In a typical scenario, the hardware modeler receives input stimuli from the software simulator. The hardware modeler then exercises the device and sends the output response back to the software simulator. Using actual hardware devices reduces the expense of generating and maintaining software models and provides an environment to support application software development. However, the hardware device must exist, which means hardware modeling has limited use in the initial stages of design in which hardware implementations are not available. Also, it is sometimes difficult for a slave hardware modeler to preserve accurate real-time device operating response characteristics within a master non-real-time software simulation environment. For example, some hardware devices may not be able to retain state information between invocations, so the hardware modeler must save the history of previous inputs and reapply them to bring the hardware device to the correct state in order to apply a new input.

Another technique for addressing the growing computational demands of simulation is via *simulation engines*. A simulation engine can be viewed as an extension of the simulation acceleration techniques of hardware modeling. With a hardware modeler, the simulation algorithm executes in software, and component evaluation executes in dedicated hardware. With a simulation engine, the simulation algorithm and component evaluation execute in dedicated hardware. Simulation engines are typically two to three orders of magnitude faster than software simulation [43].

7.5.3 Emulation

Emulation, also called ''computer-aided prototyping,'' verifies a design by realizing the design in ''preproduction'' hardware and exercising the hardware. The term preproduction hardware means nonoptimized hardware providing the correct functional behavior, but not necessarily the correct performance. That is, emulation hardware may be slower, require more area, or dissipate more power than production hardware. Presently, preproduction hardware commonly involves some form of *programmable logic devices*, typically *field-programmable gate arrays*. Programmable logic devices provide generic combinational and sequential digital system logic that can be programmed to realize a wide variety of designs [44].

Emulation begins by partitioning the design; each design segment is realized by a programmable logic device. The design segments are then interconnected to realize a preproduction implementation and exercised with a series of input test vectors.

Emulation offers the advantage of providing prototype hardware early in the design cycle to check for errors or inconsistencies in initial functional specifications. Problems can be isolated and design modifications can be accommodated easily by reprogramming the logic devices. Emulation can support functional verification at a computational rate much greater than conventional simulation. However, emulation does not generally support performance verification because, as explained previously, prototype hardware typically does not operate at production clock rates.

7.6 Test

Figure 7.22 shows that the test task generally follows the verification task. Although the verification and test tasks both seek to check for correct function and performance, verification focuses on a model of the design before manufacturing, whereas test focuses on the actual hardware after manufacturing. Thus, the primary objective of test is to detect a faulty device by applying input test stimuli and observing expected results [47,53].

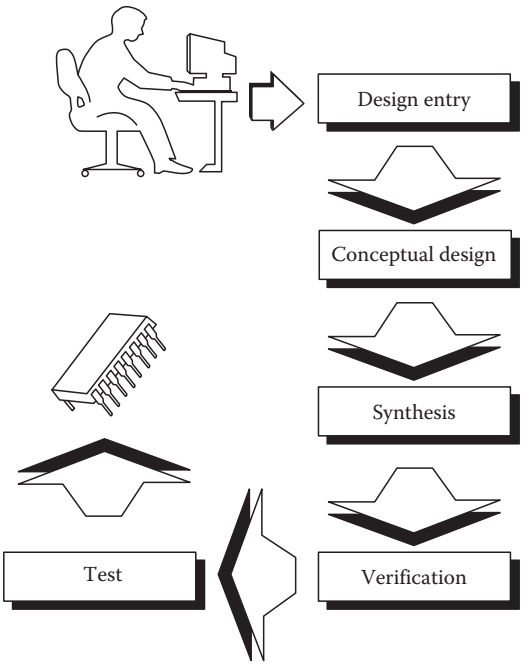


FIGURE 7.22 Design process.

The test task is difficult because designs are growing in complexity; more components provide more opportunity for manufacturing defects. Test is also challenged by new microelectronic fabrication processes having new failure modes which again provides more opportunity for manufacturing defects. New microelectronic fabrication processes also offer higher levels of integration with fewer access points to probe internal electrical nodes. To illustrate the growing demands of test, Table 7.2 shows the general proportional relationship between manufacturing defects and required fault coverage, i.e., quality of testing. Figure 7.23 shows the escalating costs of testing equipment.

Testing involves three general testing techniques or strategies: functional, parametric, and fault. Functional testing checks that the hardware device realizes the correct I–O digital system behavior. Parametric testing checks that the hardware device realizes the correct performance specifications, such as speed or power dissipation, and electrical specifications, such as voltage polarities and current sinking/sourcing limitation. Finally, fault testing checks for manufacturing defects or ''faults.''

TABLE 7.2 Fault Coverage and Defect Rate

	Defect Rate		
Microelectronic Fabrication Process	70% Fault Coverage	90% Fault Coverage	99% Fault Coverage
2 μm			
90% yield	3%	1%	0.1%
1.5 μm			
50% yield	19%	7%	0.7%
1 μm			
10% yield	50%	21%	2%

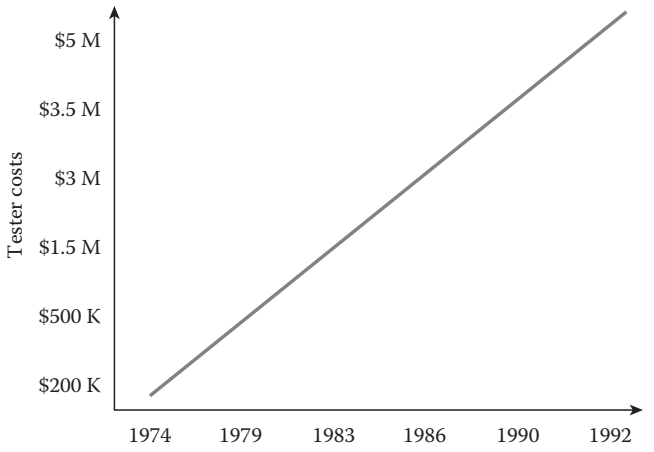


FIGURE 7.23 Integrated circuit tester equipment cost.

7.6.1 Fault Modeling

A fault is a manufacturing or aging defect that causes a device to operative incorrectly or to fail. A sample listing of common integrated circuit physical faults is given next:

- Wiring faults
- Dielectric faults
- Threshold faults
- Soft faults

Wiring faults are unwanted opens and shorts. Two wires or networks that should be electrically connected but are not constitute an open. Two wires or networks that should not be electrically connected but are constitute a short. Wiring faults can be caused by manufacturing defects such as metallization or etching problems, or aging defects, such as corrosion or electromigration. Dielectric faults are electrical isolation defects that can be caused by masking defects, material impurities or imperfections, and electrostatic discharge. Threshold faults occur when the turn-on and turn-off voltage potentials of electrical devices exceed allowed ranges. The faulty devices cannot be properly operated, which results in component failure. Soft faults occur when radiation exposure temporarily changes electrical charge distributions. Such changes can alter circuit voltage potentials, which can, in turn, change logical values, also called ''dropping bits.'' Radiation effects are called ''soft'' faults because the hardware is not permanently damaged [54].

To simplify the task of fault testing, the physical faults described above are translated into logical faults. Typically, a single logical fault covers several physical faults. A popular logical fault model is the *single stuck line* fault model. The single stuck line fault model considers faults where any single signal line or wire is permanently set to a logic 0, ''stuck-at-0,'' or a logic 1, ''stuck-at-1.'' These signal or interconnect faults are assumed to be time invariant.

Building on the single stuck line fault model, the *multiple stuck line* fault model considers multiple signal wire stuck-at-0/stuck-at-1 faults. The multiple stuck line fault model is more expressive and can cover more physical faults than the single stuck line model. However, fault testing for the multiple stuck line fault model is more difficult because of the exponential growth in the possible combinations of multiple signal faults.

Stuck fault models do not address all physical faults because not all physical faults result in signal lines permanently set to low or high voltages, i.e., stuck-at-0 or stuck-at-1 logic faults. Thus, other fault models

have been developed to address specific failure mechanisms. For example, the *bridging* fault model addresses electrical shorts that cause unwanted coupling or spurious feedback loops. As another example, the *pattern-sensitive* fault model addresses wiring and dielectric faults that yield unwanted interference between physically adjacent signals. Pattern-sensitive faults are generally most prevalent in high-density memories incorporating low signal:noise ratios and can be difficult to detect because they are often data-pattern and data-rate dependent. In other words, the part fails only under certain combinations of input stimuli and only under certain operating conditions.

7.6.2 Fault Testing

Having identified and categorized the physical faults that may cause device malfunction or failure and determined how the physical faults relate to logical faults, the next task is to develop tests to detect these faults. When the tests are generated by a computer program, this activity is called "automatic test program generation." Examples of fault testing techniques are listed next:

- Stuck-at techniques
- Scan techniques
- Signature techniques
- Ad hoc techniques
- Coding techniques
- Electrical monitoring techniques

The following paragraphs review these testing strategies.

Basic stuck-at techniques generate input stimuli for fault testing combinational digital systems. Three of the most popular stuck-at fault testing techniques are the D algorithm, the path-oriented decision-making (Podem) algorithm, and the fan algorithm. These algorithms first identify a circuit fault (e.g., stuck-at-0 or stuck-at-1) and then try to generate an input stimulus that detects the fault and makes the fault visible as an output. Detecting a fault is often called "fault sensitization" and making a fault visible is often called "fault propagation." To illustrate this process, consider the simple combinational design in Figure 7.24 [46,49,50]. The design is defective because a manufacturing defect has caused the output of the `and` gate to be permanently tied to ground, i.e., stuck-at-0, using a positive logic convention. To sensitize the fault, the inputs A and B should both be set to 1, which should force the `and` gate output to a 1 for a good circuit. To propagate the fault, the inputs C and D should both be set to 0, which will force the `xor` gate output to 1, again for a good circuit. Thus, if $A = 1$, $B = 1$, $C = 0$, and $D = 0$ in Figure 7.24, then a good circuit would yield a 1, but the defective circuit yields a 0 which detects the stuck-at-0 fault at the `and` gate output.

Sequential automatic test program generation is a more difficult task than combinational automatic test program generation because exercising or sensitizing a particular circuit path to detect the presence of a possible manufacturing fault may require a sequence of input test vectors. One technique for testing sequential digital systems is called scan fault testing. Scan fault testing is called a "design-for-testability" technique because it modifies or constrains the design in a manner that facilitates fault testing. Scan techniques impose a logic design discipline that all state registers be connected in one or more chains to form "scan rings," as shown in Figure 7.25 [48]. During normal device operation, the scan rings are disabled and the registers serve as conventional memory (state) storage elements. During test operation, the scan rings are enabled and

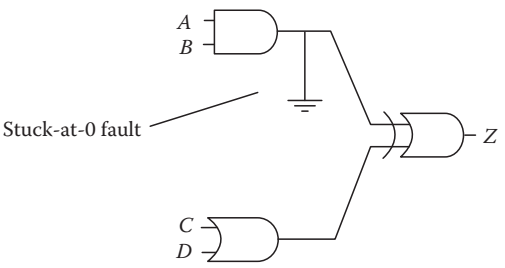


FIGURE 7.24 Combinational logic stuck-at fault testing.

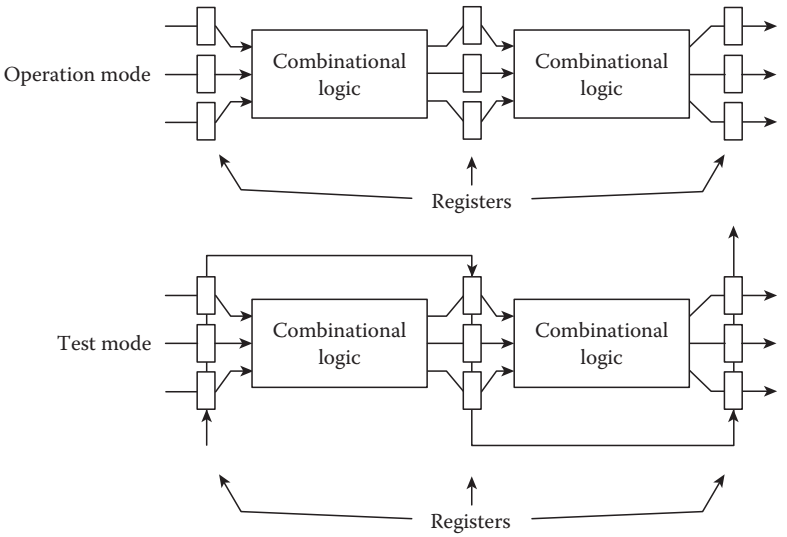


FIGURE 7.25 Scan-based design-for-testability.

stimulus test vectors are shifted into the memory elements to set the state of the digital system. The digital system is exercised for one clock cycle and then the results are shifted out of the scan ring to record the response.

The principal advantage of scan design-for-testability is that the scan ring decouples stages of combinational logic, thereby transforming a sequential design into effectively a combinational design and correspondingly a difficult sequential test task into a simpler combinational test task. However, fault tests must still be generated for the combinational logic, and shifting in stimuli and shifting out response requires time. Also, scan paths require additional hardware resources that can impose a performance (speed) penalty. To address these limitations, ''partial'' scan techniques have been developed that offer a compromise between testability and performance. Instead of connecting every register into scan chains, Figure 7.26 shows that a partial scan selectively connects a subset of registers into scan chains. Registers in critical performance circuit paths are typically excluded and registers providing control and observability to portions of the design are typically included. Similar to full scan test operation, stimulus test vectors are shifted into the memory elements to set the state of the digital system. Then, a partial scan test exercises the digital system for multiple clock cycles (two clock cycles for the partial scan shown in Figure 7.26), and then the results are shifted out of the scan ring to record the response.

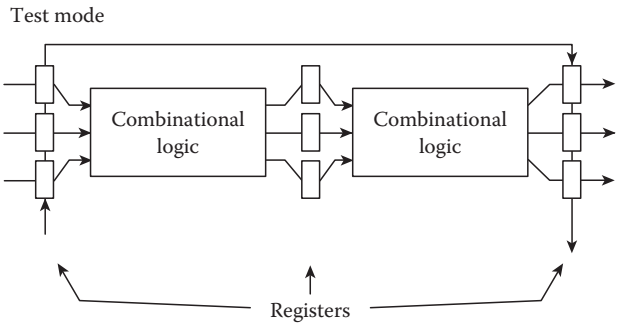


FIGURE 7.26 Partial scan-based design-for-testability.

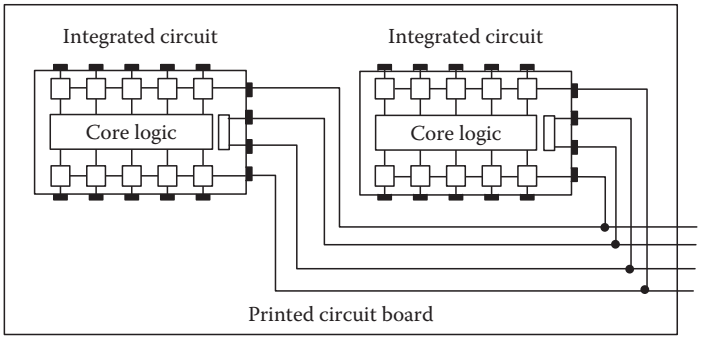


FIGURE 7.27 Boundary scan.

A variation of partial scan design-for-testability, called ''boundary scan,'' has been defined for testing integrated circuits on printed circuit boards. Printed circuit board manufacturing developments, such as fine-lead components, surface mount assembly, and multichip modules, yield high-density boards with fewer access points to probe individual pins. Such printed circuit boards are difficult to test. As the name implies, boundary scan imposes a design discipline on printed circuit board components, typically integrated circuits, such that the I–O pins on the integrated circuits can be connected into scan chains. Figure 7.27 shows that each integrated circuit configured for boundary scan contains scan registers between the I–O pins and the core logic to enable the printed circuit board test bus to control and observe the behavior of individual integrated circuits [51].

Another design-for-testability technique is signature analysis, also called ''built-in-self-test.'' Signature testing techniques use additional logic, typically linear feedback shift registers, to automatically generate pseudorandom test vectors. The output responses are compressed into a single vector and compared to a known good vector. If the output response vector does not exactly match the known good vector, then the design is considered faulty. Matching the output response vector and a known good vector does not guarantee correct hardware; however, if enough pseudorandom test vectors are exercised, then the chances are acceptably small of obtaining a false positive result. Signature analysis is often used to test memories [45].

Ad hoc testing techniques selectively insert test hardware and access points into a design to improve observability and controllability. Typical candidates for additional access points include storage elements (set and reset controls), major system communication buses, and feedback loops. Ad hoc testing techniques, also called behavioral testing techniques, can avoid the performance penalty of more structured, logical fault testing techniques, such as scan testing, and more closely mimics the actions of an expert test engineer. However, the number of access points is often restricted by integrated circuit or printed circuit board I–O pin limitations.

Coding test techniques encode signal information so that errors can be detected and possibly corrected. Although often implemented in software, coding techniques can also be implemented in hardware. For example, a simple coding technique called ''parity checking'' is often implemented in hardware. Parity checking adds an extra bit to multibit data. The parity bit is set such that the total number of logic 1's in the multibit data and parity bit is either an even number (even parity) or an odd number (odd parity). An error has occurred if an even parity encoded signal contains an odd number of logic 1's or an odd parity encoded signal contains an even number of logic 1's. Coding techniques are used extensively to detect and correct transmission errors on system buses and networks, storage errors in system memory, and computational errors in processors [52].

Finally, the electrical monitoring testing technique, also called current/voltage testing, relies on the simple observation that an out-of-range current or voltage often indicates a defective or bad part. Possibly a short or open is present, causing a particular I–O signal to have the wrong voltage or current.

Current testing (I_{ddq} testing) is particularly useful for digital systems using CMOS integrated circuit technology. Normally, CMOS circuits yield very low static or quiescent currents. However, physical faults, such as gate oxide defects, can increase static current by several orders of magnitude. Such a substantial change in static current is straightforward to detect. The principal advantages of current testing are that the tests are simple and the fault models address detailed transistor-level defects. However, current testing requires that enough time be allotted between input stimuli to allow the circuit to reach a static state, which slows down testing and causes problems with circuits that cannot be tested at scaled clock rates.

7.7 Frameworks

The previous sections discussed various types of design automation programs ranging from initial design entry tasks to final manufacturing test tasks. As the number and sophistication of design automation programs increases, the systems aspects of how the tools should be integrated with each other and underlying computing platforms become increasingly important. The first commercial design automation system product offerings in the 1970s consisted primarily of design automation programs and an associated computing platform ''bundled'' together in turnkey systems. These initial product offerings offered little flexibility to ''mix-and-match'' design automation programs and computing products from different source to take advantage of state-of-the-art capabilities and construct a design automation system tailored to particular end-user requirements. In response to these limitations, vendor offerings in the 1980s started to address *open systems*. Vendors ''unbundled'' software and hardware and introduced interoperability mechanisms to enable a design automation program to execute on different vendor systems. The interoperability mechanisms are collectively called ''frameworks'' [56].

Frameworks support design automation systems potentially involving many users, application programs, and host computing environments or platforms. Frameworks manage the complexities of a complete design methodology by coordinating and conducting the logistics of design automation programs and design data. Supporting the entire design cycle, also called ''concurrent engineering,'' improves productivity by globally optimizing the utilization of resources and the minimization of design errors and associated costs.

A general definition of a framework is given next:

> A CAD framework is a software infrastructure that provides a common operating environment for CAD tools. A framework should enable users to launch and manage tools; create, organize, and manage data; graphically view the entire design process; and perform design management tasks such as configuration and version management. Among the key elements of a CAD framework are platform-independent graphics and user interfaces, inter-tool communications, and design data and process management services. (CAD Framework Initiative, CFI)*

Figure 7.28 illustrates that a framework is essentially a domain-specific (i.e., electronic systems) layer of operating system software that facilitates ''plug-compatible'' design automation programs. Framework services are specific to electronic systems design and are mapped into the more generic services provided by general-purpose computing platforms [55].

User interface services provide a common and consistent ''look-and-feel'' to application programs. User interface services include support for consistent information display styles using menus and/or windows. These services also include support for consistent command styles using programming function keys and/or mouse buttons. Application program services provide support for program interactive/batch invocation and normal/abnormal termination. Application services also provide support for

* CAD Framework Initiative is a consortium of companies established to define and promote industry standards for design automation system interoperability.

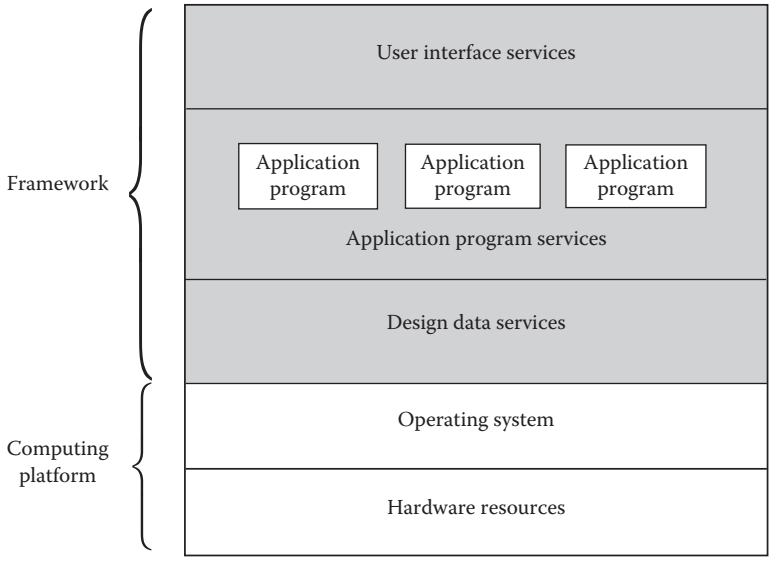


FIGURE 7.28 Framework architecture.

program-to-program communication and process management. Process management implements a design methodology that defines application-to-application and application-to-data dependencies. Process management describes a general sequencing of design automation programs and provisions for iterations. In other words, process management ensures a designer is working with the right tool at the right time with the right data.

Finally, a framework provides design data services to support access, storage, configuration, and integrity operations. Figure 7.29 shows relationships among relational, network, and object-oriented data management schemes. Due to the comparatively large size of design data and the length of design operations, framework data services are evolving toward object-oriented paradigms [57–59].

Object-oriented paradigms match data structures and operations with design automation objects and tasks, respectively. Figure 7.30 shows that hardware devices, components, and products become natural choices for software ''objects.''

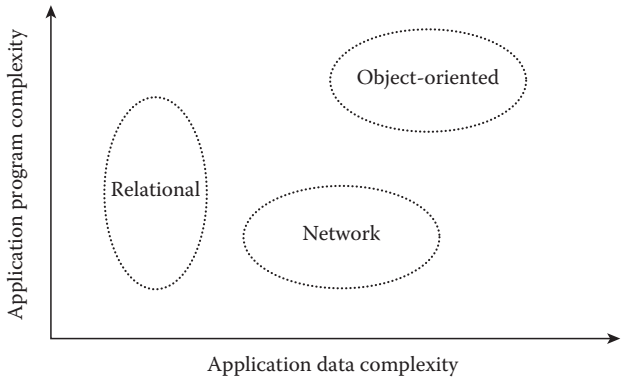


FIGURE 7.29 Data management technology.

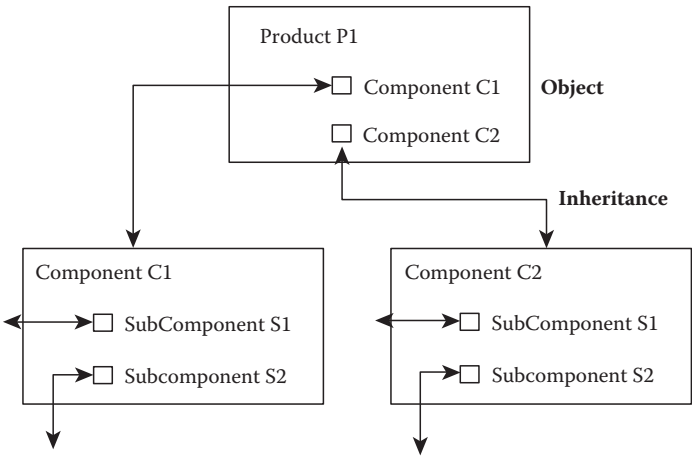


FIGURE 7.30 Object-oriented modeling.

The hierarchical relationships between hardware units become natural choices for software ''inheritance.'' Finally, design automation tasks and subtask that operate on hardware units, such as display or simulate, become natural choices for software ''methods.''

7.8 Summary

Design automation technology offers the potential of serving as a powerful fulcrum in leveraging the skills of a designer against the growing demands of electronic system design and manufacturing. Design automation programs help to relieve the designer of the burden of tedious, repetitive tasks that can be labor intensive and error prone.

Design automation technology can be broken down into several topical areas, such as design entry, conceptual design, synthesis, verification, testing, and frameworks. Each topical area has developed an extensive body of knowledge and experience.

Design entry defines a desire specification. Conceptual design refines the specification into a design plan. Synthesis refines the design plan into an implementation. Verification checks that the implementation faithfully realizes the desired specification. Testing checks that the manufactured part performs functionally and parametrically correctly. Finally, frameworks enable individual design automation programs to operate collectively and cohesively within a larger computing system environment.

References

1. D. Barbe, Ed. *Very Large Scale Integration* (*VLSI*)—*Fundamentals and Applications*, New York: Springer-Verlag, 1980.
2. T. Dillinger, *VLSI Engineering*, Englewood Cliffs, NJ: Prentice-Hall, 1988.
3. E. Hollis, *Design of VLSI Gate Array Integrated Circuits*, Englewood Cliffs, NJ: Prentice Hall, 1987.
4. S. Sapiro, *Handbook of Design Automation*, Englewood Cliffs, NJ: Prentice Hall, 1986.
5. S. Trimberger, *An Introduction to CAD for VLSI*, San Jose, CA: Domancloud Publishers, 1990.
6. G. Birtwistle and P. Subrahmanyan, *VLSI Specification, Verification, and Synthesis*, Boston, MA: Kluwer Academic, 1988.
7. A. Dewey, VHSIC hardware description language development program, *Proc. Design Autom. Conf.*, June 1983.
8. A. Dewey, VHDL: Towards a unified view of design, *IEEE Design Test Comput.*, June 1992.

9. J. Douglas-Young, *Complete Guide to Reading Schematic Diagrams*, Englewood Cliffs, NJ: Prentice Hall, 1988.
10. C. Liu, *Elements of Discrete Mathematics*, New York: McGraw-Hill, 1985.
11. M. Pechet, Ed., *Handbook of Electrical Package Design*, New York: Marcel Dekker, 1991.
12. J. Peterson, *Petri Net Theory and Modeling of Systems*, Englewood Cliffs, NJ: Prentice Hall, 1981.
13. B. Spinks, *Introduction to Integrated Circuit Layout*, Englewood Cliffs, NJ: Prentice Hall, 1985.
14. N. Bose, *Digital Filters: Theory and Applications*, New York: North-Holland, 1985.
15. X. Chen and M. Bushnell, A module area estimator for VLSI layout, *Proc. Design Autom. Conf.*, June 1988.
16. A. Dewey, *Principles of VLSI Systems Planning: A Framework for Conceptual Design*, Boston: Kluwer Academic, 1990.
17. W. Donath, Wiring space estimation for rectangular gate arrays, *Proc. Int. Conf. VLSI*, 1981.
18. W. Heller, Wirability—Designing wiring space for chips and chip packages, *IEEE Design Test Comput.*, Aug. 1984.
19. C. Leiserson, Area-efficient VLSI computation, PhD dissertation, Pittsburgh, Carnegie Mellon University, 1981.
20. A. Oppenheim and R. Schafer, *Digital Signal Processing*, Englewood Cliffs, NJ: Prentice Hall, 1975.
21. L. Rabiner, O. Hermann, and D. Chan, Practical design rules for optimum finite impulse response low-pass digital filters, *Bell Syst. Tech. J.*, July–Aug. 1973.
22. E. Sacerdoti, Planning in a hierarchy of abstraction spaces, *Artif. Intell.*, Sept. 1974.
23. M. Stefik, Planning with constraints, *Artif. Intell.*, Feb. 1980.
24. C. Thompson, Area-time complexity for VLSI, *Proc. Caltech Conf. VLSI*, Jan. 1979.
25. L. Valiant, Universality considerations in VLSI circuits, *IEEE Trans. Comput.*, Feb. 1981.
26. R. Ayres, *VLSI: Silicon Compilation and the Art of Automatic Microchip Design*, Englewood Cliffs, NJ: Prentice Hall, 1983.
27. R. Brayton et al., *Logic Minimization Algorithms for VLSI Synthesis*, Boston, MA: Kluwer Academic, 1992.
28. R. Camposano and W. Wolfe, *High-Level VLSI Synthesis*, Boston, MA: Kluwer Academic, 1991.
29. D. Gajski, Ed., *Silicon Compilation*, Reading, MA: Addison-Wesley, 1988.
30. D. Gajski et al., *High-Level Synthesis—Introduction to Chip and System Design*, Boston, MA: Kluwer Academic, 1992.
31. P. Paulin and J. Knight, Force-directed scheduling for the behavioral synthesis of ASIC's, *IEEE Design Test Comput.*, Oct. 1989.
32. B. Preas, M. Lorenzetti, and B. Ackland, Eds., *Physical Design Automation of VLSI Systems*, New York: Benjamin Cummings, 1988.
33. T. Sasao, Ed., *Logic Synthesis and Optimization*, Boston, MA: Kluwer Academic, 1993.
34. R. Bryant, Simulation on distributed systems, *Proc. Int. Conf. Distributed Systems*, 1979.
35. J. Butler, Ed., *Multiple-Valued Logic in VLSI Design*, New York: IEEE Computer Society Press, 1991.
36. K. Chandy and J. Misra, Asynchronous distributed simulation via a sequence of parallel computations, *Commun. ACM*, April 1981.
37. W. Hahn and K. Fischer, High performance computing for digital design simulation, *VLSI*85, Amsterdam: Elsevier, 1985.
38. R. McHaney, *Computer Simulation: A Practical Perspective*, New York: Academic Press, 1991.
39. T. McWilliams and L. Widdoes, SCALD—Structured computer aided logic design, *Proc. Design Autom. Conf.*, June 1978.
40. U. Pooch, *Discrete Event Simulation: A Practical Approach*, Boca Raton, FL: CRC Press, 1993.
41. T. Sasiki, et al., Hierarchical design and verification for large digital systems, *Proc. Design Autom. Conf.*, June 1978.
42. J. Sifakis, Ed., *Automatic Verification Methods for Finite State Systems*, New York: Springer-Verlag, 1990.

43. S. Takasaki, F. Hirose, and A. Yamada, Logic simulation engines in Japan, *IEEE Design Test Comput.*, Oct. 1989.
44. S. Walters, Computer-aided prototyping for ASIC-based synthesis, *IEEE Design Test Comput.*, June 1991.
45. V. Agrawal, C. Kime, and S. Saluja, A tutorial on built-in self test, *IEEE Design Test Comput.*, June 1993.
46. M. Breuer and A. Friedman, *Diagnosis and Reliable Design of Digital Systems*, New York: Computer Science Press, 1976.
47. A. Buckroyd, *Computer Integrated Testing*, New York: John Wiley & Sons, 1989.
48. E. Eichelberger and T. Williams, A logic design structure for LSI testability, *Proc. Design Autom. Conf.*, June 1977.
49. H. Fujiwara and T. Shimono, On the acceleration of test generation algorithms, *IEEE Trans. Comput.*, Dec. 1983.
50. P. Goel, An implicit enumeration algorithm to generalize tests for combinational logic circuits, *IEEE Trans. Comput.*, March 1981.
51. K. Parker, The impact of boundary scan on board test, *IEEE Design Test Comput.*, Aug. 1989.
52. W. Peterson and E. Weldon, *Error-Correcting Codes*, Cambridge, MA: MIT Press, 1972.
53. M. Weyerer and G. Goldemund, *Testability of Electronic Circuits*, Englewood Cliffs, NJ: Prentice Hall, 1992.
54. G. Zobrist, *VLSI Fault Modeling and Testing Technologies*, New York: Ablex Publishing, 1993.
55. T. Barnes, *Electronic CAD Frameworks*, Boston, MA: Kluwer Academic, 1992.
56. D. Bedworth, M. Henderson, and P. Wolfe, *Computer-Integrated Design and Manufacturing*, New York: McGraw-Hill, 1991.
57. R. Gupta and E. Horowitz, Eds., *Object-Oriented Databases with Applications to CASE Networks, and VLSI CAD*, Englewood Cliffs, NJ: Prentice Hall, 1990.
58. W. Kim, *Object-Oriented Concepts, Databases, and Applications*, Reading, MA: Addison-Wesley, 1989.
59. E. Nahouranii and F. Petry, Eds., *Object-Oriented Databases*, Los Alamitos, CA: IEEE Computer Society Press, 1991.

8

Computer-Aided Analysis

8.1 Circuit Simulation Using SPICE and SUPREM 8-1
Introduction • DC (Steady-State) Analysis • AC Analysis • Transient Analysis • Process and Device Simulation • Process Simulation • Device Simulation
Appendix A 8-20
Circuit Analysis Software • Process and Device Simulators
References 8-20
8.2 Parameter Extraction for Analog Circuit Simulation 8-21
Introduction • MOS DC Models • BSIM Extraction Strategy in Detail • Bipolar DC Model • MOS and Bipolar Capacitance Models • Bipolar High-Frequency Model • Miscellaneous Topics
References 8-45

J. Gregory Rollins
Technology Modeling Associates, Inc.

Peter Bendix
Technology Modeling Associates, Inc.

8.1 Circuit Simulation Using SPICE and SUPREM

8.1.1 Introduction

Computer-aided simulation is a powerful aid during the design or analysis of electronic circuits and semiconductor devices. The first part of this chapter focuses on analog circuit simulation. The second part covers simulations of semiconductor processing and devices. While the main emphasis is on analog circuits, the same simulation techniques may, of course, be applied to digital circuits (which are, after all, composed of analog circuits). The main limitation will be the size of these circuits because the techniques presented here provide a very detailed analysis of the circuit in question and, therefore, would be too costly in terms of computer resources to analyze a large digital system.

The most widely known and used circuit simulation program is SPICE (simulation program with integrated circuit emphasis). This program was first written at the University of California at Berkeley by Laurence Nagel in 1975. Research in the area of circuit simulation is California at many universities and industrial sites. Commercial versions of SPICE or related programs are available on a wide variety of computing platforms, from small personal computers to large mainframes. A list of some commercial simulator vendors can be found in the Appendix A.

It is possible to simulate virtually any type of circuit using a program like SPICE. The programs have built-in elements for resistors, capacitors, inductors, dependent and independent voltage and current sources, diodes, MOSFETs, JFETs, bipolar junction transistors (BJTs), transmission lines, transformers, and even transformers with saturating cores in some versions. Found in commercial versions are libraries of standard components which have all necessary parameters prefitted to typical specifications.

These libraries include items such as discrete transistors, op-amps, phase-locked loops, voltage regulators, logic integrated circuits (ICs) and saturating transformer cores.

Computer-aided circuit simulation is now considered an essential step in the design of ICs, because without simulation the number of ''trial runs'' necessary to produce a working IC would greatly increase the cost of the IC. Simulation provides other advantages, however:

- Ability to measure ''inaccessible'' voltages and currents. Because a mathematical model is used all voltages and currents are available. No loading problems are associated with placing a voltmeter or oscilloscope in the middle of the circuit, with measuring difficult one-shot wave forms, or probing a microscopic die.
- Mathematically ideal elements are available. Creating an ideal voltage or current source is trivial with a simulator, but impossible in the laboratory. In addition, all component values are exact and no parasitic elements exist.
- It is easy to change the values of components or the configuration of the circuit. Unsoldering leads or redesigning IC masks are unnecessary.

Unfortunately, computer-aided simulation has its own problems:

- Real circuits are distributed systems, not the ''lumped elements models'' which are assumed by simulators. Real circuits, therefore, have resistive, capacitive, and inductive parasitic elements present besides the intended components. In high-speed circuits these parasitic elements are often the dominant performance-limiting elements in the circuit, and must be painstakingly modeled.
- Suitable predefined numerical modes have not yet been developed for certain types of devices or electrical phenomena. The software user may be required, therefore, to create his or her own models which are available in the simulator. (An example is the solid-state thyristor which may be created from a NPN and PNP bipolar transistor.)
- Numerical methods used may place constraints on the form of the model equations used.

The following sections consider the three primary simulation modes: DC, AC, and transient analysis. In each section an overview is given of the numerical techniques used. Some examples are then given, followed by a brief discussion of common pitfalls.

8.1.2 DC (Steady-State) Analysis

DC analysis calculates the state of a circuit with fixed (nontime varying) inputs after an infinite period of time. DC analysis is useful to determine the operating point (Q-point) of a circuit, power consumption, regulation and output voltage of power supplies, transfer functions, noise margin and fan-out in logic gates, and many other types of analysis. In addition, DC analysis is used to find the starting point for AC and transient analysis. To perform the analysis the simulator performs the following steps:

1. All capacitors are removed from the circuit (replaced with opens).
2. All inductors are replaced with shorts.
3. Modified nodal analysis is used to construct the nonlinear circuit equations. This results in one equation for each circuit node plus one equation for each voltage source. Modified nodal analysis is used rather than standard nodal analysis because an ideal voltage source or inductance cannot be represented using normal nodal analysis. To represent the voltage sources, loop equations (one for each voltage source or inductor), are included as well as the standard node equations. The node voltages and voltage source currents, then, represent the quantities which are solved for. These form a vector $\mathbf{x}$. The circuit equations can also be represented as a vector $\mathbf{F}(\mathbf{x}) = \mathbf{0}$.
4. Because the equations are nonlinear, Newton's method (or a variant thereof) is used to solve the equations. Newton's method is given by the following equations:

$$J = \left.\frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right|_{\mathbf{x}^i}$$
$$\mathbf{x}^{i+1} = \mathbf{x}^i - J^{-1} \cdot \mathbf{F}(\mathbf{x}^i)$$

Here, if $\mathbf{x}^i$ is an estimate of the solution, $\mathbf{x}^{i+1}$ is a better estimate. The equations are used iteratively, and hopefully the vector $\mathbf{x}$ converges to the correct solution. The square matrix J of partial derivatives is called the Jacobian of the system. Most of the work in calculating the solution is involved in calculating J and its inverse J^{-1}. It may take as many a 100 iterations for the process to converge to a solution. Parameters control this process in most simulation programs (see the .OPTIONS statement in SPICE). For example, the maximum number of iterations allowed and error limits which must be satisfied before the process is considered to be converged, but normally the default limits are appropriate.

Example 8.1: Simulation Voltage Regulator

We shall now consider simulation of the type 723 voltage regulator IC, shown in Figure 8.1. We wish to simulate the IC and calculate the sensitivity of the output *IV* characteristic and verify that the output current follows a "fold-back" type characteristic under overload conditions.

The IC itself contains a voltage reference source and operational amplifier. Simple models for these elements are used here rather than representing them in their full form, using transistors to illustrate model development. The use of simplified models can also greatly reduce the simulation effort. (For example, the simple op-amp used here requires only eight nodes and ten components, yet realizes many advanced features.)

Note in Figure 8.1 that the numbers next to the wires represent the circuit nodes. These numbers are used to describe the circuit to the simulator. In most SPICE-type simulators the nodes are represented by numbers, with the ground node being node zero. Referring to Figure 8.2, the 723 regulator and its internal op-amp are represented by subcircuits. Each subcircuit has its own set of nodes and components. Subcircuits are useful for encapsulating sections of a circuit or when a certain section needs to be used repeatedly.

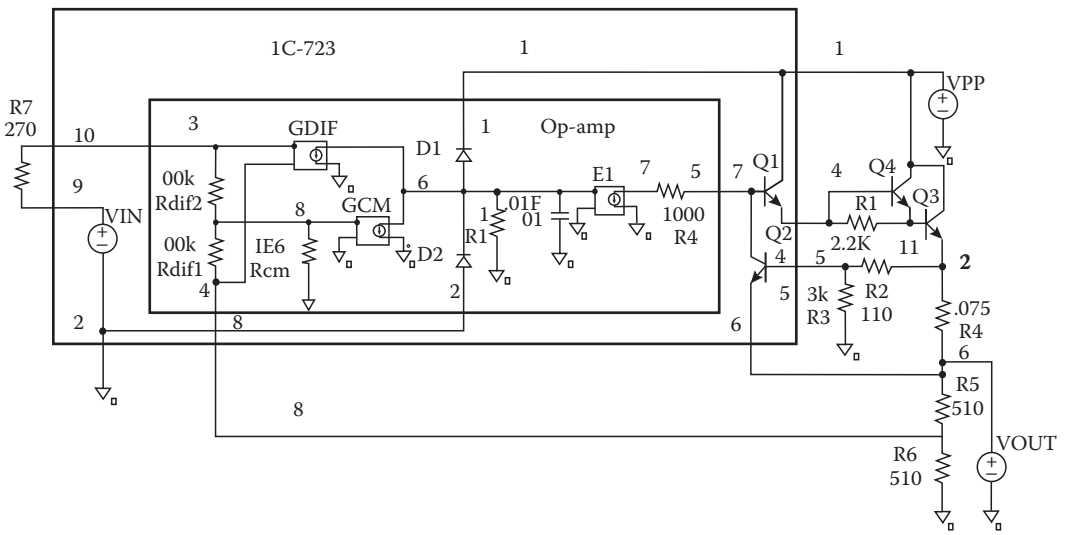


FIGURE 8.1 Regulator circuit to be used for DC analysis, created using PSPICE.

```
Regulator circuit.
*Complete circuit*
*Load source*
vout 6 0
*Power input*
vpp 1 0 11
x1 1 0 4 5 6 7 8 9 10 ic723
*Series Pass transistors*
q3 1 4 11 mq3
q4 1 11 2 mq4
r1 4 11 2.2k
r2 5 2 110
r3 5 0 3k
r4 2 6 0.075
r5 6 8 510
r6 8 0 510
r7 9 10 270
*Control cards*
.op
.model mq3 npn(is = 1e–9 bf = 30
+ br = 5 ikf = 50m)
model mq4 npn(is = 1e–6 bf = 30
+ br = 5 ikf = 10)
.dc vout 1 5.5 .01
.plot dc i(vout)
.probe
.subckt ic723 1 2 4 5 6 7 8 9 10
*Type 723 voltage regulator*
x1 1 2 10 8 7 opamp
*Internal voltage reference*
vr 9 2 2.5
q1 3 7 4 mm
q2 7 5 6 mm
.model mm npn (is = 1e–12 bf = 100)
+br = 5
.ends ic723
*Ideal opamp with limiting
.subckt opamp 1 2 3 4 5
*                vcc vee +in –in out
rdif1 3 8 le5
rdif2 4 8 le5
rcm 8 0 le6
*Common mode gain*
gcm 6 0 8 0 le–1
*Differential mode gain*
gdif 6 0 4 3 1 00
r1 6 0 1
*Single pole response*
c1 6 0 .01
d1 6 1 ideal
d2 2 6 ideal
e1 7 0 6 0 1
rout 5 7 le3
.model ideal d (is = le–6 n = .01)
.ends opamp
```

FIGURE 8.2 SPICE input listing of regulator circuit shown in Figure 8.1.

The following properties are modeled in the op-amp:

1. Common mode gain
2. Differential mode gain
3. Input impedance
4. Output impedance
5. Dominant pole
6. Output voltage clipping

The input terminals of the op-amp connect to a "T" resistance network, which sets the common and differential mode input resistance. Therefore, the common mode resistance is RCM + RDIF = 1.1E6 and the differential mode resistance is RDIF1 + RDIF2 = 2.0E5.

Dependent current sources are used to create the main gain elements. Because these sources force current into a $1-\Omega$ resistor, the voltage gain is Gm*R at low frequency. In the differential mode, this gives (GDIF*R1 = 100). In the common mode, this gives (GCM*R1*(RCM/(RDIF1 + RCM = 0.0909). The two diodes D1 and D2 implement clipping by preventing the voltage at node 6 from exceeding VCC or going below VEE. The diodes are made "ideal" by reducing the ideality factor n. Note that the diode current is $I_d = I_s[\exp(V_d/(nV_t)) - 1]$, where V_t is the thermal voltage (0.026 V). Thus, reducing n makes the diode turn on at a lower voltage.

A single pole is created by placing a capacitor (C1) in parallel with resistor R1. The pole frequency is therefore given by 1.0/2 * π * R1 * C1). Finally, the output is driven by the voltage-controlled voltage source E1 (which has a voltage gain of unity), through the output resistor R4. The output resistance of the op-amp is therefore equal to R4.

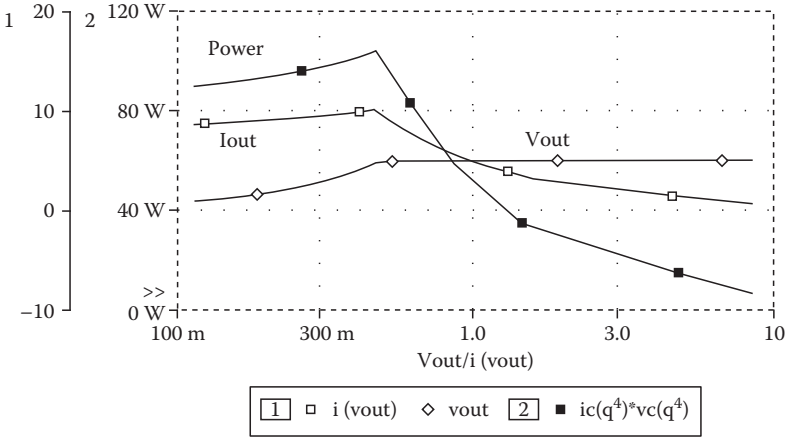


FIGURE 8.3 Output characteristics of regulator circuit using PSPICE.

To observe the output voltage as a function of resistance, the regulator is loaded with a voltage source (VOUT) and the voltage source is swept from 0.05 to 6.0 V. A plot of output voltage vs. resistance can then be obtained by plotting VOUT vs. VOUT/I(VOUT) (using PROBE in this case; see Figure 8.3). Note that for this circuit, even though a current source would seem a more natural choice, a voltage source must be used as a load rather than a current source because the output characteristic curve is multi-valued in current. If a current source were used it would not be possible to easily simulate the entire curve. Of course, many other interesting quantities can be plotted; for example, the power dissipated in the pass transistor can be approximated by plotting IC(Q3)*VC(Q3).

Several restrictions exist as to what constitutes a valid circuit, and in most cases the simulators will complain if the restrictions are violated:

1. All nodes must have a DC path to ground.
2. Voltage sources must not be connected in a loop.
3. Current sources may not be connected in series.
4. Each node must be connected to at least two elements.

For these simulations, PSPICE was used running on an IBM PC. The simulation took <1 min of CPU time.

Pitfalls. Many SPICE users forget that the first line in the input file is used as the title. Therefore, if the first line of the file is a circuit component, the component name will be used as the title and the component will not be included in the circuit. Convergence problems are sometimes experience if ''difficult'' bias conditions are created. An example of such a condition is if a diode is placed in the circuit backwards, resulting in a large forward bias voltage, SPICE will have trouble resolving the current. Another difficult case is if a current source were used instead of a voltage to bias the output in the previous example. If the user then tried to increase the output current above 10 A, SPICE would not be able to converge because the regulator will not allow such a large current.

8.1.3 AC Analysis

AC analysis uses phasor analysis to calculate the frequency response of a circuit. The analysis is useful for calculating the gain, 3 dB frequency input and output impedance, and noise of a circuit as a function of frequency, bias conditions, temperature, etc.

8.1.3.1 Numerical Method

1. DC solution is performed to calculate the Q-point for the circuit.
2. Linearized circuit is constructed at the Q-point. To do this, all nonlinear elements are replaced by their linearized equivalents. For example, a nonlinear current source $I = aV_1^2 + bV_2^3$ would be replaced by a linear voltage controlled current source $I = V_1(2aV_{1q}) + V_2(3bV_{2q}^2)$.
3. All inductors and capacitors are replaced by complex impedances, and conductances evaluated at the frequency of interest.
4. Nodal analysis is now used to reduce the circuit to a linear algebraic complex matrix. The AC node voltages may now be found by applying an excitation vector (which represents the independent voltage and current sources) and using Gaussian elimination (with complex arithmetic) to calculate the node voltages.

AC analysis does have limitations and the following types of nonlinear or large signal problems cannot be modeled:

1. Distortion due to nonlinearities such as clipping, etc.
2. Slew rate-limiting effects
3. Analog mixers
4. Oscillators

Noise analysis is performed by including noise sources in the models. Typical noise sources include thermal noise in resistors $I_n^2 = 4kT\Delta f/R$, and shot $I_n^2 = 2qI_d\Delta f$, and flicker noise in semiconductor devices. Here, T is temperature in Kelvin, k is Boltzmann's constant, and Δf is the bandwidth of the circuit. These noise sources are inserted as independent current sources, $\mathrm{In}_j(f)$ into the AC model. The resulting current due to the noise source is then calculated at a user-specified summation node(s) by multiplying by the gain function between the noise source and the summation node $A_{js}(f)$. This procedure is repeated for each noise source, and then the contributions at the reference node are root mean squared (RMS) summed to give the total noise at the reference node. The equivalent input noise is then easily calculated from the transfer function between the circuit input and the reference node $A_{is}(f)$. The equation describing the input noise is therefore:

$$I_i = \frac{1}{A_{is}(f)} \sqrt{\sum_j [A_{js}(f)\mathrm{In}_j(f)]^2}$$

Example 8.2: Cascode Amplifier with Macro Models

Here, we find the gain, bandwidth, input impedance, and output noise of a cascode amplifier. The circuit for the amplifier is shown in Figure 8.5. The circuit is assumed to be fabricated in a monolithic IC process, so it will be necessary to consider some of the parasitics of the IC process. A cross-section of a typical IC bipolar transistor is shown in Figure 8.4 along with some of the parasitic elements. These parasitic elements are easily included in the amplifier by creating a "macro model" for each transistor. The macro model is then implemented in SPICE form using subcircuits.

The PSPICE circuit simulator allows the user to define parameters which are passed into the subcircuits. This capability is very useful in this case because the resistor model will vary, depending on the value of the resistor. It if is assumed for a certain resistor type that the width w (measured perpendicular to current flow) of the resistor is fixed, e.g., to the minimum line width of the process, then the resistance must be proportional to the length (l) of the resistor ($R \propto l/w$). The parasitic capacitance of the resistor, on the other hand, is proportional to the junction area of the resistor, and therefore to the value of the resistance as well ($C \propto lw \propto R$). Using parameterized subcircuits these relations are easily implemented, and one subcircuit can be used to represent many different resistors (see Figure 8.6). Here,

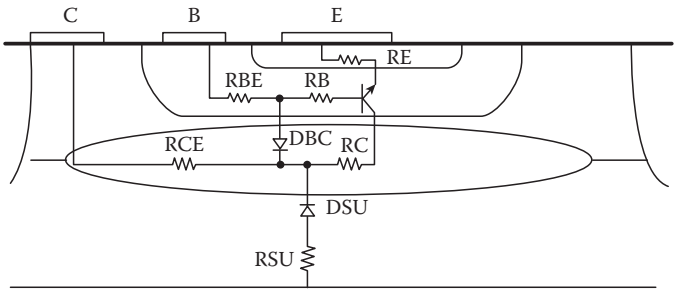


FIGURE 8.4 BJT cross-section with macro model elements.

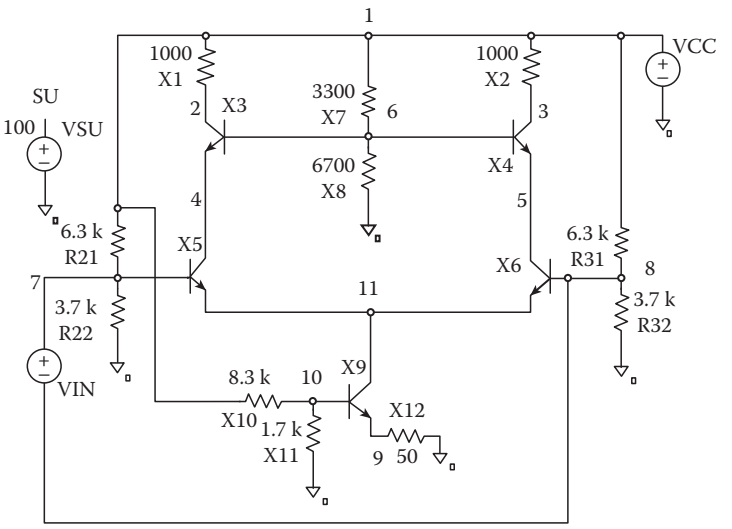


FIGURE 8.5 Cascode amplifier for AC analysis, created using PSPICE.

we represent the capacitance using two diodes one at each end of the resistor. This was done because the resistor junction capacitance is voltage dependent.

The input to the circuit is a voltage source (VIN), applied differentially to the amplifier. The output will be taken differentially across the collectors of the two upper transistors at nodes 2 and 3. The input impedance of the amplifier can be calculated as VIN/I(VIN) or because VIN = 1.0 just as 1/I(VIN). These quantities are shown plotted using PROBE in Figure 8.7. It can be seen that the gain of the amplifier falls off at high frequency as expected. The input impedance also drops because parasitic capacitances shunt the input.

It is also requested in Figure 8.6 that noise analysis be performed at every 20th frequency point. A portion of the noise printout is shown in Figure 8.8. It can be seen that the simulator calculates the noise contributions of each component in the circuit at the specified frequencies and displays them. The total noise at the specified summing node (differentially across nodes 2 and 3) in this case is also calculated as well as the equivalent noise referenced back to the input. This example took <1 min on an IBM PC.

Pitfalls. Many novice users will forget that AC analysis is a linear analysis. They will, for example, apply a 1 V signal to an amplifier with 5 V power supplies and a gain of 1000 and be surprised when SPICE tells them that the output voltage is 1000 V. Of course, the voltage generated in a simple amplifier must be less

```
Cascode amp with macro models.
*P type substrate is node 100
vcc 1 0 10
vsu 100 0 0
vin 7 8 ac 1
x1 1 2 100 icr PARAMS: val = 1k
x2 1 3 100 icr PARAMS: val = 1k
x3 2 6 4 100 tran
x4 3 6 5 100 tran
x5 4 7 11 100 tran
x6 5 8 11 100 tran
*cascode base bias divider*
x7 1 6 100 icr PARAMS: val = 3.3k
x8 6 0 100 icr PARAMS: val = 6.7k
*input bias dividers*
r21 1 7 6.3k
r22 7 0 3.7k
r31 1 8 6.3k
r32 8 0 3.7k
*Current Source*
x9 11 10 9 100 tran
x10 9 0 100 icr PARAMS: val = 100
x11 9 0 100 icr PARAMS: val = 100
x12 1 10 100 icr PARAMS: val = 8.3k
x13 10 0 100 icr PARAMS: val = 1.7k
.op
.noise v(2,3) vin 8
.ac dec 8 le6 le10
.probe

.subckt tran 1 2 3 4 PARAMS: val = 1
q1 5 2 3 mq{val}
rc 1 5 20/{val}
dbc 2 5 mdbc {val}
dsc 4 5 mdcs {val}
.model mq npn (is = le–15 bf = 100 br = 5
+ vaf =  30 var = 10 ikf = 5e–3 ikr =  le–4
+ re = 5 rb = 100 rc = 50 cje = .1p cjc = .05p
+ tf = 100p tr = 1n)
.model mdbc d (is = 1e–16 cjo =  .05p)
.model mdcs d (is = 1e–16 cjo =  .01p)
+ rs = 1000)
.ends tran

.subckt icr 1 2 3 PARAMS: val = 1000
rr 1 2 {val}
d1 3 1 md {val/1000}
d2 3 1 md {val/1000}
.model md d (is = le–16 cjo = 10f rs = ik)
.ends icr
```

FIGURE 8.6 SPICE input listing for cascode amplifier of Figure 8.5.

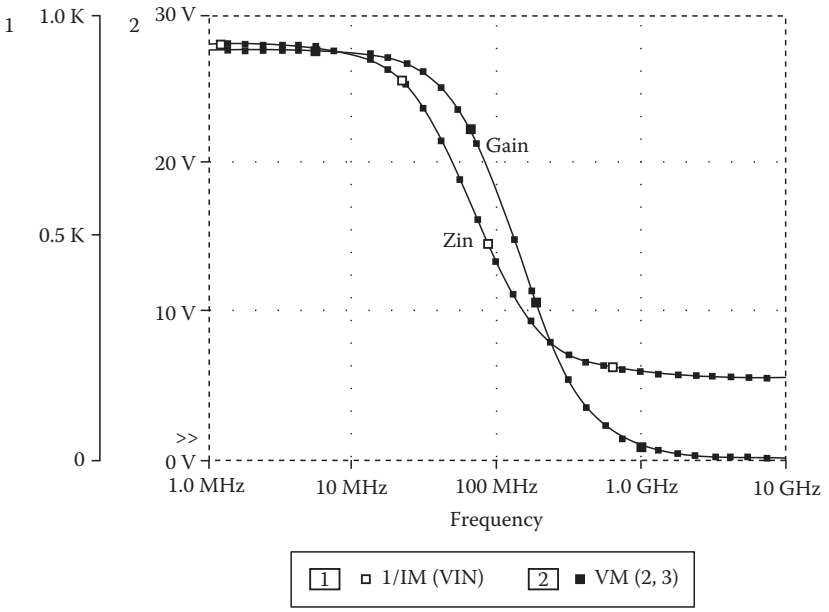


FIGURE 8.7 Gain and input impedance of cascode amplifier.

**** NOISE ANALYSIS TEMPERATURE = 27.000 DEG C
FREQUENCY = 1.000E+09 HZ

**** DIODE SQUARED NOISE VOLTAGES (SQ V/HZ)

	x1.d1	x1.d2	x2.d1	x2.d2	x3.dsc	x4.dsc
RS	0.000E+00	0.000E+00	0.000E+00	0.000E+00	6.098E−21	6.098E−21
ID	0.000E+00	0.000E+00	0.000E+00	0.000E+00	2.746E−24	2.746E−24
FN	0.000E+00	0.000E+00	0.000E+00	0.000E+00	0.000E+00	0.000E+00
TOTAL	0.000E+00	0.000E+00	0.000E+00	0.000E+00	6.1000E−21	6.100E−21

**** TRANSISTOR SQUARED NOISE VOLTAGES (SQ V/HZ)

	x3.q1	x4.q1	x5.q1	x6.q1	x7.q1
RB	1.519E−19	1.519E−19	1.522E−16	1.522E−16	0.000E+00
RC	3.185E−20	3.185E−20	7.277E−20	7.277E−20	0.000E+00
RE	8.056E−21	8.056E−21	7.418E−18	7.418E−18	0.000E+00
IB	2.190E−18	2.190E−18	3.390E−18	3.390E−18	0.000E+00
IC	8.543E−17	8.543E−17	1.611E−16	1.611E−16	0.000E+00
FN	0.000E+00	0.000E+00	0.000E+00	0.000E+00	0.000E+00
TOTAL	8.782E−17	8.782E−17	3.242E−16	3.242E−19	0.000E+00

****TOTAL OUTPUT NOISE VOLTAGE = 8.554E−16 SQ V/HZ
= 2.925E−08 V/RT HZ

TRANSFER FUNCTION VALUE:
V(2,3)/vin = 9.582E+00
EQUIVALENT INPUT NOISE AT vin = 3.052E−09 V/RT HZ

FIGURE 8.8 Noise analysis results for cascode amplifier.

than the power supply voltage, but to examine such clipping effects, transient analysis must be used. Likewise, selection of a proper *Q*-point is important. If the amplifier is biased in a saturated portion of its response and AC analysis is performed, the gain reported will be much smaller than the actual large signal gain.

8.1.4 Transient Analysis

Transient analysis is the most powerful analysis capability of a simulator because the transient response is so hard to calculate analytically. Transient analysis can be used for many types of analysis, such as switching speed, distortion, basic operation of certain circuits like switching power supplies. Transient analysis is also the most CPU intensive and can require 100 or 1000 times the CPU time as a DC or AC analysis.

8.1.4.1 Numerical Method

In a transient analysis time is discretized into intervals called time steps. Typically, the time steps are of unequal length, with the smallest steps being taken during portions of the analysis when the circuit voltages and currents are changing most rapidly. The capacitors and inductors in the circuit are then replaced by voltage and current sources based on the following procedure.

The current in a capacitor is given by $I_c = C\mathrm{d}V_c/\mathrm{d}t$. The time derivative can be approximated by a difference equation.

$$I_c^k + I_c^{k-1} = 2C\frac{V_c^k - V_c^{k-1}}{t^k - t^{k-1}}$$

In this equation, the superscript k represents the number of the time step. Here, k is the time step we are presently solving for and $(k-1)$ is the previous time step. This equation can be solved to give the capacitor current at the present time step.

$$I_c^k = V_c^k(2C/\Delta t) - V_c^{k-1}(2C/\Delta t) - I_c^{k-1}$$

Here $\Delta t = t^k - t^{k-1}$, or the length of the time step. As time steps are advanced, $V_c^{k-1} \rightarrow V_c^k; I_c^{k-1} \rightarrow I_c^k$. Note that the second two terms on the right hand side of the above equation are dependent only on the capacitor voltage and current from the previous time step, and are therefore fixed constants as far as the present step is concerned. The first term is effectively a conductance ($g = 2C/\Delta t$) multiplied by the capacitor voltage, and the second two terms could be represented by an independent current source. The entire transient model for the capacitor therefore consists of a conductance in parallel with two current sources (the numerical values of these are, of course, different at each time step). Once the capacitors and inductors have been replaced as indicated, the normal method of DC analysis is used. One complete DC analysis must be performed for each time point. This is the reason that transient analysis is so CPU intensive. The method outlined here is the trapezoidal time integration method and is used as the default in SPICE.

Example 8.3: Phase-Locked Loop Circuit

Figure 8.9 shows the phase-locked loop (PLL) circuit. The first analysis considers only the analog multiplier portion (also called a phase detector). The second analysis demonstrates the operation of the entire PLL. For the first analyses we wish to show that the analog multiplier does indeed multiply its input voltages. To do this, sinusoidal signals of 0.5 V amplitude are applied to the inputs, one at 2 MHz and the second at 3 MHz. The analysis is performed in the time domain and a Fourier analysis is used to analyze the output. Because the circuit functions as a multiplier, the output should be

$$V_{out} = AV_a \sin(2e6\pi t)V_b \sin(3e6\pi t) = AV_aV_b[\cos(1e6\pi t) - \cos(5e6\pi t)]$$

The 1 and 5 MHz components are of primary interest. Feedthrough of the original signal will also occur, which will give 2 and 3 MHz components in the output. Other forms of nonlinear distortion will

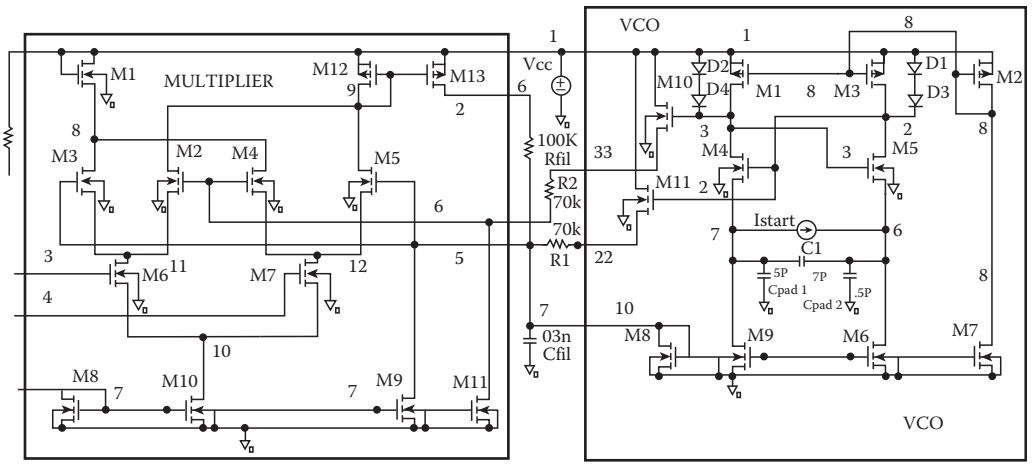


FIGURE 8.9 Phase-locked loop circuit for transient analysis, created with PSPICE.

```
Analog Multiplier Circuit.

Vcc 1 0 10.0
Rbias 1 2 800k
x1 1 9 3 4 5 6 2 10 mult
rdum 9 0 1meg

rin1 4 7 10k
vbias 7 0 2
vin 3 4 sin(0.5 2meg 0 0)

rin2 5 8 10k
vbias2 8 0 5.0
vin2 5 6 sin(0.5 3meg 0 0)

.tran 5n 3u 0 5n
.four 1meg 20 v(10)
.op
.probe
.options defl = 4u defas = 200p
defad = 200p

.subckt mult      1  2   3  4   5   6   7    8
* Pwr Iout In1 In2 Pin1 Pin2 Ibias
Vout
*load resistor*
m1 1 1 8 0 nmos w = 30u 1 = 4u
*Upper diff pairs*
m2 9 5 11 0 nmos w = 60u 1= 4u
m3 8 6 11 0 nmos w = 60u 1= 4u
m4 8 5 12 0 nmos w = 60u 1= 4u
m5 9 6 12 0 nmos w = 60u 1= 4u
*Lower diff pairs*
m6 11 3 10 0 nmos w = 60u 1= 4u
m7 12 4 10 0 nmos w = 60u 1= 4u
*Drive Current Mirror*
m8 7 7 0 0 nmos w = 60u 1= 4u
m9 5 7 0 0 nmos w = 60u 1= 4u
m10 10 7 0 0 nmos w = 60u 1= 4u
m11 6 7 0 0 nmos w = 60u 1= 4u
*Output current Mirror*
m12 9 9 1 1 pmos w = 60u 1= 4u
m13 2 9 1 1 pmos w = 60u 1= 4u

.model nmos nmos (level = 2 tox = 5e–8
+ nsub = 2e 15 tpg = 1 vto = .9 uo = 450
+ ucrit = 8e4 uexp = .15 cgso = 5.2e–10
+ cgdo = 5.2e–10)
.model pmos pmos (level = 2 tox = 5e–8
+ nsub = 2e 15 tpg = –1 vto = –.9 uo = 200
+ ucrit = 8e4 uexp = .15 cgso = 5.2e–10
+ cgdo = 5.2e–10)
.ends mult
```

FIGURE 8.10 SPICE input file for analog multiplier portion of PLL.

give components at higher frequencies. The SPICE input deck is shown in Figure 8.10. The output of the Fourier analysis is shown in Figure 8.11. It can be seen that the components at 1 and 5 MHz dominate and are a factor of 3 larger than the next largest component. The phase of the 1 and 5 MHz component is also offset by approximately 90° from the input (at 0°), as expected.

Simulation of the entire PLL will now be performed. The SPICE input deck is own in Figure 8.12. The phase detector and voltage-controlled oscillator are modeled in separate subcircuits. The phase

FOURIER COMPONENTS OF TRANSIENT RESPONSE V(10)

DC COMPONENT = 8.125452E+00

NO	FREQ HZ	FOURIER COMP.	NORMALIZED COMP.	PHASE (DEG)	NORMALIZED PHASE (DEG)
1	1.000E+06	9.230E–02	1.000E+00	8.973E+01	0.000E+00
2	2.000E+06	6.462E–03	7.002E–02	–1.339E+01	–1.031E+02
3	3.000E+06	3.207E–02	3.474E–01	6.515E+01	–2.458E+01
4	4.000E+06	1.097E–02	1.189E–01	–1.604E+02	–2.501E+02
5	5.000E+06	8.229E–02	8.916E–01	–1.074E+02	–1.971E+02
6	6.000E+06	1.550E–02	1.680E–01	–1.683E+02	–2.580E+02
7	7.000E+06	3.695E–02	4.004E–01	7.984E+01	–9.886E+00
8	8.000E+06	7.943E–03	8.606E–02	1.302E+02	4.044E+00
9	9.000E+06	1.962E–02	2.126E–01	–1.149E+02	–2.046E+02

FIGURE 8.11 Results of transient and Fourier analyses of analog multiplier.

```
Phase locked loop circuit.
Vcc 1 0 10.0
Rbias 1 2 800k
rin 1 4 5 100k
vbias 5 0 2
Rfil 6 7 100k
Cfil 6 0 .03n
x1 1 6 3 4 18 19 2 10 mult
r1 8 18 70k
r2 9 19 70k
vsens 7 17 0
x2 1 8 9 17 vco
vin 3 4 sffm(0 1 600k 2 60k)
.tran .05u 60u
.probe
.options acct defl = 4u defas = 200p
+ defad = 200p

.subckt vco 1 22 33 10
*          Pwr out1 out2 Ict1

* P current mirror
m1 2 8 1 1 pmos w = 10u 1 = 4u
m2 8 8 1 1 pmos w = 10u 1 = 4u
m3 3 8 1 1 pmos w = 10u 1 = 4u

*Oscillator
m4 3 2 7 0 nmos w = 10u 1 = 4u
m5 2 3 6 0 nmos w = 10u 1 = 4u

*N current mirror
m6 6 10 0 0 nmos w = 20u 1 = 4u
m7 8 10 0 0 nmos w = 4u 1 = 4u
m8 10 10 0 0 nmos w = 20u 1 = 4u
m9 7 10 0 0 nmos w = 20u 1 = 4u

*source follower buffers
m10 13 33 0 nmos w = 80u 1 = 4u
m11 1 2 22 0 nmos w = 80u 1 = 4u

*Frequency setting capacitor*
c1 6 7 7pf
cpad6 6 0.5pf
cpad7 7 0.5pf

*Diode swing limiters*
d1 1 5 md
d2 1 4 md
d3 5 2 md
d4 4 3 md
*Pulse to start VCO*
istart 6 0 pulse (0 400u. 1us.01us
+.01us .3us 100)
```

FIGURE 8.12 SPICE input listing for phase-locked loop circuit.

detector, or multiplier, subcircuit is the same as that of Figure 8.10 and is omitted from Figure 8.12 for brevity. Examine the VCO subcircuit and note the PULSE-type current source ISTART connected across the capacitor. The source gives a current pulse 0.3E-6 s wide at the start of the simulation to start the VCO running. To start a transient simulation SPICE first computes a DC operating point (to find the initial voltages V_c^{k-1} on the capacitors). As this DC point is a valid, although not necessarily stable, solution, an oscillator will remain at this point indefinitely unless some perturbation is applied to start the oscillations. Remember, this is an ideal mathematical model and no noise sources or asymmetries exist that would start a real oscillator—it must be done manually. The capacitor C1 would have to be placed off-chip, and bond pad capacitance (CPAD1 and CPAD2) have been included at the capacitor nodes. Including the pad capacitances is very important if a small capacitor C1 is used for high-frequency operation.

In this example, the PLL is to be used as a FM detector circuit and the FM signal is applied to the input using a single frequency FM voltage source. The carrier frequency is 600 kHz and the modulation frequency is 60 kHz. Figure 8.13 shows the input voltage and the output voltage of the PLL at the VCO output and at the phase detector output. It can be seen that after a brief starting transient, the PLL locks onto the input signal and that the phase detector output has a strong 60 kHz component. This example took 251 s on a Sun SPARC-2 workstation (3046 time steps, with an average of 5 N iterations per time step).

Pitfalls. Occasionally, SPICE will fail and give the message ''Timestep too small in transient analysis,'' which means that the process of Newton iterations at certain time steps could not be made to converge. One of the most common causes of this is the specification of a capacitor with a value that is much too large, for example, specifying a 1 F capacitor instead of a 1 pF capacitor (an easy mistake to make by not adding the ''p'' in the value specification). Unfortunately, we usually have no way to tell which capacitor is at fault from the type of failure generated other than to manually search the input deck.

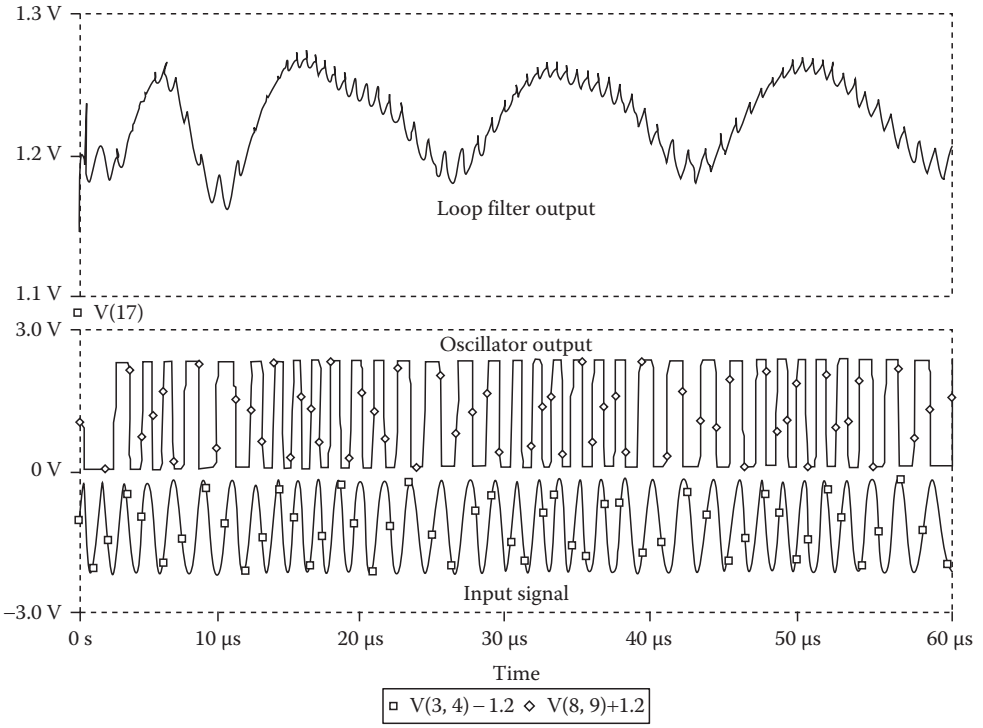


FIGURE 8.13 Transient analysis results of PLL circuit, created using PSPICE.

Other transient failures are caused by MOSFET models. Some models contain discontinuous capacitances (with respect to voltage) and others do not conserve charge. These models can vary from version to version so it is best check the user's guide.

8.1.5 Process and Device Simulation

Process and device simulation are the steps that precede analog circuit simulation in the overall simulation flow (see Figure 8.14). The simulators are also different in that they are not measurement driven as are analog circuit simulators. The input to a process simulator is the sequence of process steps performed (times, temperatures, gas concentrations) as well as the mask dimensions. The output from the process simulator is a detailed description of the solid-state device (doping profiles, oxide thickness, junction depths, etc.). This input to the device simulator is the detailed description generated by the process simulator (or via measurement). The output of the device simulator is the electrical characteristics of the device (I–V curves, capacitances, switching transient curves).

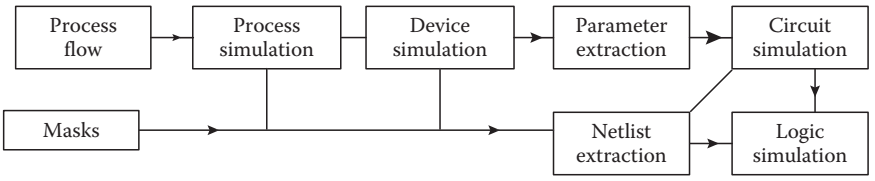


FIGURE 8.14 Data flow for complete process-device-circuit modeling.

Process and device simulation are becoming increasingly important and widely used during the IC design process. A number of reasons exist for this:

- As device dimensions shrink, second-order effects can become dominant. Modeling of these effects is difficult using analytical models.
- Computers have greatly improved, allowing time-consuming calculations to be performed in a reasonable amount of time.
- Simulation allows access to impossible to measure physical characteristics.
- Analytic models are not available for certain devices, for example, thyristors, heterojunction devices, and IGBTS.
- Analytic models have not been developed for certain physical phenomena, for example, single event upset, hot electron aging effects, latchup, and snap-back.
- Simulation runs can be used to replace split lot runs. As the cost of fabricate test devices increases, this advantage becomes more important.
- Simulation can be used to help device, process, and circuit designers understand how their devices and process work.

Clearly, process and device simulation is a topic which can be been the topic of entire texts. The following sections attempt to provide an introduction to this type of simulation, give several examples showing what the simulations can accomplish, and provide references to additional sources of information.

8.1.6 Process Simulation

IC processing involves a number of steps which are designed to deposit (deposition, ion implantation), remove (etching), redistribute (diffusion), or transform (oxidation) the material of which the IC is made. Most process simulation work has been in the areas of diffusion, oxidation, and ion implantation; however, programs are available that can simulate the exposure and development of photo-resist, the associated optical systems, as well as gas and liquid phase deposition and etch.

A number of programs are available (either from universities or commercial vendors) which can model silicon processing. The best known program is SUPREM-IV, which was developed at Stanford University (Stanford, CA). SUPREM-IV is capable of simulating oxidation, diffusion, and ion implantation, and has simple models for deposition and etch. In the following section a very brief discussion of the governing equations used in SUPREM will be given along with the results of an example simulation showing the power of the simulator.

8.1.6.1 Diffusion

The main equation governing the movement of electrically charged impurities (acceptors in this case) in the crystal is the diffusion equation:

$$\frac{\partial C}{\partial t} = \nabla \cdot \left(D\nabla C - \frac{DqC_a}{kT}\mathbf{E} \right)$$

Here

C is the concentration (#/cm^3) of impurities
C_a is the number of electrically active impurities (#/cm^3)
q is the electron charge
k is Boltzmann's constant
T is temperature in Kelvin
D is the diffusion constant
$\mathbf{E}$ is the built-in electric field.

The built-in electric field E in (V/cm) can be found from

$$\mathbf{E} = -\frac{kT}{q}\frac{1}{n}\nabla n$$

In this equation, n is the electron concentration (#/cm^3), which in turn can be calculated from the number of electrically active impurities (C_a). The diffusion constant (D) is dependent on many factors. In silicon the following expression is commonly used:

$$D = F_{IV}\left(D_x + D_+\frac{n_i}{n} + D_-\frac{n}{n_i} + D_=\left(\frac{n}{n_i}\right)^2\right)$$

The four D components represent the different possible charges states for the impurity: (x) neutral, ($+$) positive, ($-$) negative, ($=$) doubly negatively charged. n_i is the intrinsic carrier concentration, which depends only on temperature. Each D component is in turn given by an expression of the type

$$D = A\exp\left(-\frac{B}{kT}\right)$$

Here, A and B are experimentally determined constants, different for each type of impurity (x, +g, $-$, $=$). B is the activation energy for the process. This expression derives from the Maxwellian distribution of particle energies and will be seen many times in process simulation. It is easily seen that the diffusion process is strongly influenced by temperature. The term F_{IV} is an enhancement factor which is dependent on the concentration of interstitials and vacancies within the crystal lattice (an interstitial is an extra silicon atom which is not located on a regular lattice site; a vacancy is a missing silicon atom which results in an empty lattice site) $F_{IV} \propto C_I + C_V$. The concentration of vacancies, C_V, and interstitials, C_I, are in turn determined by their own diffusion equation:

$$\frac{\partial C_v}{\partial t} = +\nabla \cdot D_V \cdot \nabla C_V - R + G$$

In this equation, D_V is another diffusion constant of the form $A\exp(-B/kT)$. R and G represent the recombination and generation of vacancies and interstitials. Note that an interstitial and a vacancy may recombine and in the process destroy each other, or an interstitial and a vacancy pair may be simultaneously generated by knocking a silicon atom off its lattice site. Recombination can occur anywhere in the device via a bulk recombination process $R = A(C_V C_1)\exp(-b/kT)$. Generation occurs where there is damage to the crystal structure, in particular at interfaces where oxide is being grown or in regions where ion implantation has occurred, as the high-energy ions can knock silicon atoms off their lattice sites.

8.1.6.2 Oxidation

Oxidation is a process whereby silicon reacts with oxygen (or with water) to form new silicon dioxide. Conservation of the oxidant requires the following equation:

$$\frac{dy}{dt} = \frac{\mathbf{F}}{N}$$

Here

F is the flux of oxidant (#/cm^2/s)
N is the number of oxidant atoms required to make up a cubic centimeter of oxide
dy/dt is the velocity with which the Si–SiO_2, interface moves into the silicon.

In general, the greater the concentration of oxidant (C_0), the faster the growth of the oxide and the greater the flux of oxidant needed at the Si–SiO_2, interface. Thus, $F = k_s C_0$.

The flux of oxidant into the oxide from the gaseous environment is given by

$$F = h(HP_{ox} - C_0)$$

Here

H is a constant
P is the partial pressure of oxygen in the gas
C_0 is the concentration of oxidant in the oxide at the surface
h is of the form $A \exp(-B/kT)$.

Finally, the moment of the oxidant within the already existing oxide is governed by diffusion: $\mathbf{F} = D_0 \nabla C$ When all these equations are combined, it is found that (in the one-dimensional case) oxides grow linearly $dy/dt \propto t$ when the oxide is thin and the oxidant can move easily through the existing oxide. As the oxide grows thicker $dy/dt \propto \sqrt{t}$ because the movement of the oxidant through the existing oxide becomes the rate-limiting step.

Modeling two-dimensional oxidation is a challenging task. The newly created oxide must "flow" away from the interface where it is being generated. This flow of oxide is similar to the flow of a very thick or viscous liquid and can be modeled by a creeping flow equation:

$$\nabla^2 V \propto \nabla P$$
$$\nabla \cdot \mathrm{V} = 0$$

V is the velocity at which the oxide is moving and P is the hydrostatic pressure. The second equation results from the incompressibility of the oxide. The varying pressure P within the oxide leads to mechanical stress, and the oxidant diffusion constant D_0 and the oxide growth rate constant k_s are both dependent on this stress. The oxidant flow and the oxide flow are therefore coupled because the oxide flow depends on the rate at which oxide is generated at the interface and the rate at which the new oxide is generated depends on the availability of oxidant, which is controlled by the mechanical stress.

8.1.6.3 Ion Implantation

Ion implantation is normally modeled in one of two ways. The first involves tables of moments of the final distribution of the ions that are typically generated by experiment. These tables are dependent on the energy and the type of ion being implanted. The second method involves Monte Carlo simulation of the implantation process. In Monte Carlo simulation, the trajectories of individual ions are followed as they interact with (bounce off) the silicon atoms in the lattice. The trajectories of the ions, and the recoiling Si atoms (which can strike more Si atoms) are followed until all come to rest within the lattice. Typically several thousand trajectories are simulated (each will be different due to the random probabilities used in the Monte Carlo method) to build up the final distribution of implanted ions. Monte Carlo has advantages in that the damage to the lattice can be calculated during the implant process. This damage creates interstitials and vacancies that affect impurity diffusion, as was seen earlier. Monte Carlo can also model channeling, which is a process whereby the trajectories of the ions align with the crystal planes, resulting in greater penetration of the ion than in a simple amorphous target. Monte Carlo has the disadvantage of being CPU intensive.

Process simulation is always done in the transient mode using time steps as was done with transient circuit simulation. Because partial differential equations are involved, rather than ordinary differential equations, spatial discretization is needed as well. To numerically solve the problem, the differential equations are discretized on a grid. Either rectangular or triangular grids in one, two, or

three dimensions are commonly used. This discretization process results in the conversion of the partial differential equations into a set of nonlinear algebraic equations. The nonlinear equations are then solved using a Newton method in a way very similar to the method used for the circuit equations in SPICE.

Example 8.4: NMOS Transistor

In this example, the process steps used to fabricate a typical NMOS transistor will be simulated using SUPREM-4. These steps are

1. Grow initial oxide (30 min at 1000 K).
2. Deposit nitride layer (a nitride layer will prevent oxidation of the underlying silicon).
3. Etch holes in nitride layer.
4. Implant P + channel stop (boron dose = 5e12, energy = 50 keV).
5. Grow the field oxide (180 min at 1000 K wet O_2).
6. Remove all nitride.
7. Perform P channel implant (boron dose = 1e11, energy = 40 keV).
8. Deposit and etch polysilicon for gate.
9. Oxidize the polysilicon (30 min at 1000 K, dry O_2).
10. Implant the light doped drain (arsenic dose = 5e13 energy = 50 keV).
11. Deposit sidewall space oxide.
12. Implant source and drain (arsenic, dose = 1e15, energy = 200 keV).
13. Deposit oxide layer and etch contact holes.
14. Deposit and etch metal.

The top 4 μm of the completed structure, as generated by SUPREM-4, is shown in Figure 8.15. The actual simulation structure used is 200 μm deep to allow correct modeling of the diffusion of the vacancies and interstitials. The gate is at the center of the device. Notice how the edges of the gate have lifted up due to the diffusion of oxidant under the edges of the polysilicon (the polysilicon, as deposited in step 8, is flat). The dashed contours show the concentration of dopants in both the oxide and silicon layers. The short dashes indicate N-type material, while the longer dashes indicate P-type material. This entire simulation requires about 30 min on a Sun SPARC-2 workstation.

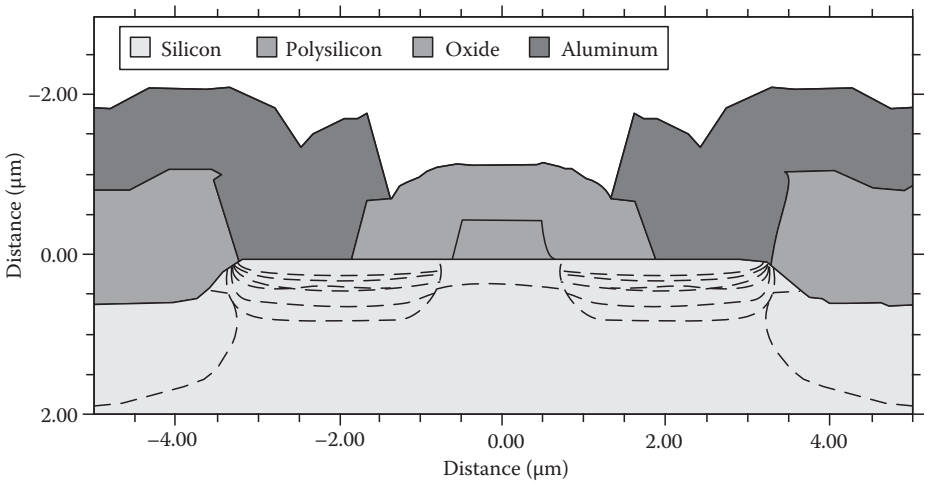


FIGURE 8.15 Complete NMOS transistor cross section generated by process simulation, created with TMA SUPREM-4.

8.1.7 Device Simulation

Device simulation uses a different approach from that of conventional lumped circuit models to determine the electrical device characteristics. Whereas with analytic or empirical models all characteristics are determined by fitting a set of adjustable parameters to measured data, device simulators determine the electrical behavior by numerically solving the underlying set of differential equations. The first of these equations is the Poisson equation, which describes the electrostatic potential within the device.

$$\nabla \cdot \varepsilon \cdot \nabla \Psi = q\left(N_{\mathrm{a}}^{-} - N_{\mathrm{d}}^{+} - p + n - Q_{\mathrm{f}}\right)$$

N_{d} and N_{a} are the concentration of donors and acceptors, i.e., the N- and P-type dopants. Q_{f} is the concentration of fixed charge due, for example, to traps or interface charge. The electron and hole concentrations are given by n and p, respectively, and Ψ is the electrostatic potential.

A set of continuity equations describes the conservation of electrons and holes:

$$\frac{\partial n}{\partial t} = \left(\frac{1}{q}\nabla \cdot J_n - R + G\right)$$

$$\frac{\partial p}{\partial t} = \left(-\frac{1}{q}\nabla \cdot J_p - R + G\right)$$

In these equations, R and G describe the recombination and generation rates for the electrons and holes. The recombination process is influenced by factors such as the number of electrons and holes present as well as the doping and temperature. The generation rate is also dependent upon the carrier concentrations, but is most strongly influenced by the electric field, with increasing electric fields giving larger generation rates. Because this generation process is included, device simulators are capable of modeling the breakdown of devices at high voltage. J_n and J_p are the electron and hole current densities (in amperes per square centimeter). These current densities are given by another set of equations:

$$\mathbf{J}_n = q\mu\left(-n\nabla\Psi + \frac{kT_n}{q}\nabla n\right)$$

$$\mathbf{J}_p = q\mu\left(-p\nabla\Psi - \frac{kT_p}{q}\nabla p\right)$$

In this equation, k is Boltzmann's constant, μ is the carrier mobility, which is actually a complex function of the doping, n, p, electric field, temperature, and other factors. In silicon the electron mobility will range between 50 and 1000 and the hole mobility will normally be a factor of 2 smaller. In other semiconductors such as gallium arsenide the electron mobility can be as high as 5000. T_n and T_p are the electron and hole mean temperatures, which describe the average carrier energy. In many models these default to the device temperature (300 K). In the first term the current is proportional to the electric field ($\nabla\Psi$), and this term represents the drift of carriers with the electric field. In the second term the current is proportional to the gradient of the carrier concentration (∇n), so this term represents the diffusion of carriers from regions of high concentration to those of low concentration. The model is therefore called the drift-diffusion model.

In devices in which self-heating effects are important, a lattice heat equation can also be solved to give the internal device temperature:

$$\sigma(T)\frac{\partial T}{\partial t} = H + \nabla \cdot \lambda(T) \cdot \nabla T$$

$$H = -(J_n + J_p) \cdot \nabla\Psi + H_{\mathrm{R}}$$

where H is the heat generation term, which includes resistive (Joule) heating as well as recombination heating, H_R. The terms $\sigma(T)$, $\lambda(T)$ represent the specific heat and the thermal conductivity of the material (both temperature dependent). Inclusion of the heat equation is essential in many power device problems.

As with process simulation partial differential equations are involved, therefore, a spatial discretization is required. As with circuit simulation problems, various types of analysis are available:

- Steady-state (DC), used to calculate characteristic curves of MOSFETs, BJTs diodes, etc.
- AC analysis, used to calculate capacitances, *Y*-parameters, small signal gains, and *S*-parameters
- Transient analysis, used for calculation of switching and large signal behavior, and special types of analysis such as radiation effects

Example 8.5: NMOS IV Curves

The structure generated in the previous SUPREM-IV simulation is now passed into the device simulator and bias voltages are applied to the gate and drain. Models were included which account for Auger and Shockley Reed Hall recombination, doping and electric field-dependent mobility, and impact ionization. The set of drain characteristics obtained is shown in Figure 8.16. Observe how the curves bend upward at high V_{ds} as the device breaks down. The $V_{gs} = 1$ curve has a negative slope at $I_d = 1.5\text{e-}4$ A as the device enters snap-back. It is possible to model this type of behavior because impact ionization is included in the model.

Figure 8.17 shows the internal behavior of the device with $V_{gs} = 3$ V and $I_d = 3\text{e-}4$ A. The filled contours indicate impact ionization, with the highest rate being near the edge of the drain right beneath the gate. This is to be expected because this is the region in which the electric field is largest due to the drain depletion region. The dark lines indicate current flow from the source to the drain. Some current also flows from the drain to the substrate. This substrate current consists of holes generated by the impact ionization. The triangular grid used in the simulation can be seen in the source, drain, and gate electrodes. A similar grid was used in the oxide and silicon regions.

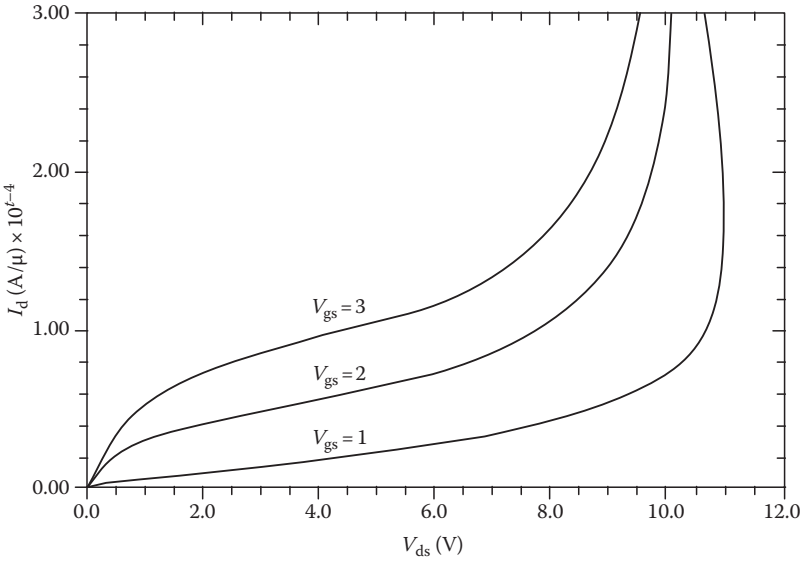


FIGURE 8.16 I_d vs. V_{ds} curves generated by device simulation, created with TMA MEDICI.

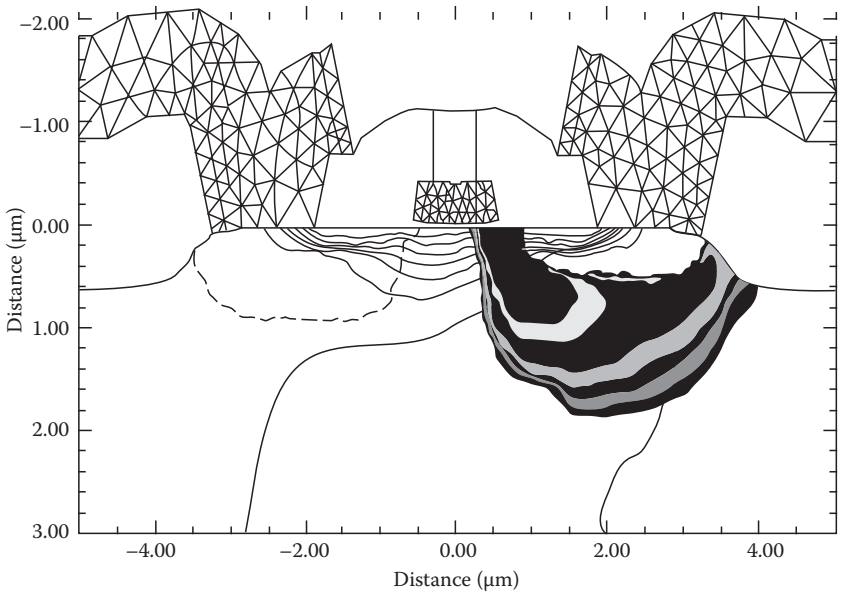


FIGURE 8.17 Internal behavior of MOSFET under bias, created with TMA MEDICI.

Appendix A

A.1 Circuit Analysis Software

SPICE2, SPICE3: University of California, Berkeley, CA
PSPICE: MicroSim Corporation, Irvine, CA (used in this chapter)
HSPICE: Meta Software, Campbell, CA
IsSPICE: Intusoft, San Pedro, CA
SPECTRE: Cadence Design Systems, San Jose, CA
SABRE: Analogy, Beaverton, OR

A.2 Process and Device Simulators

SUPREM-4, PISCES: Stanford University, Palo Alto, CA
MINIMOS: Technical University, Vienna, Austria
SUPREM-4, MEDICI, DAVINCI: Technology Modeling Associates, Palo Alto, CA (used in this chapter)
SEMICAD: Dawn Technologies, Sunnyvale, CA

References

1. P. Antognetti and G. Massobrio, *Semiconductor Device Modeling with SPICE*, New York: McGraw-Hill, 1988.
2. P. W. Tuinenga, *SPICE, A Guide to Circuit Simulation and Analysis Using PSPICE*, Englewood Cliffs, NJ: Prentice Hall, 1988.
3. J. A. Connelly and P. Choi, *Macromodeling with SPICE*, Englewood Cliffs, NJ: Prentice Hall, 1992.
4. S. Selberherr, *Analysis and Simulation of Semiconductor Devices*, Berlin: Springer-Verlag, 1984.
5. R. Dutton and Z. Yu, *Technology CAD, Computer Simulation of IC Process and Devices*, Boston: Kluwer Academic, 1993.

8.2 Parameter Extraction for Analog Circuit Simulation

Peter Bendix

8.2.1 Introduction

8.2.1.1 Definition of Device Modeling

We use various terms such as device characterization, parameter extraction, optimization, and model fitting to address an important engineering task. In all of these, we start with a mathematical model that describes the transistor behavior. The model has a number of parameters that are varied or adjusted to match the I–V (current–voltage) characteristics of a particular transistor or set of transistors. The act of determining the appropriate set of model parameters is what we call device modeling. We then use the model with the particular set of parameters that represent our transistors in a circuit simulator such as SPICE* to simulate how circuits with our kinds of transistors will behave. Usually, the models are supplied by the circuit simulator we chose. Occasionally, we may want to modify these models or construct our own models. In this case we need access to the circuit simulator model subroutines as well as the program that performs the device characterization.

Most people believe the preceding description covers device modeling. However, we feel this is a very narrow definition. One can obtain much more information by characterizing a semiconductor process than just providing models for circuit simulation. If the characterization is done correctly, it can provide detailed information about the fabrication process. This type of information is useful for process architects as an aid in developing future processes. In addition, the wealth of information from the parameters obtained can be used by process and product engineers to monitor wafers routinely. This provides much more information than what is usually obtained by doing a few crude electrical tests for process monitoring. Finally, circuit designers can use device characterization as a means of optimizing a circuit for maximum performance. Therefore, instead of just being a tool used by specialized device physicists, device characterization is a tool for process architects, process and product engineers, and circuit designers. It is a tool that can and should be used by most people concerned with semiconductor manufacture and design.

8.2.1.2 Steps Involved in Device Characterization

Device characterization begins with a test chip. Without the proper test chip structures, proper device modeling cannot be done from measured data. A good test chip for MOS technology would include transistors of varying geometries, gate oxide capacitance structures, junction diode capacitance structures, and overlap capacitance structures. This would be a minimal test chip. Additional structures might include ring oscillators and other circuits for checking the AC performance of the models obtained. It is very important that the transistors be well designed and their geometries be chosen appropriate for the technology as well as the desired device model. For bipolar technology modeling, the layout of structures used to get *S*-parameter measurements is very critical. Although a complete test chip description is beyond the scope of this book, be aware that even perfect device models cannot correct for a poor test chip.

Next, we need data that represent the behavior of a transistor or set of transistors of different sizes. These data can come from direct measurement or they can be produced by a device simulator such as PISCES.† Typically, one does voltage sweeps on various nodes of the transistor and measures the output current at some or all of these nodes. For example, in an MOS transistor, one might sweep the gate voltage and measure the resulting drain current, holding the drain and bulk voltages fixed. The equipment used to do DC device characterization measurements is usually a DC parametric tester.

* SPICE is a circuit simulation program from the Department of Electrical Engineering and Computer Science at the University of California, Berkeley.

† PISCES is a process simulation program from the Department of Electrical Engineering at Stanford University, Stanford, CA.

This equipment usually has a set of SMUs (source/measure units) that can force voltage and measure current, or force current and measure voltage. The measuring equipment can be manually run or controlled by a computer. In either case the measured data must be put onto a hard disk.

For oxide capacitors, junction diodes, and overlap capacitors, a capacitance or inductance, capacitance, resistance (LCR) meter is used to do the measurements. For junction capacitance models, a junction diode (source to bulk, for example) has a reverse bias sweep done and the capacitance is measured at each bias point. For oxide and overlap capacitances, only a single capacitance measurement at one bias voltage is required.

In a more automated environment, switching matrices, semiautomated probers, and temperature-controlled chucks are added. The same equipment is used for bipolar devices. In addition, network analyzers are often used for *S*-parameter characterization to obtain the related AC transit time parameters of bipolar transistors. A typical collection of measuring equipment might look like Figure 8.18.

It is also possible to use a device simulator like PISCES, or a combination of a process simulator like SUPREM-IV* coupled to a device simulator, to provide the simulated results. The process recipe steps are fed into SUPREM-IV, which produces a device structure and associated profiles of all dopants; this information is then fed into the device simulator along with a description of voltage sweeps to be applied to various device electrodes. The output of the device simulator is a set of IV characteristics which closely represent what would have been measured on a real device. The benefits of using simulation over measurement are that no expensive measurement equipment or fabricated wafers are necessary. This can be very helpful when trying to predict the device characteristics of a new fabrication process before any wafers have been produced.

Once the measured (or simulated) data are available, parameter extraction software is used to find the best set of model parameter values to fit the data. This can be done by extraction, optimization, or a combination of the two. Extraction is simpler and quicker, but not as accurate as optimization. In extraction simplifying assumptions are made about the model equations and parameter values are extracted using slope and intercept techniques, for example. In optimization general least-squares curve fitting routines are employed to vary a set or subset of parameters to fit the measured data. Extraction and optimization are covered in more detail in the following sections.

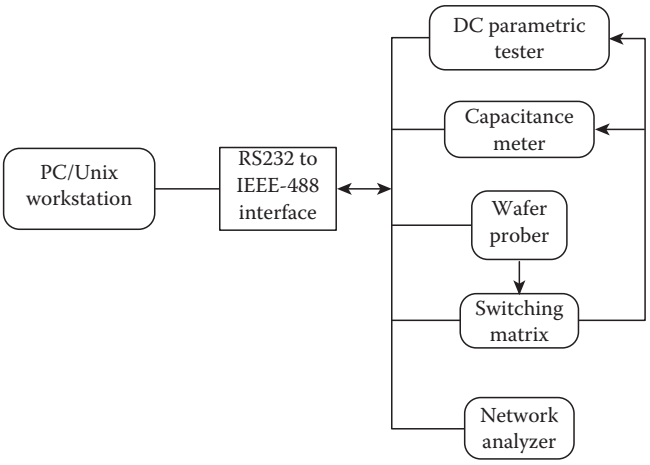


FIGURE 8.18 Typical hardware measuring equipment for device characterization.

* SUPREM-IV is a process simulation program from the Department of Electrical Engineering at Stanford University, Stanford, CA.

8.2.1.3 Least-Squares Curve Fitting (Analytical)

We begin this section by showing how to do least-squares curve fitting by analytical solutions, using a simple example to illustrate the method. We then discuss least-squares curve fitting using numerical solutions in the next section. We can only find analytical solutions to simple problems. The more complex ones must rely on numerical techniques.

Assume a collection of measured data, $m_1, \ldots, m_n$. For simplicity, let these measured data values be functions of a single variable, v, which was varied from v_1 through v_n, measuring each m_i data point at each variable value v_i, i running from 1 to n. For example, the m_i data points might be drain current of an MOS transistor, and the v_i might be the corresponding values of gate voltage. Assume that we have a model for calculating simulated values of the measured data points, and let these simulated values be denoted by $s_1, \ldots, s_n$. We define the least-squares, root mean square (RMS) error as

$$Error_{\text{rms}} = \left[\frac{\sum_{i=1}^{n} \{\text{weight}_i (s_i - m_i)\}^2}{\sum_{i=1}^{n} \{\text{weight}_i m_i\}^2} \right]^{1/2} \tag{8.1}$$

where a weighting term is included for each data point. The goal is to have the simulated data match the measured data as closely as possible, which means we want to minimize the RMS error. Actually, what we have called RMS error is really the relative RMS error, but the two terms are used synonymously. There is another way of expressing the error, called the absolute RMS error, defined as follows:

$$Error_{\text{rms}} = \left[\frac{\sum_{i=1}^{n} \{\text{weight}_i (s_i - m_i)\}^2}{\sum_{i=1}^{n} \{\text{weight}_i \; m_{\min}\}^2} \right]^{1/2} \tag{8.2}$$

where we have used the term $m_{\min}$ in the denominator to represent some minimum value of the measured data. The absolute RMS error is usually used when the measured values approach zero to avoid problems with small or zero denominators in Equation 8.1. For everything that follows, we consider only the relative RMS error. The best result is obtained by combining the relative RMS formula with the absolute RMS formula by taking the maximum of the denominator from Equation 8.1 or 8.2.

We have a simple expression for calculating the simulated data points, s_i, in terms of the input variable, v, and a number of model parameters, $p_1, \ldots, p_m$. That is,

$$s_i = f(v_i, p_1, \ldots, p_m) \tag{8.3}$$

where f is some function. Minimizing the RMS error function is equivalent to minimizing its square. Also, we can ignore the term in the denominator of Equation 8.1 as concerns minimizing, because it is a normalization term. In this spirit, we can define a new error term,

$$\text{Error} = (Error_{\text{rms}})^2 \left[\sum_{i=1}^{n} \{\text{weight}_i m_i\}^2 \right] \tag{8.4}$$

and claim that minimizing Error is equivalent to minimizing $Error_{\text{rms}}$. To minimize Error, we set all partial derivatives of it with respect to each model parameter equal to zero; that is, write

$$\frac{\partial(Error)}{\partial p_j} = 0 \quad \text{for } j = 1, \ldots, m \tag{8.5}$$

Then, solve the preceding equations for the value of p_j.

Least-squares curve fitting. Analytic example**:** Write a simple expression for the drain current of an MOS transistor in the linear region in terms of a single variable, the gate voltage, V_{gs}, and also in terms of two model parameters, β and V_{th};

$$I_{ds} = \beta(V_{gs} - V_{th})V_{ds} \tag{8.6}$$

We denote the measured drain current by I_{meas}. In terms of our previous notation, we have the following substitutions:

$$\begin{aligned} s &\rightarrow I_{ds} \\ v &\rightarrow V_{gs} \\ p_1 &\rightarrow \beta \\ p_2 &\rightarrow V_{th} \end{aligned}$$

We have two conditions to satisfy for minimizing the error:

$$\frac{\partial(Error)}{\partial \beta} = 0 \tag{8.7}$$

$$\frac{\partial(Error)}{\partial V_{th}} = 0 \tag{8.8}$$

Equations 8.7 and 8.8 imply, respectively,

$$\sum_{i=1}^{n} [\beta(V_{gs_i} - V_{th})V_{ds} - I_{meas_i}](V_{gs_i} - V_{th}) = 0 \tag{8.9}$$

$$\sum_{i=1}^{n} [\beta(V_{gs_i} - V_{th})V_{ds} - I_{meas_i}] = 0 \tag{8.10}$$

Solving Equations 8.9 and 8.10 is straightforward but tedious. The result is

$$V_{th} = \frac{[\text{Term}_1\text{Term}_2 - \text{Term}_3\text{Term}_4]}{[n\text{Term}_1 - \text{Term}_2\text{Term}_4]} \tag{8.11}$$

$$\beta = \left(\frac{1}{V_{ds}}\right)\frac{[n\text{Term}_1 - \text{Term}_2\text{Term}_4]}{\left[n\text{Term}_3 - \text{Term}_2^2\right]} \tag{8.12}$$

where n is the number of data points that i is summed over, and

$$\text{Term}_1 = \sum_{i=1}^{n} V_{gs_i} I_{meas_i} \tag{8.13}$$

$$\text{Term}_2 = \sum_{i=1}^{n} V_{\text{gs}_i} \tag{8.14}$$

$$\text{Term}_3 = \sum_{i=1}^{n} V_{\text{gs}_i}^2 \tag{8.15}$$

and

$$\text{Term}_4 = \sum_{i=1}^{n} I_{\text{meas}_i} \tag{8.16}$$

Thus, analytical solutions can become quite messy, even for the simplest model expressions. In fact, it is usually not possible to solve the system of equations analytically for more complicated models. Then, we must rely on numerical solutions.

One further word of caution is required. One may try comparing analytical solutions to those obtained by numerical techniques. They almost always will not match. The reason is that the weighting functions used to calculate $Error_{\text{rms}}$ are different for the two cases. In order to compare the two techniques, one must know the exact algorithm that the numerical least-squares curve fitting routine is using and be able to match the weighting functions. Often weighting is implicit in the numerical least-squares curve fitting algorithm that is not explicitly obvious.

8.2.1.4 Least-Squares Curve Fitting (Numerical)

For almost all practical applications we are forced to do least-squares curve fitting numerically, because the analytic solutions as previously discussed are not obtainable in closed form. What we are calling least-squares curve fitting is more generally known as nonlinear optimization. Many fine references on this topic are available. We highlight only the main features here, and paraphrase parts of [2] in what follows.

Basically, we want to solve the optimization problem whose goal is to minimize the function $F(\mathbf{x})$ where F is some arbitrary function of the vector $\mathbf{x}$. If extra conditions exist on the components of $\mathbf{x}$, we are then solving a constrained, nonlinear optimization problem. When using least-squares fitting for model parameter optimization, we usually constrain the range of the fitting parameters. This type of constraint is a very simple one and is a special case of what is called nonlinear constrained optimization. This or the unconstrained approach are the types normally used for model parameter optimization.

For simplicity, let F be a function of a single variable, x, instead of the vector, $\mathbf{x}$. We state without proof that a necessary condition for a minimum of $F(x)$ to exist is that

$$F' = \frac{\partial F}{\partial x} = 0 \tag{8.17}$$

This means we solve the minimization problem if we can solve for the zeroes of F'. This is exactly what we did when we went through the least-squares analytical method.

One numerical approach to solving for the zeroes of a function, f, is Newton's method. This is an iterative method in which we write

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{8.18}$$

Steepest descents method. Next consider the more general case in which F is now a function of many variables, $\mathbf{x}$; i.e., F is multivariate. We want to consider numerical techniques for finding the minimum of

F by iteration. We also would like to take iteration steps that decrease the value of F at each step. This imposes what is called a descent condition:

$$F(\mathbf{x}_{k+1}) < F(\mathbf{x}_k) \tag{8.19}$$

where k is the iteration number. Solving the minimization problem amounts to choosing a search direction for the vector $\mathbf{x}$ such that at each iteration F decreases until some convergence criterion is satisfied with $\mathbf{x}_k$ as the solution. Let us write F as a Taylor series expanded around the point $\mathbf{x}$, with $\mathbf{p}$ a vector of unit length and h a scalar

$$F(\mathbf{x} + h\mathbf{p}) = F(\mathbf{x}) + hg^{\mathrm{T}}(\mathbf{x})\mathbf{p} + \frac{1}{2}h^2\mathbf{p}^{\mathrm{T}}G(\mathbf{x})\mathbf{p} + \cdots \tag{8.20}$$

where in Equation 8.20, $g(\mathbf{x})$ is the gradient (first derivative in the $\mathbf{p}$ direction) of $F(\mathbf{x})$ and $G(\mathbf{x})$ is the curvature (second derivative in the $\mathbf{p}$ direction) of $F(\mathbf{x})$. The uppercase "T" means matrix transpose.

We want to choose a direction and step length for the next iteration in which

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h_k\mathbf{p}_k \tag{8.21}$$

where in Equation 8.21, $\mathbf{p}$ is a unit vector in the search direction and h is the step size. In the method of steepest descents we set

$$\mathbf{p}_k = -g(\mathbf{x}_k) \tag{8.22}$$

That is, the method of steepest descents chooses the search direction to be opposite the first derivative in the direction $\mathbf{x}_k$.

Newton's method. The method of steepest descents is a first derivative method. Newton's method is a second derivative method (this is not to be confused with the simple Newton's method discussed previously). Referring to Equation 8.20, the Newton direction of search is defined to be the vector $\mathbf{p}$, which satisfies

$$G(\mathbf{x}_k)\mathbf{p}_k = -g(\mathbf{x}_k) \tag{8.23}$$

Finding $\mathbf{p}_k$ from the preceding equation involves calculating second derivatives to compute G.

Gauss–Newton method. We want to modify Newton's method so that second derivative calculations are avoided. To do this, consider the so-called least-squares problem in which F can be written as

$$F(\mathbf{x}) = \sum [f_i(\mathbf{x})]^2 \tag{8.24}$$

In this special case, we can write

$$g(\mathbf{x}) = J^{\mathbf{T}}(\mathbf{x})f(\mathbf{x}) \tag{8.25}$$

$$G(\mathbf{x}) = J^{\mathbf{T}}(\mathbf{x})J(\mathbf{x}) + Q(\mathbf{x}) \tag{8.26}$$

where J is the Jacobian of f and

$$Q(\mathbf{x}) = \sum f_i(\mathbf{x})G_i(\mathbf{x}) \tag{8.27}$$

where G_i is the so-called Hessian of f. Substituting Equations 8.25 and 8.26 into Equation 8.23 gives

$$\left[J^{\mathrm{T}}(\mathbf{x}_k)J(\mathbf{x}_k) + Q_k\right]\mathbf{p}_k = -J^{\mathbf{T}}(\mathbf{x}_k)f(\mathbf{x}_k) \tag{8.28}$$

If we throw away the Q term, Equation 8.28 becomes

$$\left[J^{\mathrm{T}}(\mathbf{x}_k)J(\mathbf{x}_k)\right]\mathbf{p}_k = -J^{\mathrm{T}}(\mathbf{x}_k)f(\mathbf{x}_k) \tag{8.29}$$

a condition for finding the search direction, $\mathbf{p}_k$, which does not involve second derivatives. Solving Equation 8.29 is tantamount to solving the linear least-squares problem of minimizing

$$F(\mathbf{x}) = \sum [\, j_i(\mathbf{x}_k)\mathbf{p}_k + f_i(\mathbf{x}_k)]^2 \tag{8.30}$$

When the search direction, $\mathbf{p}_k$, is found from Equation 8.29, we call this the Gauss–Newton method.

Levenberg–Marquardt algorithm. Both the method of steepest descents and the Gauss–Newton method have advantages and disadvantages. Briefly stated, the method of steepest descents will move toward the correct solution very quickly, but due to the large steps it takes, it can jump past the solution and not converge. The Gauss–Newton method, however, moves more slowly toward the correct solution, but because it tends to take smaller steps, it will converge better as its approaches the solution. The Levenberg–Marquardt algorithm was designed to combine the benefits of both the steepest descents and Gauss–Newton methods.

The Levenberg–Marquardt search direction is found by solving

$$\left[J^{\mathrm{T}}(\mathbf{x}_k)J(\mathbf{x}_k) + \lambda_k I\right]\mathbf{p}_k = -J^{\mathrm{T}}(\mathbf{x}_k)f(\mathbf{x}_k) \tag{8.31}$$

where λ_k is a nonnegative scalar and I is the identity matrix. Note that Equation 8.31 becomes Equation 8.29 when λ_k is zero, so that the Levenberg–Marquardt search direction becomes the Gauss–Newton in this case; as $\lambda_k \to \infty$, $\mathbf{p}_k$ becomes parallel to $-J^{\mathrm{T}}(\mathbf{x}_k)\, f(\mathbf{x}_k)$, which is just $-g(\mathbf{x}_k)$, the direction of steepest descents. Generally speaking, in the Levenberg–Marquardt algorithm the value of λ_k is varied continuously, starting with a value near the steepest descents direction (large λ_k) and moving toward smaller values of λ_k as the solution is approached.

8.2.1.5 Extraction (as Opposed to Optimization)

The terms ''extraction'' and ''optimization'' are, unfortunately, used interchangeably in the semiconductor industry; however, strictly speaking, they are not the same. By optimization, we mean using generalized least-squares curve fitting methods such as the Levenberg–Marquardt algorithm to find a set of model parameters. By extraction, we mean any technique that does not use general least-squares fitting methods. This is a somewhat loose interpretation of the term extraction, but perhaps the following discussion will justify it.

Suppose we have measured the drain current of an MOS transistor in the linear region at zero backbias (the bulk node bias) and we want to find the threshold voltage by extraction (as opposed to optimization). We could plot the drain current vs. gate voltage, draw a tangent line at the point where the slope of I_{ds} is a maximum, and find the V_{gs} axis intercept of this tangent line. This would give us a crude value of the threshold voltage, V_{th} (neglecting terms such as $V_{\mathrm{ds}}/2$). This would be a graphic extraction technique; it would not involve using least-squares optimization.

We could also apply the preceding graphic technique in an equivalent algorithmic way, without drawing any graphs. Let us write a linear region equation for the drain current

$$I_{\mathrm{ds}} = \beta(V_{\mathrm{gs}} - V_{\mathrm{th}})V_{\mathrm{ds}}[1 + Ke^{-\alpha(V_{\mathrm{gs}} - V_{\mathrm{th}})}] \tag{8.32}$$

In Equation 8.32, the exponential term is the subthreshold contribution to the drain current; it becomes negligible for $V_{\mathrm{gs}} \gg V_{\mathrm{th}}$. To find the value of V_{gs} where the maximum slope occurs, we set the derivative of I_{ds} with respect to V_{gs} equal to zero and solve for V_{gs}. The solution is

$$V_{gs}(\text{maximum slope}) = V_{th} + \left[\frac{1+K}{2\alpha K}\right] \tag{8.33}$$

where in obtaining Equation 8.33 we used the approximation

$$e^{-\alpha(V_{gs}-V_{th})} \cong 1 - \alpha(V_{gs} - V_{th}) \tag{8.34}$$

If we rewrite (8.33) for V_{th} in terms of V_{gs} (maximum slope) we have

$$V_{th} = V_{gs}(\text{maximum slope}) - \left[\frac{1+K}{2\alpha K}\right] \tag{8.35}$$

Therefore, the approach is to use a measurement search routine that seeks the maximum slope, $\Delta I_{ds}/\Delta V_{gs}$, where the Δ is calculated as numerical derivatives, and having found V_{gs} (maximum slope) through measurement, use Equation 8.35 to calculate V_{th} from V_{gs} (maximum slope).

We could continue by including the effects of mobility degradation, modifying Equation 8.32 to

$$I_{ds} = \left[\frac{\beta}{1+\theta(V_{gs} - V_{th})}\right](V_{gs} - V_{th})V_{ds}[1 + Ke^{-\alpha(V_{gs}-V_{th})}] \tag{8.36}$$

where we have now included the effects of mobility degradation with the θ term. The extraction analysis can be done again by including this term, with a somewhat different solution obtained using suitable approximations.

The main point is that we write the equations we want and then solve them by whatever approximations we choose, as long as these approximations allow us to get the extracted results in closed form. This is parameter extraction.

8.2.1.6 Extraction vs. Optimization

Extraction has the advantage of being much faster than optimization, but it is not always as accurate. It is also much harder to supply extraction routines for models that are being developed. Each time you make a change in the model, you must make suitable changes in the corresponding extraction routine. For optimization, however, no changes are necessary other than the change in the model itself, because least-squares curve fitting routines are completely general. Also, if anything goes wrong in the extraction algorithm (and no access to the source code is available), almost nothing can be done to correct the problem. With optimization, one can always change the range of data, weighting, upper and lower bounds, etc. A least-squares curve fitting program can be steered toward a correct solution.

Novices at device characterization find least-squares curve fitting somewhat frustrating because a certain amount of user intervention and intuition is necessary to obtain the correct results. These beginners prefer extraction methods because they do not have to do anything. However, after being burned by extraction routines that do not work, a more experienced user will usually prefer the flexibility, control, and accuracy that optimization provides.

Commercial software is available that provides both extraction and optimization together. The idea here is to first use extraction techniques to make reasonable initial guesses and then use these results as a starting point for optimization, because optimization can give very poor results if poor initial guesses for the parameters are used. Nothing is wrong with using extraction techniques to provide initial guesses for optimization, but for an experienced user this is rarely necessary, assuming that the least-squares curve fitting routine is robust (converges well) and the experienced user has some knowledge of the process under characterization. Software that relies heavily on extraction may do so because of the nonrobustness of its optimizer.

These comments apply when an experienced user is doing optimization locally, not globally. For global optimization (a technique we do not recommend), the previous comparisons between extraction and optimization are not valid. The following section contains more detail about local vs. global optimization.

8.2.1.7 Strategies: General Discussion

The most naive way of using an optimization would be to take all the measured data for all devices, put them into one big file, and fit to all these data with all model parameters simultaneously. Even for a very high quality, robust optimization program the chances of this method converging are slight. Even if the program does converge, it is almost certain that the values of the parameters will be very unphysical. This kind of approach is an extreme case of global optimization. We call any optimization technique that tries to fit with parameters to data outside their region of applicability a global approach. That is, if we try to fit to saturation region data with linear region parameters such as threshold voltage, mobility, etc., we are using a global approach. In general, we advise avoiding global approaches, although in the strategies described later, sometimes the rules are bent a little.

Our recommended approach is to fit subsets of relevant parameters to corresponding subsets of relevant data in a way that makes physical sense. For example, in the MOS level 3 model, VT0 is defined as the threshold voltage of a long, wide transistor at zero back-bias. It does not make sense to use this parameter to fit to a short channel transistor, or to fit at nonzero back-bias values, or to fit to anywhere outside the linear region. In addition, subsets of parameters should be obtained in the proper order so that those obtained at a later step do not affect those obtained at earlier steps. That is, we would not obtain saturation region parameters before we have obtained linear region parameters because the values of the linear region parameters would influence the saturation region fits; we should have to go back and reoptimize on the saturation region parameters after obtaining the linear region parameters. Finally, never use optimization to obtain a parameter value when the parameter can be measured directly. For example, the MOS oxide thickness, TOX, is a model parameter, but we would never use optimization to find it. Always measure its value directly on a large oxide capacitor provided on the test chip. The recommended procedure for proper device characterization follows:

1. Have all the appropriate structures necessary on your test chip. Without this, the job cannot be performed properly.
2. Always measure whatever parameters are directly measurable. Never use optimization for these.
3. Fit the subset of parameters to corresponding subsets of data, and do so in physically meaningful ways.
4. Fit the parameters in the proper order so that those obtained later do not affect those obtained previously. If this is not possible, iteration may be necessary.

Naturally, a good strategy cannot be mounted if one is not intimately familiar with the model used. There is no substitute for learning as much about the model as possible. Without this knowledge, one must rely on strategies provided by software vendors, and these vary widely in quality.

Finally, no one can provide a completely general strategy applicable to all models and all process technologies. At some point the strategy must be tailored to suit the available technology and circuit performance requirements. This not only requires familiarity with the available device models, but also information from the circuit designers and process architects.

8.2.2 MOS DC Models

8.2.2.1 Available MOS Models

A number of MOS models have been provided over time with the original circuit simulation program, SPICE. In addition, some commercially available circuit simulation programs have introduced their own proprietary models, most notably HSPICE.* This section concentrates on the standard MOS models

* HSPICE is a commercially available, SPICE-like circuit simulation from Meta Software, Campbell, CA.

provided by UC Berkeley's SPICE, not only because they have become the standard models used by all circuit simulation programs, but also because the proprietary models provided by commercial vendors are not well documented and no source code is available for these models to investigate them thoroughly.

MOS levels 1, 2, and 3. Originally, SPICE came with three MOS models known as level 1, level 2, and level 3. The level 1 MOS model is a very crude first-order model that is rarely used. The level 2 and level 3 MOS models are extensions of the level 1 model and have been used extensively in the past and present [11]. These two models contain about 15 DC parameters each and are usually considered useful for digital circuit simulation down to 1 μm channel length technologies. They can fit the drain current for wide transistors of varying length with reasonable accuracy (about 5% RMS error), but have very little advanced fitting capability for analog application. They have only one parameter for fitting the subthreshold region, and no parameters for fitting the derivative of drain current with respect to drain voltage, G_{ds} (usually considered critical for analog applications). They also have no ability to vary the mobility degradation with back-bias, so the fits to I_{ds} in the saturation region at back-bias are not very good. Finally, these models do not interpolate well over device geometry, e.g., if a fit is made to a wide-long device and a wide-short device, and then one observes how the models track for lengths between these two extremes, they usually do not perform well. For narrow devices, they can be quite poor as well. Level 3 has very little practical advantage over level 2, although the level 2 model is proclaimed to be more physically based, whereas the level 3 model is called semiempirical. If only one can be used, perhaps levels 3 is slightly better because it runs somewhat faster and does not have quite such an annoying kink in the transition region from linear to saturation as does level 2.

***Berkeley short-channel IGFET model* (*BSIM*).** To overcome the many shortcomings of level 2 and level 3, the BSIM and BSIM2 models were introduced. The most fundamental difference between these and the level 2 and 3 models is that BSIM and BSIM2 use a different approach to incorporating the geometry dependence [3,6]. In levels 2 and 3, the geometry dependence is built directly into the model equations. In BSIM and BSIM2, each parameter (except for a very few) is written as a sum of three terms:

$$\text{Parameter} = \text{Par}_0 + \frac{\text{Par}_L}{L_{\text{eff}}} + \frac{\text{Par}_W}{W_{\text{eff}}} \tag{8.37}$$

where

Par_0 is the zero-order term
Par_L accounts for the length dependence of the parameter
Par_W accounts for the width dependence
L_{eff} and W_{eff} are the effective channel width and length, respectively

This approach has a large influence on the device characterization strategy, as discussed later. Because of this tripling of the number of parameters and for other reasons as well, the BSIM model has about 54 DC parameters and the BSIM2 model has over 100.

The original goal of the BSIM model was to fit better than the level 2 and 3 models for submicron channel lengths, over a wider range of geometries, in the subthreshold region, and for nonzero back-bias. Without question, BSIM can fit individual devices better than level 2 and level 3. It also fits the subthreshold region better and it fits better for nonzero back-biases. However, its greatest shortcoming is its inability to fit over a large geometry variation. This occurs because Equation 8.37 is a truncated Taylor series in $1/L_{\text{eff}}$ and $1/W_{\text{eff}}$ terms, and in order to fit better over varying geometries, higher power terms in $1/L_{\text{eff}}$ and $1/W_{\text{eff}}$ are needed. In addition, no provision was put into the BSIM model for fitting G_{ds}, so its usefulness for analog applications is questionable. Many of the BSIM model parameters are unphysical, so it is very hard to understand the significance of these model parameters. This has profound implications for generating skew models (fast and slow models to represent the process corners) and for incorporating temperature dependence. Another flaw of the BSIM model is its wild

behavior for certain values of the model parameters. If model parameters are not specified for level 2 or 3, they will default to values that will at least force the model to behave well. For BSIM, not specifying certain model parameters, setting them to zero, or various combinations of values can cause the model to become very ill-behaved.

BSIM2. The BSIM2 model was developed to address the shortcomings of the BSIM model. This was basically an extension of the BSIM model, removing certain parameters that had very little effect, fixing fundamental problems such as currents varying the wrong way as a function of certain parameters, adding more unphysical fitting parameters, and adding parameters to allow fitting G_{ds}. BSIM2 does fit better than BSIM, but with more than twice as many parameters as BSIM, it should. However, it does not address the crucial problem of fitting large geometry variations. Its major strengths over BSIM are fitting the subthreshold region better, and fitting G_{ds} better. Most of the other shortcomings of BSIM are also present in BSIM2, and the large number of parameters in BSIM2 makes it a real chore to use in device characterization.

BSIM3. Realizing the shortcomings of BSIM2, UC Berkeley recently introduced the BSIM3 model. This is an unfortunate choice of name because it implies BSIM3 is related to BSIM and BSIM2. In reality, BSIM3 is an entirely new model that in some sense is related more to levels 2 and 3 than BSIM or BSIM2. The BSIM3 model abandons the length and width dependence approach of BSIM and BSIM2, preferring to go back to incorporating the geometry dependence directly into the model equations, as do levels 2 and 3. In addition, BSIM3 is a more physically based model, with about 30 fitting parameters (the model has many more parameters, but the majority of these can be left untouched for fitting), making it more manageable, and it has abundant parameters for fitting G_{ds}, making it a strong candidate for analog applications.

It is an evolving model, so perhaps it is unfair to criticize it at this early stage. Its greatest shortcoming is, again, the inability to fit well over a wide range of geometries. It is hoped that future modifications will address this problem. In all fairness, however, it is a large order to ask a model to be physically based, have not too many parameters, be well behaved for all default values of the parameters, fit well over temperature, fit G_{ds}, fit over a wide range of geometries, and still fit individual geometries as well as a model with over 100 parameters, such as BSIM2. Some of these features were compromised in developing BSIM3.

Proprietary models. A number of other models are available from commercial circuit simulator vendors, the literature, etc. Some circuit simulators also offer the ability to add a researcher's own models. In general, we caution against using proprietary models, especially those which are supplied without source code and complete documentation. Without an intimate knowledge of the model equations, it is very difficult to develop a good device characterization strategy. Also, incorporating such models into device characterization software is almost impossible. To circumvent this problem, many characterization programs have the ability to call the entire circuit simulator as a subroutine in order to exercise the proprietary model subroutines. This can slow program execution by a factor of 20 or more, seriously impacting the time required to characterize a technology. Also, if proprietary models are used without source code, the circuit simulator results can never be checked against another circuit simulator. Therefore, we want to stress the importance of using standard models. If these do not meet the individual requirements, the next best approach is to incorporate a proprietary model whose source code one has access to. This requires being able to add the individual model not only to circuit simulators, but also to device characterization programs; it can become a very large task.

8.2.2.2 MOS Level 3 Extraction Strategy in Detail

The strategy discussed here is one that we consider to be a good one, in the spirit of our earlier comments. Note, however, that this is not the only possible strategy for the level 3 model. The idea here is to illustrate basic concepts so that this strategy can be refined to meet particular individual requirements.

In order to do a DC characterization, the minimum requirement is one each of the wide-long, wide-short, and narrow-long devices. We list the steps of the procedure and then discuss them in more detail.

Step 1. Fit the wide-long device in the linear region at zero back-bias, at V_{gs} values above the subthreshold region, with parameters VT0 (threshold voltage), U0 (mobility), and THETA (mobility degradation with V_{gs}).

Step 2. Fit the wide-short device in the linear region at zero back-bias, at V_{gs} values above the subthreshold region, with parameters VT0, LD (length encroachment), and THETA. When finished with this step, replace VT0 and THETA with the values from step 1, but keep the value of LD.

Step 3. Fit the narrow-long device in the linear region at zero back-bias, at V_{gs} values above the subthreshold region, with parameters VT0, DW (width encroachment), and THETA. When finished with this step, replace VT0 and THETA with the values from step 1, but keep the value of DW.

Step 4. Fit the wide-short device in the linear region at zero back-bias, at V_{gs} values above the subthreshold region, with parameters RS and RD (source and drain series resistance).

Step 5. Fit the wide-long device in the linear region at all back-biases, at V_{gs} values above the subthreshold region, with parameter NSUB (channel doping affects long channel variation of threshold voltage back-bias).

Step 6. Fit the wide-short device in the linear region at all back-biases, at V_{gs} values above the subthreshold region, with parameter XJ (erroneously called the junction depth; affects short-channel variation of threshold voltage with back-bias).

Step 7. Fit the narrow-long device in the linear region at zero back-bias, at V_{gs} values above the subthreshold region, with parameter DELTA (narrow channel correction to threshold voltage).

Step 8. Fit the wide-short device in the saturation region at zero back-bias (or all back-biases) with parameters VMAX (velocity saturation), KAPPA (saturation region slope fitting parameter), and ETA (V_{ds} dependence of threshold voltage).

Step 9. Fit the wide-short device in the subthreshold region at whatever back-bias and drain voltage is appropriate (usually zero back-bias and low V_{ds}) with parameter NFS (subthreshold slope fitting parameter). One may need to fit with VT0 also and then VT0 is replaced after this step with the value of VT0 obtained from step 1.

This completes the DC characterization steps for the MOS level 3 model. One would then go on to do the junction and overlap capacitance terms (discussed later). Note that this model has no parameters for fitting over temperature, although temperature dependence is built into the model that the user cannot control.

In Step 1, VT0, U0, and THETA are defined in the model for a wide-long device at zero back-bias. They are zero-order fundamental parameters without any short or narrow channel corrections. We therefore fit them to a wide-long device. It is absolutely necessary that such a device be on the test chip. Without it, one cannot obtain these parameters properly. The subthreshold region must be avoided also because these parameters do not control the model behavior in subthreshold.

In Step 2, we use LD to fit the slope of the linear region curve, holding U0 fixed from step 1. We also fit with VT0 and THETA because without them the fitting will not work. However, we want only the value of LD that fits the slope, so we throw away VT0 and THETA, replacing them with the values from step 1.

Step 3 is the same as step 2, except that we are getting the width encroachment instead of the length.

In Step 1, the value of THETA that fits the high V_{gs} portion of the wide-long device linear region curve was found. Because the channel length of a long transistor is very large, the source and drain series resistances have almost no effect here, but for a short-channel device, the series resistance will also affect the high V_{gs} portion of the linear region curve. Therefore, in step 4 we fix THETA from step 1 and use RS and RD to fit the wide-short device in the linear region, high V_{gs} portion of the curve.

In Step 5, we fit with NSUB to get the variation of threshold voltage with back-bias. We will get better results if we restrict ourselves to lower values of V_{gs} (but still above subthreshold) because no mobility

degradation adjustment exists with back-bias, and therefore the fit may not be very good at higher V_{gs} values for the nonzero back bias curves.

Step 6 is just like step 5, except we are fitting the short-channel device. Some people think that the value of XJ should be the true junction depth. This is not true. The parameter XJ is loosely related to the junction depth, but XJ is really the short-channel correction to NSUB. Do not be surprised if XJ is not equal to the true junction depth.

Step 7 uses DELTA to make the narrow channel correction to the threshold voltage. This step is quite straightforward.

Step 8 is the only step that fits in the saturation region. The use of parameters VMAX and KAPPA is obvious, but one may question using ETA to fit in the saturation region. The parameter ETA adjusts the threshold voltage with respect to V_{ds}, and as such one could argue that ETA should be used to fit measurements of I_{ds} sweeping V_{gs} and stepping V_{ds} to high values. In doing so, one will corrupt the fit in the saturation region, and usually we want to fit the saturation region better at the expense of the linear region.

Step 9 uses NFS to fit the slope of the $\log(I_{ds})$ vs. V_{gs} curve. Often, the value of VT0 obtained from step 1 will prevent one from obtaining a good fit in the subthreshold region. If this happens, try fitting with VT0 and NFS, but replacing the final value of VT0 with that from step 1 at the end, keeping only NFS from this final step.

The preceding steps illustrate the concepts of fitting relevant subsets of parameters to relevant subsets of data to obtain physical values of the parameters, as well as fitting parameters in the proper order so that those obtained in the later steps will affect those obtained in earlier steps minimally. Please refer to Figures 8.19 and 8.20 for how the resulting fits typically appear (all graphs showing levels fits are provided by the device modeling software package Aurora, from Technology Modeling Associates, Inc., Palo Alto, CA).

An experienced person may notice that we have neglected some parameters. For example, we did not use parameters KP and GAMMA. This means KP will be calculated from U0, and GAMMA will be calculated from NSUB. In a sense U0 and NSUB are more fundamental parameters than KP and GAMMA. For example, KP depends on U0 and TOX; GAMMA depends on NSUB and TOX. If one is trying to obtain skew models, it is much more advantageous to analyze statistical distributions of

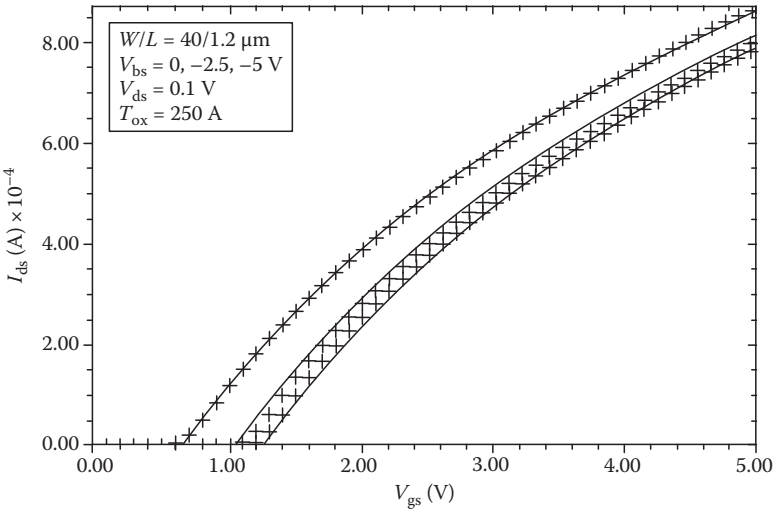


FIGURE 8.19 Typical MOS level 3 linear region measured and simulated plots at various V_{bs} values for a wide-short device.

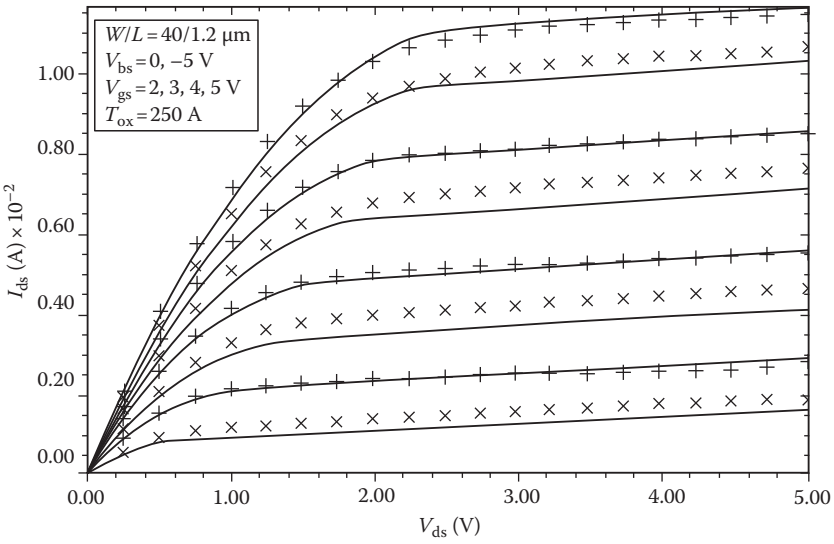


FIGURE 8.20 Typical MOS level 3 saturation region measured and simulated plots at various V_{gs} and V_{bs} values for a wide-short device.

parameters that depend on a single effect than those that depend on multiple effects. KP will depend on mobility and oxide thickness; U0 is therefore a more fundamental parameter. We also did not obtain parameter PHI, so it will be calculated from NSUB. The level 3 model is very insensitive to PHI, so using it for curve fitting is pointless. This illustrates the importance of being very familiar with the model equations. The kind of judgments described here cannot be made without such knowledge.

Test chip warnings. The following hints will greatly assist in properly performing device characterization:

1. Include a wide-long device; without this, the results will not be physically correct.
2. All MOS transistors with the same width should be drawn with their sources and drains identical. No difference should be seen in the number of source/drain contacts, contact spacing, source/drain contact overlap, poly gate to contact spacing, etc. In Figure 8.21, *c* is the contact size, *cs* is the contact space, *cov* is the contact overlap of diffusion, and *c2p* is the contact to poly spacing. All these dimensions should be identical for devices of different *L* but identical *W*. If not, extracting the series resistance will become more difficult.

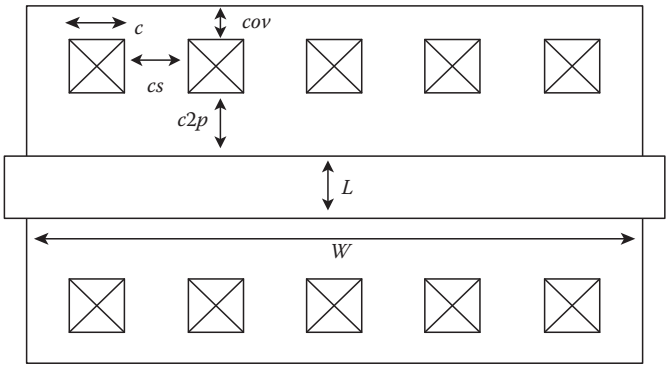


FIGURE 8.21 Mask layers showing dimensions that should be identical for all MOS devices of the same width.

3. Draw devices in pairs. That is, if the wide-long device is $W/L = 20/20$, make the wide-short device the same width as the wide-long device; e.g., make the short device 20/1, not 19/1. If the narrow-long device is 2/20, make the narrow-short device of the same width; i.e., make it 2/1, not 3/1, and similarly for the lengths. (Make the wide-short and the narrow-short devices have the same length.)
4. Draw structures for measuring the junction capacitances and overlap capacitances. These are discussed later in Section 8.2.5.

8.2.3 BSIM Extraction Strategy in Detail

All MOS model strategies have basic features in common; namely, fit the linear region at zero back-bias to get the basic zero-order parameters, fit the linear region at nonzero back-bias, fit the saturation region at zero back-bias, fit the saturation region at nonzero back-bias, and then fit the subthreshold region. It is possible to extend the type of strategy we covered for level 3 to the BSIM model, but that is not the way BSIM was intended to be used.

The triplet sets of parameters for incorporating geometry dependence into the BSIM model, Equation 8.37, allow an alternate strategy. We obtain sets of parameters without geometry dependence by fitting to individual devices without using the Par_L and Par_W terms. We do this for each device size individually. This produces sets of parameters relevant to each individual device. So, for device number 1 of width $W(1)$ and length $L(1)$ we would have a value for the parameter VFB which we will call VFB(1); for device number n of width $W(n)$ and length $L(n)$ we will have VFB(n). To get the Par_0, Par_L and Par_W terms, we fit to the parameters themselves, rather than the measured data. That is, to get VFB_0, VFB_L, and VFB_W, we fit to the "data points" VFB(1),...,VFB(n) with parameters VFB_0, VFB_L, and VFB_W using Equation 8.37, where L_{eff} and W_{eff} are different for each index, 1 through n.

Note that as L and W become very large, the parameters must approach Par_0. This suggests that we use the parameter values for the wide-long device as the Par_0 terms and only fit the other geometry sizes to get the Par_W and Par_L terms. For example, if we have obtained VFB(1) for our first device which is our wide-long device, we would set $\text{VFB}_0 = \text{VFB}(1)$, and then fit to VFB(2),...,VFB(n) with parameters VFB_L and VFB_W, and similarly for all the other triplets of parameters. In order to use a general least-squares optimization program in this way the software must be capable of specifying parameters as targets, as well as measured data points.

We now list a basic strategy for the BSIM model:

Step 1. Fit the wide-long device in the linear region at zero back-bias, at V_{gs} values above the subthreshold region, with parameters VFB (flatband voltage), MUZ (mobility), and U0 (mobility degradation), with DL (length encroachment) and DW (width encroachment) set to zero.

Step 2. Fit the wide-short device in the linear region at zero back-bias, at V_{gs} values above the subthreshold region, with parameters VFB, U0, and DL.

Step 3. Fit the narrow-long device in the linear region at zero back-bias, at V_{gs} values above the subthreshold region, with parameters VFB, U0, and DW.

Step 4. Refit the wide-long device in the linear region at zero back-bias, at V_{gs} values above the subthreshold region, with parameters VFB, MUZ, and U0, now that DL and DW are known.

Step 5. Fit the wide-short device in the linear region at zero back-bias, at V_{gs} values above the subthreshold region, with parameters VFB, RS, and RD. When finished, replace the value of VFB with the value found in step 4.

Step 6. Fit the wide-long device in the linear region at all back-biases, at V_{gs} values above the subthreshold region, with parameters K1 (first-order body effect), K2 (second-order body effect), U0, and X2U0 (V_{bs} dependence of U0).

Step 7. Fit the wide-long device in the saturation region at zero back-bias with parameters U0, ETA (V_{ds} dependence of threshold voltage), MUS (mobility in saturation), U1 (V_{ds} dependence of mobility), and X3MS (V_{ds} dependence cf MUS).

Step 8. Fit the wide-long device in the saturation region at all back-biases with parameter X2MS (V_{bs} dependence of MUS).

Step 9. Fit the wide-long device in the subthreshold region at zero back-bias and low V_{ds} value with parameter N0; then fit the subthreshold region nonzero back-bias low V_{ds} data with parameter NB; and finally fit the subthreshold region data at higher V_{ds} values with parameter ND. Or, fit all the subthreshold data simultaneously with parameters N0, NB, and ND.

Repeat steps 6 through 10 for all the other geometries, with the result of sets of geometry-independent parameters for each different size device. Then follow the procedure described previously for obtaining the geometry-dependent terms Par_0, Par_L, and Par_W.

In the preceding strategy, we have omitted various parameters either because they have minimal effect or because they have the wrong effect and were modified in the BSIM2 model. Because of the higher complexity of the BSIM model over the level 3 model, many more strategies are possible than the one just listed. One may be able to find variations of the above strategy that suit the individual technology better. Whatever modifications are made, the general spirit of the above strategy probably will remain.

Some prefer to use a more global approach with BSIM, fitting to measured data with Par_L and Par_W terms directly. Although this is certainly possible, it is definitely not a recommended approach. It represents the worst form of blind curve fitting, with no regard for physical correctness or understanding. The BSIM model was originally developed with the idea of obtaining the model parameters via extraction as opposed to optimization. In fact, UC Berkeley provides software for obtaining BSIM parameters using extraction algorithms, with no optimization at all. As stated previously, this has the advantage of being relatively fast and easy. Unfortunately, it does not always work. One of the major drawbacks of the BSIM model is that certain values of the parameters can cause the model to produce negative values of G_{ds} in saturation. This is highly undesirable, not only from a modeling standpoint, but also because of the convergence problems it can cause in circuit simulators. If an extraction strategy is used that does not guarantee nonnegative G_{ds}, very little can be done to fix the problem when G_{ds} becomes negative. Of course, the extraction algorithms can be modified, but this is difficult and time consuming. With optimization strategies, one can weight the fitting for G_{ds} more heavily and thus force the model to produce nonnegative G_{ds}. We, therefore, do not favor extraction strategies for BSIM, or anything else. As with most things in life, minimal effort provides minimal rewards.

8.2.3.1 BSIM2 Extraction Strategy

We do not cover the BSIM2 strategy in complete detail because it is very similar to the BSIM strategy, except more parameters are involved. The major difference in the two models is the inclusion of extra terms in BSIM2 for fitting G_{ds} (refer to Figure 8.22, which shows how badly BSIM typically fits $1/G_{ds}$ vs. V_{ds}). Basically, the BSIM2 strategy follows the BSIM strategy for the extraction of parameters not related to G_{ds}. Once these have been obtained, the last part of the strategy includes steps for fitting to G_{ds} with parameters that account for channel length modulation (CLM) and hot electron effects. The way this proceeds in BSIM2 is to fit I_{ds} first, and then parameters MU2, MU3, and MU4 are used to fit to $1/G_{ds}$ vs. V_{ds} curves for families of V_{ds} and V_{bs}. This can be a very time-consuming and frustrating experience, because fitting to $1/G_{ds}$ is quite difficult. Also, the equations describing how G_{ds} is modeled with MU2, MU3, and MU4 are very unphysical and the interplay between the parameters makes fitting awkward. The reader is referred to Figure 8.23, which shows how BSIM2 typically fits $1/G_{ds}$ vs. V_{ds}. BSIM2 is certainly better than BSIM, but it has its own problems fitting $1/G_{ds}$.

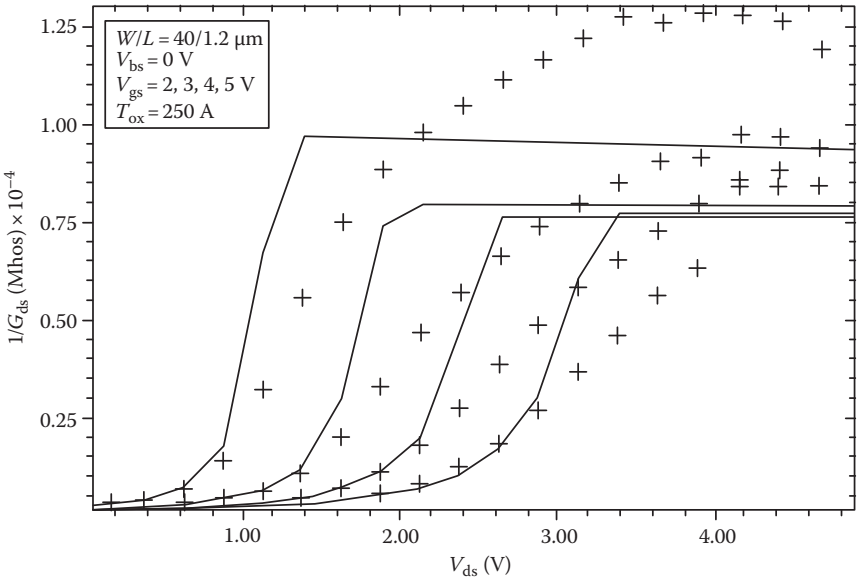


FIGURE 8.22 Typical BSIM $1/G_{ds}$ vs. V_{ds} measured and simulated plots at various V_{gs} values for a wide-short device.

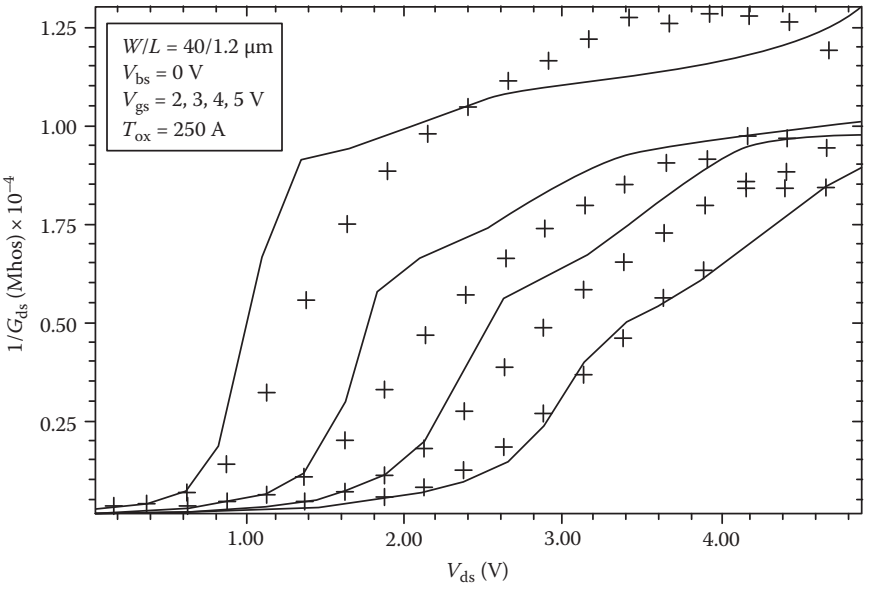


FIGURE 8.23 Typical BSIM2 $1/G_{ds}$ vs. V_{ds} measured and simulated plots at various V_{gs} values for a wide-short device.

8.2.3.2 BSIM3 Comments

The BSIM3 model is very new and will undoubtedly change in the future [5]. We will not list a BSIM3 strategy here, but focus instead on the features of the model that make it appealing for analog modeling.

BSIM3 has terms for fitting G_{ds} that relate to CLM, drain-induced barrier lowering (DIBL), and hot electron effects. They are incorporated completely differently from the G_{ds} fitting parameters of BSIM2.

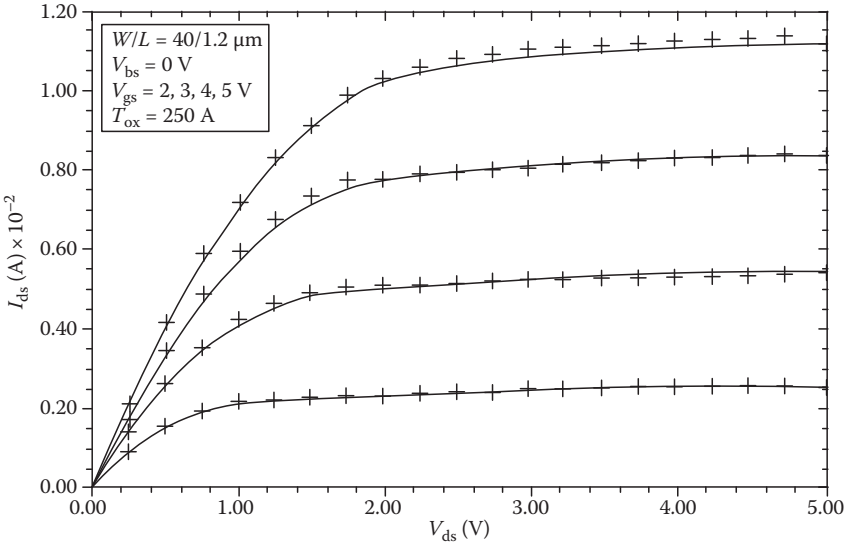


FIGURE 8.24 Typical BSIM3 saturation region measured and simulated plots at various V_{gs} values for a wide-short device.

In BSIM3, these parameters enter through a generalized Early voltage relation, with the drain current in saturation written as

$$I_{ds} = I_{d\,sat}\left[1 + \frac{(V_{ds} - V_{d\,sat})}{V_A}\right] \tag{8.38}$$

where V* is a generalized early voltage made up of three terms as

$$\frac{1}{V_A} = \frac{1}{V_{ACLM}} + \frac{1}{V_{ADIBL}} + \frac{1}{V_{AHCE}} \tag{8.39}$$

with the terms in Equation 8.39 representing generalized early voltages for CLM, DIBL, and hot carrier effects (HCE). This formulation is more physically appealing than the one used in BSIM2, making it easier to fit $1/G_{ds}$ vs. V_{ds}. curves with BSIM2. Figures 8.24 and 8.25 show how BSIM3 typically fits I_{ds} vs. V_{ds} and $1/G_{ds}$ vs. V_{ds}.

Most of the model parameters for BSIM3 have physical significance so they are obtained in the spirit of the parameters for the level 2 and 3 models. The incorporation of temperature dependence is also easier in BSIM3 because the parameters are more physical. All this, coupled with the fact that about 30 parameters exists for BSIM3 as compared to over 100 for BSIM2, makes BSIM3 a logical choice for analog design. However, BSIM3 is evolving, and shortcomings to the model may still exist that may be corrected in later revisions.

8.2.3.3 Which MOS Model to Use?

Many MOS models are available in circuit simulators, and the novice is bewildered as to which model is appropriate. No single answer exists, but some questions must be asked before making a choice:

1. What kind of technology am I characterizing?
2. How accurate a model do I need?

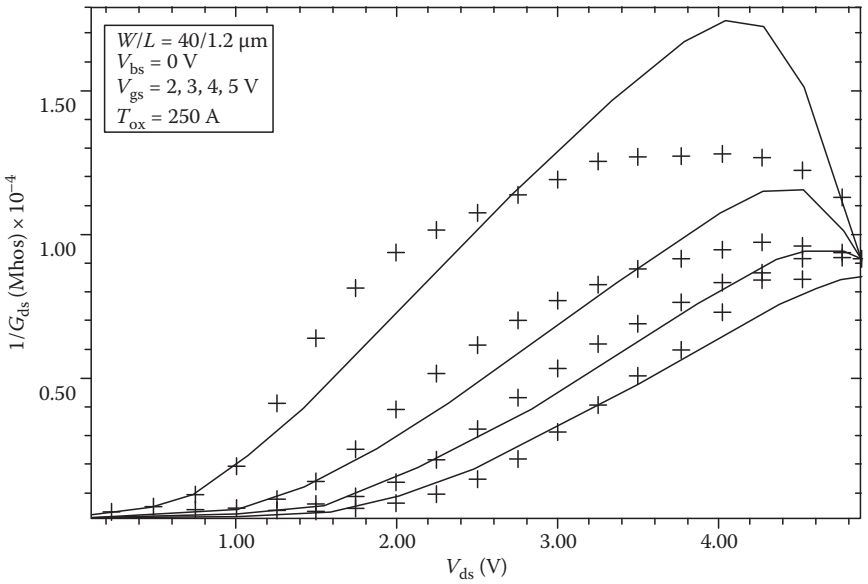


FIGURE 8.25 Typical BSIM3 $1/G_{ds}$ vs. V_{ds} measured and simulated plots at various V_{gs} values for a wide-short device.

3. Do I want to understand the technology?
4. How important are the skew model files (fast and slow parameter files)?
5. How experienced am I? Do I have the expertise to handle a more complicated model?
6. How much time can I spend doing device characterization?
7. Do I need to use this model in more than one circuit simulator?
8. Is the subthreshold region important?
9. Is fitting G_{ds} important?

Let us approach each question with regard to the models available. If the technology is not submicron, perhaps a simpler model such as level 3 is capable of doing everything needed. If the technology is deep submicron, then use a more complicated model such as BSIM, BSIM2, or BSIM3. If high accuracy is required, then the best choice is BSIM3, mainly because it is more physical than all the other models and is capable of fitting better.

For a good physical understanding of the process being characterized, BSIM and BSIM2 are not good choices. These are the least physically based of all the models. The level 2 and 3 models have good physical interpretation for most of the parameters, although they are relatively simple models. BSIM3 is also more physically based, with many more parameters than level 2 or 3, so it is probably the best choice.

If meaningful skew models need to be generated, then BSIM and BSIM2 are very difficult to use, again, because of their unphysical parameter sets. Usually, the simplest physically based model is the best for skew model generation. A more complicated physically based model such as BSIM3 may also be difficult to use for skew model generation.

If the user is inexperienced, none of the BSIM models should be used until the user's expertise improves. Our device is to practice using simpler models before tackling the harder ones.

If time is critical, the simpler models will definitely be much faster for use in characterization. The more complicated models require more measurements over wider ranges of voltages as well as wider ranges of geometries. This, coupled with the larger number of parameters, means they will take some time with which to work. The BSIM2 model will take longer than all the rest, especially if the G_{ds} fitting parameters are to be used.

The characterization results may need to be used in more than one circuit simulator. For example, if a foundry must supply models to various customers, they may be using different circuit simulators. In this case, proprietary models applicable to a single circuit simulator should not be used. Also, circuit designers may want to check the circuit simulation results on more than one circuit simulator. It is better to use standard Berkeley models (level 2, level 3, BSIM, BSIM2, and BSIM3) in such cases.

If the subthreshold region is important, then level 2 or level 3 cannot be used, and probably not even BSIM. BSIM2 or BSIM3 must be used instead. These two models have enough parameters for fitting the subthreshold region.

If fitting G_{ds} is important, BSIM2 and BSIM3 are, again, the only choices. None of the other models have enough parameters for fitting G_{ds}.

Finally, if a very unusual technology is to be characterized, none of the standard models may be appropriate. In this case commercially available specialized models or the user's own models must be used. This will be a large task, so the goals must justify the effort.

8.2.4 Bipolar DC Model

The standard bipolar model used by all circuit simulators is the Gummel–Poon model, often called the BJT (bipolar junction transistor) model [4]. Most circuit simulator vendors have added various extra parameters to this model, which we will not discuss, but they all have the basic parameters introduced by the original UC Berkeley SPICE model.

We now list a basic strategy for the BJT model:

Step 1. Fit a Gummel plot (log(IC) and log(IB) vs. VBE) for the IC curve in the straight line portion of the curve (low to middle VBE values) with parameters IS and NF. Note that it is best if the Gummel plots are done with VCB = 0.

Step 2. Fit plots of IC vs. VCE, stepping IB, in the high VCE region of the curves with parameters VAF, BF, and IKF.

Step 3. Fit plots of IC vs. VCE, stepping IB, in the low VCE region of the curves with parameter RC.

Step 4. Fit a Gummel plot for the IB curve for VBE values in the mid- to high-range with parameter BF.

Step 5. Fit a Gummel plot for the IB curve for low VBE values with parameters ISE and NE.

Step 6. Try to obtain parameter RE by direct measurement or by simulation using a device simulator such as PISCES. It is also best if RB can be obtained as a function of current by measurement on special structures, or by simulation. One may also obtain RB by *S*-parameter measurements (discussed later). Failing this, RB may be obtained by fitting to the Gummel plot for the IB curve for VBE values in the mid- to high-range with parameters RB, RBM, and IRB. This is a difficult step to perform with an optimization program.

Step 7. Fit a Gummel plot for the IC curve for high VBE values with parameter IKF.

Step 8. At this point, all the DC parameters have been obtained for the device in the forward direction. Next, obtain the reverse direction data by interchanging the emitter and collector to get the reverse Gummel and IC vs. VCE data.

Repeat Step 2 on the reverse data to get VAR, BR, IKR.
Repeat Step 4 on the reverse data to get BR.
Repeat Step 5 on the reverse data to get ISC, NC.
Repeat Step 7 on the reverse data to get IKR.

Fitting the reverse direction parameters is very frustrating because the BJT model is very poor in the reverse direction. After seeing how poor the fits can be, one may decide to obtain only a few of the reverse parameters.

8.2.5 MOS and Bipolar Capacitance Models

8.2.5.1 MOS Junction Capacitance Model

The MOS junction capacitance model accounts for bottom wall (area component) and sidewall (periphery component) effects. The total junction capacitance is written as a sum of the two terms

$$\mathrm{CJ(total)} = \frac{A(\mathrm{CJ})}{1 + [(VR/\mathrm{PB})]^{\mathrm{MJ}}} + \frac{P(\mathrm{CJSW})}{1 + [(VR/\mathrm{PB})]^{\mathrm{MJSW}}} \tag{8.40}$$

where

A is the area
P is the periphery of the junction diode capacitor
VR is the reverse bias (a positive number) across the device

The parameters of this model are CJ, CJSW, PB, MJ, and MJSW.

It is very easy to find parameter values for this model. Typically, one designs special large junction diodes on the test chip, large enough so that the capacitances being measured are many tens of picofarads. This means the diodes have typical dimensions of hundreds of microns. Two junction diodes are needed—one with a large area:periphery ratio, and another with a small area:periphery ratio. This is usually done by drawing a rectangular device for the large area:periphery ratio, and a multifingered device for the small area:periphery ratio (see Figure 8.26).

The strategy for this model consists of a single step, fitting to all the data simultaneously with all the parameters. Sometimes it is helpful to fix PB to some nominal value such as 0.7 or 0.8, rather than optimize on PB, because the model is not very sensitive to PB.

8.2.5.2 BJT Junction Capacitance Model

The BJT model is identical to the MOS, except it uses only the bottom wall (area term), not the sidewall. Three sets of BJT junction diodes need to be characterized: emitter–base, collector–base, and collector–substrate. Each of these has three model parameters to fit. For example, for the emitter–base junction diode, the parameters are CJE, VJE, and MJE (CJE corresponds to CJ, VJE to PB, and MJE to MJ of the MOS model). Similarly, for the collector–base junction diode, we have CJC, VJC, and MJC. Finally, for the collector–substrate junction diode, we have CJS, VJS, and MJS.

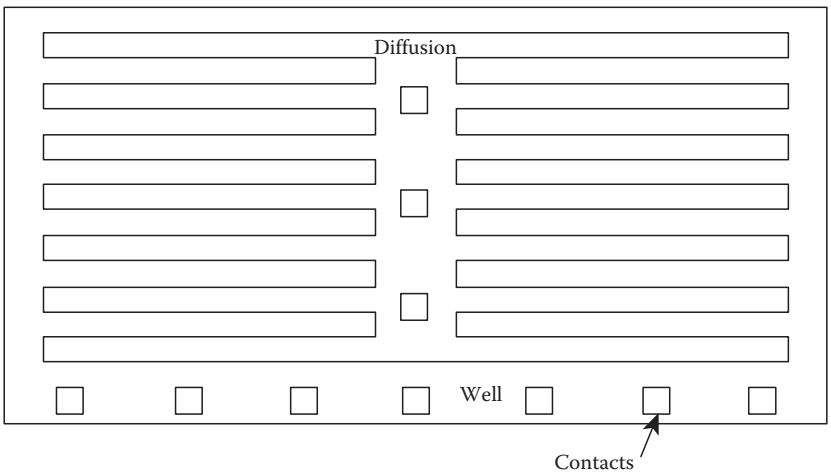


FIGURE 8.26 Mask layers showing the periphery multifingered junction diode structure.

These parameters are all fit in a single strategy step on the relevant data. Again, it is best if large test structures can be drawn rather than the values measured directly on transistor, because the transistor junction diodes have such small areas that the measurements are prone to accuracy problems.

8.2.5.3 MOS Overlap Capacitance Model

In addition to the junction capacitance model, the MOS models include terms for the overlap or Miller capacitance. This represents the capacitance associated with the poly gate overlap over the source and drain regions. As for the junction diodes, special test chip structures are necessary for obtaining the overlap capacitance terms CGS0, CGD0, and CGB0. These structures usually consist of large, multi-fingered devices that look just like the junction capacitance periphery diodes, except that where the junction diodes are diffusion over well structures, the overlap capacitors are poly over diffusion.

8.2.6 Bipolar High-Frequency Model

The SPICE BJT model has five parameters for describing the high-frequency behavior of bipolar devices. These are parameters TF, XTF, ITF, VTF, and PTF, all of which are associated with the base transit time, or equivalently, the cutoff frequency. Parameter TF is the forward transit time, XTF and ITF modify TF as a function of VBE (or IC), and VTF modifies TF as function of VBE (or IC), and VTF modifies TF as a function of VCB. Parameter PTF represents the excess phase at the cutoff frequency. The expression used by SPICE for the transit time is

$$\tau_f = \mathrm{TF}\left[1 + \mathrm{XTF}\left\{\frac{\mathrm{IF}}{\mathrm{IF} + \mathrm{ITF}}\right\}^2\right] e^{[\mathrm{VBC}/\{(1.44(\mathrm{VTF}))\}]} \tag{8.41}$$

where

$$\mathrm{IF} = \mathrm{IS}[e^{\{(q)(V_{\mathrm{be}})\}/\{(NF)(k)(T)\}} - 1] \tag{8.42}$$

SPICE uses the transit time to calculate the diffusion capacitance of the emitter–base junction in forward bias.

Obtaining the bipolar high frequency parameters is very complicated, time consuming, and difficult. It also requires very expensive measuring equipment, including a network analyzer for measuring *S*-parameters, and a quality high-frequency wafer prober with high-frequency probes. Having measured the *S*-parameters, after suitable data manipulation and optimization it is possible to find the high-frequency SPICE parameters.

It is beyond the scope of this book to cover high-frequency bipolar theory completely. We will, however, list the basic steps involved in obtaining the high frequency bipolar transit time parameters:

Step 1. Measure the *S*-parameters over an appropriate frequency range to go beyond the cutoff frequency, f_T. Do these measurements over a large range of IC values and stepping over families of VCB values as well. Be sure to include VCB = 0 in these data. Be sure that the network analyzer being used is well calibrated. These measurements can be done in either common emitter or common collector mode. Each has advantages and disadvantages.

Step 2. De-embed the *S*-parameter measurements by measurements dummy pad structures that duplicate the layout of the bipolar transistors, including pads and interconnect, but with no devices. The de-embedding procedure subtracts the effects of the pads and interconnects from the actual devices. This step is very important if reliable data are desired [10].

Step 3. Calculate the current gain, β from the *S*-parameter data. From plots of β vs. frequency, find the cutoff frequency, f_T. Also calculate parameter PTF directly from these data.

Step 4. Calculate τ_f from f_T by removing the effect of RE, RC, and the junction capacitances. This produces tables of τ_f vs. IC, or equivalently, τ_f vs. VBE, over families of VCB.

Step 5. Optimize on τ_f vs. VBE data over families of VCB with parameters TF, XTF, ITF, and VTF.

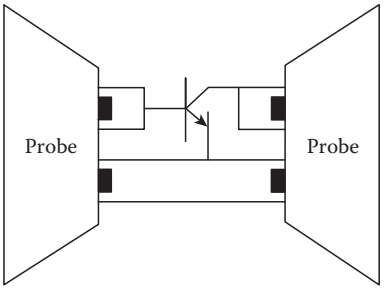


FIGURE 8.27 Typical bipolar transistor test chip layout for measuring *S*-parameters with high-frequency probes. This shows the common emitter configuration.

Note that many of the preceding steps are done automatically by some commercial device modeling programs.

It is very important to have well-designed and appropriate test structures for measuring *S*-parameters. The pad layout and spacing is very critical because of the special size and shape of the high-frequency probes (see Figure 8.27). The test chip structures must be layed out separately for common emitter and common collector modes if measurements are necessary for both these modes.

It is also possible to use the *S*-parameter measurements to obtain the base resistance as a function of collector current. From this information one can optimize to obtain parameters RB, RBM, and IRB. However, the base resistance obtained from *S*-parameters is not a true DC base resistance, and the model is expecting DC values of base resistance.

8.2.7 Miscellaneous Topics

8.2.7.1 Skew Parameter Files

This chapter discusses obtaining model parameters for a single wafer, usually one that has been chosen to represent a typical wafer for the technology being characterized. The parameter values obtained from this wafer correspond to a typical case. Circuit designers also want to simulate circuits with parameter values representing the extremes of process variation, the so-called fast and slow corners, or skew parameter files. These represent the best and worst case of the process variation over time.

Skew parameter values are obtained usually by tracking a few key parameters, measuring many wafers over a long period of time. The standard deviation of these key parameters is found and added to or subtracted from the typical parameter values to obtain the skew models. This method is extremely crude and will not normally produce a realistic skew model. It will almost always overestimate the process spread, because the various model parameters are not independent—they are correlated.

Obtaining realistic skew parameter values, taking into account all the subtle correlations between parameters, is more difficult. In fact, skew model generation is often more an art than a science. Many attempts have been made to utilize techniques from a branch of statistics called multivariate analysis [1]. In this approach, principal component or factor analysis is used to find parameters that are linear combinations of the original parameters. Only the first few of these new parameters will be kept; the others will be discarded because they have less significance. This new set will have fewer parameters than the original set and therefore will be more manageable in terms of finding their skews. The user sometimes must make many choices in the way the common factors are utilized, resulting in different users obtaining different results.

Unfortunately, a great deal of physical intuition is often required to use this approach effectively. To date, we have only seen it applied to the simpler MOS models such as level 3. It is not known if this is a viable approach for a much more complicated model such as BSIM2 [7].

8.2.7.2 Macro Modeling

Sometimes, a device is designed and manufactured that cannot be modeled by a single transistor model. In such cases, one may try to simulate the behavior of a single device by using many basic device elements together, representing the final device. For example, a single real bipolar device might be modeled using a combination of several ideal bipolar devices, diodes, resistors, and capacitors. This would be a macro model representing the real bipolar device.

Macro modeling usually uses least-squares optimization techniques. One chooses model parameter values to represent the ideal devices in the technology, and then optimizes on geometry size, capacitance size, resistance size, etc., to obtain the final macro model. This is similar to the optimization techniques used to find model parameters for a standard model, but in this case we are calling on an entire circuit simulator rather than a model subroutine to provide the output characteristics of our macro model.

Obviously, macro modeling can be very computation intensive. A complex macro model can take a very long time to optimize. Also, it may be impractical to use a macro model for every real device in the circuit. For example, if the circuit in question has 100,000 real transistors and each of these is being represented by a macro model with 10 components, the number of circuit nodes introduced by the macro model ($>10 \times 10{,}000$) might be prohibitive in terms of simulation time. Nevertheless, for critical components, macro modeling can provide tremendous insight.

8.2.7.3 Modeling in a TCAD Environment

Programs that do optimization are part of a larger set of software that is sometimes called TCAD (technology computer-aided design). Typically, TCAD encompasses software for process simulation (doping distributions), device simulation (electrical characteristics), device modeling, and circuit simulation. Other elements of TCAD can include lithography simulation, interconnect modeling (capacitance, resistance, and inductance), etc.

In the past, each of the software components of TCAD were used as stand-alone programs, with very little communication between the components. The trend now is to incorporate all these pieces into a complete environment that allows them to communicate with each other seamlessly, efficiently, and graphically, so that the user is not burdened with keeping track of file names, data, etc. In such a system, one can easily set up split lots and simulate from process steps up through circuit simulation. One will also be able to close the loop, feeding circuit simulation results back to the process steps and run optimization on an entire process to obtain optimal circuit performance characteristics (see Figure 8.28).

The future of device modeling will surely be pushed in this direction. Device modeling tools will be used not only separately, but within a total TCAD environment, intimately connected with process, device, interconnect, and circuit simulation tools.

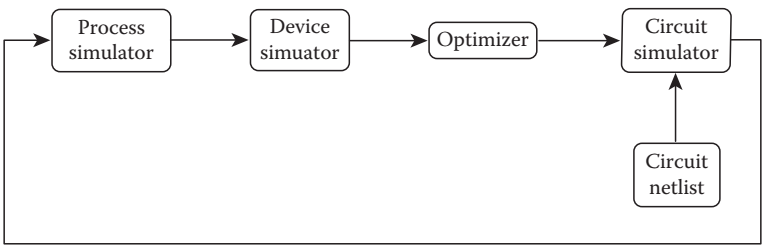


FIGURE 8.28 Block diagram of a TCAD system.

References

1. W. R. Dillon and M. Goldstein, *Multivariate Analysis Methods and Applications*, New York: John Wiley & Sons, 1984.
2. P. E. Gill, W. Murray, and M. Wright, *Practical Optimization*, Orlando, FL: Academic Press, 1981.
3. J. S. Duster, J.-C. Jeng, P. K. Ko, and C. Hu, User's guide for BSIM2 parameter extraction program and the SPICE3 with BSIM implementation, Electronic Research Laboratory, Berkeley, CA: University of California, 1988.
4. I. Getreu, *Modeling the Bipolar Transistor*, Beaverton, OR: Tektronix, 1976.
5. J.-H. Huang, Z. H. Liu, M.-C. Jeng, P. K. Ko, and C. Hu, BSIM3 manual, Berkeley, CA: University of California, 1993.
6. M.-C. Jeng, P. M. Lee, M. M. Kuo, P. K. Ko, and C. Hu, Theory, algorithms, and user's guide for BSIM and SCALP, Version 2.0, Electronic Research Laboratory, Berkeley, CA: University of California, 1987.
7. J. A. Power, A. Mathewson, and W. A. Lane, An approach for relating model parameter variabilities to process fluctuations, *Proc. IEEE Int. Conf. Microelectronic Test Struct.*, vol. 6, Mar. Barcelona, Spain, 1993.
8. W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C*, Cambridge, U.K.: Cambridge University Press, 1988.
9. B. J. Sheu, D. L. Scharfetter, P. K. Ko, and M.-C. Jeng, BSIM: Berkeley Short-Channel IGFET Model for MOS Transistors, *IEEE J. Solid-State Circuits*, SC-22(4), Aug. 1987.
10. P. J. van Wijnen, *On the Characterization and Optimization of High-Speed Silicon Bipolar Transistors*, Beaverton, OR: Cascade Microtech, Inc., 1991.
11. A. Vladimirescu and S. Liu, The simulation of MOS integrated circuits using SPICE2, memorandum no. UCB/ERL M80/7, Berkeley, CA: University of California, 1980.

Other recommended publications which are useful in device characterization are

L. W. Nagel. SPICE2: A computer program to simulate semiconductor circuits, memorandum no. ERL-M520, Berkeley, CA: University of California, 1975.

G. Massobrio and P. Antognetti, *Semiconductor Device Modeling with SPICE*, New York: McGraw-Hill, 1993.

9

Analog Circuit Simulation

9.1 Introduction 9-1
9.2 Purpose of Simulation 9-1
9.3 Netlists 9-2
9.4 Formulation of the Circuit Equations 9-3
9.5 Modified Nodal Analysis 9-4
9.6 Active Device Models 9-5
9.7 Types of Analysis 9-7
DC (Steady-State) Analysis • AC Analysis • Transient Analysis
9.8 Verilog-A 9-13
9.9 Fast Simulation Methods 9-18
9.10 Commercially Available Simulators 9-19
References 9-19

J. Gregory Rollins
Technical Modeling Associates, Inc.

9.1 Introduction

Analog circuit simulation usually means simulation of analog circuits or very detailed simulation of digital circuits. The most widely known and used circuit simulation program is SPICE (simulation program with integrated circuit emphasis) of which it is estimated that there are over 100,000 copies in use. SPICE was first written at the University of California at Berkeley in 1975, and was based on the combined work of many researchers over a number of years. Research in the area of circuit simulation continues at many universities and industrial sites. Commercial versions of SPICE or related programs are available on a wide variety of computing platforms, from small personal computers to large mainframes. A list of some commercial simulator vendors can be found in the Appendix. The focus of this chapter is the simulators and the theory behind them. Examples are also given to illustrate their use.

9.2 Purpose of Simulation

Computer-aided simulation is a powerful aid for the design or analysis of VLSI circuits. Here, the main emphasis will be on analog circuits; however, the same simulation techniques may be applied to digital circuits, which are composed of analog circuits. The main limitation will be the size of these circuits because the techniques presented here provide a very detailed analysis of the circuit in question and, therefore, would be too costly in terms of computer resources to analyze a large digital system. However, some of the techniques used to analyze digital systems (like iterated timing analysis or relaxation methods) are closely related to the methods used in SPICE.

It is possible to simulate almost any type of circuit SPICE. The programs have built-in elements for resistors, capacitors, inductors, dependent and independent voltage and current sources, diodes, MOSFETs, JFETs, BJTs, transmission lines, and transformers. Commercial versions have libraries of standard components, which have all necessary parameters prefitted to typical specifications. These libraries include items such as discrete transistors, op-amps, phase-locked loops, voltage regulators, logic integrated circuits, and saturating transformer cores. Versions are also available that allow the inclusion of digital models (mixed-mode simulation) or behavioral models that allow the easy modeling of mathematical equations and relations.

Computer-aided circuit simulation is now considered an essential step in the design of modern integrated circuits. Without simulation, the number of ''trial runs'' necessary to produce a working IC would greatly increase the cost of the IC and the critical time-to-market. Simulation provides other advantages, including

- The ability to measure ''inaccessible'' voltages and currents that are buried inside a tiny chip or inside a single transistor.
- No loading problems are associated with placing a voltmeter or oscilloscope in the middle of the circuit, measuring difficult one-shot waveforms or probing a microscopic die.
- Mathematically ideal elements are available. Creating an ideal voltage or current source is trivial with a simulator, but impossible in the laboratory. In addition, all component values are exact and no parasitic elements exist.
- It is easy to change the values of components or the configuration of the circuit.

Unfortunately, computer-aided simulation has it own set of problems, including

- Real circuits are distributed systems, not the ''lumped element models'' which are assumed by simulators. Real circuits, therefore, have resistive, capacitive, and inductive parasitic elements present in addition to the intended components. In high-speed circuits, these parasitic elements can be the dominant performance-limiting elements in the circuit, and they must be painstakingly modeled.
- Suitable predefined numerical models have not yet been developed for certain types of devices or electrical phenomena. The software user may be required, therefore, to create his or her own models out of other models that are available in the simulator. (An example is the solid-state thyristor, which may be created from an npn and pnp bipolar transistor.)
- The numerical methods used may place constraints on the form of the model equations used. In addition, convergence difficulties can arise, making the simulators difficult to use.
- There are small errors associated with the solution of the equations and other errors in fitting the nonlinear models to the transistors that make up the circuit.

9.3 Netlists

Before simulating, a circuit must be coded into a netlist. Figure 9.1 shows the circuit for a simple differential pair. Circuit nodes are formed wherever two or more elements meet. This particular circuit has seven nodes, which are numbered 0–6. The ground or datum node is traditionally numbered as zero. The circuit elements (or branches) connect the nodes.

The netlist provides a description of the topography of a circuit and is simply a list of the branches (or elements) that make up the circuit. Typically, the elements may be entered in any order and each has a unique name, a list of nodes, and either a value or model identifier. For the differential amplifier of Figure 9.1, the netlist is shown in Figure 9.2.

The first line gives the title of the circuit (and is required in many simulators). The next three lines define the three voltage sources. The letter V at the beginning tells SPICE that this is a voltage source element. The list of nodes (two in this case) is next followed by the value in volts. The syntax for the resistor is similar to that of the voltage source; the starting letter R in the names of the resistors tells SPICE that these

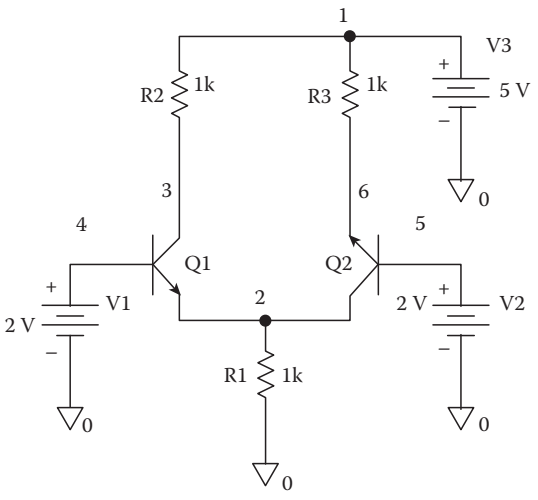


FIGURE 9.1 Circuit for differential pair.

are resistors. SPICE also understands that the abbreviation ''k'' after a value means 1000. For the two transistors Q1 and Q2, the starting letter Q indicates a bipolar transistor. Q1 and Q2 each have three nodes and in SPICE, the convention for their ordering is collector, base, and emitter. So, for Q1, the collector is connected to node 3, the base to node 4, and the emitter to node 2. The final entry ''m2n2222'' is a reference to the model for the bipolar transistor (note that both Q1 and Q2 reference the same model). The ''.model'' statement at the end of the listing defines this model. The model type is npn (for an npn bipolar junction transistor), and a list of ''parameter = value'' entries follow. These entries define the numerical values of constants in the mathematical models which are used for the

```
Differential pair circuit
V1 4 0 2V
V2 5 0 2V
V3 1 0 5V
R1 2 0 1k
R2 3 1 1K
R3 6 1 1K
Q1 3 4 2 m2n2222
Q2 6 5 2 mq2n2222
.model m2n2222 NPN IS = 1e-12 BF = 100 BR = 5 TF = 100pS
```

FIGURE 9.2 Netlist for differential pair.

bipolar transistor. (Models will be discussed later in more detail.) Most commercial circuit simulation packages come with ''schematic capture'' software that allows the user to draw the circuit by placing and connecting the elements with the mouse.

9.4 Formulation of the Circuit Equations

In SPICE, the circuits are represented by a system of ordinary differential equations. These equations are then solved using several different numerical techniques. The equations are constructed using Kirchhoff's voltage and current laws (KVL and KCL). The first system of equations pertains to the currents flowing into each node. One equation is written for each node in the circuit (except for the ground node), so the following equation is really a system of N equations for the N nodes in the circuit. The subscript i denotes the node index.

$$0 = F_i(V) = G_i(V) + \frac{\partial Q_i(V)}{\partial t} + W_i(t) \tag{9.1}$$

where

V is an N-dimensional vector that represents the voltages at the nodes

Q is another vector which represents the electrical charge (in Coulombs) at each node

The term W represents any independent current sources that may be attached to the nodes and has units of amperes

The function $G(V)$ represents the currents that flow into the nodes as a result of the voltages V

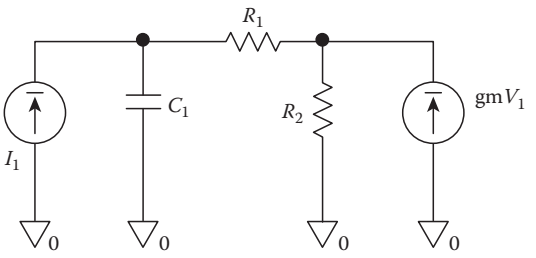


FIGURE 9.3 Example circuit for nodal analysis.

If the equations are formulated properly, a system of N equations in N unknown results.

For example, for the circuit of Figure 9.3 which has two nodes, we need to write two equations. At node 1:

$$0 = (V_1 - V_2)/R_1 + \frac{\mathrm{d}(C_1 V_1)}{\mathrm{d}t} + I_1 \qquad (9.2)$$

We can identify $G(V)$ as $(V_1 - V_2)/R$, the term $Q(V)$ is $C_1 V_1$ and $W(t)$ is simply I_1. Likewise at node 2:

$$0 = (V_2 - V_1)/R_1 + V_2/R_2 + gmV_1 \qquad (9.3)$$

In this example, G and Q are simple linear terms; however, in general, they can be nonlinear functions of the voltage vector V.

9.5 Modified Nodal Analysis

Normal nodal analysis that uses only KCL cannot be used to represent ideal voltage sources or inductors. This is so because the branch current in these elements cannot be expressed as a function of the branch voltage. To resolve this problem, KVL is used to write a loop equation around each inductor or voltage source. Consider Figure 9.4 for an example of this procedure. The unknowns to be solved for are the voltage V_1 at node 1, V_2 the voltage at node 2, V_3 the voltage at node 3, the current flowing through voltage source V_1 which we shall call I_x, and the current flowing in the inductor L_1 which we shall call I_1 The system of equations is

$$\begin{aligned} 0 &= V_1/R_1 + I_x \\ 0 &= V_2/R_2 - I_z + I_l \\ 0 &= V_3/R_3 - I_l \\ 0 &= V_1 - V_x + V_2 \\ 0 &= V_2 + \frac{\mathrm{d}(L_1 I_1)}{\mathrm{d}t} - V_3 \end{aligned} \qquad (9.4)$$

The use of modified nodal analysis does have the disadvantage of requiring that an additional equation be included for each inductor or voltage source, but has the advantage that ideal voltage sources can be used.

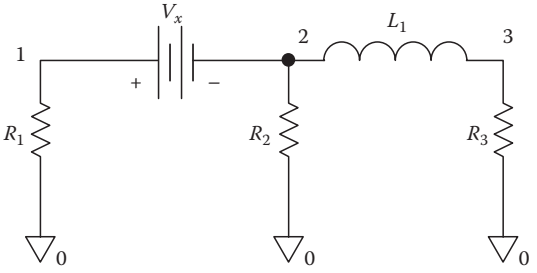


FIGURE 9.4 Circuit for modified nodal analysis.

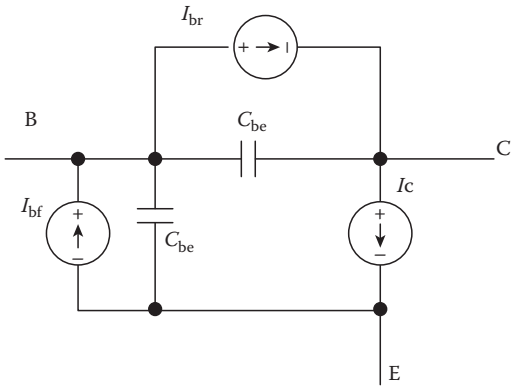


FIGURE 9.5 Ebers–Moll model for the bipolar transistor.

The total number of equations to be solved is therefore the number of nodes plus the number of voltages sources and inductors.

9.6 Active Device Models

VLSI circuits contain active devices like transistors or diodes, which act as amplifiers. These devices are normally described by a complicated set of nonlinear equations. We shall consider a simple model for the bipolar transistor—the Ebers–Moll model. This model is one of the first developed, and while it is too simple for practical application, it is useful for discussion.

A schematic of the Ebers–Moll model is shown in Figure 9.5. The model contains three nonlinear voltage-dependent current sources I_c, I_{bf}, and I_{br} and two nonlinear capacitances C_{be} and C_{bc}. The current flowing in the three current sources are given by the following equations:

$$I_c = I_s(\exp(V_{be}/V_t)) - \exp(V_{ce}/V_t) \tag{9.5}$$

$$I_{bf} = \frac{I_s}{B_f}(\exp(V_{be}/V_t) - 1) \tag{9.6}$$

$$I_{br} = \frac{I_s}{B_r}(\exp(V_{bc}/V_t) - 1) \tag{9.7}$$

The voltages V_{be} and V_{bc} are the voltages between base and emitter and the base and collector, respectively. I_s, B_f, and B_r are three user-defined parameters that govern the DC operation of the BJT. V_t is the "thermal voltage" or kT/q, which has the numerical value of approximately 0.26 V at room temperature. Observe that in the normal forward active mode, where $V_{be} > 0$ and $V_{ce} < 0$, I_{br} and the second term in I_c vanish and the current gain of the BJT, which is defined as I_c/I_b becomes numerically equal to B_f. Likewise, in the reverse mode where $V_{ce} > 0$ and $V_{be} < 0$, the reverse gain (I_e/I_b) is equal to B_r.

The two capacitances in Figure 9.5 contribute charge to the emitter, base, and collector, and this charge is given by the following equations:

$$Q_{be} = \tau_f I_s(\exp V_{be}/V_t - 1) + C_{je}\int_0^{V_{be}} (1 - V/V_{je})^{-m_e} \tag{9.8}$$

$$Q_{bc} = \tau_r I_s(\exp V_{bc}/V_t - 1) + C_{jc}\int_0^{V_{bc}} (1 - V/V_{jc})^{-m_c} \tag{9.9}$$

Q_{be} contributes positive charge to the base and negative charge to the emitter. Q_{bc} contributes positive charge to the base and negative charge to the collector. The first term in each charge expression is due to charge injected into the base from the emitter for Q_{be} and from the collector into the base for Q_{bc}. Observe that the exponential terms in the charge terms are identical to the term in I_c. This is so because the injected charge is proportional to the current flowing into the transistor. The terms τ_f and τ_r are the forward and reverse transit times, respectively, and correspond to the amount of time it takes the electrons (or holes) to cross the base. The second term in the charge expression (the term with the integral)

corresponds to the charge in the depletion region of the base–emitter junction for Q_{be} and in the base–collector junction for Q_{bc}. Recall that the depletion width in a pn junction is a function of the applied voltage. The terms V_{je} and V_{jc} are the ''built-in'' potentials with units of volts for the base–emitter and base–collector junctions. The terms m_c and m_e are the grading coefficients for the two junctions and are related to how rapidly the material changes from n-type to p-type across the junction.

This ''simple'' model has eleven constants I_s, B_f, B_r, C_{je}, C_{jc}, M_e, M_c, V_{je}, V_{jc}, T_f, and T_r that must be specified by the user. Typically, these constants would be extracted from measured I–V and C–V data taken from real transistors using a fitting or optimization procedure (typically a nonlinear least-squares fitting method is needed). The Ebers–Moll model has a number of shortcomings that are addressed in newer models like Gummel–Poon, Mextram, and VBIC. The Gummel–Poon model has over 40 parameters that must be adjusted to get a good fit to data in all regions of operation.

Models for MOS devices are even more complicated than the bipolar models. Modeling the MOSFET is more difficult than the bipolar transistor because it is often necessary to use a different equation for each of the four regions of operation (off, subthreshold, linear, and saturation) and the drain current and capacitance are functions of three voltages (V_{ds}, V_{bs}, and V_{gs}) rather than just two (V_{be} and V_{ce}) as in the case of the BJT. If a Newton–Raphson solver is to be used, the I–V characteristics and capacitances must be continuous and it is best if their first derivatives are continuous as well. Furthermore, MOS models contain the width (W) and length (L) of the MOSFET channel as parameters; and for the best utility the model should remain accurate for many values of W and L. This property is referred to as ''scalability.''

Over the years, literally hundreds of different MOS models have been developed. However, for modern VLSI devices, only three or four are commonly used today. These are the SPICE Level-3 MOS model, the HSPICE Level-28 model (which is a proprietary model developed by Meta Software), the public domain BSIM3 model developed at UC, Berkeley, and MOS9 developed at Phillips. These models are supported by many of the ''silicon foundries,'' that is, parameters for the models are provided to chip designers by the foundries. BSIM3 has been observed to provide a good fit to measured data and its I–V curves to be smooth and continuous (thereby resulting in good simulator convergence). The main drawback of BSIM3 is that it has over 100 parameters which are related in intricate ways, making extraction of the parameter set a difficult process.

A process known as ''binning'' is used to provide greater accuracy. When binning is used, a different set of model parameters is used for each range of the channel length and width (L and W). An example of this is shown in Figure 9.6. For a given type of MOSFET, 12 complete sets of model parameters are extracted and each is valid for a given range. For example, in Figure 9.6, the set represented by the number ''11'' would only be valid for channel lengths between 0.8 and 2.0 μm and for channel widths between 0.5 and 0.8 μm. Thus, for a typical BSIM3 model with about 60 parameters, $12 \times 60 = 720$ parameters would need to be extracted in all and this just for one type of device.

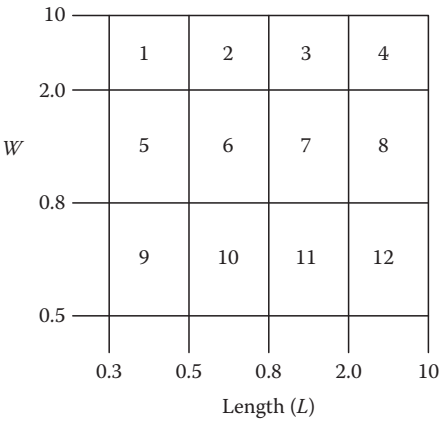


FIGURE 9.6 Binning of MOS parameters.

Many commercial simulators contain other types of models besides the traditional R, L, C, MOS, and BJT devices. Some simulators contain ''behavioral'' models which are useful for systems design or integration tasks; examples of these are integrators, multipliers, summation, and Laplace operator blocks. Some simulators are provided with libraries of prefitted models for commercially available operational amplifiers, logic chips, and discrete devices. Some programs allow ''mixed-mode'' simulation, which is a combination of logic simulation (which normally allows only a few discrete voltage states) and analog circuit simulation.

9.7 Types of Analysis

For analog circuits, three methods of analysis are commonly used: DC, AC, and transient analysis. DC analysis is used to examine the steady-state operation of a circuit; that is, what the circuit voltages and currents would be if all inputs were held constant for an infinite time. AC analysis (or sinusoidal steady state) examines circuit performance in the frequency domain using phasor analysis. Transient analysis is performed in the time domain and is the most powerful and computationally intensive of the three. For special applications, other methods of analysis are available such as the harmonic-balance method, which is useful for detailed analysis of nonlinear effects in circuits excited by purely periodic signals (like mixers and RF amplifiers).

9.7.1 DC (Steady-State) Analysis

DC analysis calculates the steady-state response of a circuit (with all inductors shorted and capacitors removed). DC analysis is used to determine the operating point (*Q*-point) of a circuit, power consumption, regulation and output voltage of power supplies, transfer functions, noise margin and fanout in logic gates, and many other types of analysis. In addition, a DC solution must be calculated to find the starting point for AC and transient analysis.

To calculate the DC solution, we need to solve Kirchhoff's equations formulated earlier. Unfortunately, since the circuit elements will be nonlinear in most cases, a system of transcendental equations will normally result and it is impossible to solve this system analytically. The method that has met with the most success is Newton's method or one of its derivatives.

9.7.1.1 Newton's Method

Newton's method is actually quite simple. We need is to solve the system of equations $F(X) = 0$ for X, where both F and X are vectors of dimension N. (F is the system of equations from modified nodal analysis, and X is the vector of voltages and current that we are solving for.) Newton's method states that given an initial guess for X^i, we can obtain a better guess X^{i+1} from the equation:

$$X^{i+1} = X^i - [J(X^i)]^{-1} F(X^i) \tag{9.10}$$

Note that all terms on the right side of the equation are functions only of the vector X^i. The term $J(X)$ is a $N \times N$ square matrix of partial derivatives of F, called the Jacobian. Each term in J is given by

$$J_{i,j} = \frac{\partial F_i(X)}{\partial X_j} \tag{9.11}$$

We assemble the Jacobian matrix for the circuit at the same time that we assemble the circuit equations. Analytic derivatives are used in most simulators.

The -1 in Equation 9.10 indicates that we need to invert the Jacobian matrix before multiplying by the vector F. Of course, we do not need to actually invert J to solve the problem; we only need to solve the linear problem $F = YJ$ for the vector Y and then calculate $X^{i+1} = X^i - Y$. A direct method such as the LU decomposition is usually employed to solve the linear system.

For the small circuit of Figure 9.3, analyzed in steady state (without the capacitor), the Jacobian entries are

$$\begin{array}{ll} J_{1,1} = 1/R_1 & J_{1,2} = -1/R_1 \\ J_{2,1} = 1/R_1 + gm & J_{2,2} = 1/R_1 + 1/R_2 \end{array} \tag{9.12}$$

```
Stead state analysis of differential pair.
V1 4 0 2V
V2 5 0 2V
V3 1 0 5V
R1 2 0 1k
R2 3 1 1K
R3 6 1 1K
Q1 3 4 2 m2n2222
Q2 6 5 2 m2n2222
.model m2n2222 NPN IS = 1e-12 BF = 100 BR = 5 TF = 100pS
.dc V1  1.0  3.0  0.01
```

FIGURE 9.7 Input file for DC sweep of V_1.

For a passive circuit (i.e., a circuit without gain), the Jacobian will be symmetric and for any row, the diagonal entry will be greater than the sum of all the other entries.

Newton's method converges quadratically, provided that the initial guess X^i is sufficiently close to the true solution. Quadratically implies that if the distance between X^i and the true solution is d, then the distance between X^{i+1} and the true solution will be d^2. Of course, we are assuming that d is small to start with. Still, programs like SPICE may require 50 or more iterations to achieve convergence. The reason for this is that, often times, the initial guess is poor and quadratic convergence is not obtained until the last few iterations. There are additional complications like the fact that the model equations can become invalid for certain voltages. For example, the BJT model will "explode" if a junction is forward-biased by more than 1 V or so since: $\exp(1/Vt) = 5e16$. Special limiting or damping methods must be used to keep the voltages and currents to within reasonable limits.

9.7.1.2 Example Simulation

Most circuit simulators allow the user to ramp one or more voltage sources and plot the voltage at any node or the current in certain branches. Returning to the differential pair of Figure 9.1, we can perform a DC analysis by simply adding a .dc statement (see Figure 9.7). A plot of the differential output voltage (between the two collectors) and the voltage at the two emitters is shown in Figure 9.8. Observe that the output voltage is zero when the differential pair is "balanced" with 2.0 V on both inputs. The output saturates at both high and low values for V_1, illustrating the nonlinear nature of the analysis. This simulation was run using the PSPICE package from MicroSim Corporation. The simulation run is a few seconds on a 486 type PC.

9.7.2 AC Analysis

AC analysis is performed in the frequency domain under the assumption that all signals are represented as a DC component V_{dc} plus a small sinusoidal component V_{ac}

$$V = V_{dc} + V_{ac} \exp(j\omega t) \tag{9.13}$$

where

$j = \sqrt{-1}$

ω is the radial frequency $(2\pi f)$

V_{ac} is a complex number

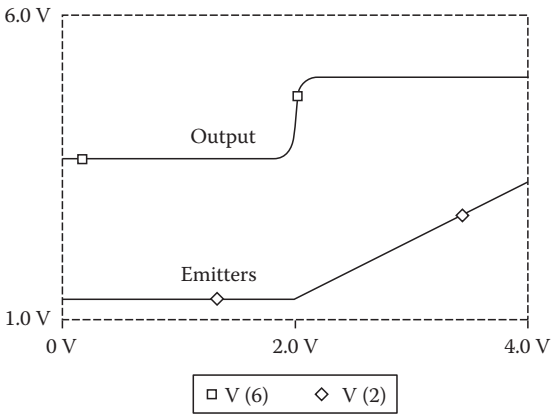


FIGURE 9.8 Output from DC analysis.

Expanding Equation 9.1 about the DC bias point V_{dc} (also referred to as the Q point), we obtain

$$F(V) = F(V_{dc}) + W_{dc} + W_{ac} + \frac{\partial G(V_{dc})}{\partial V_{dc}} V_{ac} + \frac{\partial}{\partial t}\left(\frac{\partial Q(V_{dc})}{\partial V_{dc}}\right) V_{ac} + \alpha V_{ac}^2 \Lambda \tag{9.14}$$

The series has an infinite number of terms; however, we assume that if V_{ac} is sufficiently small, all terms above first order can be neglected. The first two terms on the right-hand side are the DC solution and, when taken together, yield zero. The third term W_{ac} is the vector of independent AC sources which drive the circuit. The partial derivative in the fourth term is the Jacobian element, and the derivative of Q in parentheses is the capacitance at the node. When we substitute the exponential into Equation 9.14, each term will have an exponential term that can be canceled. The result of all these simplifications is the familiar result:

$$0 = W_{ac} + JV_{ac} + j\omega C V_{ac} \tag{9.15}$$

This equation contains only linear terms which are equal to the partial derivatives of the original problem evaluated at the Q point. Therefore, before we can solve the AC problem, we must calculate the DC bias point. Rearranging terms slightly, we obtain

$$V_{ac} = -(J + j\omega C)^{-1} W_{ac} \tag{9.16}$$

The solution at a given frequency can be obtained from a single matrix inversion. The matrix, however, is complex but normally the complex terms share a sparsity pattern similar to the real terms. It is normally possible (in FORTRAN and C++) to create a suitable linear solver by taking the linear solver which is used to calculate the DC solution and substituting ''complex'' variables for ''real'' variables. Since there is no nonlinear iteration, there are no convergence problems and AC analysis is straightforward and foolproof.

The same type of analysis can be applied to the equations for modified nodal analysis. The unknowns will of course be currents and the driving sources voltage sources.

$$I_{ac} = -(J + j\omega L)^{-1} E_{ac} \tag{9.17}$$

The only things that must be remembered with AC analysis are

1. AC solution is sensitive to the Q point, so if an amplifier is biased near its saturated DC output level, the AC gain will be smaller than if the amplifier were biased near the center of its range.
2. This is a linear analysis and therefore ''clipping'' and slew rate effects are not modeled. For example, if a 1 V AC signal is applied to the input of a small signal amplifier with a gain of 100 and a power supply voltage of 5 V, AC analysis will predict an output voltage of 100 V. This is of course impossible since the output voltage cannot exceed the power supply voltage of 5 V. If you want to include these effects, use transient analysis.

9.7.2.1 AC Analysis Example

In the following example, we will analyze the differential pair using AC analysis to determine its frequency response. To perform this analysis in SPICE, we need to only specify which sources are the AC-driving sources (by adding the magnitude of the AC signal at the end) and specify the frequency range on the .AC statement (see Figure 9.9). SPICE lets the user specify the range as linear or ''decade,'' indicating that we desire a logarithmic frequency scale. The first number is the number of frequency points per decade. The second number is the starting frequency, and the third number is the ending frequency.

```
AC analysis of differential pair.
V1 4 0 2V    AC 1
V2 5 0 2V
V3 1 0 5V
R1 2 0 1k
R2 3 1 1K
R3 6 1 1K
Q1 3 4 2 m2n2222
Q2 6 5 2 m2n2222
.model m2n2222 NPN IS = 1e-12 BF = 100 BR = 5 TF = 100pS
.AC DEC  10 1e3 1e9
```

FIGURE 9.9 Input file for AC analysis.

Figure 9.10 shows the results of the analysis. The gain begins to roll off at about 30 MHz due to the parasitic capacitances within the transistor models. The input impedance (which is plotted in kΩ) begins to roll off at a much lower frequency. The reduction in input impedance is due to the increasing current that flows in the base–emitter capacitance as the current increases. SPICE does not have a method of calculating input impedance, so we have calculated it as $Z = V_{\text{in}}/I(V_{\text{in}})$, where $V_{\text{in}} = 1.0$, using the postprocessing capability of PSPICE. This analysis took about 2 s on a 486-type PC.

9.7.2.2 Noise Analysis

Noise is a problem primarily in circuits that are designed for the amplification of small signals like the RF and IF amplifiers of a receiver. Noise is the result of random fluctuations in the currents which flow in the circuit and is generated in every circuit element. In circuit simulation, noise analysis, is an extension of AC analysis. During noise analysis, it is assumed that every circuit element contributes some small noise component either as a voltage V_{n} in series with the element or as a current I_{n} across the element. Since the noise sources are small in comparison to the DC signal levels, AC small signal analysis is an applicable analysis method.

Different models have been developed for the noise sources. In a resistor, thermal noise is the most important component. Thermal noise is due to the random motion of the electrons:

$$I_n^2 = \frac{4kT\Delta f}{R} \tag{9.18}$$

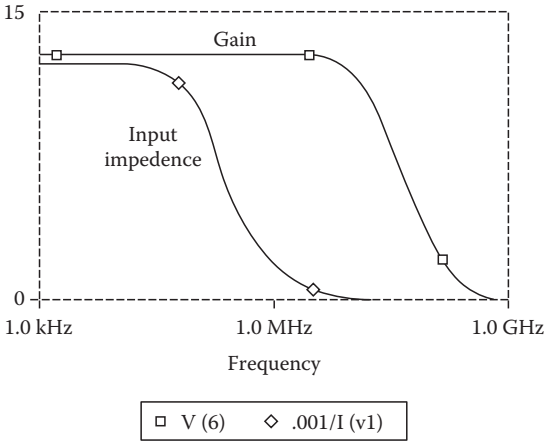


FIGURE 9.10 Gain and input impedance calculated by AC analysis.

where

T is the temperature
k is Boltzman's constant
Δf is the bandwidth of the circuit

In a semiconductor diode, shot noise is important. Shot noise is related to the probability that an electron will surmount the semiconductor barrier energy and be transported across the junction:

$$I_n^2 = 2qI_d\Delta f \tag{9.19}$$

Other types of noise occur in diodes and transistors; examples are flicker and popcorn noise. Noise sources, in general, are frequency-dependent.

Noise signals will be amplified or attenuated as they pass through different circuits. Normally, noise is referenced to some output point called the ''summing node.'' This would normally be the output of the amplifier where we would actually measure the noise. We can call the gain between the summing node and the current flowing in an element j in the circuit $A_j(f)$. Here, f is the analysis frequency since the gain will normally be frequency dependent.

Noise signals are random and uncorrelated to each other so their magnitudes must be root-mean-squared summed rather than simply summed. Summing all noise sources in a circuit yields:

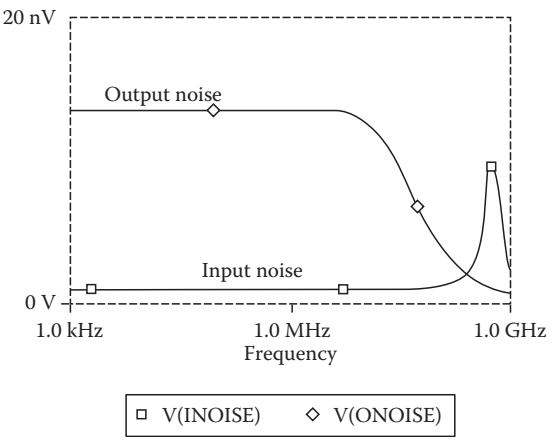


FIGURE 9.11 Noise referenced to output and input.

$$I_n(f) = \sqrt{\sum_j A_j^2(f) I_j^2(f)} \tag{9.20}$$

It is also common to reference noise back to the amplifier input and this is easily calculated by dividing the preceding expression by the amplifier gain. Specifying noise analysis in SPICE is simple. All the user needs to do is add a statement specifying the summing node and the input source. SPICE then calculates the noise at each as a function of frequency

```
.noise  v([6])  V1
```
(9.21)

See Figure 9.11 for example output. Many circuit simulators will also list the noise contributions of each element as part of the output. This is particularly helpful in locating the source of noise problems.

9.7.3 Transient Analysis

Transient analysis is the most powerful analysis capability because the transient response of a circuit is so difficult to calculate analytically. Transient analysis can be used for many types of analysis, such as switching speed, distortion, and checking the operation of circuits such as logic gates, oscillators, phase-locked loops, or switching power supplies. Transient analysis is also the most CPU intensive and can require 100 or 1000 times the CPU time of DC or AC analysis.

9.7.3.1 Numerical Method

In transient analysis, time is discretized into intervals called time steps. Typically, the time steps are of unequal length, with the smallest steps being taken during intervals where the circuit voltages and currents are changing most rapidly. The following procedure is used to discretize the time-dependent terms in Equation 9.1.

Time derivatives are replaced by difference operators, the simplest of which is the forward difference operator:

$$\frac{dQ(t_k)}{dt} = \frac{Q(t_{k+1}) - Q(t_k)}{h} \tag{9.22}$$

where h is the time step given by $h = t_{k+1} - t_k$. We can easily solve for the charge $Q(t_{k+1})$ at the next time point:

$$Q(t_{k+1}) = Q(t_k) - h(G_i(V(t_k)) + W_i(t_k)) \tag{9.23}$$

using only values from past time points. This means that it would be possible to solve the system simply by plugging in the updated values for V each time. This can be done without any matrix assembly or inversion and is very nice. (Note for simple linear capacitors, $V = Q/C$ at each node, so it is easy to get V back from Q.) However, this approach is undesirable for circuit simulation for two reasons: (1) The charge Q, which is a ''state variable'' of the system, is not a convenient choice since some nodes may not have capacitors (or inductors) attached, in which case they will not have Q values and (2) It turns out that forward (or explicit) time discretization methods like this one are unstable for ''stiff'' systems, and most circuit problems result in ''stiff systems.'' The term ''stiff system'' refers to a system that has greatly varying time constants.

To overcome the stiffness problem, we must use implicit time discretization methods which, in essence, mean that the G and W terms in the above equations must be evaluated at t_{k+1}. Since G is nonlinear, we will need to use Newton's method once again.

The most popular implicit method is the trapezoidal method. The trapezoidal method has the advantage of requiring information only from one past time point and, furthermore, has the smallest error of any method requiring one past time point. The trapezoidal method states that if I is the current in a capacitor, then

$$I(t_{k+1}) = \frac{\mathrm{d}Q}{\mathrm{d}t} = 2\frac{Q(V(t_{k+1})) - Q(V(t_k))}{h} - I(t_k) \tag{9.24}$$

Therefore, we need to only substitute Equation 9.24 into Equation 9.1 to solve the transient problem. Observe that we are solving for the voltages $V(t_{k+1})$, and all terms involving t_k are constant and will not be included in the Jacobian matrix. An equivalent electrical model for the capacitor is shown in Figure 9.12. Therefore, the solution of the transient problem is in effect a series of DC solutions where the values of some of the elements depend on voltages from the previous time points.

All modern circuit simulators feature automatic time step control. This feature selects small time steps during intervals where changes occur rapidly and large time steps in intervals where there is little change. The most commonly used method of time step selection is based on the local truncation error (LTE) for each time step. For the trapezoidal rule, the LTE is given by

$$\varepsilon = \frac{h^3}{12}\frac{\mathrm{d}^3 x}{\mathrm{d}t^3}(\xi) \tag{9.25}$$

and represents the maximum error introduced by the trapezoidal method at each time step. If the error (ε) is larger than some preset value, the step size is reduced. If the error is smaller, then the step size is increased. In addition, most simulators select time points so that they coincide with the edges of pulse-type waveforms.

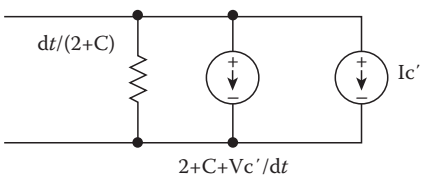


FIGURE 9.12 Electrical model for a capacitor; the two current sources are independent sources. The prime (′) indicates values from a preceding time point.

9.7.3.2 Transient Analysis Examples

As a simple example, we return to the differential pair and apply a sine wave differentially to the input. The amplitude (2 V p-p) is selected to drive the amplifier into saturation.

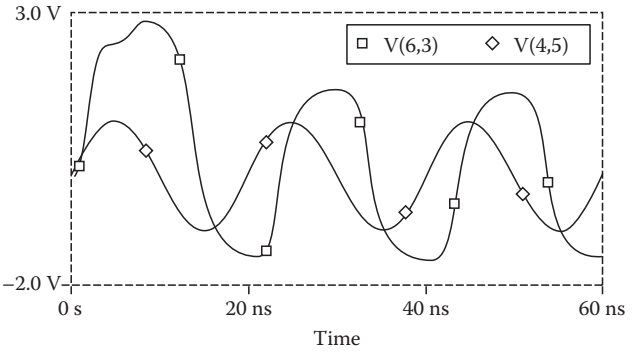


FIGURE 9.13 Transient response V(6,3) of differential amplifier to sinusoidal input at V(4,5).

In addition, we make the frequency (50 MHz) high enough to see phase shift effects. The output signal is therefore clipped due to the nonlinearities and shifted in phase due to the capacitive elements in the transistor models (see Figure 9.13). The first cycle shows extra distortion since it takes time for the ''zero-state'' response to die out. This simulation, using PSPICE, runs in about one second on a 486-type computer.

9.8 Verilog-A

Verilog-A is a new language designed for simulation of analog circuits at various levels. Mathematical equations can be entered directly as well as normal SPICE-type circuit elements.

Groups of equations and elements can be combined into reusable ''modules'' that are similar to subcircuits. Special functions are also provided for converting analog signals into digital equivalents, and vice versa. Systems-type elements such as Laplace operators, integrators, and differentiators are also provided. This makes it possible to perform new types of modeling that were not possible in simulators like SPICE:

- Equations can be used to construct new models for electrical devices (for example, the Ebers–Moll model described earlier could be easily implemented).
- Behavioral models for complex circuits like op-amps, comparators, phase detectors, etc. can be constructed. These models can capture the key behavior of a circuit and yet be simulated in a small fraction of the time it takes to simulate at the circuit level.
- Special interface elements make it possible to connect an analog block to a digital simulator, making mixed-mode simulation possible. Verilog-A is related to and compatible with the popular Verilog-D modeling language for digital circuits.

As an example, consider a phase-locked loop circuit which is designed as an 50X frequency multiplier. A block diagram for the PLL is shown in Figure 9.14 and the Verilog-A input listing is shown in Figures 9.15 and 9.16.

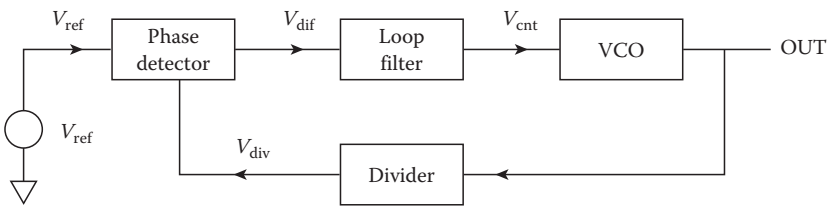


FIGURE 9.14 Block diagram of phase-locked loop.

```
‘include “electrical.h”
‘timescale 100ns/10ns

module top;                                  // Top level Module
        electrical Vdd, Vref, Vdif, Vcnt, Vout, Vdiv;
        pd #(.gain(15u), .rout (2e6), dir(–l)) p1(Vdif, Vdiv, Vref, Vdd);
        filter f1 (Vdif, Vcnt);
        vco #(.gain(2e7), .center(3e7)) v1 (Vout, Vcntm Vdd);
        divide #(.count(50)) d1 (Vdiv, Vout, Vdd);
        analog begin
                V(Vdd) <+ 5.0;
                V(Vref) <+ 2.5+2.5*sin(6.238*1e6*$realtime);
        end
endmodule

module pd (out, vco, ref, vdd);          //Phase detector Module.
        inout out, vco, ref, vdd;
        electrical out,vco, ref, vdd;
        parameter real gain = 15u, rout = 5e6;          // gain & output impedance
        parameter integer dir = 1;                      // 1,–1 pos or neg edge trigger
        integer state;
        analog begin
                @ (cross((V(ref) – V(vdd)/2),dir)) state = state – 1;
                @ (cross((V(vco) – V(vdd)/2),dir)) state = state + 1;
                if (state > 1 ) state = 1;
                if (state < –1) state = –1;
                if (state ! = 0)   I(out) <+ transition (state * gain, 0, 0, 0);
              I(out) <+ V(out)/rout;
        end
endmodule
```

FIGURE 9.15 Part one of Verilog-A listing for PLL.

Simulation of this system at the circuit level is very time consuming due to the extreme difference in frequencies. The phase detector operates at a low frequency of 1.0 MHz, while the VCO operates at close to 50 MHz. However, we need to simulate enough complete cycles at the phase detector output to verify that the circuit correctly locks onto the reference signal.

The circuit is broken up into five blocks or modules: the ‘‘top module,’’ VCO, divider, phase detector, and loop filter. The VCO has a simple linear dependence of frequency on the VCO input voltage and produces a sinusoidal output voltage. The VCO frequency is calculated by the simple expression freq = center + gain* ($V_{in} - V_{min}$). Center and gain are parameters which can be passed in when the VCO is created within the top module by the special syntax `‘‘#(.gain(2e7),.center(3e7)’’` in the top module. If the parameters are not specified when the module is created, then the default values specified within the module are used instead. The special V() operator is used to obtain the voltage at a node (in this case `V(in)` and `V(Vdd)`). The sinusoidal output is created using the SIN and IDT operators. SIN calculates the sine of its argument. I_{dt} calculates the integral of its argument with respect to time. The amplitude of the output is taken from the V_{dd} input, thus making it easy to integrate the VCO block with others. Given that $V_{dd} = 5$ V, gain = 2e7 Hz/V, center = 3e7 Hz, and in = 1.8, the final expression for the VCO output is

$$V_{out} = 2.5 + 2.5\sin\left(3e7 + 2e7\int 2\pi(V_{in} - 1.8)dt\right) \tag{9.26}$$

The phase detector functions as a charge pump which drives current into or out of the loop filter, depending on the phase difference between its two inputs. The `@cross(V1,dir)` function becomes

```
module divide (out, in, vdd);      // Divider module....
        inout out, in;
        electrical out, in;
        parameter real count = 4; // divide by this.
    integer n, state;
        analog begin

                @(cross((V(in) – V(vdd)/2.0),1)) n = n + 1;
                if (n >= count/2) begin
                        if (state == 0) state = 1;
                        else state = 0;
                        n = 0;
                end
                V(out) <+ transition(state*5, 0, 0, 0);
        end
endmodule

module vco (vout, vin, vdd);       // VCO module
        inout vin, vout, vdd;
        electrical vin, vout, vdd;
        parameter real gain = 5e6, center = 40e6, vmin = 1.8;
        real freq, vinp;
        analog begin
                vinp = V(vin);
                if (vinp < vm) vinp = vmin;
                freq = center + gain* (vinp-vmin);
                V(vout) <+ V(vdd)/2.0*sin(6.28318531*idt(freq,0)) + V(vdd)/2.0;
        end
endmodule

'language SPICE
.SUBCKT filter Vin Vout
Rf2 Vin Vout 100
Rfilter Vin VC2 200000
C1 VC2 0 58p
C2 Vin 0 5p
.ends
'endlanguage
```

FIGURE 9.16 Part two of Verilog-A PLL listing.

true whenever signal V_1 crosses zero in the direction specified by `dir`. This either increments or decrements the variable STATE. The ''transition'' function is used to convert the STATE signal, which is essentially digital and changes abruptly into a smoothly changing analog signal which can be applied to the rest of the circuit. The ''<+'' (or contribution) operator adds the current specified by the equation on the right to the node on the left. Therefore, the phase detector block forces current into the output node whenever the VCO signal leads the reference signal and forcing current out of the output node whenever the reference leads the VCO signal. The phase detector also has an output resistance which is specified by parameter ROUT.

The loop filter is a simple SPICE subcircuit composed of two resistors and one capacitor. Of course, this subcircuit could contain other types of elements as well and can even contain other Verilog-A modules. The divider block simply counts zero crossings and, when the count reaches the preset divisor, the output of the divider is toggled from 0 to 1, or vice versa. The transition function is used to ensure that a smooth, continuous analog output is generated by the divider.

This PLL was simulated using AMS from Antrim Design Systems. The results of the simulations are shown in Figure 9.17. The top of Figure 9.17 shows the output from the loop filter (V_{cnt}). After a few cycles, the PLL has locked onto the reference signal. The DC value of the loop filter output is

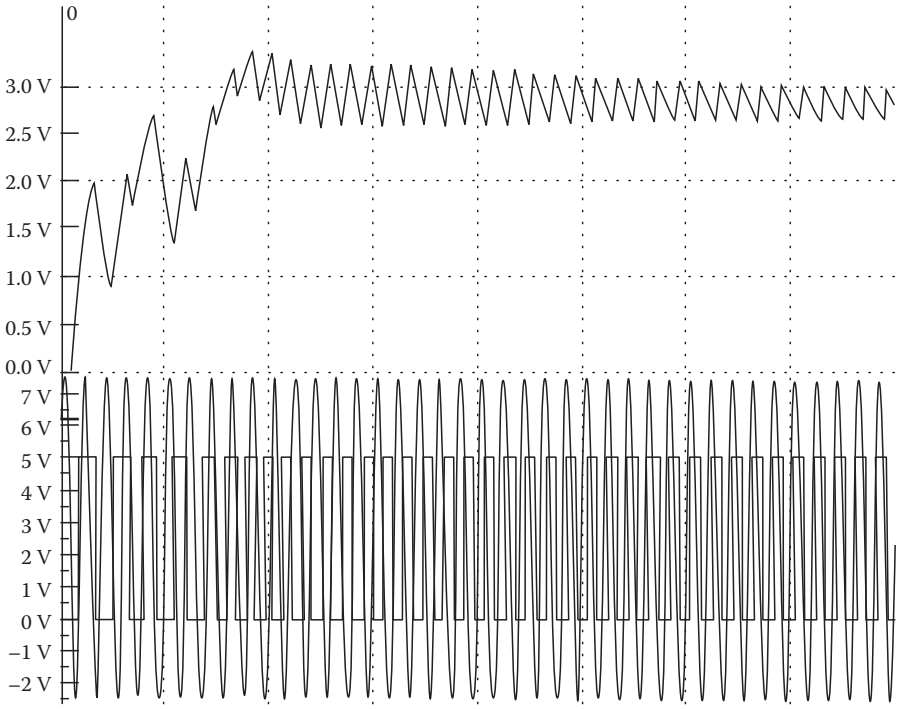


FIGURE 9.17 Loop filter output (top) V_{ref} and divider output (bottom) from PLL simulation.

approximately 2.8 V. Referring back to the VCO model, this gives an output frequency of 2e7*(2.8 − 1.8) + 3e7 = 50 MHz, which is as expected. The lower portion of Figure 9.17 shows the divider output (V_{div}) and the reference signal (V_{ref}). It can be seen that the two signals are locked in phase. Figure 9.18 shows the VCO output and the divider output. As expected, the VCO frequency is 50 times the divider frequency.

The behavioral models used in this example are extremely simple ones. Typically, more complex models must be used to accurately simulate the operation of an actual PLL. A better model might include effects such as the nonlinear dependence of the VCO frequency on the input voltage, the effects on signals introduced through power supply lines, delays in the divider and phase detector, and finite signal rise and fall times. These models can be built up from measurements, or transistor-level simulation of the underlying blocks (a process known as characterization). Of course, during the simulation, any of the behavioral blocks could be replaced by detailed transistor-level models or complex Verilog-D digital models.

Another Verilog-A example is shown in Figure 9.19. Here, the Ebers–Moll model developed earlier is implemented as a module. This module can then be used in a circuit in the same way as the normal built-in models. Verilog-A takes care of calculating all the derivatives needed to form the Jacobian matrix. The ''parameter'' entries can be used in the same way as the parameters on a SPICE.MODEL statement. Observe the special ''ddt'' operator. This operator is used to take the time derivative of its argument. In this case, the time derivative of the charge (a current) is calculated and summed in with the other DC components. The ''$limexp'' operation is a special limited exponential operator designed to give better convergence when modeling pn junctions. Of course, this module could be expanded and additional features could be added.

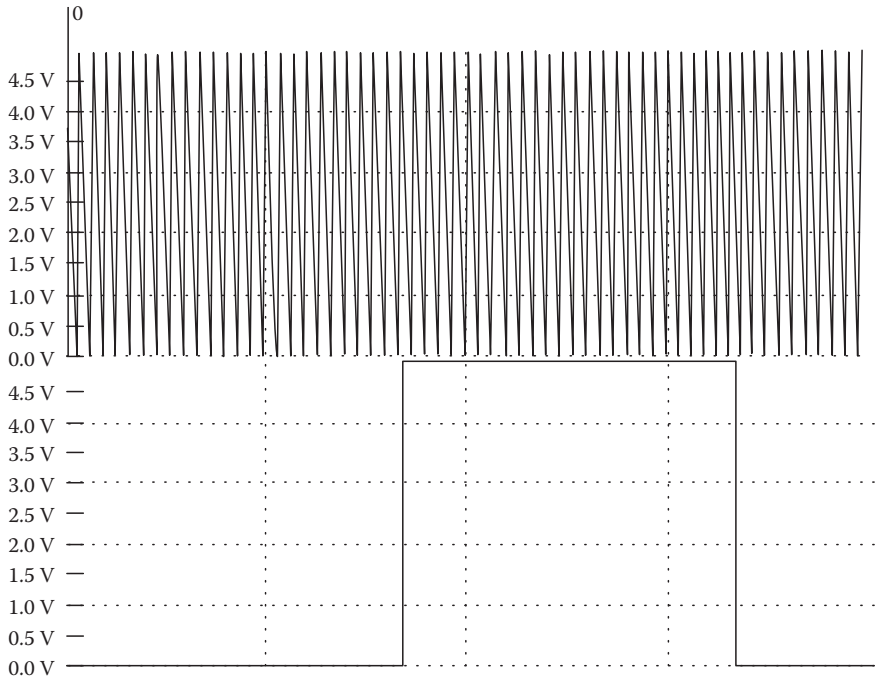


FIGURE 9.18 VCO output and divider output from PLL simulation.

```
module mybjt (C,B,E);
        inout C,B,E;
        electrical C,B,E;
        parameter real is = le–16, bf = 100, br = 10, tf = 1n, tr = l0n, cje = 1p cjc = lp,
                vje = 0.75, vjc = 0.75, mje = 0.33 mjc = 0.33;
        real ibc, ibe, qbc, qbe, vbe, vbc, cc, cb, x, y;
        analog begin
                vbe = V(B,E);
                vbc = V(B,C);
                ibe = is*($limexp(vbe/$vt)–1);
                ibc = is*($limexp(vbc/$vt)–1);
                cb = ibe/bf + ibc/br;
                cc = ibe – ibc – ibc/br
                if(vbe < 0) begin
                        x = 1–vbe/vje;
                        y = exp(–mje*ln(x));
                        qbe = vje*cje*(1–x*y)/(l.0–mje)+tf*ibe;
                end
                else qbe = cje*(vbe*+0.5*mje*vbe*vbe/vje) + tf*ibe;
                if(vbc < 0) begin
                        x = 1 –vbc/vjc;
                        y = exp(–mjc*ln(x));
                        qbc = vjc*cjc*(l–x*y)/(1.0–mjc) + tr*ibc;
                end
                else qbc = cjc*(vbc+0.5*mjc*vbc*vbc/vjc) + tr*ibc;
                ibe = ddt(qbe);
                ibc = ddt(qbc);
                I(C,E) <+ cc – ibc;
                I(B,E) <+ cb + ibe + ibc;
        end
endmodule
```

FIGURE 9.19 Verilog-A implementation of the Ebers–Moll model.

9.9 Fast Simulation Methods

As circuits get larger, simulation times become larger. In addition, as integrated circuit feature sizes shrink, second-order effects become more important and many circuit designers would like to be able to simulate large digital systems at the transistor level (requiring 10,000 to 100,000 nodes). Numerical studies in early versions of SPICE showed that the linear solution time could be reduced to 26% for relatively small circuits with careful coding. The remainder is used during the assembly of the matrix, primarily for model evaluation. The same studies found that the CPU time for the matrix solution was proportional to $n^{1.24}$, where n is the number of nodes. The matrix assembly time on the other hand should increase linearly with node count. Circuits have since grown much bigger, but the models (particularly for MOS devices) have also become more complicated.

Matrix assembly time can be reduced by a number of methods. One method is to simplify the models; however, accuracy will be lost as well. A better way is to precompute the charge and current characteristics for the complicated models and store them into tables. During simulation, the actual current and charges can be found from table lookup and interpolation, which can be done quickly and efficiently. However, there are some problems:

1. To assure convergence of Newton's method, both the charge and current functions and their derivatives must be continuous. This rules out most simple interpolation schemes and means that something like a cubic spline must be used.
2. The tables can become large. A MOS device has four terminals, which means that all tables will be functions of three independent variables. In addition, the MOSFET requires four separate tables (I_d, Q_g, Q_d, Q_b). If we are lucky, we can account for simple parameteric variations (like channel width) by a simple multiplying factor. However, if there are more complex dependencies as is the case with channel length, oxide thickness, temperature, or device type, we will need one complete set of tables for each device.

If the voltages applied to an element do not change from the past iteration to the present iteration, then there is no need to recompute the element currents, charges, and their derivatives. This method is referred to as taking advantage of latency and can result in large CPU time savings in logic circuits, particularly if coupled with a method that only refractors part of the Jacobian matrix. The tricky part is knowing when the changes in voltage can be ignored. Consider, for example, the input to a high-gain op-amp, here ignoring a microvolt change at the input could result in a large error at the output. Use of sophisticated latency-determining methods could also cut into the savings.

Another set of methods are the waveform relaxation techniques which increase efficiency by temporarily ignoring couplings between nodes. The simplest version of the method is as follows. Consider a circuit with n nodes which requires m time points for its solution. The circuit can be represented by the vector equation:

$$F_i(V(t)) + \frac{dQ_i(V(t))}{dt} = 0 \tag{9.27}$$

Using trapezoidal time integration gives a new function:

$$W_i(V(k)) = F_i(V(k)) + F_i(V(k-1)) + 2[Q_i(V(k)) - Q_i(V(k-1))]/dt = 0 \tag{9.28}$$

We need to find the $V(k)$ that makes W zero for all k time points at all i nodes. The normal method solves for all n nodes simultaneously at each time point before advancing k. Waveform relaxation solves for all m time points at a single node (calculates the waveform at that node) before advancing to the next node. An outer loop is used to assure that all the individual nodal waveforms are consistent with each other.

Waveform relaxation is extremely efficient as long as the number of outer loops is small. The number of iterations will be small if the equations are solved in the correct order; that is, starting on nodes which are signal sources and following the direction of signal propagation through the circuit. This way, the waveform at node $i+1$ will depend strongly on the waveform at node i, but the waveform at node i will depend weakly on the signal at node $i+1$. The method is particularly effective if signal propagation is unidirectional, as is sometimes the case in logic circuits. During practical implementation, the total simulation interval is divided into several subintervals and the subintervals are solved sequentially. This reduces the total number of time points which must be stored. Variants of the method solve small numbers of tightly coupled nodes as a group; such a group might include all the nodes in a TTL gate or in a small feedback loop. Large feedback loops can be handled by making the simulation time for each subinterval less than the time required for a signal to propagate around the loop.

The efficiency of this method can be further improved using different time steps at different nodes, yielding a multirate method. This way, during a given interval, small time steps are used at active nodes while large steps are used at inactive nodes (taking advantage of latency).

9.10 Commercially Available Simulators

The simulations in this chapter were performed with the evaluation version of PSPICE from Microsim and AMS from Antrim design systems. The following vendors market circuit simulation software. The different programs have strengths in different areas and most vendors allow you to try their software in-house for an ''evaluation period'' before you buy.

SPICE2-SPICE3	University of California, Berkeley, CA
AMS	Antrim Design Systems, Scotts Valley, CA, www.antrim.com
PSPICE	Orcad Corporation, Irvine, CA, www.orcad.com
HSPICE	Avant! Corporation, Fremont, CA, www.avanticorp.com
ISPICE	Intusoft, SanPedro, CA, www.intusoft.com
SABER	Analogy, Beaverton, OR, www.analogy.com
SPECTRE	Cadence Design Systems, San Jose, CA, www.cadence.com
TIMEMILL	Synopsys Corporation, Sunnyvale, CA, www.synopsys.com
ACCUSIM II	Mentor Graphics, Wilsonville, OR, www.mentorg.com

References

On general circuit simulation:

1. J. Vlach and K. Singhal, *Computer Methods for Circuit Analysis and Design*, Van Nostrand Reinhold, New York, 1983.
2. A. E. Ruehli, Ed., *Circuit Analysis, Simulation and Design, Part 1*, Elsevier Science, B.V. North Holland, 1981.
3. L. Nagel, SPICE2: A computer program to simulate semiconductor circuits, PhD thesis, University of California, Berkeley, CA, 1975.
4. K. Kundert, *The Designers Guide to SPICE and SPECTRE*, Kluwer Academic, Boston, MA, 1995.
5. P. W. Tuinenga, *SPICE, A Guide to Circuit Simulation and Analysis Using PSPICE*, Prentice Hall, Englewood Cliffs, NJ, 1988.
6. J. A. Connelley and P. Choi, *Macromodeling with SPICE*, Prentice Hall, Engelwood Cliffs, NJ, 1992.
7. P. Gray and R. Meyer, *Analysis and Design of Analog Integrated Circuits*, Wiley, New York, 1977.
8. A. Vladimiresch, *The SPICE Book*, Wiley, New York, 1994.

On modern techniques:

9. A. E. Ruehli, Ed., *Circuit Analysis, Simulation and Design, Part 2*, Elsevier Science, B.V. North Holland, 1981.

On device models:

10. P. Antognetti and G. Massobrio, *Semiconductor Modeling with SPICE2*, 2nd edn., McGraw-Hill, New York, 1993.
11. D. Foty, *MOSFET Modeling with SPICE*, Prentice Hall, Englewood Cliffs, NJ, 1997.
12. BSIM3 Users Guide, www-device.EECS.Berkeley.EDU/~bsim3.
13. MOS-9 models, www-us2.semiconductors.philips.com/Philils_Models.

On Verilog-A

14. D. Fitzpatrick, *Analog Behavior Modeling with the Verilog-A Language*, Kluwer Academic, Boston, MA, 1997.

II

Design Automation

Wai-Kai Chen
University of Illinois at Chicago

10 Internet-Based Microelectronic Design Automation Framework
Moon-Jung Chung and Heechul Kim **10**-1
Introduction • Functional Requirements of Framework • IMEDA System • Formal Representation of Design Process • Execution Environment of the Framework • Implementation • Conclusion • References

11 System-Level Design *Alice C. Parker, Yosef Tirat-Gefen, and Suhrid A. Wadekar* **11**-1
Introduction • System Specification • System Partitioning • Scheduling and Allocating Tasks to Processing Modules • Allocating and Scheduling Storage Modules • Selecting Implementation and Packaging Styles for System Modules • Interconnection Strategy • Word-Length Determination • Predicting System Characteristics • Survey of Research in System Design • References

12 Performance Modeling and Analysis Using VHDL and SystemC
Robert H. Klenke, Jonathan A. Andrews, and James H. Aylor **12**-1
Introduction • ADEPT Design Environment • Simple Example of an ADEPT Performance Model • Mixed-Level Modeling • Performance and Mixed-Level Modeling Using SystemC • Conclusions • References

13 Embedded Computing Systems and Hardware/Software Codesign
Wayne Wolf **13**-1
Introduction • Uses of Microprocessors • Embedded System Architectures • Hardware/Software Codesign •

14 Design Automation Technology Roadmap *Donald R. Cottrell* **14**-1
Introduction • Design Automation: Historical Perspective • The Future • Summary • References

10

Internet-Based Microelectronic Design Automation Framework

10.1 Introduction **10**-1
10.2 Functional Requirements of Framework **10**-3
Building Blocks of Process • Functional Requirements of Workflow Management • Process Specification • Execution Environment • Literature Surveys
10.3 IMEDA System **10**-9
10.4 Formal Representation of Design Process **10**-12
Process Flow Graph • Process Grammars
10.5 Execution Environment of the Framework **10**-16
Cockpit Program • Manager Programs • Execution Example • Scheduling
10.6 Implementation **10**-23
System Cockpit • External Tools • Communications Model • User Interface
10.7 Conclusion **10**-30
References **10**-30

Moon-Jung Chung
Michigan State University

Heechul Kim
Hankuk University of Foreign Studies

10.1 Introduction

As the complexity of VLSI systems continues to increase, the microelectronic industry must possess an ability to reconfigure design and manufacturing resources and integrate design activities so that it can quickly adapt to the market changes and new technology. Gaining this ability imposes a twofold challenge: (1) to coordinate design activities that are geographically separated and (2) to represent an immense amount of knowledge from various disciplines in a unified format. The Internet can provide the catalyst by abridging many design activities with the resources around the world not only to exchange information but also to communicate ideas and methodologies.

In this chapter, we present a collaborative engineering framework that coordinates distributed design activities through the Internet. Engineers can represent, exchange, and access the design knowledge and carry out design activities. The crux of the framework is the formal representation of process flow using the process grammar, which provides the theoretical foundation for representation, abstraction, manipulation, and execution of design processes. The abstraction of process representation provides mechanisms to represent hierarchical decomposition and alternative methods, which enable designers to manipulate

the process flow diagram and select the best method. In the framework, the process information is layered into separate specification and execution levels so that designers can capture processes and execute them dynamically. As the framework is being executed, a designer can be informed of the current status of design such as updating and tracing design changes and be able to handling exception. The framework can improve design productivity by accessing, reusing, and revising the previous process for a similar. The cockpit of our framework interfaces with engineers to perform design tasks and to negotiate design trade-off. The framework has the capability to launch whiteboards that enable the engineers in a distributed environment to view the common process flows and data and to concurrently execute dynamic activities such as process refinement, selection of alternative process, and design reviews. The proposed framework has a provision for various browsers where the tasks and data used in one activity can be organized and retrieved later for other activities.

One of the predominant challenges for microelectronic design is to handle the increased complexity of VLSI systems. At the turn of the century, it is expected that there will be 100 million transistors in a single chip with 0.1 μm features, which will require an even shorter design time (Spiller and Newton, 1997). This increase of chip complexity has given impetus to trends such as system on a chip, embedded system, and hardware/software codesign. To cope with this challenge, industry uses custom-off-the-shelf (COTS) components, relies on design reuse, and practices outsourcing design. In addition, design is highly modularized and carried out by many specialized teams in a geographically distributed environment. Multifacets of design and manufacturing, such as manufacturability and low power, should be considered at the early stage of design. It is a major challenge to coordinate these design activities (Fairbairn, 1994). The difficulties are caused by due to the interdependencies among the activities, the delay in obtaining distant information, the inability to respond to errors and changes quickly, and general lack of communications. At the same time, the industry must contend with decreased expenditures on manufacturing facilities while maintaining rapid responses to market and technology changes.

To meet this challenge, the U.S. government has launched several programs. The Rapid Prototyping of Application Specific Signal Processor (RASSP) Program was initiated by the Department of Defense to bring about the timely design and manufacturing of signal processors. One of the main goals of the RASSP program was to provide an effective design environment to achieve a four-time improvement in the development cycle of digital systems (Chung et al., 1996). DARPA also initiated a program to develop and demonstrate key software elements for Integrated Product and Process Development (IPPD) and agile manufacturing applications. One of the foci of the earlier program was the development of infrastructure for distributed design and manufacturing. Recently, the program is continued to Rapid Design Exploration & Optimization (RaDEO) to support research, development, and demonstration of enabling technologies, tools, and infrastructure for the next generation of design environments for complex electromechanical systems. The design environment of RaDEO is planned to provide cognitive support to engineers by vastly improving their ability to explore, generate, track, store, and analyze design alternatives (Lyons, 1997).

The new information technologies, such as the Internet and mobile computing, are changing the way we communicate and conduct business. More and more design centers use PCs, and link them on the Internet/intranet. The web-based communication allows people to collaborate across space and time, between humans, humans and computers, and computers in a shared virtual world (Berners-Lee et al., 1994). This emerging technology holds the key to enhance design and manufacturing activities. The Internet can be used as the medium of a virtual environment where concepts and methodologies can be discussed, accessed, and improved by the participating engineers. Through the medium, resources and activities can be reorganized, reconfigured, and integrated by the participating organizations. This new paradigm certainly impacts the traditional means for designing and manufacturing a complex product. Using Java, programs can be implemented in a platform-independent way so that they can be executed in any machine with a Web browser. Common Object Request Broker Architecture (CORBA) (Yang and Duddy, 1996) provides distributed services for tools to communicate through the Internet (Vogel and Duddy). Designers may be able to execute remote tools through the Internet and see the visualization of design data (Erkes et al., 1996; Chan et al., 1998; Chung and Kwon, 1998).

Even though the potential impact of this technology will be great on computer aided design, Electronic Design Automation (EDA) industry has been slow in adapting this new technology (Spiller and Newton, 1997). Until recently, EDA frameworks used to be a collection of point tools. These complete suites of tools are integrated tightly by the framework using their proprietary technology. These frameworks have been suitable enough to carry out a routine task where the process of design is fixed. However, new tools appear constantly. To mix and match various tools outside of a particular framework is very difficult. Moreover, tools, expertise, and materials for design and manufacturing of a single system are dispersed geographically. Now we have reached the stage where a single tool or framework is not sufficient enough to handle the increasing complexity of a chip and emerging new technology. A new framework is necessary which is open and scalable. It must support collaborative design activities so that designers can add new tools to the framework, and interface them with other CAD systems. There are two key functions of the framework: (1) managing the process and (2) maintaining the relationship among many design representations. For design data management, refer to Katz et al. (1987). In this chapter, we will focus on the process management aspect.

To cope with the complex process of VLSI system design, we need a higher level of viewing of a complete process, i.e., the abstraction of process by hiding all details that need not to be considered for the purpose at hand. As pointed out in National Institute of Standards and Technology reports (Schlenoff et al., 1996; Knutilla et al., 1998), a ''unified process specification language'' should have the following major requirements: abstraction, alternative task, complex groups of tasks, and complex sequences.

In this chapter, we first review the functional requirements of the process management in VLSI system design. We then present the Internet-based Microelectronic Design Automation (IMEDA) System. IMEDA is a web-based collaborative engineering framework where engineers can represent, exchange, and access design knowledge and perform the design activities through the Internet. The crux of the framework is a formal representation of process flow using process grammar. Similar to the language grammar, production rules of the process grammar map tasks into admissible process flows (Baldwin and Chung, 1995a). The production rules allow a complex activity to be represented more concisely with a small number of high-level tasks. The process grammar provides the theoretical foundation for representation, abstraction, manipulation, and execution of design and manufacturing processes. It facilitates the communication at an appropriate level of complexity. The abstraction mechanism provides a natural way of browsing the process repository and facilitates process reuse and improvement. The strong theoretical foundation of our approach allows users to analyze and predict the behavior of a particular process. The cockpit of our framework interfaces with engineers to perform design tasks and to negotiate design trade-off. The framework guides the designer in selecting tools and design methodologies, and it generates process configurations that provide optimal solutions with a given set of constraints. The just-in-time binding and the location transparency of tools maximize the utilization of company resources. The framework is equipped with whiteboards so that engineers in a distributed environment can view the common process flows and data and concurrently execute dynamic activities such as process refinement, selection of alternative processes, and design reviews. With the grammar, the framework gracefully handles exceptions and alternative productions. A layered approach is used to separate the specification of design process and execution parameters. One of the main advantages of this separation is freeing designers from the overspecification and graceful exception handling. The framework, implemented using Java, is open and extensible. New process, tools, and user-defined process knowledge and constraints can be added easily.

10.2 Functional Requirements of Framework

Design methodology is defined as a collection of principles and procedures employed in the design of engineering systems. Baldwin and Chung (1995a) define design methodology management as selecting and executing methodologies so that the input specifications are transformed into desired output specifications. Kleinfeldth et al. (1994) state that ''design methodology management provides for the

definition, presentation, execution, and control of design methodology in a flexible, configured way.'' Given a methodology, we can select a process or processes for that particular methodology.

Each design activity, whether big or small, can be treated as a task. A complex design task is hierarchically decomposed into simpler subtasks, and each subtask in turn may be further decomposed. Each task can be considered as a transformation from input specification to output specification. The term *workflow* is used to represent the details of a process including its *structure in terms of all the required tasks and their interdependencies*. Some process may be ill-structured, and capturing it as a workflow may not be easy. Exceptions, conditional executions, and human involvement during the process make it difficult to model the process as a workflow.

There can be many different tools or alternative processes to accomplish a task. Thus, a design process requires design decisions such as selecting tools and processes as well as selecting appropriate design parameters. At a very high level of design, the input specifications and constraints are very general and may even be ill-structured. As we continue to decompose and perform the tasks based on design decisions, the output specifications are refined and the constraints on each task become more restrictive. When the output of a task does not meet certain requirements or constraints, a new process, tools, or parameters must be selected. Therefore, the design process is typically iterative and based on previous design experience. Design process is also a collaborative process, involving many different engineering activities and requiring the coordination among engineers, their activities, and the design results.

Until recently, it was the designer's responsibility to determine which tools to use and in what order to use them. However, managing the design process itself has become difficult, since each tool has its own capabilities and limitations. Moreover, new tools are developed and new processes are introduced continually. The situation is further aggravated because of incompatible assumptions and data formats between tools. To manage the process, we need a framework to monitor the process, carry out design tasks, support cooperative teamwork, and maintain the relationship among many design representations (Katz et al., 1987; Chiueh and Katz, 1990). The framework must support concurrent engineering activities by integrating various CAD tools and process and component libraries into a seamless environment. Figure 10.1 shows the RASSP enterprise system architecture (Welsh et al., 1995). It integrates tools, tool frameworks, and data management functions into an enterprise environment. The key functionality of the RASSP system is managing the RASSP design methodology by ''process automation'', that is, controlling CAD program execution through workflow.

10.2.1 Building Blocks of Process

The lowest level of a building block of a design process is a tool. A *tool* is an unbreakable unit of a CAD program. It usually performs a specific task by transforming given input specifications into output specifications. A *task* is defined as design activities that include information about what tools to use and how to use them. It can be decomposed into smaller subtasks. The simplest form of the task, called an *atomic task*, is the one that cannot be decomposed into subtasks. In essence, an atomic task is defined as an encapsulated tool. A task is called *logical* if it is not atomic. A workflow of a logical task describes the details of how the task is decomposed into subtasks, and the data and control dependencies such as the relationship between design data used in the subtasks. For a given task, there can be several workflows, each of which denotes a possible way of accomplishing the task. A *methodology* is a collection of workflow supported together with information on which workflow should be selected in a particular instance.

10.2.2 Functional Requirements of Workflow Management

To be effective, a framework must integrate many design automation tools and allow the designer to specify acceptable methodologies and tools together with information such as when and how they may be used. Such a framework must not only guide the designer in selecting tools and design methodologies, but also aid the designer in constructing a workflow that is suitable to complete the design under given

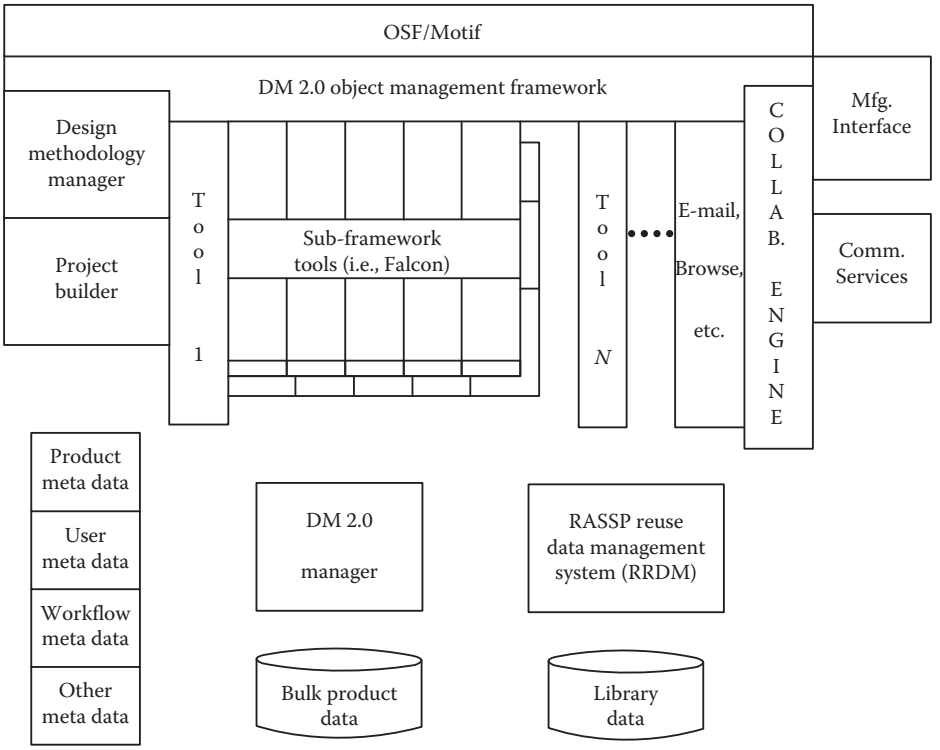


FIGURE 10.1 RASSP enterprise system architecture.

constraints. The constructed workflow should guarantee that required steps are not skipped; built-in design checks are incorporated into the workflow. The framework must also keep the relationships between various design representations, maintain the consistency between designs and support cooperative teamwork, and allow the designer to interact with the system to adjust design parameters or to modify the previous design process. The framework must be extendible to accommodate rapidly changing technologies and emerging new tools. Such a framework can facilitate developing new hardware systems as well as redesigning a system from a previous design.

During a design process, a particular methodology or workflow selected by a designer must be based on available tools, resources (computing and human), and design data. For example, a company may impose a rule that if input is a VHDL behavioral description, then designers should use Model Technology's VHDL simulator, but if the input is Verlig, they must use ViewLogic simulator. Or, if a component uses Xilinx, then all other components must also use Xilinx. Methodology must be driven by local expertise and individual preference, which in turn, are based on the designer's experience.

The process management should not constrain the designer. Instead, it must free designers from routine tasks, and guide the execution of workflow. User interaction and a designer's freedom are especially important when exceptions are encountered during the execution of flows, or when designers are going to modify the workflow locally. The system must support such activities through ''controlled interactions'' with designers.

Process management can be divided into two parts:

- Formal specification of supported methodologies and tools that must show the tasks and data involved in a workflow and their relationships
- Execution environment that helps designers to construct workflow and execute them

10.2.3 Process Specification

Methodology management must provide facilities to specify design processes. Specification of processes involves tasks and their structures (i.e., workflow). The task involved and the flow of process, that is the way the process can be accomplished in terms of its subtasks, must be defined. Processes must be encapsulated and presented to designers in a usable way. Designers want an environment to guide them in building a workflow and to help them execute it during the design process. Designers must be able to browse related processes, and compare, analyze, and modify them.

10.2.3.1 Tasks

Designers should be able to define the tasks that can be logical or atomic, organize the defined tasks, and retrieve them. Task abstraction refers to using and viewing a task for specific purposes and ignoring the irrelevant aspects of the task. In general, object-oriented approaches are used for this purpose. Abstraction of the task may be accomplished by defining tasks in terms of ''the operations the task is performing'' without detailing the operations themselves. Abstraction of tasks allows users to clearly see the behavior of them and use them without knowing the details of their internal implementations. Using the generalization–specialization (GS) hierarchy (Chung and Kim, 1990), similar tasks can be grouped together. In the hierarchy, a node in the lower level inherits its attributes from its predecessors. By inheriting the behavior of a task, the program can be shared, and by inheriting the representation of a task (in terms of its flow), the structure (workflow) can be shared. The Process Handbook (Malone et al., in press) embodies concepts of specialization and decomposition to represent processes.

There are various approaches associated with binding a specific tool to an atomic task. A tool can be bound to a task statically at the compile time, or dynamically at the run time based on available resources and constraints. When a new tool is installed, designers should be able to modify the existing bindings. The simplest approach is to modify the source code or write a script file and recompile the system. The ideal case is plug and play, meaning that CAD vendors address the need of tool interoperability, e.g., the Tool Encapsulation Specification (TES) proposed by CFI (1995).

10.2.3.2 Workflow

To define a workflow, we must specify the tasks involved in the workflow, data, and their relationship. A set of workflows defined by methodology developers enforces the user to follow the flows imposed by the company or group. Flows may also serve to guide users in developing their own flows. Designers would retrieve the cataloged flows, modify them, and use them for their own purposes based on the guidelines imposed by the developer. It is necessary to generate legal flows. A blackboard approach was used in (Lander et al., 1996) to generate a particular flow suitable for a given task. In Nelsis (ten Bosch et al., 1991), branches of a flow are explicitly represented using ''or'' nodes and ''merge'' nodes. A task can be accomplished in various ways. It is necessary to represent alternative methodologies for the task succinctly so that designers can access alternative methodologies and select the best one based on what-if analysis. IDEF3.X (IDEF) is used to graphically model workflow in RASSP environment. Figure 10.2 shows an example of workflow using IDEF3.X. A node denotes a task. It has inputs, outputs, mechanisms, and conditions. IDEF definition has been around for 20 years mainly to capture flat modeling such as a shop floor process. IDEF specification, however, requires complete information such as control mechanisms and scheduling at the specification time, making the captured process difficult to understand. In IDEF, ''or'' nodes are used to represent the alternative paths. It does not have an explicit mechanism to represent alternative workflow. IDEF is ideal only for documenting the current practice and not suitable for executing iterative process which is determined during the execution of the process. Perhaps, the most important aspect missing from most process management systems is the abstraction mechanism (Schlenoff et al., 1996).

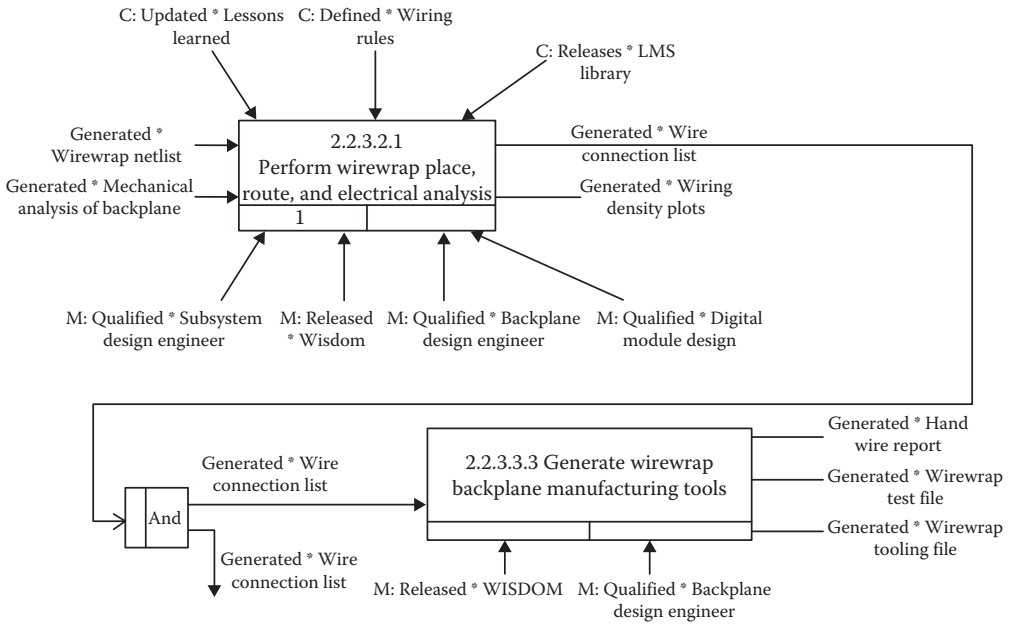


FIGURE 10.2 Workflow example using IDEF definition.

10.2.4 Execution Environment

The execution environment provides dynamic execution of tasks and tools and binds data to tools, either manually or automatically. Few frameworks separate the execution environment from the specification of design process. There are several modes in which a task can be executed (Kleinfeldth et al., 1994): manual mode, manual execution of flow, automatic flow execution, and automatic flow generation. In manual flow execution, the environment executes a task in the context of a flow. In an automatic flow execution environment, tasks are executed based on the order specified on the flow graph. In automatic flow generation, the framework generates workflow dynamically and executes them without the guidance of designers. Many frameworks use blackboard or knowledge-based approaches to generate workflow. However, it is important for designers to be able to analyze the workflow created and share it with others. That is, repeatability and predictability are important factors if frameworks support dynamic creation of workflow.

Each task may be associated with pre and postconditions. Before a task is executed, the precondition of the task is evaluated. If the condition is not satisfied, the framework either waits until the condition is met, or aborts the task and selects another alternative. After the task is executed, its postcondition is evaluated to determine if the result meets the exit criteria. If the evaluation is unsatisfactory, another alternative should be tried.

When a task is complex involving many subtasks and each subtask in turn has many alternatives, generating a workflow for the task that would successfully accomplish the task is not easy. If the first try of an alternative is not successful, another alternative should be tried. In some cases, backtrack occurs which nullifies all the executions of previous workflow.

10.2.5 Literature Surveys

Many systems have been proposed to generate design process (Knapp and Parker, 1991) and manage workflow (Dellen et al., 1997; Lavana et al., 1997; Schurmann and Altmeyer, 1997; Sutton and Director,

1998). Many of them use the Web technology to coordinate various activities in business (Andreoli, 1998), manufacturing (Berners-Lee et al., 1994; Cutkosy et al., 1996; Erkes et al., 1996), and microelectronic design (Rastogi et al., 1993; Chan et al., 1998). WELD (Chan et al., 1998) is a network infrastructure for a distributed design system that offers users the ability to create a customizable and adaptable virtual design system that can couple tools, libraries, design, and validation services. It provides support not only for designing but also for manufacturing, consulting, component acquisition, and product distribution, encompassing the developments of companies, universities, and individuals throughout the world. Lavana et al. (1997) proposed an Internet-based collaborative design. They use Petri nets as a modeling tool for describing and executing workflow. User teams, at different sites, control the workflow execution by selection of its path. Minerva II (Sutton and Director, 1998) is a software tool that provides design process management capabilities serving multiple designers working with multiple CAD frameworks. The proposed system generates design plan and realizes unified design process management across multiple CAD frameworks and potentially across multiple design disciplines. ExPro (Rastogi et al., 1993) is an expert-system-based process management system for the semiconductor design process.

There are several systems that automatically determine what tools to execute. OASIS (1992) uses Unix make file style to describe a set of rules for controlling individual design steps. The Design Planning Engine of the ADAM system (Knapp and Parker, 1986, 1991) produces a plan graph using a forward chaining approach. Acceptable methodologies are specified by listing preconditions and postconditions for each tool in a lisp-like language. Estimation programs are used to guide the chaining. Ulysses (Bushnell and Director, 1986) and Cadweld (Daniel and Director, 1991) are blackboard systems used to control design processes. A knowledge source, which encapsulates each tool, views the information on the blackboard and determines when the tool would be appropriate. The task management is integrated into the CAD framework and Task Model is interpreted by a blackboard architecture instead of a fixed inference mechanism. Minerva (Jacome and Director, 1992) and the OCT task manager (Chiueh and Katz, 1990) use hierarchical strategies for planning the design process. Hierarchical planning strategies take advantage of knowledge about how to perform abstract tasks which involve several subtasks.

To represent design process and workflow, many languages and schema have been proposed. NELSIS (ten Bosch et al., 1991) framework is based on a central, object-oriented database and on a flow management. It uses a dataflow graph as Flow Model and provides the hierarchical definition and execution of design flow. PLAYOUT (Schurmann and Altmeyer, 1997) framework is based on separate Task and Flow Models which are highly interrelated among themselves and the Product Model. In Barthelmann (1996), graph grammar is proposed in defining the task of software process management. Westfechtel (1996) proposed ''process-net'' to generate the process flow dynamically. However, in many of these systems, the relationship between task and data is not explicitly represented. Therefore, representing the case in which a task generates more than one datum and each of them goes to a different task is not easy. In Schurmann and Altmeyer (1997), Task Model (describing the I/O behavior of design tools) is used as a link between the Product Model and the Flow Model. The proposed system integrates data and process management to provide traceability. Many systems use IDEF to represent a process (Chung et al., 1996; Stavas et al.; IDEF). IDEF specification, however, requires complete information such as control mechanisms and scheduling at the specification time, making the captured process difficult to understand.

Although there are many other systems that address the problem of managing process, most proposed system use either a rule-based approach or a hard-coded process flow. They frequently require source code modification for any change in process. Moreover, they do not have mathematical formalism. Without the formalism, it is difficult to handle the iterative nature of the engineering process and to simulate the causal effects of any changes in parameters and resources. Consequently, coordinating the dynamic nature of processes is not well supported in most systems. It is difficult to analyze the rationale how an output is generated and where a failure has occurred. They also lack a systematic way of generating all permissible process flows at any level of abstraction while providing means to hide

the details of the flow when they are not needed. Most systems have the tendency to overspecify the flow information, requiring complete details of a process flow before executing the process. In most real situations, the complete flow information may not be known after the process has been executed: they are limited in their ability to address the underlying problem of process flexibility. They are rather rigid and not centered on users, and do not handle exceptions gracefully. Thus, the major functions for the collaborative framework such as adding new tools and sharing and improving the process flow cannot be realized. Most of them are weak in at least one of the following criteria suggested by NIST (Schlenoff et al., 1996): process abstraction, alternative tasks, complex groups of tasks, and complex sequences.

10.3 IMEDA System

The Internet-based Microelectronic Design Automation (IMEDA) System is a general management framework for performing various tasks in design and manufacturing of complex microelectronic systems. It provides a means to integrate many specialized tools such as CAD and analysis packages, and allows the designer to specify acceptable methodologies and tools together with information such as when and how they may be used. IMEDA is a collaborative engineering framework that coordinates design activities distributed geographically. The framework facilitates the flow of multimedia data sets representing design process, production, and management information among the organizational units of a virtual enterprise. IMEDA uses process grammar (Baldwin and Chung, 1995a) to represent the dynamic behavior of the design and manufacturing process. In a sense, IMEDA is similar to agent-based approach such as Redux (Petrie, 1996). Redux, however, does not provide process abstraction mechanism or facility to display the process flow explicitly.

The major functionality of the framework includes

- Formal representation of the design process using the process grammar that captures a complex sequence of activities of microelectronic design
- Execution environment that selects a process, elaborates the process, invokes tools, pre- and postevaluates the productions if the results meet the criterion, and notifies designers
- User interface that allows designers to interact with the framework, guides the design process, and edits the process and productions
- Tool integration and communication mechanism using Internet Socket and HTTP
- Access control that provides a mechanism to secure the activity and notification and approval that provide the mechanisms to disperse design changes to, and responses from, subscribers

IMEDA is a distributed framework design knowledge, including process information, manager programs, etc., are maintained in a distributed fashion by local servers. Figure 10.3 illustrates how IMEDA links tools and sites for distributed design activities. The main components of IMEDA are

- **System Cockpit:** It controls all interactions between the user and the system and between the system components. The cockpit will be implemented as a Java applet and may be executable on any platform for which a Java-enabled browser is available. It keeps track of the current design status and informs the user of possible actions. It allows users to collaboratively create and edit process flows, production libraries, and design data.
- **Manager Programs:** These encapsulate design knowledge. Using preevaluation functions, managers estimate the possibility of success for each alternative. They invoke tools and call postevaluation functions to determine if a tool's output meets the specified requirements. The interface servers allow cockpits and other Java-coded programs to view and manipulate production, task and design data libraries. Manager programs must be maintained by tool integrators to reflect site-specific information such as company design practices and different ways of installing tools.

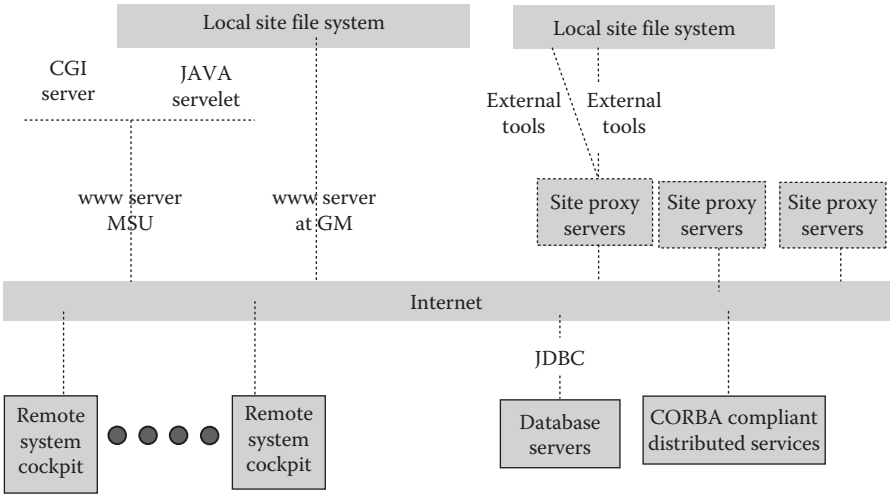


FIGURE 10.3 Architecture of IMEDA.

- **Browsers:** The task browser organizes the tasks in a GS hierarchy and contains all the productions available for each task. The data-specification browser organizes the data-specifications in a GS hierarchy and contains all the children.
- **External Tools:** These programs are the objects invoked by the framework during DM activities. Each atomic task in a process flow is bound to an external tool. External tools are written typically by the domain experts.
- **Site Proxy Server:** Any physical site that will host external tools must have a site proxy server running. These servers provide an interface between the cockpit and the external tools. The site server receives requests from system cockpits, and invokes the appropriate tool. Following the tool completion, the site server notifies the requesting cockpit, returning results, etc.
- **CGI Servers and Java Servlets:** The system cockpit may also access modules and services provided by CGI servers or the more recently introduced Java servlets. Currently, the system integrates modules of this type as direct components of the system (as opposed to external tools that may vary with the flow).
- **Database Servers:** Access to component data is a very important function. Using an API called JDBC, the framework can directly access virtually any commercially available database server remotely.
- **Whiteboard:** The shared cockpit or ''whiteboard'' is a communication medium to share information among users in a distributed environment. It allows designers to interact with the system and guides the design process collaboratively. Designers will be able to examine design results and current process flows, post messages, and carry out design activities both concurrently and collaboratively. Three types of whiteboards are the process board, the chat board, and the freeform drawing board. Their functionality includes (1) process board to the common process flow graph indicating the current task being executed and the intermediate results arrived at before the current task; (2) drawing board to load visual design data, and to design and simulate process; and (3) chat board to allow participants to communicate with each other via text-based dialog box.

IMEDA uses a methodology specification based on a process flow graphs and process grammars (Baldwin and Chung, 1995). Process grammars are the means for transforming high-level process flow graphs into progressively more detailed graphs by applying a set of substitution rules, called productions,

to nodes that represent logical tasks. It provides not only the process aspect of design activities but also a mechanism to coordinate them. The formalism in process grammar facilitates abstraction mechanisms to represent hierarchical decomposition and alternative methods, which enable designers to manipulate the process flow diagram and select the best method. The formalism provides the theoretical foundations for the development of IMEDA.

IMEDA contains the database of admissible flows, called process specifications. With the initial task, constraints, and execution environment parameters, including personal profile, IMEDA guides designers in constructing process flow graphs in a top-down manner by applying productions. It also provides designers with the ability to discover process configurations that provide optimal solutions. It maintains consistency among designs and allows the designer to interact with the system and adjust design parameters, or modify the previous design process. As the framework is being executed, a designer can be informed of the current status of design such as updating and tracing design changes and be able to handling exception.

Real-world processes are typically very complex by their very nature; IMEDA provides designers the ability to analyze, organize, and optimize processes in a way never before possible. More importantly, the framework can improve design productivity by accessing, reusing, and revising the previous process for a similar design.

The unique features of our framework include

Process Abstraction/Modeling: Process grammars provide abstraction mechanism for modeling admissible process flows. The abstraction mechanism allows a complex activity to be represented more concisely with a small number of higher-level tasks, providing a natural way of browsing the process repository. The strong theoretical foundation of our approach allows users to analyze and predict the behavior of a particular process. With the grammar, the process flow gracefully handles exceptions and alternative productions. When a task has alternative productions, backtracking occurs to select other productions.

Separation of Process Specification and Execution Environment: Execution environment information such as complex control parameters and constraints is hidden from the process specification. The information of these two layers is merely linked together to show the current task being executed on a process flow. The represented process flow can be executed in both automatic and manual modes. In the automatic mode, the framework executes all possible combinations to find a solution. In the manual mode, users can explore design space.

Communication and Collaboration: To promote real-time collaboration among participants, the framework is equipped with the whiteboard, a communication medium to share information. Users can browse related processes, compare them with other processes, analyze, and simulate them. Locally managed process flows and productions can be integrated by the framework in the central server. The framework manages the production rules governing the higher level tasks, while lower level tasks and their productions are managed by local servers. This permits the framework to be effective in orchestrating a large-scale activity.

Efficient Search of Design Process and Solution: IMEDA is able to select the best process and generate a process plan, or select a production dynamically and create a process flow. The process grammar easily captures design alternatives. The execution environment selects and executes the best one. If the selected process does not meet the requirement, then the framework backtracks and selects another alternative. This backtrack occurs recursively until a solution is found. If you allow a designer to select the best solution among many feasible ones, the framework may generate many multiple versions of the solution.

Process Simulation: The quality of a product depends on the tools (maturity, speed, and special strength of the tool), process (or workflow selected), and design data (selected from the reuse library). Our framework predicts the quality of results (product) and assesses the risk and reliability. This information can be used to select the best process/workflow suitable for a project.

Parallel Execution of Several Processes and Multiple Versions: To reduce the design time and risk, it is necessary to execute independent tasks in parallel whenever they are available. Sometimes, it is necessary to investigate several alternatives simultaneously to reduce the design time and risk. Or the designer may want to execute multiple versions with different design parameters. The key issue in this case is scheduling the tasks to optimize the resource requirements.

Life Cycle Support of Process Management: The process can be regarded as a product. A process (such as airplane designing or shipbuilding) may last many years. During this time, it may be necessary for the process itself to be modified because of new tools and technologies. Life cycle support includes updating the process dynamically, and testing/validating the design process, version history and configuration management of the design process. Tests and validations of the design processes, the simulation of processes, and impact analysis are necessary tools.

10.4 Formal Representation of Design Process*

IMEDA uses a methodology specification based on a process flow graphs and process grammars (Baldwin and Chung, 1995). The grammar is an extension of graph grammar originally proposed by Ehrig (1979) and has been applied to interconnection network (Derk and DeBrunner, 1998) and software engineering (Heiman et al., 1997).

10.4.1 Process Flow Graph

A process flow graph depicts tasks, data, and the relationships among them, describing the sequence of tasks for an activity. Three basic symbols are used to represent a process flow graph. Oval nodes represent logical tasks, two-concentric oval nodes represent atomic tasks, rectangular nodes represent data specifications and diamond nodes represent selectors. A task that can be decomposed into subtasks is called *logical*. Logical task nodes represent abstract tasks that could be done with several different tools or tool combinations. A task that cannot be decomposed is *atomic*. An atomic task node, commonly called a tool invocation, represents a run of an application program.

A *selector* is a task node that selects data or parameter. Data specifications are design data, where the output specification produced by a task can be consumed by another task as an input specification. Each data specification node, identified by a rectangle, is labeled with a data specification type. Using the graphical elements of the flow graph, engineers can create a process flow in a top-down fashion. These elements can be combined into a process flow graph using directed arcs. The result is a bipartite acyclic directed graph that identifies clearly the task and data flow relationships among the tasks in a design activity. The set of edges indicates those data specifications used and produced by each task. Each specification must have at most one incoming edge. Data specifications with no incoming edges are inputs of the design exercise. $T(G)$, $S(G)$, and $E(G)$ are the sets of task nodes, specification nodes, and edges of graph G, respectively. Figure 10.4 shows a process flow graph that describes a possible rapid prototyping design process, in which a state diagram is transformed into a field-programmable gate array (FPGA) configuration file.

The various specification types form a class hierarchy where each child is a specialization of the parent. There may be several incompatible children. For example, VHDL and Verilog descriptions are both children of simulation models. We utilize these specification types to avoid data format incompatibilities between tools (Figure 10.5a). Process flow graphs can describe design processes to varying levels of detail. A graph containing many logical nodes abstractly describes what should be done without describing how

* Materials in this section are excerpted from Baldwin, R. and Chung, M.J., *IEEE Comput.*, February, 54, 1995a. With permission.

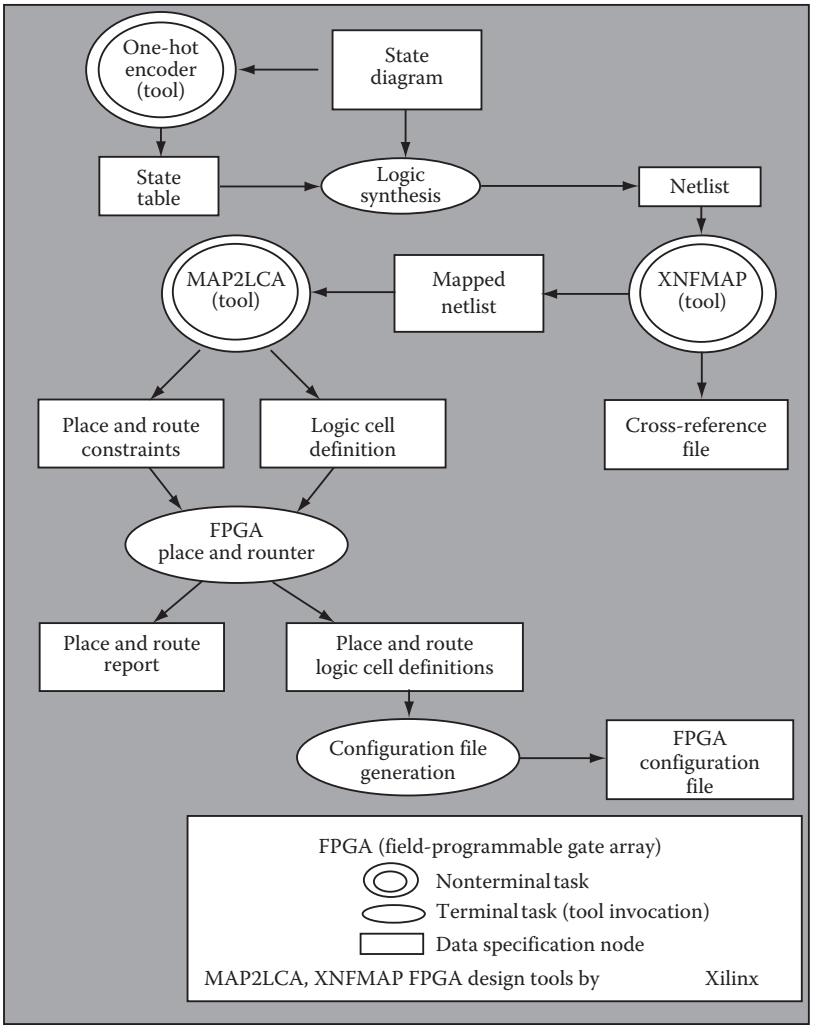


FIGURE 10.4 Sample process flow graph in which a state diagram is transformed into a field-programmable gate array configuration file.

it should be done (i.e., specifying which tools to use). Conversely, a graph in which all task nodes are atomic completely describes a methodology.

In our prototype, we use the following definitions: In(N) is the set of input nodes of node N: In(N) = $\{M \mid (M,N) \in E\}$. Out(N) is the set of output nodes of node N: Out(N) = $\{M \mid (N,M) \in E\}$. $I(G)$ is the set of input specifications of graph G: $\{N \in S(G) \mid \text{In}(N) = \emptyset\}$.

10.4.2 Process Grammars

The designer specifies the overall objectives with the initial graph that lists available input specifications, desired output specifications, and the logical tasks to be performed. By means of process grammars, logical task nodes are replaced by the flows of detailed subtasks and intermediate specifications. The output specification nodes are also replaced by nodes that may have a child specification type.

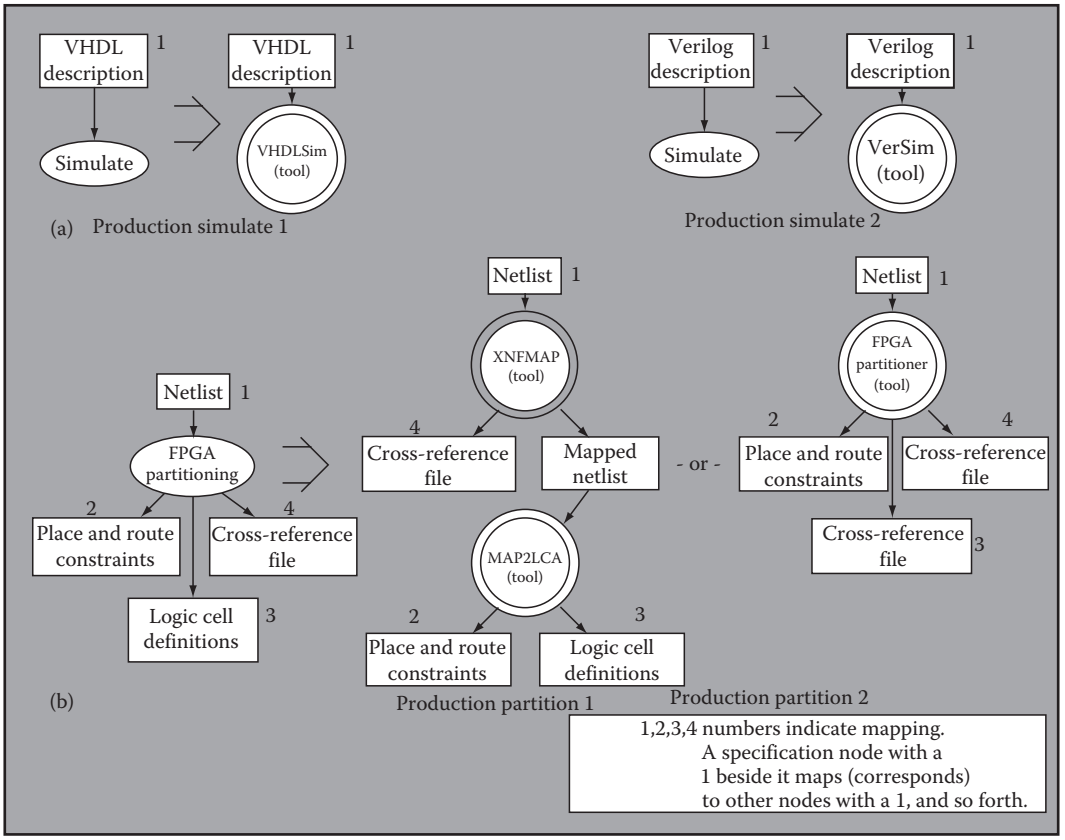


FIGURE 10.5 Graph production from a design process grammar. Two simulation alternatives based on input format are portrayed in (a); two partition alternatives representing different processes for an abstract task are portrayed in (b).

The productions in a graph grammar permit the replacement of one subgraph by another. A production in a design process grammar can be expressed formally as a tuple $P = (G_{LHS}, G_{RHS}, \sigma_{in}, \sigma_{out})$, where G_{LHS} and G_{RHS} are process flow graphs for the left side and the right side of the production, respectively, such that (1) G_{LHS} has one logical task node representing the task to be replaced, (2) σ_{in} is a mapping from the input specifications $I(G_{LHS})$ to $I(G_{RHS})$, indicating the relationship between two input specifications (each input specification of $I(G_{RHS})$ is a subtype of $I(G_{LHS})$), and (3) σ_{out} is a mapping from the output specifications of G_{LHS} to output specifications of G_{RHS} indicating the correspondence between them (each output specification must be mapped to a specification with the same type or a subtype). Figure 10.5 illustrates productions for two tasks, simulate and FPGA partitioning. The mappings are indicated by the numbers beside the specification nodes. Alternative productions may be necessary to handle different input specification types (as in Figure 10.5a), or because they represent different processes—separated by the word ''or''—for performing the abstract task (as in Figure 10.5b).

Let A be the logical task node in G_{LHS} and A' be a logical task node in the original process flow graph G such that A has the same task label as A'. The production rule P can be applied to A', which means that A' can be replaced with G_{RHS} only if each input and output specifications of A' matches to input and output specifications of G_{LHS}, respectively. If there are several production rules with the same left side flow graph, it implies that there are alternative production rules for the logical task. Formally, the production matches A' if

1. A' has the same task label as A.
2. There is a mapping ρ_{in}, from In(A) to In(A'), indicating how the inputs should be mapped. For all nodes $N \in$ In(A), $\rho_{in}(N)$ should have the same type as N or a subtype.
3. There is a mapping, ρ_{out}, from Out(A') to Out(A), indicating how the outputs should be mapped. For all nodes $N \in$ Out(A'), $\rho_{out}(N)$ should have the same type as N or a subtype.

The mappings are used to determine how edges that connected the replaced subgraph to the remainder should be redirected to nodes in the new subgraph. Once a match is found in graph G, the production is applied as follows:

1. Insert $G_{RHS} - I(G_{RHS})$ into G. The inputs of the replaced tasks are not replaced.
2. For every N in $I(G_{RHS})$ and edge (N,M) in G_{RHS}, add edge ($\rho_{in}[\sigma_{in}(N)]$,$M$) to G. That is to connect the inputs of A' to the new task nodes that will use them.
3. For every N in Out(A') and edge (N,M) in G, replace edge (N,M) with edge ($\sigma_{out}[\rho_{out}(N)]$,$M$). That is to connect the new output nodes to the tasks that will use them.
4. Remove A' and Out(A') from G, along with all edges incident on them.

Figure 10.6 illustrates a derivation in which the FPGA partitioning task is planned, using a production from Figure 10.5b.

The process grammar provides mechanism of specifying alternative methods for a logical task. A high-level flow graph can then be decomposed into detailed flow graphs by applying *production rules* to a logical task. A *production rule* is a substitution that permits the replacement of a logical task node with a flow graph that represents a possible way of performing the task. The concept of applying productions to

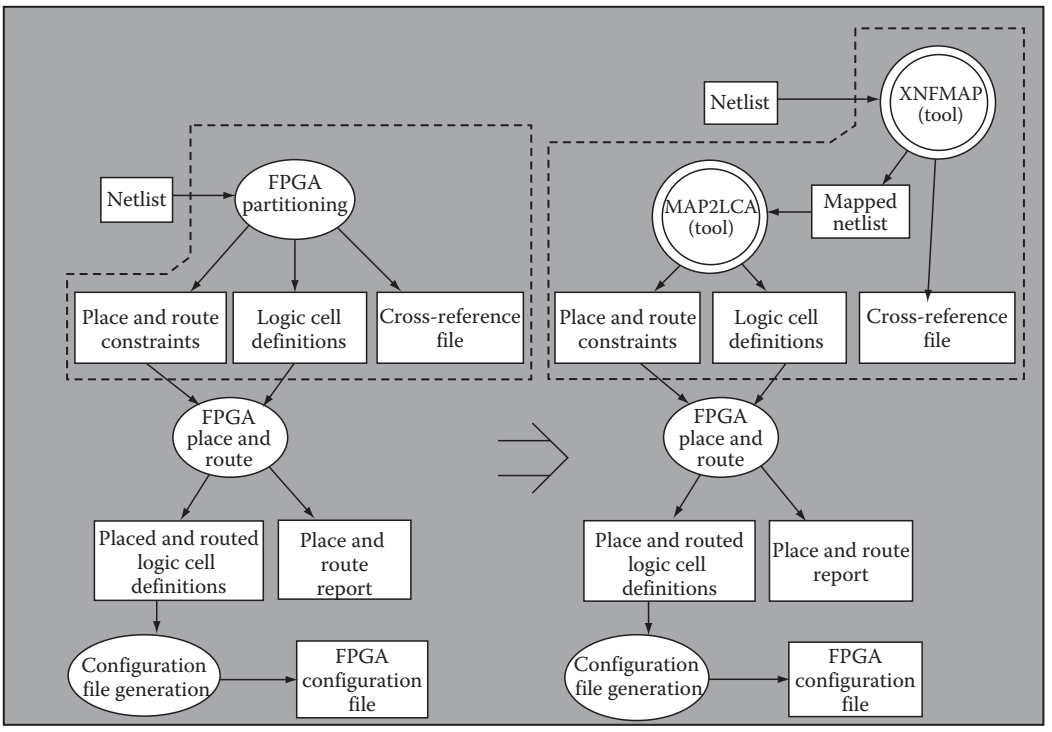


FIGURE 10.6 Sample graph derivation. Nodes in the outlined region, left, are replaced with nodes in the outlined region, right, according to production partition 1 in Figure 10.5.

logical tasks is somewhat analogous to the idea of productions in traditional (i.e., nongraph) grammars. In this sense, logical tasks correspond to *logical symbols* in grammar, and atomic tasks correspond to *terminal symbols*.

10.5 Execution Environment of the Framework

Figure 10.7 illustrates the architecture of our proposed system, which applies the theory developed in the previous section. Decisions to select or invoke tools are split between the designers and a set of manager programs, where manager programs are making the routine decisions and the designers make decisions that requires higher-level thinking. A program called Cockpit coordinates the interaction among manager programs and the designers. Tool sets and methodology preferences will differ among sites and over time. Therefore, our assumption is that each unit designates a person (or group) to act as system integrator, who writes and maintains the tool-dependent code in the system. We provide the tool-independent code and template to simplify the task of writing tool-dependent code.

10.5.1 Cockpit Program

The designer interacts with Cockpit, a program which keeps track of the current process flow graph and informs the designer of possible actions such as productions that could be applied or tasks that could be executed. Cockpit contains no task-specific knowledge; its information about the design process comes entirely from a file of graph productions. When new tools are acquired or new design processes are developed, the system integrator modifies this file by adding, deleting, and editing productions.

To assist the designer in choosing an appropriate action, Cockpit interacts with several manager programs which encapsulate design knowledge. There are two types of manager programs: task managers and production managers. Task managers invoke tools and determine which productions to execute for

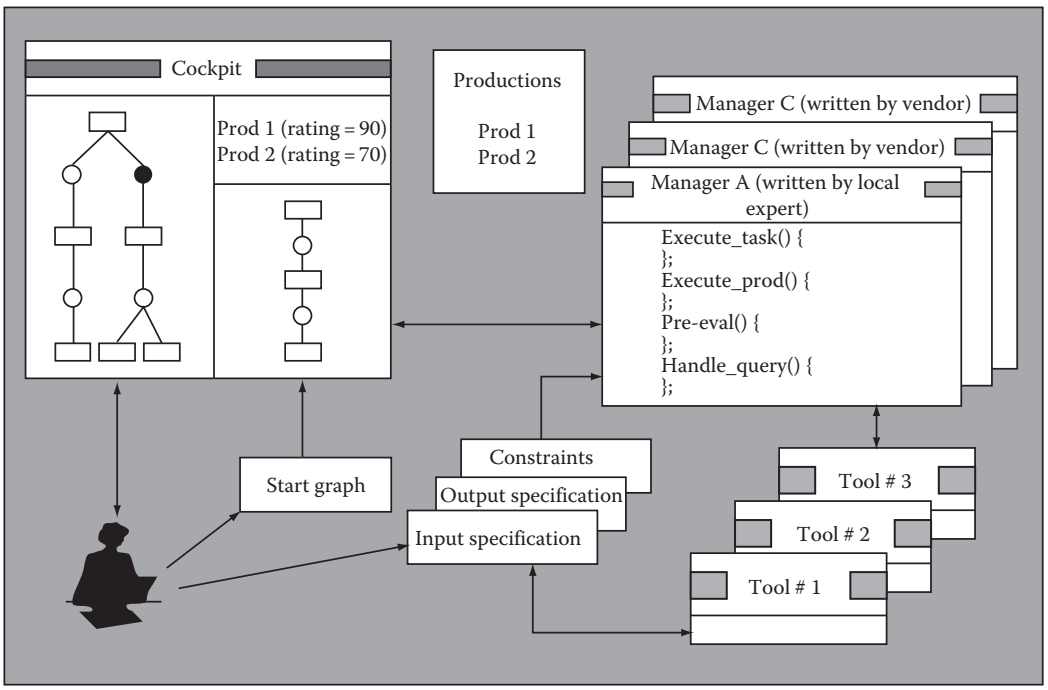


FIGURE 10.7 Proposed system based on Cockpit.

logical task nodes. Production managers provide ratings for the productions and schedule the execution of tasks on the right-hand side of the production. Managers communicate with each other using messages issued by Cockpit.

Our prototype system operates as follows. Cockpit reads the initial process flow graph from an input file generated by using a text editor. Cockpit then iteratively identifies when productions can be applied to logical task nodes and requests that the production managers assign the ratings to indicate how appropriate the productions are for those tasks. The process flow graph and the ratings of possible production applications are displayed for the designer, who directs Cockpit through a graphical user interface to apply a production or execute a task at any time. When asked to execute a task, Cockpit sends a message to a task manager. For an atomic task node, the task manager simply invokes the corresponding tool. For a logical task, the task manager must choose one or more productions, as identified by a Cockpit. The Cockpit applies the production and requests that the production manager executes it.

10.5.2 Manager Programs

Manager programs must be maintained by system integrators to reflect site-specific information, such as company design practices and tool installation methods. Typically, a manager program has its own thread. A Cockpit may have several manager programs, and therefore multithreads. We define a communication protocol between Cockpit and manager programs and provide templates for manager programs. The manager programs provide five operations: preevaluation, tool invocation, logical task execution, production execution, and query handling. Each operation described below corresponds to a C++ or Java function in the templates, which system integrators can customize as needed.

Preevaluation: Production managers assign ratings to help designers and task managers select the most appropriate productions. The rating indicates the likelihood of success from applying this production. The strategies used by the system integrator provide most of the code to handle the rating. In some cases, it may be sufficient to assign ratings statically, based on the success of past productions. These static ratings can be adjusted downward when the production has already been tried unsuccessfully on this task node (which could be determined using the query mechanism). Alternatively, the ratings may be an arbitrarily complex function of parameters obtained through the query mechanism or by examining the input files. Sophisticated manager programs may continuously gather and analyze process metrics that indicate those conditions leading to success, adjust adjusting ratings accordingly.

Tool Invocation: Atomic task mangers must invoke the corresponding software tool when requested by Cockpit, then determine whether the tool completed successfully. In many cases, information may be predetermined and entered in a standard template, which uses the tool's result status to determine success. In other cases, the manager must determine tool parameters using task-specific knowledge or determine success by checking task-specific constraints. Either situation would require further customization of the manager program.

Logical Task Execution: Logical task managers for logical tasks must select productions to execute the logical task. Cockpit informs the task manager of available productions and their ratings. The task manager can either direct Cockpit to apply and execute one or more productions, or it can decide that none of the productions is worthwhile and report failure. The task manager can also request that the productions be reevaluated when new information has been generated that might influence the ratings, such as a production's failure. If a production succeeds, the task manager checks any constraints; if they are satisfied, it reports success.

Production Execution: Production managers execute each task on the right-hand side of the production at the appropriate time and possibly check constraints. If one of the tasks fails or a constraint is violated, backtracking can occur. The production manager can use task-specific knowledge to determine which

tasks to repeat. If the production manager cannot handle the failure itself, it reports the failure to Cockpit, and the managers of higher level tasks and productions attempt to handle it.

Query Handling: Both production and task managers participate in the query mechanism. A production manager can send queries to its parent (the task manager for the logical task being performed) or to one of its children (a task manager of a subtask). Similarly, a task manager can send a query to its parent production manager or to one of its children (a production manager of the production it executed). The manager templates define C functions, which take string arguments, for sending these queries. System integrators call these functions but do not need to modify them. The manager templates also contain functions which are modified by system integrators for responding to queries. Common queries can be handled by template code; for example, a production manager can frequently ask its parent whether the production has already been attempted for that task and whether it succeeded. The manager template handles any unrecognized query from a child manager by forwarding it to the parent manager. Code must be added to handle queries for task-specific information such as the estimated circuit area or latency.

10.5.3 Execution Example

Now we describe a synthesis scenario that illustrates our prototype architecture in use. In this scenario, the objective is to design a controller from a state diagram, which will ultimately be done following the process flow graph in Figure 10.4. There are performance and cost constraints on the design, and the requirement to produce a prototype quickly. The productions used are intended to be representative but not unique. For simplicity, we assume that a single designer is performing the design with, therefore, only one Cockpit.

The start graph for this scenario contains only the primary task, chip synthesis, and specification nodes for its inputs and outputs (like the graph in the left in Figure 10.8). Cockpit tells us that the production of Figure 10.8 can be applied. We ask Cockpit to apply it. The chip synthesis node is then replaced by nodes for state encoding, logic synthesis, and physical synthesis, along with intermediate specification nodes.

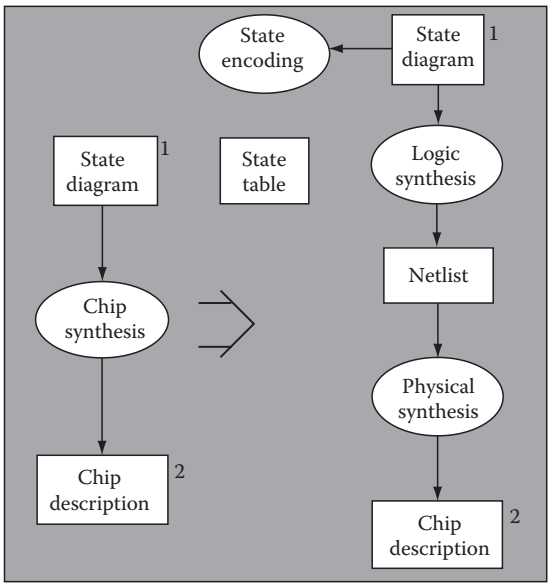


FIGURE 10.8 Productions for chip synthesis.

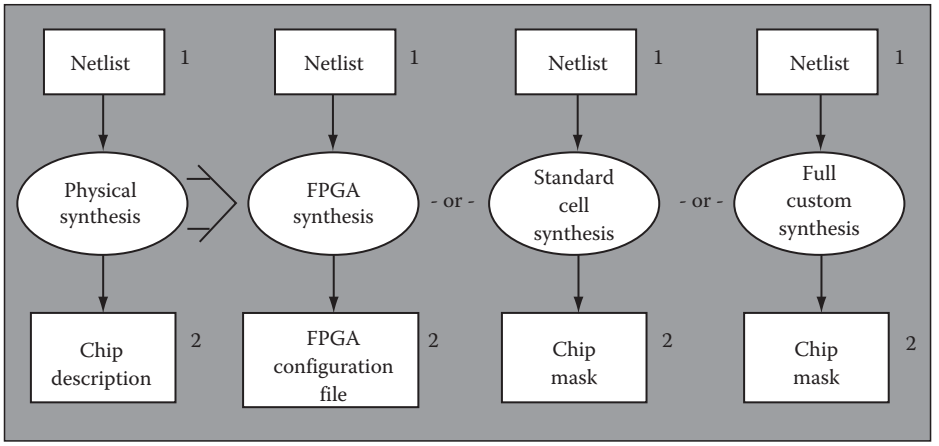


FIGURE 10.9 Productions for physical synthesis.

Next, we want to plan the physical synthesis task. Tasks can be planned in an order other than they are to be performed. Cockpit determines that any of the productions shown in Figure 10.9 may be applied, then queries each production's task manager program asking it to rate the production's appropriateness in the current situation. Based on the need to implement the design quickly, the productions for standard cell synthesis and full custom synthesis are rated low while the production for FPGA synthesis is rated high. Ratings are displayed to help us decide.

When we plan the state encoding task, Cockpit finds two productions: one to use the tool Minbits encoder and the other to use the tool One-hot encoder. One-hot encoder works well for FPGAs, while Minbits encoder works better for other technologies. To assign proper ratings to these productions, their production managers must find out which implementation technology will be used. First, they send a query to their parent manager, the state encoding task manager. This manager forwards the message to its parent, the chip synthesis production manager. In turn, this manager forwards the query to the physical synthesis task manager for an answer. All messages are routed by Cockpit, which is aware of the entire task hierarchy. This sequence of actions is illustrated in Figure 10.10.

After further planning and tool invocations, a netlist is produced for our controller. The next step is the FPGA synthesis task. We apply the production in Figure 10.11 and proceed to the FPGA partitioning task. The knowledge to automate this task has already been encoded into the requisite manager programs, so we direct Cockpit to execute the FPGA partitioning task. It finds the two productions illustrated in Figure 10.5b and requests their ratings, Next, Cockpit sends an execute message, along with the ratings, to the FPGA partitioning task manager. This manager's strategy is to always execute the highest-rated production, which in this case is production Partition 1. (Other task managers might have asked that both productions be executed or, if neither were promising, immediately reported failure.) This sequence of actions is shown in Figure 10.12.

Because the Partition 1 manager used an as-soon-as-possible task scheduling strategy, it asks Cockpit to execute XNFMAP immediately. The other subtask, MAP2LCA, is executed when XNFMAP complete successfully. After both tasks complete successfully, Cockpit reports success to the FPGA partitioning task manager. This action sequence is illustrated in Figure 10.13.

10.5.4 Scheduling

In this section, we describe a detailed description and discussion of auto-mode scheduling, including the implementation of the *linear scheduler*. The ability to search through the configuration space of a design process for a design configuration that meets user-specified constraints is important. For example,

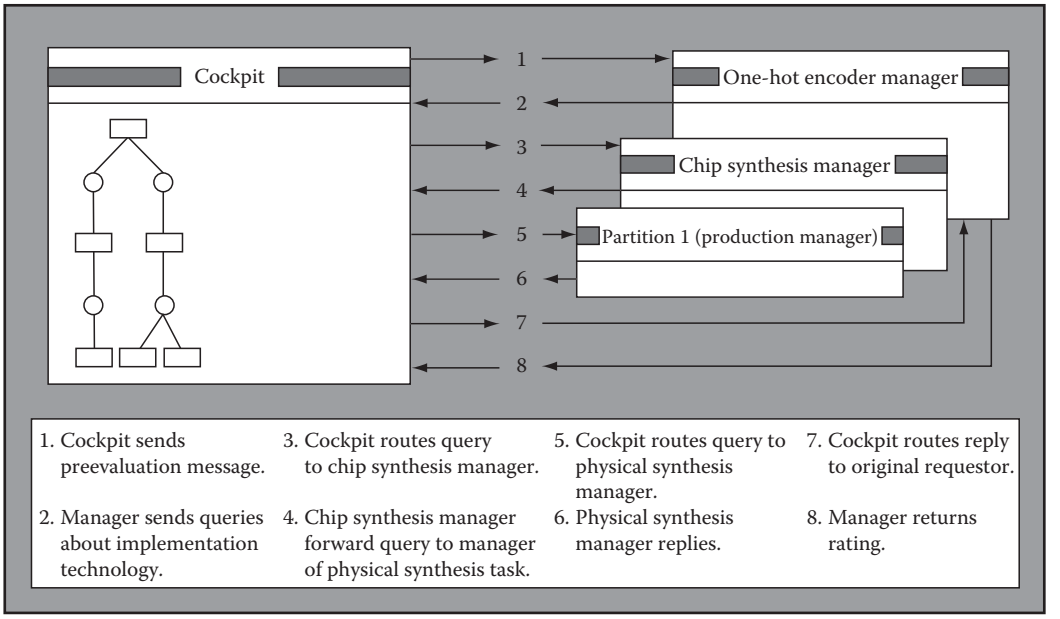


FIGURE 10.10 Sequence of actions for query handling.

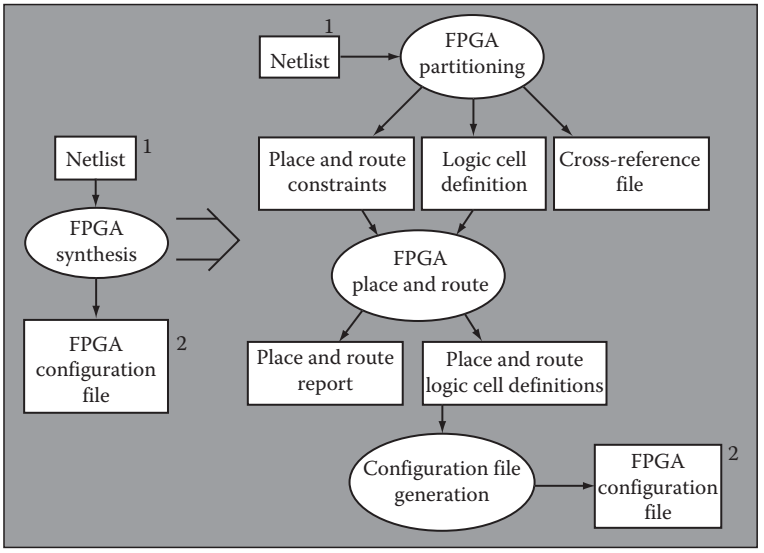


FIGURE 10.11 Production for field-programmable gate array.

assume that a user has defined a process for designing a digital filter with several different alternative ways of performing logical tasks such as ''FPGA Partitioning'' and ''Select the Filter Architecture.'' One constraint that an engineer may wish to place on the design might be: ''Find a process configuration that produces a filter that has maximum delay at most 10 ns.'' Given such a constraint, the framework must search through the configuration space of the filter design process, looking for a sequence of valid atomic tasks that produces a filter with ''maximum delay at most 10 ns.'' We call the framework component that

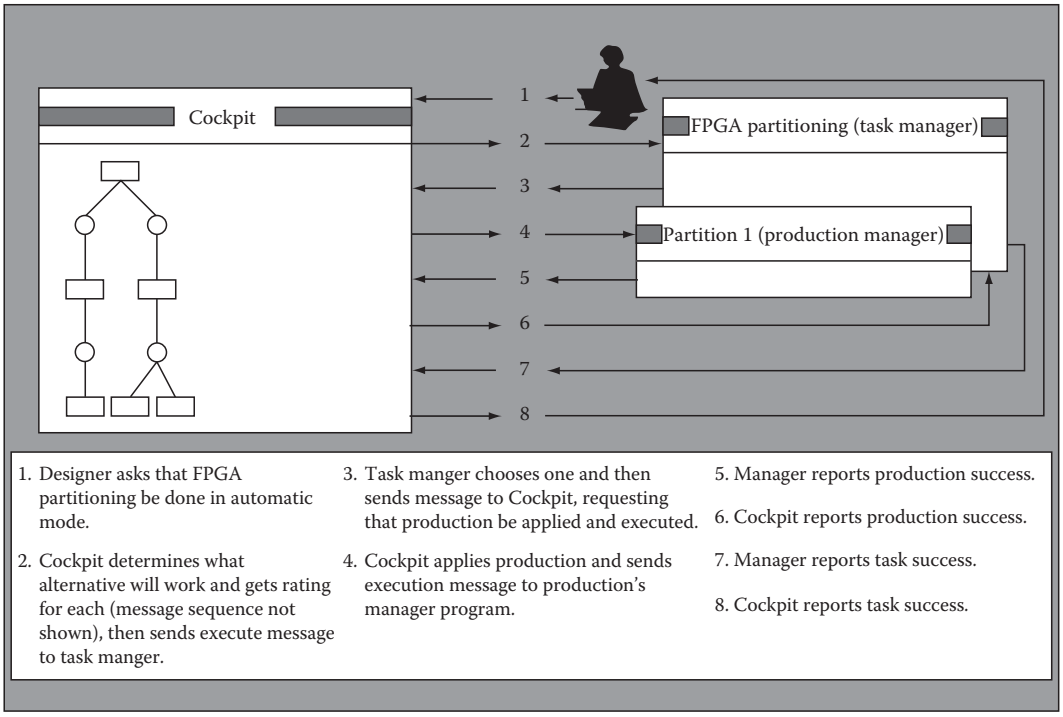


FIGURE 10.12 Sequence of action during automatic task execution.

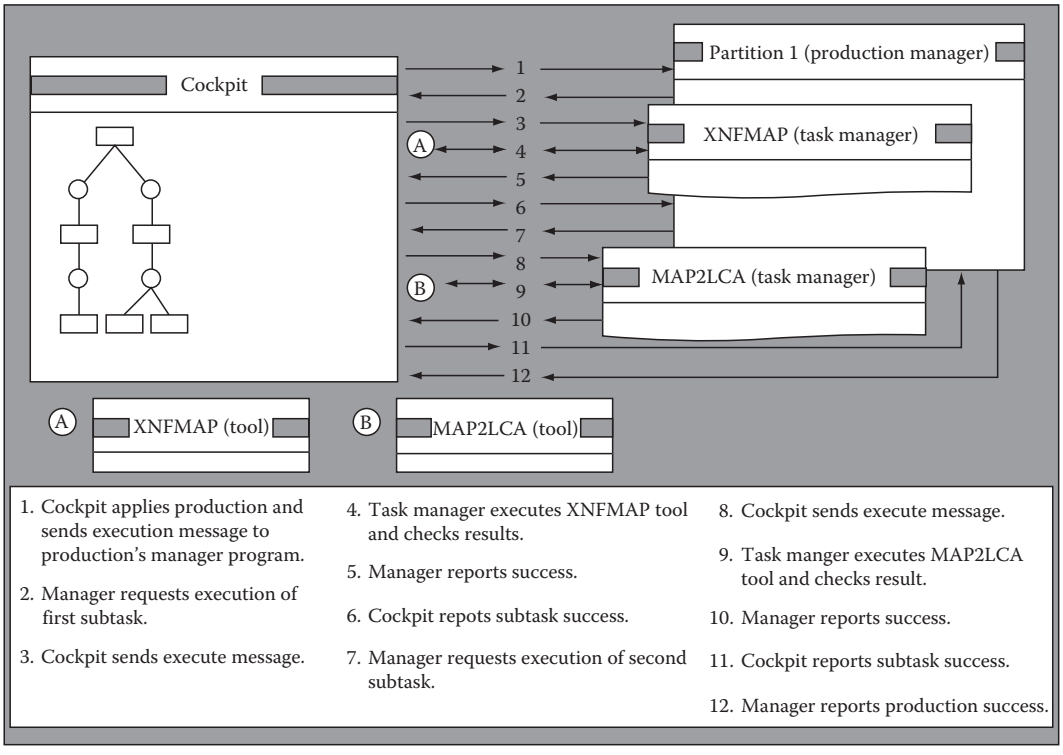


FIGURE 10.13 Sequence of actions during automatic production execution.

performs this search a *scheduler*. There are, of course, many different ways of searching through the design process configuration space. In general, a successful scheduler will provide the following functionality:

- **Completeness (Identification of Successful Configurations):** Given a particular configuration of a process, the correct scheduler will be able to conclusively determine whether the configuration meets user-specified constraints. The scheduler must guarantee that before reporting failure, all possible process configurations have been considered, and if there is a successful configuration, the algorithm must find it.
- **Reasonable Performance:** The configuration space of a process grows exponentially (in the number of tasks). Ideally, a scheduler will be able to search the configuration space using an algorithm that requires less than exponential time.

The Linear Scheduling Algorithm is very simple yet complete and meets most of the above criteria. In this algorithm, for each process flow graph (corresponding to an initial process flow graph or a production), it has a scheduler. Each scheduler is a separate thread with a *Task Schedule List* (TSL) representing the order in which tasks are to be executed. The tasks in a scheduler's TSL are called its children tasks. A scheduler also has a *task pointer* to indicate the child task being executed in the TSL. The algorithm is recursive such that with each new instantiation of a production of a given task, a new scheduler is created to manage the flow graph representing the production selected. A liner scheduler creates a TSL by performing a topological sort of the initial process flow graph and executes its children tasks in order. If a child task is atomic, the scheduler executes the task without creating a new scheduler; otherwise, it selects a new alternative, creates a new child scheduler to manage the selected alternative, and waits for a signal from the child scheduler indicating success or failure. When a child task execution is successful, the scheduler increments the *task pointer* in its TSL and proceeds to execute the next task. If a scheduler reaches the end of its TSL, it signals success to its own parent, and awaits signals from its parent if it should terminate itself (all successful) or rollback (try another to find new configurations). If a child task fails, the scheduler tries another alternative for the task. If there is no alternatives left, it rolls back (by decrementing the task pointer) until it finds a logical task that has another alternative to try. If a scheduler rolls back to the beginning of the TSL and cannot find an alternative, then its flow has failed. In this case, it signals a failure to its parent and terminates itself.

In the linear scheduling algorithm, each scheduler can send or receive any of five signals: PROCEED, ROLLBACK, CHILD-SUCCESS, CHILD-FAILURE, and DIE. These signals comprise scheduler-to-scheduler communication, including self-signaling. Each of the five signals is discussed below.

- **PROCEED:** This signal tells the scheduler to execute the next task in the TSL. It can be self-sent or received from a parent scheduler. For example, a scheduler increments its task pointer and sends itself a PROCEED signal when a child task succeeds, whereas it sends a PROCEED signal to its children to start its execution.
- **ROLLBACK:** This is signaled when a task execution has failed. This signal may be self-sent or received from a parent scheduler. Scheduler self-signals ROLLBACK whenever a child task fails. A Rollback can result in either trying the next alternative of a logical task, or decrementing the task pointer and trying the previous task in the TSL. If rollback results in decrementing the task pointer to point to a child task node which has received a success-signal, the parent scheduler will send a rollback signal to that child task scheduler.
- **CHILD-SUCCESS:** A child scheduler sends a CHILD-SUCCESS to its parent scheduler if it has successfully completed the execution of all of the tasks in its TSL. After sending the child-success signal, the scheduler remains active, listening for possible rollback signals from the parent. After receiving a child-success signal, parent schedulers self-send a proceed signal.
- **CHILD-FAILURE:** A child-failure signal is sent from a child scheduler to its parent in the event that the child's managed flow fails. After sending a child-failure signal, children

schedulers terminate. Upon receiving child-failure signals, parent scheduler self-send a rollback signal.

- **DIE:** This signal may be either self-sent, or sent from parent schedulers to their children schedulers.

10.6 Implementation

In this section, a high level description of the major components of IMEDA and their organization and functionality will be presented. Detailed explanations of the key concepts involved in the architecture of the Process Management Framework will also be discussed, including external tool integration, the tool invocation process, the Java File System, and state properties.

10.6.1 System Cockpit

The System Cockpit, as its name suggests, is shown in Figure 10.14 where nearly all user interaction with the framework takes place. It is here that users create, modify, save, load, and simulate process flow graphs representing design processes. This system component has been implemented as a Java applet. As such, it is possible to run the cockpit in any Java-enabled Web browser such as Netscape's Navigator

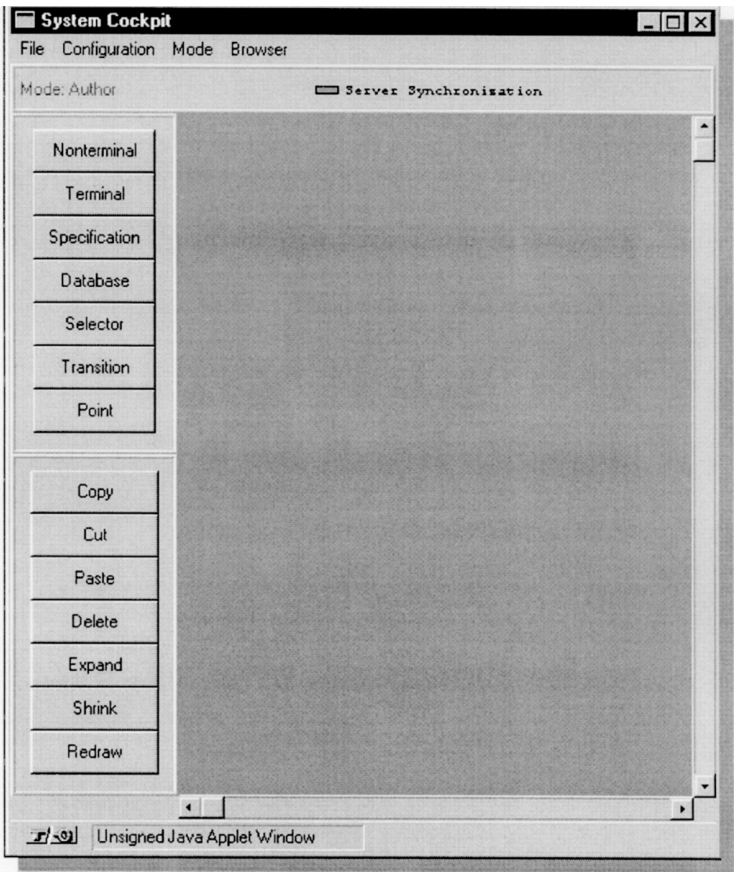


FIGURE 10.14 System Cockpit window.

or Microsoft's Internet Explorer. It is also possible to run the cockpit in some Java-enabled operating systems such as IBM's OS/2.

Each cockpit component also has the following components:

- **Root Flow:** Every cockpit has a Root Flow. The Root Flow is the flow currently being edited in the Cockpit's Flow Edit Panel. Notice that the Root Flow may change as a result of applying a production to a flow graph, in which case the Root Flow becomes a derivation of itself.
- **Flow Edit Panel:** The Flow Edit Panel is the interactive Graphical User Interface for creating and editing process flow graphs. This component also acts as a display for animating process simulations performed by various schedulers such as the manual or auto-mode linear scheduler.
- **Class Directory:** The Cockpit has two Class Directories: Task Directory and the Specification Directory. These directories provide the ''browser'' capabilities of the framework, allowing users to create reusable general-to-specific hierarchies of task classes. Class Directories are implemented using a *tree* structure.
- **Production Database:** The Production Database acts as a warehouse for logical task productions. These productions document the alternative methods available for completing a logical task. Each Production Database has a list of Productions. The Production Database is implemented as a tree-like structure, with Productions being on the root trunk, and Alternatives being leaves.
- **Browser:** Browsers provide the tree-like graphical user interface for users to edit both Class Directories and Databases. There are three Browsers: Database Browser for accessing the Production Database, Directory Browser for accessing the Task Directory, and Directory Browser for accessing the Spec Browser. Both Database Browsers and Directory Browsers inherit properties from object Browser, and offer the user nearly identical editing environments and visual representations. This deliberate consolidation of Browser interfaces allowed us to provide designers with an interface that was consistent and easier to learn.
- **Menu:** A user typically performs and accesses most of the system's key function from the Cockpit's Menu.
- **Scheduler:** The Cockpit has one or more schedulers. Schedulers are responsible for searching the configuration space of a design process for configurations that meet user specified design constraints. The Scheduler animates its process simulations by displaying them in the Flow Edit Panel of the Cockpit.

10.6.2 External Tools

External Tools are the concrete entities to which atomic tasks from a production flow are bound. When a flow task object is expanded in the Cockpit Applet (during process simulation), the corresponding external tool is invoked. The external tool uses a series of inputs and produces a series of outputs (contained in files). These inputs and outputs are similarly bound to specifications in a production flow. Outputs from one tool are typically used as inputs for another. IMEDA can handle the transfer of input and output files between remote sites. The site proxy servers, in conjunction with a remote file server (also running at each site) automatically handle the transfer of files from one system to another. External tools may be implemented using any language, and on any platform that has the capability of running a site server. While performing benchmark tests of IMEDA, we used external tools written in C, Fortran, Perl, csh (a Unix shell script), Java applications, and Mathematica scripts.

10.6.2.1 External Tool Integration

One of the primary functionality of IMEDA is the integration of user-defined external tools into an abstract process flow. IMEDA then uses these tools both in simulating the process flow to find a flow configuration that meets specific constraints, and in managing selected flow configurations during actual design execution.

There are two steps to integrating tools with a process flow defined in IMEDA: *association* and *execution*. Association involves ''linking'' or ''binding'' an abstract flow item (e.g., an atomic task) to an external tool. Execution describes the various steps that IMEDA takes to actually invoke the external tool and process the results.

10.6.2.2 Binding Tools

External tools may be bound to three types of flow objects: atomic tasks, selectors, and multiple version selectors. Binding an external tool to a flow object is a simple and straightforward job, involving simply defining certain properties in the flow object.

The following properties must be defined in an object that is to be bound to an external tool:

- **SITE:** Due to the fact that IMEDA can execute tools on remote systems, it is necessary to specify the site where the tool is located on. Typically, a default SITE will be specified in the system defaults, and making it unnecessary to define the site property unless the default is to be overriden. Note that the actual site ID specified by the SITE property must refer to a site that is running a site proxy server listening on that ID. See the ''Executing External Tools'' section below for more details.
- **CMDLINE:** The CMDLINE property specifies the command to be executed at the specified remote site. The CMDLINE property should include any switches or arguments that will always be sent to the external tool. Basically, the CMDLINE argument should be in the same format that would be used if the command were executed from a shell/DOS prompt.
- **WORKDIR:** The working directory of the tool is specified by the WORKDIR property. This is the directory in which IMEDA will actually execute the external tool, create temporary files, etc. This property is also quite often defined in the global system defaults, and thus may not necessarily have to be defined for every tool.
- **WRAPPERPATH:** The JDK 1.0.2 does not allow Java applications to execute a tool in an arbitrary directory. To handle remote tool execution, a wrapper is provided. It is a ''go-between'' program that would simply change directories and then execute the external tool. This program can be as simple as a DOS/NT batch file, a shell script, or a perl program. The external tool is wrapped in this simple script, and executed. Since IMEDA can execute tools at remote and heterogeneous sites, it was very difficult to create a single wrapper that would work on all platforms (WIN32, Unix, etc.). Therefor, the wrapper program may be specified for each tool, defined as global default, or a combination of the two.

Once the properties above have been defined for a flow object, the object is said to be ''bound'' to an external tool. If no site, directory, or filename is specified for the outputs of the flow object, IMEDA automatically creates unique file names, and stores the files in the working directory of the tool on the site that the tool was run. If a tool uses as inputs data items that are not specified by any other task, then the data items must be bound to static files on some site.

10.6.2.3 Executing External Tools

Once flow objects have been bound to the appropriate external tools, IMEDA can be used to perform process simulation or process management. IMEDA actually has several ''layers'' that lie between the Cockpit (a Java applet) and the external tool that is bound to a flow being viewed by a user in the Cockpit. A description of each of IMEDA components for tool invocations is listed below:

- **Tool Proxy:** The tool proxy component acts as a liaison between flow objects defined in Cockpits and the site proxy server. All communication is done transparently through the communication server utilizing TCP/IP sockets. The tool proxy ''packages'' information from Cockpit objects (atomic tasks, selectors, etc.) into string messages that the proxy server will recognize. It also listens

for and processes messages from the proxy server (through the communications server) and relays the information back to the Cockpit object that instantiated the tool proxy originally.

- **Communications Server:** Due to various security restrictions in the 1.0.2 version of Sun Microsystem's Java Development Kit (JDK), it is impossible to create TCP/IP socket connections between a Java applet and any IP address other than the address from which the applet was loaded. Therefore, it was necessary to create a ''relay server'' in order to allow cockpit applets to communicate with remote site proxy servers. The sole purpose of the communications server is to receive messages from one source and then to rebroadcast them to all parties that are connected and listening on the same channel.
- **Site Proxy Server:** Site proxy servers are responsible for receiving and processing invocation requests from tool proxies. When an invocation request is received, the site proxy server checks to see that the request is formatted correctly, starts a tool monitor to manage the external tool invocation, and returns the exit status of the external tool after it has completed.
- **Tool Monitors:** When the site proxy server receives an invocation request and invokes an external tool, it may take a significant amount of time for the tool to complete. If the proxy server had to delay the handling of other requests while waiting for each external tool to complete, IMEDA would become very inefficient. For this reason, the proxy server spawns a tool monitor for each external tool that is to be executed. The tool monitor runs as a separate thread, waiting on the tool, storing its stdout and stderr, and moving any input or output files that need moving to their appropriate site locations, and notifying the calling site proxy server when the tool has completed. This allows the site proxy server to continue receiving and processing invocation requests in a timely manner.
- **Tool Wrapper:** Tool wrapper changes directories into the specified WORKDIR, and then executes the CMDLINE.
- **External Tool:** External tools are the actual executable programs that run during a tool invocation. There is very little restriction on the nature of the external tools.

10.6.3 Communications Model

The Communications Model of IMEDA is perhaps the most complex portion of the system in some respects. This is where truly *distributed* communications come into play. One system component is communicating with another via network messages rather than function calls.

The heart of the communications model is the Communications Server. This server is implemented as a broadcast server. All incoming messages to the server are simply broadcast to all other connected parties. FlowObjects communicate with the Communications Server via ToolProxys. A ToolProxy allows a FlowObject to abstract all network communications and focus on the functionality of invoking tasks. A ToolProxy takes care of constructing a network message to invoke an external tool. That message is then sent to the Communications Server via a Communications Client. The Communication Client takes care of the low-level socket based communication complexities. Finally, the Communications Client sends the message to the Communications Server, which broadcasts the message to all connected clients. The client for which the message was intended (typically a site proxy server) decodes the message and, depending on its type, creates either a ToolMonitor (for an Invocation Message) or an External Redraw Monitor (for a Redraw Request).

The site proxy server creates these monitors to track the execution of external programs, rather than monitoring them itself. In this way, the proxy server can focus on its primary job—receiving and decoding network messages. When the monitors invoke an external tool, they must do so within a wrapper. Once the monitors have observed the termination of an external program, they gather any output on *stdout* or *stderr* and return these along with the exit code of the program to the site proxy server. The proxy server returns the results to the Communications Server, then the Communications Client, then the ToolProxy, and finally to the original calling FlowObject.

10.6.4 User Interface

The Cockpit provides both the user interface and core functionality of IMEDA. While multiple users may use different instances of the Cockpit simultaneously, there is currently no provision for direct collaboration between multiple users. Developing efficient means of real-time interaction between IMEDA users is one of the major thrusts of the next development cycle.

Currently the GUI of the Cockpit provides the following functionalities:

- **Flow Editing:** Users may create and edit process flows using the flow editor module of the Cockpit. The flow editor provides the user with a simple graphical interface that allows the use of a template of tools for ''drawing'' a flow. Flows can be optimally organized via services provided by a remote Layout Server written in Perl.
- **Production Library Maintenance:** The Cockpit provides functionality for user maintenance of collections of logical task productions, called libraries. Users may organize productions, modify input/output sets, or create/edit individual productions using flow editors.
- **Class Library Maintenance:** Users are provided with libraries of task and specification classes that are organized into a GS hierarchy. Users can instantiate a class into an actual task, specification, selector, or database when creating a flow by simply dragging the appropriate class from a class browser and dropping it onto a flow editor's canvas. The Cockpit provides the user with a simple tree structure interface to facilitate the creation and maintenance of class libraries.
- **Process Simulation:** Processes may be simulated using the Cockpit. The Cockpit provides the user with several scheduler modules that determine how the process configuration space will be explored. The schedulers control the execution of external tools (through the appropriate site proxy servers) and simulation display (flow animation for user monitoring of simulation progress). There are multiple schedulers for the user to choose from when simulating a process, including the manual scheduler, comprehensive linear scheduler, etc.
- **Process Archival:** The Cockpit allows processes to be archived on a remote server using the Java File System (JFS). The Cockpit is enabled by a JFS client interface to connect to a remote JFS server where process files are saved and loaded. While the JFS has its clear advantages, it is also awkward to not allow users to save process files, libraries, etc. on their local systems. Until version 1.1 of the Java Development Kit, local storage by a Java applet was simply not an option—the browser JVM definition did not allow access to most local resources. With version 1.1 of the JDK, however, comes the ability to electronically sign an applet. Once this has been done, users can grant privileged resource access to specific applets after a signature has been verified.

10.6.4.1 Design Flow Graph Properties

Initially, a flow graph created by a user using GUI is not associated with any system-specific information. For example, when a designer creates an atomic task node in a flow graph, there is initially no association with any external tool. The framework must provide a mechanism for users to bind flow graph entities to the external tools or activities that they represent.

We have used the concept of *properties* to allow users to bind flow graph objects to external entities. In an attempt to maintain flexibility, properties have been implemented in a very generic fashion. Users can define any number of properties for flow object. There are a number of key properties that the framework recognizes for each type of flow object. The user defines these properties to communicate needed configuration data to the framework.

A property consists of a *property label* and *property contents*. The label identifies the property, and consists of an alphanumeric string with no white space. The content of a property is any string. Currently users define properties using a freeform text input dialog, with each line defining a property. The first word on a line represents the property label, and the remainder of the line constitutes the property contents.

10.6.4.2 Property Inheritance

To further extend the flexibility of flow object properties, the framework requires that each flow object be associated with a *flow object class*. Classes allow designers to define properties that are common to all flow objects that inherit from that flow object class. Furthermore, classes are organized into a general-to-specific hierarchy, with children classes inheriting properties from parent classes.

Therefore, the properties of a particular class consist of any properties defined locally for that object, in addition to properties defined in the object's inherited class hierarchy. If a property is defined in both the flow object and one of its parent classes, the property definition in the flow object takes precedence. If a property is defined in more than one class in a class hierarchy, the ''youngest'' class (e.g., the child in a parent–child relationship) takes precedence.

Classes are defined in the Class Browsers of IMEDA. Designers that have identified a clear general-to-specific hierarchy of flow object classes can quickly create design flow graphs by dragging and dropping from class browsers onto flow design canvases. The user would then need only to overload those properties in the flow objects that are different from their respective parent classes (Figure 10.15).

For example, consider a class hierarchy of classes that all invoke the same external sort tool, but pass different flags to the tool, based on the context. It is likely that all of these tools will have properties in common, such as a common working directory and tool site. By defining these common properties in a common ancestor of all of the classes, such as *Search*, it is unnecessary to redefine the properties in the children classes.

Of course, children classes can define new properties that are not contained in the parent classes, and may also overload property definitions provided by ancestors. Following these rules, class *Insertion* would have the following properties defined: WORKDIR, SITE, WRAPPERPATH, and CMDLINE (Figure 10.16).

10.6.4.3 Macro Substitution

While performing benchmarks on IMEDA, one cumbersome aspect of the framework that users often pointed out was the need to reenter properties for tasks or specifications if, for example, a tool name or working directory changed. Finding every property that needed to be changed was a tedious job, and prone to errors.

In an attempt to deal with this problem, we came up with the idea of *property macros*. That is, a property macro is any macro that is not a key system macro. A macro is a textual substitution rule that

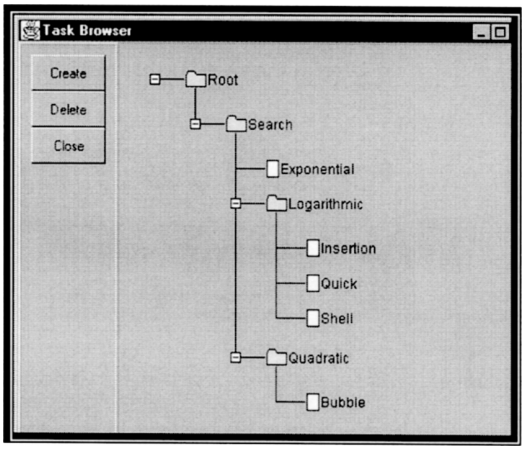


FIGURE 10.15 Task browser.

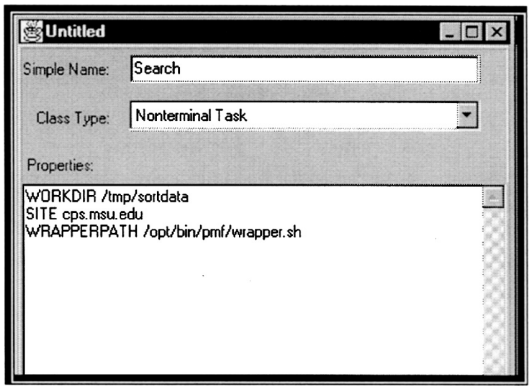


FIGURE 10.16 Property window and property inheritance.

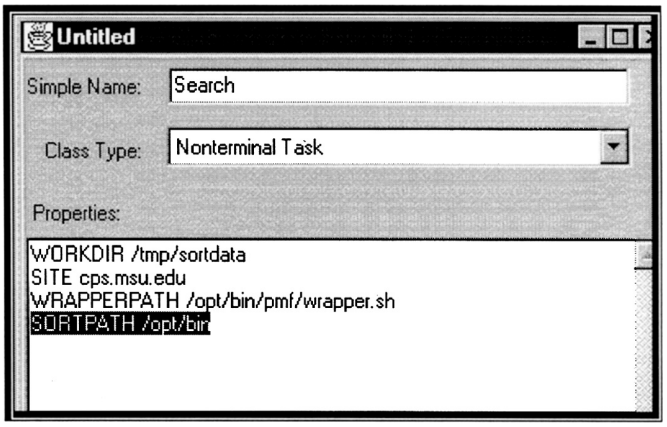


FIGURE 10.17 Macro definition.

can be created by users. By using macros in the property databases of flow objects, design flows can be made more flexible and more amiable to future changes.

As an example, consider a design flow that contains many atomic tasks bound to an external tool. Our previous example using searches is one possible scenario. On one system, the path to the external tool may be "/opt/bin/sort," while on another system the path is "/user/keyesdav/public/bin/sort." Making the flow object properties flexible is easy if a property macro named SORTPATH is defined in an ancestor of all affected flow objects (Figure 10.17). Children flow objects can then use that macro in place of a static path when specifying the flow object properties. As a further example, consider a modification to the previous "Search task hierarchy" where we define a macro SORTPATH in the Search class, and then use that macro in subsequent children classes, such as the Insertion class.

In the highlighted portion of the Property Database text area, a macro called "SORTPATH" is defined. In subsequent class' Property Databases, this macro can be used in place of a static path. This makes it easy to change the path for all tools that use the SORTPATH property macro—just the property database dialog where SORTPATH is originally defined needs to be modified (Figure 10.18).

10.6.4.4 Key Framework Properties

In our current implementation of IMEDA, there are a number of key properties defined. These properties allow users to communicate needed information to the framework in a flexible fashion. Most importantly,

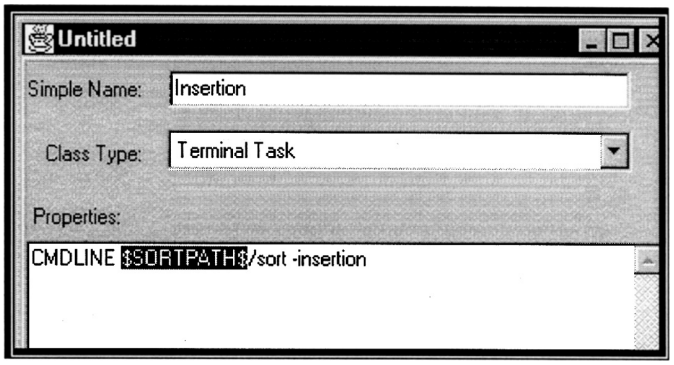


FIGURE 10.18 Macro substitution.

it allows system architects to define or modify system properties quickly. This is an important benefit when working with evolving software such as IMEDA.

10.7 Conclusion

Managing the design process is the key factor to improve the productivity in the microelectronic industry. We have presented an Internet-based Microelectronic Design Automation (IMEDA) framework to manage the design process. IMEDA uses a powerful formalism, called design process grammars, for representing design processes. We have also proposed an execution environment that utilizes this formalism to assist designers in selecting and executing appropriate design processes. The proposed approach is applicable not only in rapid prototyping but also in any environment where a design is carried out hierarchically and many alternative processes are possible.

The primary advantages of our system are

- **Formalism:** A strong theoretical foundation enables us to analyze how our system will operate with different methodologies.
- **Parallelism:** In addition to performing independent tasks within a methodology in parallel, our system also allows multiple methodologies to be executed in parallel.
- **Extensibility:** New tools can be integrated easily by adding productions and manager programs.
- **Flexibility:** Many different control strategies can be used. They can even be mixed within the same design exercise.

The prototype of IMEDA is implemented using Java. We are currently integrating more tools into our prototype system and developing manager program templates that implement more sophisticated algorithms for preevaluation, logical task execution, and query handling. Our system will become more useful as CAD vendors to adapt open software systems and allow greater tool interoperability.

References

Andreoli, J.-M., Pacull, F., and Pareschi, R., XPECT: A framework for electronic commerce, *IEEE Internet Comput.*, 1(4): 40–48, 1998.

Baldwin, R. and Chung, M.J., A formal approach to mangaging design processes, *IEEE Comput.*, 28: 54–63, February 1995a.

Baldwin, R. and Chung, M.J., Managing engineering data for complex products, *Res. Eng. Des.*, 7: 215–231, 1995b.

Barthelmann, K., Process specification and verification, *Lect. Notes Comput. Sci.*, 1073, 225–239, 1996.

Berners-Lee, T., Cailliau, R., Luotonen, A., Nielsen, H.F., and Secret, A., The World-Wide Web, *Commn. ACM*, 37(8): 76–82, 1994.

Bushnell, M.L. and Director, S.W., VLSI CAD tool integration using the Ulysses environment, *23rd ACM/IEEE Design Automation Conference*, Las Vegas, NV, pp. 55–61, 1986.

Casotto, A., Newton, A.R., and Sangiovanni-Vincentelli, A., Design management based on design traces, *27th ACM/IEEE Design Automation Conference*, Orlando, FL, pp. 136–141, 1990.

CFI, Tool Encapsulation Specification, Draft Standard, Version 2.0, released by the CFI TES Working Group, 1995.

Chan, F.L., Spiller, M.D., and Newton, A.R., WELD—An environment for web-based electronic design, *35th ACM/IEEE Design Automation Conference*, ACM Press, San Francisco, CA, pp. 146–151, June 1998.

Chiueh, T.F. and Katz, R.H., A history model for managing the VLSI design process, *International Conference on Computer-Aided Design*, Santa Clara, CA, pp. 358–361, 1990.

Chung, M.J. and Kim, S., An object-oriented VHDL environment, *Proceedings of the 27th ACM/IEEE Design Automation Conference*, Orlando, FL, June 24–28, IEEE Computer Society Press, pp. 431–436, 1990.

Chung, M.J. and Kim, S., Configuration management and version control in an object-oriented VHDL environment, *International Conference on Computer-Aided Design '91*, Santa Clara, CA, pp. 258–261, 1991.

Chung, M.J. and Kwon, P., A web-based framework for design and manufacturing a mechanical system, *1998 ASME Design Engineering Technical Conference (DETC) and Computer Engineering Conference*, Atlanta, GA, September 1998, paper number DAC-5599.

Chung, M.J., Charmichael, L., and Dukes, M., Managing a RASSP design process, *Comp. Ind.*, 30: 49–61, 1996.

Cutkosy, M.R., Tenenbaum, J.M., and Glicksman, J., Madefast: Collaborative engineering over the Internet, *Commn. ACM*, 39(9): 78–87, 1996.

Daniel, J. and Director, S.W., An object oriented approach to CAD tool control, *IEEE Trans. Comput. Aided Des.*, 10: 698–713, June 1991.

Dellen, B., Maurer, F., and Pews, G., Knowledge-based techniques to increase the flexibility of workflow management, *Data and Knowledge Engineering*, North-Holland Publishing Company, North-Holland, The Netherlands, 1997.

Derk, M.D. and DeBrunner, L.S., Reconfiguartion for fault tolerance using graph grammar, *ACM Trans. Comput. Syst.*, 16(1): 41–54, February 1998.

Di Janni, A., A monitor for complex CAD systems, *23rd Design Automation Conference*, Las Vegas, NV, pp. 145–151, 1986.

Ehrig, H., Introduction to the algebraic theory of graph grammars, *1st Workshop on Graph Grammars and Their Applications to Computer Science and Biology*, Springer, LNCS, pp. 1–69, 1979.

Erkes, J.W., Kenny, K.B., Lewis, J.W., Sarachan, B.D., Sobololewski, M.W., and Sum, R.N., Implementing shared manufacturing services on the World-Wide Web, *Commn. ACM*, 39(2): 34–45, 1996.

Fairbairn, D.G., 1994 keynote address, *31st Design Automation Conference*, San Diego, CA, pp. xvi–xvii, 1994.

Hardwick, M., Spooner, D.L., Rando, T., and Morris, K.C., Sharing manufacturing information in virtual enterprises, *Commn. ACM*, 39(2): 46–54, 1996.

Hawker, S., SEMATECH Computer Integrated Manufacturing (CIM) Framework Architecture Concepts, Principles, and Guidelines, Version 0.7.

Heiman, P. et al., Graph-based software process management, *Int. J. Software Eng. Knowledge Eng.*, 7(4): 1–24, December 1997.

Hines, K. and Borriello, G., A geographically distributed framework for embedded system design and validation, *35th Annual Design Automation Conference*, ACM Press, San Francisco, CA, pp. 140–145, June 1998.

Hsu, M. and Kleissner, C., Objectflow: Towards a process management infrastructure, *Distributed Parallel Databases*, 4: 169–194, 1996.

IDEF, http://www.idef.com.

Jacome, M.F. and Director, S.W., Design process management for CAD frameworks, *29th Design Automation Conference*, Anaheim, CA, pp. 500–505, 1992.

Jacome, M.F. and Director, S.W., A formal basis for design process planning and management, *IEEE Trans. Comput. Aided Des. Integrated Circuits Syst.*, 15(10): 1197–1211, October 1996.

Katz, R.H., Bhateja, R., E-Li Chang, E., Gedye, D., and Trijanto, V., Design version management, *IEEE Des. Test*, 4(1): 12–22, February 1987.

Kleinfeldth, S., Guiney, M., Miller, J.K., and Barnes, M., Design methodology management, *Proc. IEEE*, 82(2): 231–250, February 1994.

Knapp, D. and Parker, A., The ADAM design planning engine, *IEEE Trans. Comput. Aided Des. Integrated Circuits Syst.*, 10(7): 829–846, July 1991.

Knapp, D.W. and Parker, A.C., A design utility manager: The ADAM planning engine, *23rd ACM/IEEE Design Automation Conference*, Las Vegas, NV, pp. 48–54, 1986.

Knutilla, A., Schlenoff, C., Ray, S., Polyak, S.T., Tate, A., Chiun Cheah, S., and Anderson, R.C., Process specification language: An analysis of existing representations, NISTIR 6160, National Institute of Standards and Technology, Gaithersburg, MD, 1998.

Kocourek, C., An architecture for process modeling and execution support, *Comput. Aided Syst. Theory—EUROCAST*, Springer Berlin/Heidelberg, 1030: 199–216, 1995a.

Kocourek, C., Planning and execution support for design process, *IEEE International Symposium and Workshop on Systems Engineering of Computer Based System Proceedings*, 1995b.

Lander, S.E., Staley, S.M., and Corkill, D.D., Designing integrated engineering environments: Blackboard-based integration of design and analysis tools, *Concurrent Engineering*, 4(1): 59–71, 1996.

Lavana, H., Khetawat, A., Brglez, F., and Kozminski, K., Executable workflows: A paradigm for collaborative design on the Internet, *34th ACM/IEEE Design Automation Conference*, ACM Press, Anaheim Convention Center, Anaheim, CA, pp. 553–558, June 9–13, 1997.

Lyons, K., RaDEO Project Overview, http://www.cs.utah.edu/projects/alpha1/arpa/mind/index.html, 1997.

Malone, T.W., Crowston, K., and Herman, G.A. (Eds.), *The MIT Process Handbook*, MIT Press, Cambridge, MA, 2003.

OASIS, *OASIS Users Guide and Reference Manual*, MCNC, Research Triangle Park, NC, 1992.

Petrie, C.J., Agent based engineering, the Web, and intelligence, *Proceedings of IEEE Expert*, pp. 24–29, December 1996.

Rastogi, P., Koziki, M., and Golshani, F., ExPro: An expert system based process management system, *IEEE Trans. Semiconductor Manuf.*, 6(3): 207–218, 1993.

Schlenoff, C., Knutilla, A., and Ray, S., Unified process specification language: Requirements for modeling process, NISTIR 5910, National Institute of Standards and Technology, Gaithersburg, MD, 1996.

Schurmann, B. and Altmeyer, J., Modeling design tasks and tools—The link between product and flow model, *Proceedings of the 34th ACM/IEEE Design Automation Conference*, Anaheim, CA, pp. 564–569, June 1997.

Spiller, M.D. and Newton, A.R., EDA and network, *International Conference on Computer-Aided Design*, San Jose, CA, pp. 470–475, 1997.

Stavas, J., Wedgwood, J., Forte, M., Selvidge, W., Tuck, M.C., Bard, A., and Finnie, E., Workflow modeling for implementing complex, CAD-based, design methodologies, Proceedings of the Mentor Graphics Users Group International Conference, 1996.

Sutton, P.R. and Director, S.W., A description language for design process management, *33rd ACM/IEEE Design Automation Conference*, Las Vegas, NV, pp. 175–180, June 1996.

Sutton, P.R. and Director, S.W., Framework encapsulations: A new approach to CAD tool interoperability, *35th ACM/IEEE Design Automation Conference*, San Francisco, CA, pp. 134–139, June 15–19, 1998.

ten Bosch, K.O., Bingley, P., and Van der Wolf, P., Design flow management in the NELSIS CAD framework, *Proceedings of the 28th Design Automation Conference*, San Francisco, CA, pp. 711–716, 1991.

Toye, G., Cutkosky, M.R., Leifer, L.J., Tenenbaum, J.M., and Glicksman, J., SHARE: A methodology and environment for collaborative product development, *Proceedings of the 2nd Workshop Enabling Technology: Infrastructure Collaborative Enterprises*, Los Alamitos, CA, IEEE Computer Society Press, pp. 33–47, 1993.

Vogel, A. and Duddy, K., *Java Programming with CORBA*, Third Edition, Wiley Computer Publishing, New York, 2001.

Welsh, J., Kalathil, B., Chanda, B., Tuck, M.C., Selvidge, W., Finnie, E., and Bard, A., Integrated process control and data management in RASSP enterprise system, *Proceedings of 1995 RASSP Conference*, *Second Annual RASSP Conference*, Crystal City, VA, July 24–28, 1995.

Westfechtel, B., Integrated product and process management for engineering design applications, *Integrated Comput. Aided Eng.*, 3(1): 20–35, 1996.

Yang, Z. and Duddy, K., CORBA: A platform for distributed object computing, *ACM Oper. Syst. Rev.*, 30(2): 4–31, 1996.

11

System-Level Design

11.1 Introduction 11-1
Design Philosophies and System-Level Design • System Design Space
11.2 System Specification 11-3
11.3 System Partitioning 11-5
Constructive Partitioning Techniques • Iterative Partitioning Techniques
11.4 Scheduling and Allocating Tasks to Processing Modules 11-7
11.5 Allocating and Scheduling Storage Modules 11-8
11.6 Selecting Implementation and Packaging Styles for System Modules 11-9
11.7 Interconnection Strategy 11-9
11.8 Word-Length Determination 11-10
11.9 Predicting System Characteristics 11-10
11.10 Survey of Research in System Design 11-10
System Specification • Partitioning • Nonpipelined Design • Macropipelined Design • Genetic Algorithms • Imprecise Computation • Probabilistic Models and Stochastic Simulation • Performance Bounds Theory and Prediction • Word-Length Selection • Embedded Systems • System on Chip and Network on Chip
References 11-16

Alice C. Parker
University of Southern California

Yosef Tirat-Gefen
University of Southern California

Suhrid A. Wadekar
University of Southern California

11.1 Introduction

A *system* is a collection of interdependent operational components that together accomplish a complex task. Examples of systems range from cellular phones to camcorders to satellites. Projections point to a continuous increase in the complexity of systems in the coming years.

The term system, when used in the digital design domain, connotes many different entities. A system can consist of a processor, memory, and input/output, all on a single integrated circuit (IC), or can consist of a network of processors, geographically distributed, all performing a specific application. There can be a single clock, with modules communicating synchronously and multiple clocks with asynchronous communication or entirely asynchronous operation. The design can be general purpose, or specific to a given application, i.e., application-specific. The above variations together constitute the *system style*. System style selection is determined to a great extent by the physical technologies used, the environment in which the system operates, designer experience, and corporate culture, and is not automated to any great extent.

System-level design covers a wide range of design activities and design situations. It includes the more specific activity *system engineering*, which involves the requirements, development, test planning,

subsystem interfacing, and end-to-end analysis of systems. System-level design is sometimes called *system architecting*, a term used widely in the aerospace industry.

General-purpose system-level design involves the design of programmable digital systems including the basic modules containing storage, processors, input/output, and system controllers. At the system level, the design activities include determining the following:

- *Power budget* (the amount of power allocated to each module in the system)
- Cost and performance budget allocated to each module in the system
- Interconnection strategy
- Selection of commercial off-the-shelf (COTS) modules
- Packaging of each module
- Overall packaging strategy
- Number of processors, storage units, and input/output interfaces required
- Overall characteristics of each processor, storage unit, and I/O interface

For example, memory system design focuses on the number of memory modules required, how they are organized, and the capacity of each module. A specific system-level decision in this domain can be how to partition the memory between the processor chip and the off-chip memory. At a higher level, a similar decision might involve configuration of the complete storage hierarchy, including memory, disk drives, and archival storage.

For each general-purpose system designed, many more systems are designed to perform specific applications. *Application-specific system design* involves the same activities as described above, but can involve many more decisions, since there are usually more custom logic modules involved. Specifications for application-specific systems contain not only requirements on general capabilities, but also contain the functionality required in terms of specific tasks to be executed. Major application-specific system-level design activities include not only the above general-purpose system design activities, but also the following activities:

- Partitioning an application into multiple functional modules
- Scheduling the application tasks on shared functional modules
- Allocating functional modules to perform the application tasks
- Allocating and scheduling storage modules to contain blocks of data as they are processed
- Determining the implementation styles of functional modules
- Determining the word lengths of data necessary to achieve a given accuracy of computation
- Predicting resulting system characteristics once the system design is complete

Each of the system design tasks given in the two lists above will be described in detail below. Since the majority of system design activities are application-specific, this section will focus on system-level design of application-specific systems. Related activities, hardware–software codesign, verification, and simulation are covered in other sections.

11.1.1 Design Philosophies and System-Level Design

Many design tools have been constructed with a *top-down design* philosophy. Top-down design represents a design process whereby the design becomes increasingly detailed until final implementation is complete. Considerable prediction of resulting system characteristics is required to make the higher-level decisions with some degree of success.

Bottom-up design, on the other hand, relies on designing a set of primitive elements, and then forming more complex modules from those elements. Ultimately, the modules are assembled into a system. At each stage of the design process, there is complete knowledge of the parameters of the lower-level elements. However, the lower-level elements may be inappropriate for the tasks at hand.

System designers in industry describe the design process as being much less organized and considerably more complex than the top-down and bottom-up philosophies suggest. There is a mixture of top-down and bottom-up activities with major bottlenecks of the system receiving detailed design consideration while other parts of the system still exist only as abstract specifications. For this reason, the system-level design activities we present in detail here support such a complex design situation. Modules, elements, and components used to design at the system level might exist, or might only exist as abstract estimates along with requirements. The system can be designed after all modules have been designed and manufactured, prior to any detailed design, or with a mixture of existing and new modules.

11.1.2 System Design Space

System design, like data path design, is quite straightforward as long as the constraints are not too severe. However, most designs must solve harder problems than problems solved by existing systems. Designers must race to produce working systems faster than competitors, systems that are also less expensive. More variations in design are possible than ever before and such variations require a large *design space* to be explored. The dimensions of the design space (its axes) are system properties such as cost, power, design time, and performance. The design space contains a population of designs, each of which possesses different values of these system properties. There are literally millions of system designs for a given specification, each of which exhibits different cost, performance, power consumption, and design time. Straightforward solutions that do not attempt to optimize system properties are easy to obtain, but may be inferior to designs that are produced by system-level design tools and have undergone many iterations of design. The complexity of system design is not because system design is an inherently difficult activity, but because so many variations in design are possible and time does not permit exploration of all of them.

11.2 System Specification

Complete system specifications contain a wide range of information, including

- Constraints on the system power, performance, cost, weight, size, and delivery time
- Required functionality of the system components
- Any required information about the system structure
- Required communication between system components
- Flow of data between components
- Flow of control in the system
- Specification of input precision and desired output precision

Most systems specifications that are reasonably complete exist first in a natural language. However, natural language interfaces are not currently available with commercial system-level design tools.

More conventional system specification methods used to drive system-level design tools include formal languages, graphs, and a mixture of the two. Each of the formal system specification methods described here contains some of the information found in a complete specification, i.e., most specification methods are incomplete. The designer can provide the remaining information necessary for full system design interactively, can be entered later in the design process, or can be provided in other forms at the same time the specification is processed. The required design activities determine the specification method used for a given system design task.

There are no widely adopted formal languages for system-level hardware design although System-Level Design Language (SLDL) was developed by an industry group. Hardware descriptive languages such as VHSIC Verilog Hardware Descriptive Language (VHDL) [1] and Verilog [2] are used to describe the functionality of modules in an application-specific system. High-level synthesis tools can then synthesize

such descriptions to produce register-transfer designs. Extensions of VHDL and Verilog have been proposed to encompass more system-level design properties. Apart from system constraints, VHDL specifications can form complete system descriptions. However, the level of detail required in VHDL and to some extent in Verilog requires the designer to make some implementation decisions. In addition, some information that is explicit in more abstract specifications such as the flow of control between tasks is implicit in HDLs.

Graphical tools have been used for a number of years to describe system behavior and structure. *Block diagrams* are often used to describe system structure. Block diagrams assume that tasks have already been assigned to basic blocks and their configuration in the system has been specified. Block diagrams generally cannot represent the flow of data or control, or design constraints. The processor memory switch (PMS) notation invented by Bell and Newell was an early attempt to formalize the use of block diagrams for system specification [3].

Petri nets have been used for many years to describe system behavior using a *token-flow* model. A token-flow model represents the flow of control with tokens, which flow from one activity of the system to another. Many tokens can be active in a given model concurrently, representing asynchronous activity and parallelism, important in many system designs. Timed Petri nets have been used to model system performance, but Petri nets cannot easily be used to model other system constraints, system behavior, or any structural information.

State diagrams and graphical tools such as *State charts* [4] provide alternative methods for describing systems. Such tools provide mechanisms to describe the flow of control, but do not describe system constraints, system structure, data flow, or functionality.

Task-flow graphs, an outgrowth from the control/data-flow graphs (CDFG) used in high-level synthesis are often used for system specification. These graphs describe the flow of control and data between tasks. When used in a hierarchical fashion, task nodes in the task-flow graph can contain detailed functional information about each task, often in the form of a CDFG. Task-flow graphs contain no mechanisms for describing system constraints or system structure.

Spec charts [5] incorporate VHDL descriptions into state-chart-like notation, overcoming the lack of functional information found in state charts.

Figure 11.1 illustrates the use of block diagrams, Petri nets, task-flow graphs, and spec charts.

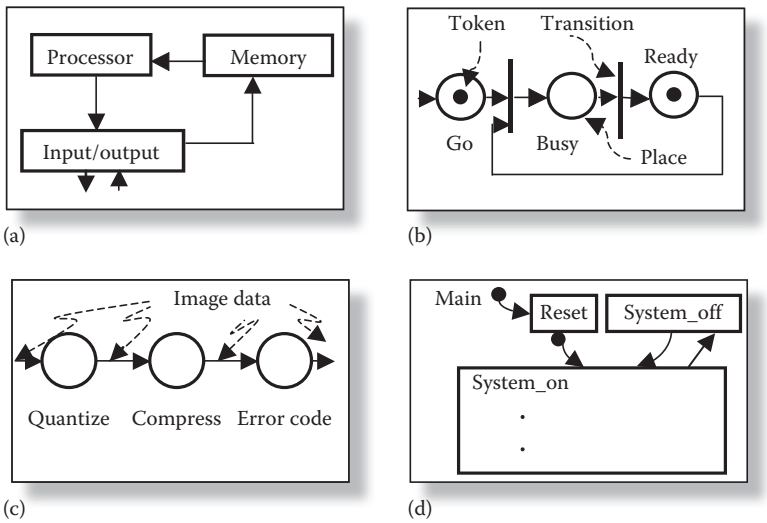


FIGURE 11.1 Use of (a) block diagrams, (b) Petri nets, (c) task-flow graphs, and (d) spec charts, shown in simplified form.

11.3 System Partitioning

Most systems are too large to fit on a single substrate. If the complexity of the system tasks and the capacities of the system modules are of the same order, then partitioning is not required. All other systems must be partitioned so that they fit into the allowed dies, packages, boards, multichip modules (MCMs), and cases. *Partitioning determines the functions, tasks, or operations in each partition of a system*. Each partition can represent a substrate, package, MCM, or larger component. Partitioning is performed with respect to a number of goals, including minimizing cost, design time or power, or maximizing performance. Any of these goals can be reformulated as specific constraints such as meeting given power requirements.

When systems are partitioned, resulting communication delays must be taken into account, affecting performance. Limitations on interconnection size must be taken into account, affecting performance as well. Pin and interconnection limitations force the multiplexing of inputs and outputs, reducing performance and sometimes affecting cost. Power consumption must also be taken into account. Power balancing between partitions and total power consumption might both be considerations. To meet market windows, system partitions can facilitate the use of COTS, programmable components, or easily fabricated components such as gate arrays. To meet cost constraints, functions that are found in the same partition might share partition resources. Such functions or tasks cannot execute concurrently, affecting performance.

Partitioning is widely used at the logic level as well as on physical designs. In these cases, much more information is known about the design properties and the interconnection structure has been determined. System partitioning is performed when information about properties of the specific components might be uncertain, and the interconnection structure undetermined. For these reasons, techniques used at lower levels must be modified to include predictions of design properties not yet known and prediction of the possible interconnection structure as a result of the partitioning.

The exact partitioning method used depends on the type of specification available. If detailed CDFG or HDL specifications are used, the partitioning method might be concerned with which register-transfer functions (e.g., add, multiply, and shift) are found in each partition. If the specification primitives are tasks, as in a task-flow graph specification, then the tasks must be assigned to partitions. Generally, the more detailed the specification, the larger the size of the partitioning problem. Powerful partitioning methods can be applied to problems of small size ($n < 100$). Weaker methods such as incremental improvement must be used when the problem size is larger.

Partitioning methods can be based on *constructive* partitioning or *iterative improvement*. Constructive partitioning involves taking an unpartitioned design and assigning operations or tasks to partitions. Basic constructive partitioning methods include *bin packing* using a first-fit decreasing heuristic, *clustering* operations into partitions by assigning nearest neighbors to the same partition until the partition is full, *random* placement into partitions, and *integer programming* approaches.

11.3.1 Constructive Partitioning Techniques

Bin packing involves creating a number of bins equal in number to the number of partitions desired and equal in size to the size of partitions desired. One common approach involves the following steps: the tasks or operations are sorted by size. The largest task in the list is placed in the first bin, and then the next largest is placed in the first bin, if it will fit, or else into the second bin. Each task is placed into the first bin in which it will fit, until all tasks have been placed in bins. More bins are added if necessary. This simple heuristic is useful to create an initial set of partitions to be improved iteratively later.

Clustering is a more powerful method to create partitions. Here is a simple clustering heuristic. Each task is ranked by the extent of ''connections'' to other tasks either owing to control flow, data flow, or physical position limitations. The most connected task is placed in the first partition and then the tasks connected to it are placed in the same partition, in the order of the strength of their connections to the

first task. Once the partition is full, the task with the most total connections remaining outside a partition is placed in a new partition, and other tasks are placed there in the order of their connections to the first task. This heuristic continues until all tasks are placed.

Random partitioning places tasks into partitions in a greedy fashion until the partitions are full. Some randomization of the choice of tasks is useful in producing a family of systems, each member of which is partitioned randomly. This family of systems can be used successfully in iterative improvement techniques for partitioning, as described later in this section.

The most powerful technique for constructive partitioning is mathematical programming. Integer and mixed integer-linear programming (MILP) techniques have been used frequently in the past for partitioning. Such powerful techniques are computationally very expensive and are successful only when the number of objects to be partitioned is small. The basic idea behind integer programming used for partitioning is the following: an integer, TP(i, j), is used to represent the assignment of tasks to partitions. When TP = 1, task i is assigned to partition j. For each task in this problem, there would be an equation

$$\sum_{j=1}^{\text{Partition total}} \text{TP}(i,j) = 1 \tag{11.1}$$

This equation states that each task must be assigned to one and only one partition. There would be many constraints of this type in the integer program, some of which were inequalities. There would be one function representing cost, performance, or other design property to be optimized. The simultaneous solution of all constraints, given some minimization or maximization goal, would yield the optimal partitioning.

Apart from the computational complexity of this technique, the formulation of the mathematical programming constraints is tedious and error prone if performed manually. The most important advantage of mathematical programming formulations is the discipline it imposes on the CAD programmer in formulating an exact definition of the CAD problem to be solved. Such problem formulations can prove useful when applied in a more practical environment, as described in Section 11.3.2.

11.3.2 Iterative Partitioning Techniques

Of the many iterative partitioning techniques available, two have been applied most successfully at the system level. These are *min-cut partitioning*, first proposed by Kernigan and Lin, and *genetic algorithms*.

Min-cut partitioning involves exchanging tasks or operations between partitions to minimize the total amount of ''interconnections'' cut. The interconnections can be computed as the sum of data flowing between partitions, or as the sum of an estimate of the actual interconnections that will be required in the system. The advantage of summing the data flowing is that it provides a quick computation, since the numbers are contained in the task-flow graph. Better partitions can be obtained if the required physical interconnections are taken into account since they are related more directly to cost and performance than is the amount of data flowing. If a partial structure exists for the design, predicting the unknown interconnections allows partitioning to be performed on a mixed design, one that contains existing parts as well as parts under design.

Genetic algorithms, highly popular for many engineering optimization problems, are especially suited to the partitioning problem. The problem formulation is similar in some ways to mathematical programming formulations. A simple genetic algorithm for partitioning is described here. In this example, a chromosome represents each partitioned system design, and each chromosome contains genes, representing information about the system. A particular gene TP(i, j) might represent the fact that task i is contained in partition j when it is equal to 1, and is set to 0 otherwise. A family of designs created by some constructive partitioning technique then undergoes mutation and crossover as new designs evolve.

A *fitness function* is used to check the quality of the design and the evolution is halted when the design is considered fit or when no improvement has occurred after some time. In the case of partitioning, the fitness function might include the estimated volume of interconnections, the predicted cost or performance of the system, or other system properties.

The reader might note some similarity between the mathematical programming formulation of the partitioning problem presented here and the genetic algorithm formulation. This similarity allows the CAD developer to create a mathematical programming model of the problem to be solved, find optimal solutions to small problems, and then create a genetic algorithm version. The genetic algorithm version can be checked against the optimal solutions found by the mathematical program. However, genetic algorithms can take into account many more details than can mathematical program formulations, can handle nonlinear relationships better, and can even handle *stochastic parameters*.*

Partitioning is most valuable when there is a mismatch between the sizes of system tasks and the capacities of system modules. When the system tasks and system modules are more closely matched, then the system design can proceed directly to *scheduling* and *allocating* tasks to processing modules.

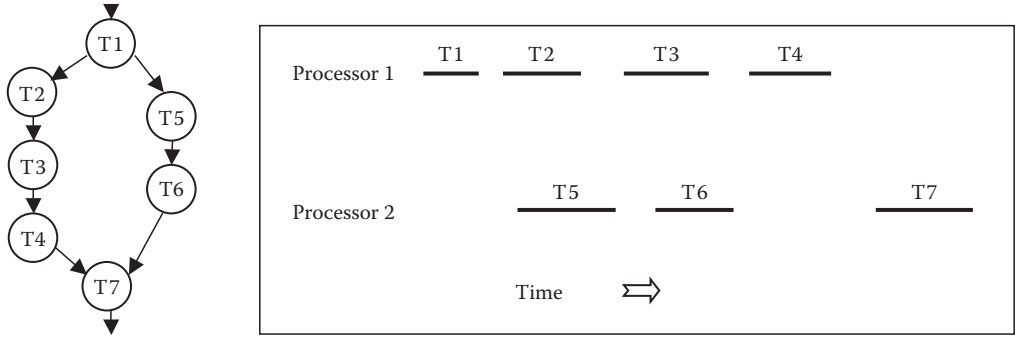
11.4 Scheduling and Allocating Tasks to Processing Modules

Scheduling and allocating tasks to processing modules involves the determination of how many processing modules are required, which modules execute which tasks, and the order in which the tasks are processed by the system. In the special case where only a single task is processed by each module, the scheduling becomes trivial. Otherwise, if the tasks share modules, the order in which the tasks are processed by the modules can affect system performance or cost. If the tasks are ordered inappropriately, some tasks might wait too long for input data, and performance might be affected. Alternatively, to meet performance constraints, additional modules must be added to perform more tasks in parallel, increasing system cost.

A variety of modules might be available to carry out each task, with differing cost and performance parameters. As each task is allocated to a module, that module is selected from a set of modules available to execute the task. This is analogous to the task *module selection*, which occurs as part of high-level synthesis. For the system design problem considered here, the modules can be general-purpose processors, special-purpose processors (e.g., signal-processing processors), or special-purpose hardware. If all (or most) modules used are general purpose, the systems synthesized are known as heterogeneous application-specific multiprocessors.

A variety of techniques can be used for scheduling and allocation of system tasks to modules. Just as with partitioning, these techniques can be constructive or iterative. Constructive scheduling techniques for system tasks include greedy techniques such as ASAP (as soon as possible) and ALAP (as late as possible). In ASAP scheduling, the tasks are scheduled as early as possible on a free processing module. The tasks scheduled first are the ones with the longest paths from their outputs to final system outputs or system completion. Such techniques, with variations, can be used to provide starting populations of system designs to be further improved iteratively. The use of such greedy techniques for system synthesis differs from the conventional use in high-level synthesis, where the system is assumed to be synchronous, with tasks scheduled into time steps. System task scheduling assumes no central clock, and tasks take a wide range of time to complete. Some tasks could even complete stochastically, with completion time being a random variable. Other tasks could complete basic calculations in a set time, but could perform a finer grain (more accurate) of computations if more time were available. A simple task-flow graph is shown in Figure 11.2, along with a Gantt chart illustrating the ASAP scheduling of tasks onto two processors. Note that two lengthy tasks are performed in parallel with three shorter tasks and that no two tasks take the same amount of time.

* Stochastic parameters represent values that are uncertain. There is a finite probability of a parameter taking a specific value that varies with time, but that probability is less than 1, in general.



(a) Task-flow graph (b) Gantt chart showing schedule

FIGURE 11.2 Example of task-flow graph and schedule.

Similar to partitioning, scheduling, allocation, and module selection, can be performed using mathematical programming. In this case, since the scheduling is asynchronous, time becomes a linear rather than integer quantity. Therefore, MILP is employed to model system-level scheduling and allocation. A typical MILP timing constraint is the following:

$$T_{\mathrm{OA}}(i) + C_{\mathrm{delay}} \leq T_{\mathrm{IR}}(j) \tag{11.2}$$

where

$T_{\mathrm{OA}}(i)$ is the time the output is available from task i
C_{delay} is the communication delay
$T_{\mathrm{IR}}(j)$ is the time the input is required by task j

Unfortunately, the actual constraints used in scheduling and allocation are mostly more complex than this, because the design choices have yet to be made. Here is another example:

$$T_{\mathrm{OA}}(i) \geq T_{\mathrm{IR}}(i) + \sum_{k} \left[P_{\mathrm{delay}}(k) * M(i, k)\right] \tag{11.3}$$

This constraint states that the time an output from task i is available is greater than or equal to the time necessary inputs are received by task i, and a processing delay P_{delay} has occurred. $M(i, k)$ indicates that task i is allocated to module k. P_{delay} can take on a range of values, depending on which of the k modules is being used to implement task i. The summation is actually a linearized select function that picks the value of P_{delay} to use depending on which value of $M(i, k)$ is set to 1.

As with partitioning, mathematical programming for scheduling and allocation is computationally intensive, and impractical for all but the smallest designs, but it does provide a baseline model of design that can be incorporated into other tools.

The most frequent technique used for iterative improvement in scheduling and allocation at the system level is a genetic algorithm. The genes can be used to represent task allocation and scheduling. To represent asynchronous scheduling accurately, time is generally represented as a linear quantity in such genes rather than an integer quantity.

11.5 Allocating and Scheduling Storage Modules

In digital systems, all data require some form of temporary or permanent storage. If the storage is shared by several data sets, the use of the storage by each data set must be scheduled. The importance of this task in system design has been overlooked in the past, but has now become an important system-level task.

Modern digital systems usually contain some multimedia tasks and data. The storage requirements for multimedia tasks sometimes result in systems where processing costs are dwarfed by storage costs, particularly caching costs. For such systems, storage must be scheduled and allocated either during or after task scheduling and allocation. If storage is scheduled and allocated concurrently with task scheduling and allocation, the total system costs are easier to determine and functional module sharing can be increased if necessary to control total costs. Alternatively, if storage allocation and scheduling are performed after task scheduling and allocation, then both programs are simpler, but the result may not be as close to optimal.

Techniques similar to those used for task scheduling and allocation can be used for storage scheduling and allocation.

11.6 Selecting Implementation and Packaging Styles for System Modules

Packaging styles can range from single-chip dual-in-line packages (DIPs) to MCMs, boards, racks, and cases. Implementation styles include general-purpose processor, special-purpose programmable processor (e.g., signal processor), COTS modules, field programmable gate arrays (FPGAs), gate array, standard cell, and custom ICs. System cost, performance, power, and design time constraints determine selection of implementation and packaging styles for many system designs. Tight performance constraints favor custom ICs, packaged in MCMs. Tight cost constraints favor off-the-shelf processors and gate array implementations, with small substrates and inexpensive packaging. Tight power constraints favor custom circuits. Tight design time constraints favor COTS modules and FPGAs. If a single design property has high priority, the designer can select the appropriate implementation style and packaging technology. If, however, design time is crucial, but the system to be designed must process video signals in real time, then trade-offs in packaging and implementation style must be made. The optimality of system cost and power consumption might be sacrificed: the entire design might be built with FPGAs, with much parallel processing and at great cost and large size. Because time-to-market is so important, early market entry systems may sacrifice the optimality of many system parameters initially, and then improve them in the next version of the product.

Selection of implementation styles and packaging can be accomplished by adding some design parameters to the scheduling and allocation program, if that program is not already computationally intensive. The parameters added would include a variable indicating that a particular

- Functional module was assigned a certain implementation style
- Storage module was assigned a certain implementation style
- Functional module was assigned a certain packaging style
- Storage module was assigned a certain packaging style

Some economy of processing could be obtained if certain implementation styles precluded certain packaging styles.

11.7 Interconnection Strategy

Modules in a digital system are usually interconnected in some carefully architected, consistent manner. If point-to-point interconnections are used, they are used throughout the system, or in a subsystem. In the same manner, buses are not broken arbitrarily to insert point-to-point connections or rings. For this reason, digital system design programs usually assume an interconnection style and determine the system performance relative to that style. The most common interconnection styles are bus, point-to-point, and ring.

11.8 Word-Length Determination

Functional specifications for system tasks are frequently detailed enough to contain the algorithm to be implemented. To determine the implementation costs of each system task, knowledge of the word widths to be used is important as system cost varies almost quadratically with word width.

Tools to automatically select task word width are currently experimental, but the potential for future commercial tools exists.

In typical hardware implementations of an arithmetic-intensive algorithm, designers must determine the word lengths of resources such as adders, multipliers, and registers. Wadekar and Parker [6] in a recent publication, present algorithm-level optimization techniques to select distinct word lengths for each computation. These techniques meet the desired accuracy and minimize the design cost for the given performance constraints. The cost reduction is possible by avoiding unnecessary bit-level computations that do not contribute significantly to the accuracy of the final results. At the algorithm level, determining the necessary and sufficient precision of an individual computation is a difficult task since the precision of various predecessor/successor operations can be traded off to achieve the same desired precision in the final result. This is achieved using a mathematical model [7] and a genetic selection mechanism [6]. There is a distinct advantage to word-length optimization at the algorithmic level. The optimized operation word lengths can be used to guide high-level synthesis or designers to achieve an efficient utilization of resources of distinct word lengths and costs. Specifically, only a few resources of larger word lengths and high cost may be needed for operations requiring high precision to meet the final accuracy requirement. Other relatively low-precision operations may be executed by resources of smaller word lengths. If there is no timing conflict, a large word-length resource can also execute a small word-length operation, thus improving the overall resource utilization further. These high-level design decisions cannot be made without the knowledge of word lengths prior to synthesis.

11.9 Predicting System Characteristics

In system-level design, early prediction gives designers the freedom to make numerous high-level choices (such as die size, package type, and latency of the pipeline) with confidence that the final implementation will meet power and energy as well as cost and performance constraints. These predictions can guide power budgeting and subsequent synthesis of various system components, which is critical in synthesizing systems that have low power dissipation, or long battery life. The use by synthesis programs of performance and cost lower bounds allows smaller solution spaces to be searched, which leads to faster computation of the optimal solution.

System cost, performance, power consumption, and design time can be computed if the properties of each system module are known. System design using existing modules requires little prediction. If system design is performed prior to design of any of the contained system modules, however, their properties must be predicted or estimated. Owing to the complexities of prediction techniques, describing these techniques is a subject worthy of an entire chapter. A brief survey of related readings is found in Section 11.10.

The register-transfer and subsequent lower-level power prediction techniques such as gate- and transistor-level techniques are essential for validation before fabricating the circuit. However, these techniques are less efficient for system-level design as a design must be generated before prediction can be done.

11.10 Survey of Research in System Design

Many researchers have investigated the problem of system design, dating back to the early 1970s. This section highlights work that is distinctive, along with tutorial articles covering relevant topics.

Much good research is not referenced here, and the reader is reminded that the field is dynamic, with new techniques and tools appearing almost daily.

Issues in top-down versus bottom-up design approaches were highlighted in the design experiment reported by Gupta et al. [8].

11.10.1 System Specification

System specification has received little attention historically except in the specific area of software specifications. Several researchers have proposed natural language interfaces capable of processing system specifications and creating internal representations of the systems that are considerably more structured. Of note is the work by Granacki [9] and Cyre [10]. One noteworthy approach is the design specification language (DSL), found in the design analysis and synthesis environment [11]. One of the few books on the subject concerns the design of embedded systems—systems with hardware and software designed for a particular application set [12]. In one particular effort, Petri nets were used to specify the interface requirements in a system of communicating modules, which were then synthesized [13]. The SIERA system designed by Srivastava, Richards, and Broderson [14] supports specification, simulation, and interactive design of systems.

The Rugby model [15] represents hardware/software systems and the design process, using four dimensions to represent designs: time, computation, communication, and data. da Silva [16] describes the system data structure (SDS), used for internal representation of systems. da Silva also presents an external language for system descriptions called OSCAR. SDS is general and comprehensive, covering behavior and structure. OSCAR is a visual interface to SDS with a formal semantics and syntax based on a visual coordination language paradigm. It is used to capture the behavior and the coordination aspects of concurrent and communicating processes.

SystemC [17] provides hardware-oriented constructs as a class library implemented in standard C++. Although the use of SystemC is claimed to span design and verification from concept to implementation in hardware and software, the constructs provided are somewhat of lower level than most of the system design activities described in this chapter. Extensions to VHDL and Verilog also support some system design activities at the lower levels of system design.

11.10.2 Partitioning

Partitioning research covers a wide range of system design situations. Many early partitioning techniques dealt with assigning register-level operations to partitions. APARTY, a partitioner designed by Lagnese and Thomas, partitions CDFG designs for single-chip implementation to obtain efficient layouts [18]. Vahid [19] performed a detailed survey of techniques for assigning operations to partitions. CHOP assigns CDFG operations to partitions for multichip design of synchronous, common clocked systems [20]. Vahid and Gajski developed an early partitioner, SpecPart, which assigns processes to partitions [21]. Chen and Parker reported on a process-to-partition technique called ProPart [22].

11.10.3 Nonpipelined Design

Although research on *system design* spans more than two decades, most of the earlier works focus on single aspects of design-like task assignment, and not on the entire design problem. We cite some representative works here. These include graph theoretical approaches to task assignment [23,24], analytical modeling approaches for task assignment [25], and probabilistic modeling approaches for task partitioning [26,27], scheduling [28], and synthesis [29]. Two publications of note cover application of heuristics to system design [30,31].

Other publications of note include mathematical programming formulations for task partitioning [32] and communication channel assignment [33]. Early efforts include those done by soviet researchers since

the beginning of the 1970s such as Linsky and Kornev [34] and others, where each model only included a subset of the entire synthesis problem. Chu et al. [35] published one of the first MILP models for a subproblem of system-level design, scheduling. The program Synthesis of Systems (SOS) including a compiler for MILP models [36,37] was developed, based on a comprehensive MILP model for system synthesis. SOS takes a description of a system described using a task-flow graph, a processor library, and some cost and performance constraints, and generates an MILP model to be optimized by an MILP solver. The SOS tool generates MILP models for the design of nonperiodic (nonpipelined) heterogeneous multiprocessors. The models share a common structure, which is an extension of the previous work by Hafer and Parker for high-level synthesis of digital systems [38].

Performance bounds of solutions found by algorithms or heuristics for system-level design are proposed in many papers, including the landmark papers by Fernandez and Bussel [39], Garey and Graham [40], and more recent publications [41].

The work of Gupta et al. [8] reported the successful use of system-level design tools in the development of an application-specific heterogeneous multiprocessor (ASHM) for image processing. Gupta and Zorian [42] describe the design of systems using cores, silicon cells with at least 5000 gates. The same issue of *IEEE Design and Test of Computers* contains a number of useful articles on the design of embedded core-based systems. Li and Wolf [43] report on a model of hierarchical memory and a multiprocessor synthesis algorithm, which takes into account the hierarchical memory structure.

A major project, RASSP, is a rapid-prototyping approach whose development is funded by the US Department of Defense [44]. RASSP addresses the integrated design of hardware and software for signal-processing applications.

An early work on board-level design, MICON, is of particular interest [45]. Other research results solving similar problems with more degrees of design freedom include the research by Chen [46] and Heo [47]. GARDEN, written by Heo, finds the design with the shortest estimated time-to-market that meets cost and performance constraints.

All the MILP synthesis works cited up to this point address only the nonperiodic case.

Synthesis of ASHMs is a major activity in the general area of system synthesis. One of the most significant system-level design efforts is Lee's Ptolemy project at the University of California, Berkeley. Representative publications include papers by Lee and Bier describing a simulation environment for signal processing [48] and the paper by Kalavede et al. in 1995 [49]. Another prominent effort is the SpecSyn project of Gajski et al. [50] which is a system-level design methodology and framework.

11.10.4 Macropipelined Design

Macropipelined (periodic) multiprocessors execute tasks in a pipelined fashion, with tasks executing concurrently on different sets of data. Most research work on design of *macropipelined* multiprocessors has been restricted to homogeneous multiprocessors having negligible communication costs. This survey divides the previous contributions according to the execution mode: preemptive or *nonpreemptive*.

11.10.4.1 Nonpreemptive Mode

The nonpreemptive mode of execution assumes that each task is executed without interruption. It is used quite often in low-cost implementations. Much research has been performed on system scheduling for the nonpreemptive mode. A method to compute the minimum possible value for the initiation interval for a task-flow graph given an unlimited number of processors and no communication costs was found by Renfors and Neuvo [51].

Wang and Hu [52] use heuristics for the allocation and *full static scheduling* (meaning that each task is executed on the same processor for all iterations) of generalized perfect-rate task-flow graphs on homogeneous multiprocessors. Wang and Hu apply planning, an artificial intelligence method, to the task scheduling problem. The processor allocation problem is solved using a *conflict-graph* approach.

Gelabert and Barnwell [53] developed an optimal method to design macropipelined homogeneous multiprocessors using *cyclic-static scheduling*, where the task-to-processor mapping is not time-invariant as in the full static case, but is periodic, i.e., the tasks are successively executed by all processors. Gelabert and Barnwell assume that the delays for *intraprocessor* and *interprocessor* communications are the same, which is an idealistic scenario. Their approach is able to find an optimal implementation (minimal iteration interval) in exponential time in the worst case.

Tirat-Gefen [54], in his doctoral thesis, extended the SOS MILP model to solve for optimal macropipelined ASHMs. He also proposed an ILP model allowing simultaneous optimal retiming and processor/module selection in high- and system-level synthesis [55].

Verhauger [56] addresses the problem of periodic multidimensional scheduling. His thesis uses an ILP model to handle the design of homogeneous multiprocessors without communication costs implementing data-flow programs with nested loops. His work evaluates the complexity of the scheduling and allocation problems for the multidimensional case, which were both found to be NP-complete. Verhauger proposes a set of heuristics to handle both problems.

Passos et al. [57] evaluate the use of multidimensional retiming for synchronous data-flow graphs. However, their formalism can only be applied to homogeneous multiprocessors without communication costs.

11.10.4.2 Preemptive Mode of Execution

Peng and Shin [58] address the optimal static allocation of periodic tasks with precedence constraints and preemption on a homogeneous multiprocessor. Their approach has an exponential time complexity. Ramamritham [59] developed a heuristic method that has a more reasonable computational cost. Rate-monotonic scheduling (RMS) is a commonly used method for allocating periodic real-time tasks in distributed systems [60]. The same method can be used in homogeneous multiprocessors.

11.10.5 Genetic Algorithms

Genetic algorithms are becoming an important tool for solving the highly nonlinear problems related to system-level synthesis. The use of genetic algorithms in optimization is well discussed by Michalewicz [61] where formulations for problems such as bin packing, processor scheduling, traveling salesman, and system partitioning are outlined.

Research works involving the use of genetic algorithms to system-level synthesis problems are starting to be published, as for example, the results of

- Hou et al. [62]—scheduling of tasks in a homogeneous multiprocessor without communication costs
- Wang et al. [63]—scheduling of tasks in heterogeneous multiprocessors with communication costs, but not allowing cost versus performance trade-off, i.e., all processors have the same cost
- Ravikumar and Gupta [64]—mapping of tasks into a reconfigurable homogeneous array processor without communication costs
- Tirat-Gefen and Parker [65]—a genetic algorithm for design of ASHMs with nonnegligible communications costs specified by a nonperiodic task-flow graph representing both control and data flow
- Tirat-Gefen [54]—introduced a full-set of genetic algorithms for system-level design of ASHMs incorporating new design features such as imprecise computation and probabilistic design

11.10.6 Imprecise Computation

The main results in imprecise *computation* theory are due to Liu et al. [66] who developed polynomial time algorithms for optimal scheduling of preemptive tasks on homogeneous multiprocessors without

communications costs. Ho et al. [67] proposed an approach to minimize the total error, where the error of a task being imprecisely executed is proportional to the amount of time that its optional part was not allowed to execute, i.e., the time still needed for its full completion. Polynomial time-optimal algorithms were derived for some instances of the problem [66].

Tirat-Gefen et al. [68] presented in 1997 a new approach for ASHM design that allows trade-offs between cost, performance, and data-quality through incorporation of imprecise computation into the system-level design cycle.

11.10.7 Probabilistic Models and Stochastic Simulation

Many probabilistic models for solving different subproblems in digital design have been proposed recently. The problem of task and data-transfer scheduling on a multiprocessor when some tasks (data transfers) have nondeterministic execution times (communication times) can be modeled by PERT networks, which were introduced by Malcolm et al. [69] along with the critical path method (CPM) analysis methodology.

A survey on PERT networks and their generalization to conditional PERT networks is done by Elmaghraby [70]. In system-level design, the completion time of a PERT network corresponds to the *system latency*, whose cumulative distribution function (cdf) is a nonlinear function of the probability density distributions of the computation times of the tasks and the communication times of the data transfers in the task-flow graph.

The exact computation of the cdf of the completion time is computationally expensive for large PERT networks, therefore it is important to find approaches that approximate the value of the expected time of the completion time and its cdf. One of the first of these approaches was due to Fulkerson [71], who derived an algorithm in 1962 to find a tight estimate (lower bound) of the expected value of the completion time. Robillard and Trahan [72] proposed a different method using the characteristic function of the completion time in approximating the cdf of the completion time.

Mehrotra et al. [73] proposed a heuristic for estimating the moments of the probabilistic distribution of the system latency t_c. Kulkarni and Adlakha [74] developed an approach based on Markov processes for the same problem. Hagstrom [75] introduced an exact solution for the problem when the random variables modeling the computation and communication times are finite discrete random variables. Kamburowski [76] developed a tight upper bound on the expected completion time of a PERT network.

An approach using random graphs to model distributed computations was introduced by Indurkhya et al. [26], whose theoretical results were improved by Nicol [27]. Purushotaman and Subrahmanyam [77] proposed formal methods applied to concurrent systems with a probabilistic behavior. An example of modeling using queueing networks instead of PERT networks is given by Thomasian and Bay [78]. Estimating errors owing to the use of PERT assumptions in scheduling problems is discussed by Lukaszewicz [79].

Tirat-Gefen developed a set of genetic algorithms using stratified stochastic sampling allowing simultaneous probabilistic optimization of the scheduling and allocation of tasks and communications on ASHM with nonnegligible communication costs [54].

11.10.8 Performance Bounds Theory and Prediction

Sastry [80] developed a stochastic approach for estimation of wireability (routability) for gate arrays. Kurdahi [81] created a discrete probabilistic model for area estimation of VLSI chips designed according to a standard cell methodology. Küçükçakar [82] introduced a method for partitioning of behavioral specifications onto multiple VLSI chips using probabilistic area/performance predictors integrated into a package called BEST (behavioral estimation). BEST provides a range of prediction techniques that can be applied at the algorithm level and includes references to prior research. These predictors provide information required by Tirat-Gefen's system-level probabilistic optimization methods [54].

Lower bounds on the performance and execution time of task-flow graphs mapped to a set of available processors and communication links were developed by Liu and Liu [83] for the case of heterogeneous processors, but no communication costs and by Hwang et al. [84] for homogeneous processors with communication costs. Tight lower bounds on the number of processors and execution time for the case of homogeneous processors in the presence of communication costs were developed by Al-Mouhamed [85]. Yen and Wolf [86] provide a technique for performance estimation for real-time distributed systems.

At the system and register-transfer level, estimating power consumption by the interconnect is important [87]. Wadekar et al. [88] reported ''Freedom,'' a tool to estimate system energy and power that accounts for functional-resource, register, multiplexer, memory, input/output pads, and interconnect power. This tool employs a statistical estimation technique to associate low-level, technology-dependent, physical, and electrical parameters with expected circuit resources and interconnect. At the system level, Freedom generates predictions with high accuracy by deriving an accurate model of the load capacitance for the given target technology—a task reported as critical in high-level power prediction by Brand and Visweswariah [89]. Methods to estimate power consumption prior to high-level synthesis were also investigated by Mehra and Rabaey [90]. Liu and Svensson [91] reported a technique to estimate power consumption in CMOS VLSI chips. The reader is referred to an example of a publication that reports power prediction and optimization techniques at the register-transfer level [92].

11.10.9 Word-Length Selection

Many researchers studied word-length optimization techniques at the register-transfer level. A few example publications are cited here. These techniques can be classified as statistical techniques applied to digital filters [93], simulated annealing-based optimization of filters [94], and simulation-based optimization of filters, digital communication, and signal-processing systems [95]. Sung and Kum reported a simulation-based word-length optimization technique for fixed-point digital signal-processing systems [96]. The objective of these particular architecture-level techniques is to minimize the number of bits in the design that is related to, but not the same as the overall hardware cost.

11.10.10 Embedded Systems

Embedded systems are becoming ubiquitous. The main factor differentiating embedded systems from other electronic systems is the focus of attention on the application rather than on the system as a computing engine. Typically, the I/O in an embedded system is to end users, sensors, and actuators. Sensors provide information on environmental conditions, for example, and actuators control the mechanical portion of the system. In an autonomous vehicle, an embedded system inputs the vehicle's GPS location via a sensor, and outputs control information to actuators that control the acceleration, braking, and steering. Embedded systems are typically real-time systems, falling into the classes hard real-time systems, where failure is catastrophic, and soft real-time systems, where failure is not catastrophic.

In an embedded system, there is typically at least one general-purpose processor or microcontroller, and one or more coprocessors that are COTS chips, DSPs, FPGAs, or custom VLSI chips. The main tasks to be performed for embedded systems are partitioning into hardware/software tasks, assigning tasks to processors, scheduling the tasks, and simulation.

11.10.11 System on Chip and Network on Chip

The level of integration of microelectronics has allowed entire systems to be fabricated on a single IC. The physical design of on-chip processing elements called cores can be obtained from vendors. Thus, along with the traditional system design issues, there are issues of intellectual property (IP) for such cores.

System on Chip (SoC) tend to be used for embedded systems, where hardware/software codesign is a major activity.

Researchers and organizations are also examining the connection of large quantities of cores on a single chip, and proposing to interconnect the cores with various types of interconnection networks. Benini and DeMicheli [97] proposed the use of packet switched on-chip networks, and there has been much publication activity in this area since. Raghavan proposed a hierarchical, heterogeneous approach, and described an alternative network architecture with torus topology and token ring protocol [98].

References

1. *IEEE Standard VHDL Language Reference Manual*, IEEE Std. 1076, IEEE Press, New York, 1987.
2. Bhasker, J., *A Verilog HDL Primer*, Star Galaxy Press, Allentown, PA, 1997.
3. Bell, G. and Newell, A., *Computer Structures: Readings and Examples*, McGraw-Hill, New York, 1971.
4. Harel, D., Statecharts: A visual formalism for complex systems, *Science of Computer Programming*, 8, 231–274, 1987.
5. Vahid, F., Narayan, S., and Gajski, D.D., SpecCharts: A VHDL front-end for embedded systems, *IEEE Transactions on CAD*, 14, 694, 1995.
6. Wadekar, S.A. and Parker, A.C., Accuracy sensitive word-length selection for algorithm optimization, in *Proceedings of the International Conference on Circuit Design [ICCD]*, 1998, 54.
7. Wadekar, S.A. and Parker, A.C., Algorithm-level verification of arithmetic-intensive application-specific hardware designs for computation accuracy, in *Digest Third International High-Level Design Validation and Test Workshop*, 1998.
8. Gupta, P., Chen, C.T., DeSouza-Batista, J.C., and Parker, A.C., Experience with image compression chip design using unified system construction tools, in *Proceedings of the 31st Design Automation Conference*, 1994.
9. Granacki, J. and Parker, A.C., PHRAN—Span: A natural language interface for system specifications, in *Proceedings of the 24th Design Automation Conference*, 1987, 416.
10. Cyre, W.R., Armstrong, J.R., and Honcharik, A.J., Generating simulation models from natural language specifications, *Simulation*, 65, 239, 1995.
11. Tanir, O. and Agarwal, V.K., A specification-driven architectural design environment, *Computer*, 6, 26–35, 1995.
12. Gajski, D.D., Vahid, F., Narayan, S., and Gong, J., *Specification and Design of Embedded Systems*, Prentice Hall, Englewood Cliffs, NJ, 1994.
13. de Jong, G. and Lin, B., A communicating Petri net model for the design of concurrent asynchronous modules, *ACM/IEEE Design Automation Conference*, June 1994.
14. Srivastava, M.B., Richards, B.C., and Broderson, R.W., System-level hardware module generation, *IEEE Transactions on Very Large-Scale Integration [VLSI] Systems*, 3, 20, 1995.
15. Jantsch, A., Kumar, S., and Hemani, A., The RUGBY model: A conceptual frame for the study of modeling, analysis and synthesis concepts of electronic systems, in *Proceedings of the Conference on Design, Automation and Test in Europe*, Munich, Germany, 1999, 72–78.
16. da Silva, D., Jr., A comprehensive framework for the specification of hardware/software systems, PhD dissertation, Department of Electrical Engineering, University of Southern California, Los Angeles, CA, December 2001.
17. Grötker, T., Liao, S., Martin, G., and Swan, S., *System Design with SystemC*, Kluwer Academic, Boston, MA, 2002.
18. Lagnese, E. and Thomas, D., Architectural partitioning for system level synthesis of integrated circuits, *IEEE Transactions on Computer-Aided Design*, 1991.
19. Vahid, F., A survey of behavioral-level partitioning systems, Technical Report TR ICS 91–71, University of California, Irvine, 1991.

20. Kucukcakar, K. and Parker, A.C., Chop: A constraint-driven system-level partitioner, in *Proceedings of the 28th Design Automation Conference*, 1991, 514.
21. Vahid, F. and Gajski, D.D., Specification partitioning for system design, in *Proceedings of the 29th Design Automation Conference*, 1992.
22. Parker, A.C., Chen, C.-T., and Gupta, P., Unified system construction, in *Proceedings of the SASIMI Conference*, 1993.
23. Bokhari, S.H., Assignment problems in parallel and distributed computing, Kluwer Academic Publishers, Dordrecht, 1987.
24. Stone, H.S. and Bokhari, S.H., Control of distributed processes, *Computer*, 11, 97, 1978.
25. Haddad, E.K., Optimal load allocation for parallel and distributed processing, Technical Report TR 89–12, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA, April 1989.
26. Indurkhya, B., Stone, H.S., and Cheng, L.X., Optimal partitioning of randomly generated distributed programs, *IEEE Transactions on Software Engineering*, SE-12, 483, 1986.
27. Nicol, D.M., Optimal partitioning of random programs across two processors, *IEEE Transactions on Software Engineering*, 15, 134, 1989.
28. Lee, C.Y., Hwang, J.J., Chow, Y.C., and Anger, F.D., Multiprocessor scheduling with interprocessor communication delays, *Operations Research Letters*, 7, 141, 1988.
29. Tirat-Gefen, Y.G., Silva, D.C., and Parker, A.C., Incorporating imprecise computation into system-level design of application-specific heterogeneous multiprocessors, in *Proceedings of the 34th Design Automation Conference*, 1997.
30. DeSouza-Batista, J.C. and Parker, A.C., Optimal synthesis of application-specific heterogeneous pipelined multiprocessors, in *Proceedings of the International Conference on Application-Specific Array Processors*, 1994.
31. Mehrotra, R. and Talukdar, S.N., Scheduling of tasks for distributed processors, Technical Report DRC-18-68-84, Design Research Center, Carnegie-Mellon University, Pittsburgh, PA, December 1984.
32. Agrawal, R. and Jagadish, H.V., Partitioning techniques for large-grained parallelism, *IEEE Transactions on Computers*, 37, 1627, 1988.
33. Barthou, D., Gasperoni, F., and Schwiegelshon, U., Allocating communication channels to parallel tasks, *Environments and Tools for Parallel Scientific Computing*, Elsevier Science Publishers B.V., Amsterdam, 1993, 275.
34. Linsky, V.S. and Kornev, M.D., Construction of optimum schedules for parallel processors, *Engineering Cybernetics*, 10, 506, 1972.
35. Chu, W.W., Hollaway, L.J., and Efe, K., Task allocation in distributed data processing, *Computer*, 13, 57, 1980.
36. Prakash, S. and Parker, A.C., SOS: Synthesis of application-specific heterogeneous multiprocessor systems, *Journal of Parallel and Distributed Computing*, 16, 338, 1992.
37. Prakash, S., Synthesis of application-specific multiprocessor systems, PhD thesis, Department of Electrical Engineering and Systems, University of Southern California, Los Angeles, CA, January 1994.
38. Hafer, L. and Parker, A., Automated synthesis of digital hardware, *IEEE Transactions on Computers*, C-31, 93, 1981.
39. Fernandez, E.B. and Bussel, B., Bounds on the number of processors and time for multiprocessor optimal schedules, *IEEE Transactions on Computers*, C-22, 745, 1975.
40. Garey, M.R. and Graham, R.L., Bounds for multiprocessor scheduling with resource constraints, *SIAM Journal of Computing*, 4, 187, 1975.
41. Jaffe, J.M., Bounds on the scheduling of typed task systems, *SIAM Journal of Computing*, 9, 541, 1991.
42. Gupta, R. and Zorian, Y., Introducing core-based system design, *IEEE Design and Test of Computers*, October–December 15, 1997.

43. Li, Y. and Wolf, W., A task-level hierarchical memory model for system synthesis of multiprocessors, in *Proceedings of the Design Automation Conference*, 1997, 153.
44. *IEEE Design and Test*, Vol. 13, No. 3, Fall 1996, published by IEEE Computer Society, Washington, DC.
45. W. Birmingham and D. Siewiorek, MICON: A single board computer synthesis tool, in *Proceedings of the 21st Design Automation Conference*, 1984.
46. Chen, C-T., System-level design techniques and tools for synthesis of application-specific digital systems, PhD thesis, Department of Electrical Engineering and Systems, University of Southern California, Los Angeles, CA, January 1994.
47. Heo, D.H., Ravikumar, C.P., and Parker, A., Rapid synthesis of multi-chip systems, *in Proceedings of the 10th International Conference on VLSI Design*, 1997, 62.
48. Lee, E.A. and Bier, J.C., Architectures for statically scheduled dataflow, *Journal of Parallel and Distributed Computing*, 10, 333–348, 1990.
49. Kalavede, A., Pino, J.L., and Lee, E.A., Managing complexity in heterogeneous system specification, simulation and synthesis, *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 1995.
50. Gajski, D.D., Vahid, F., and Narayan, S., A design methodology for system-specification refinement, in *Proceedings of the European Design Automation Conference*, 1994, 458.
51. Renfors, M. and Neuvo, Y., The maximum sampling rate of digital filters under hardware speed constraints, *IEEE Transactions on Circuits and Systems*, CAS-28, 196, 1981.
52. Wang, D.J. and Hu, Y.H., Multiprocessor implementation of real-time DSP algorithms, *IEEE Transactions on Very Large-Scale Integration (VLSI) Systems*, 3, 393, 1995.
53. Gelabert, P.R. and Barnwell, T.P., Optimal automatic periodic multiprocessor scheduler for fully specified flow graphs, *IEEE Transactions on Signal Processing*, 41, 858, 1993.
54. Tirat-Gefen, Y.G., Theory and practice in system-level of application-specific heterogeneous multiprocessors, PhD dissertation, Department of Electrical and Computer Engineering, University of Southern California, Los Angeles, CA, 1997.
55. CasteloVide-e-Souza, Y.G., Potkonjak, M. and Parker, A.C., Optimal ILP-based approach for throughput optimization using algorithm/architecture matching and retiming, *Proceedings of the 32nd Design Automation Conference*, June 1995.
56. Verhauger, W.F., Multidimensional periodic scheduling, PhD thesis, Eindhoven University of Technology, Holland, 1995.
57. Passos, N.L., Sha, E.H., and Bass S.C., Optimizing DSP flow-graphs via schedule-based multidimensional retiming, *IEEE Transaction on Signal Processing*, 44, 150, 1996.
58. Peng, D.T. and Shin, K.G., Static allocation of periodic tasks with precedence constraints in distributed real-time systems, in *Proceedings of the 9th International Conference of Distributed Computing*, 1989, 190.
59. Ramamritham, K., Allocation and scheduling of precedence-related periodic tasks, *IEEE Transactions on Parallel and Distributed Systems*, 6, 1995.
60. Ramamritham, K., Stankovic, J.A., and Shiah, P.F., Efficient scheduling algorithms for real-time multiprocessors systems, *IEEE Transaction on Parallel and Distributed Systems*, 1, 184, 1990.
61. Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs*, Springer, Berlin, 1994.
62. Hou, E.S.H, Ansari, N., and Ren, H., A Genetic algorithm for multiprocessor scheduling, *IEEE Transactions on Parallel and Distributed Systems*, 5, 113, 1994.
63. Wang, L., Siegel, H.J., and Roychowdhury, V.P., A genetic-algorithm-based approach for task matching and scheduling in heterogeneous computing environments, in *Proceedings of the Heterogeneous Computing Workshop, International Parallel Processing Symposium*, 1996, 72.
64. Ravikumar, C.P. and Gupta, A., Genetic algorithm for mapping tasks onto a reconfigurable parallel processor, *IEEE Proceedings in Computing Digital Technology*, 142, 81, 1995.

65. Tirat-Gefen, Y.G. and Parker, A.C., MEGA: An approach to system-level design of application-specific heterogeneous multiprocessors, in *Proceedings of the Heterogeneous Computing Workshop, International Parallel Processing Symposium*, 1996, 105.
66. Liu, J.W.S., Lin, K.-J., Shih, W.-K., Yu, A.C.-S., Chung, J.-Y., and Zhao, W., Algorithms for scheduling imprecise computations, *IEEE Computer*, 24, 58, 1991.
67. Ho, K., Leung, J.Y-T., and Wei, W-D., Minimizing maximum weighted error for imprecise computation tasks, Technical Report UNL-CSE-92-017, Department of Computer Science and Engineering, University of Nebraska, Lincoln, 1992.
68. Tirat-Gefen, Y.G., Silva, D.C., and Parker, A.C., Incorporating imprecise computation into system-level design of application-specific heterogeneous multiprocessors, in *Proceedings of the 34th Design Automation Conference*, 1997.
69. Malcolm, D.G., Roseboom, J.H., Clark, C.E., and Fazar, W., Application of a technique for research and development program evaluation, *Operations Research*, 7, 646, 1959.
70. Elmaghraby, S.E., The theory of networks and management science: Part II, *Management Science*, 17, B.54, 1970.
71. Fulkerson, D.R., Expected critical path lengths in pert networks, *Operations Research*, 10, 808, 1962.
72. Robillard, P. and Trahan, M., The completion time of PERT networks, *Operations Research*, 25, 15, 1977.
73. Mehrotra, K., Chai, J., and Pillutla, S., A study of approximating the moments of the job completion time in PERT networks, Technical Report, School of Computer and Information Science, Syracuse University, New York, 1991.
74. Kulkarni, V.G. and Adlakha, V.G., Markov and Markov-regenerative pert networks, *Operations Research*, 34, 769, 1986.
75. Hagstrom, J.N., Computing the probability distribution of project duration in a PERT network, *Networks*, 20, 231, 1990.
76. Kamburowski, J., An upper bound on the expected completion time of PERT networks, *European Journal of Operational Research*, 21, 206, 1985.
77. Purushothaman, S. and Subrahmanyam, P.A., Reasoning about probabilistic behavior in concurrent systems, *IEEE Transactions on Software Engineering*, SE-13, 740, 1987.
78. Thomasian, A., Analytic queueing network models for parallel processing of task systems, *IEEE Transactions on Computers*, C-35, 1045, 1986.
79. Lukaszewicz, J., On the estimation of errors introduced by standard assumptions concerning the distribution of activity duration in PERT calculations, *Operations Research*, 13, 326, 1965.
80. Sastry, S. and Parker, A.C., Stochastic models for wireability analysis of gate arrays, *IEEE Transactions on Computer-Aided Design*, CAD-5, 1986.
81. Kurdahi, F.J., Techniques for area estimation of VLSI layouts, *IEEE Transaction on Computer-Aided Design*, 8, 81, 1989.
82. Küçükçakar, K. and Parker, A.C., A methodology and design tools to support system-level VLSI design, *IEEE Transactions on Very Large-Scale Integration [VLSI] Systems*, 3, 355, 1995.
83. Liu, J.W.S. and Liu, C.L., Performance analysis of multiprocessor systems containing functionally dedicated processors, *Acta Informatica* 10, 95, 1978.
84. Hwang, J.J., Chow, Y.C., Ahnger, F.D., and Lee, C.Y., Scheduling precedence graphs in systems with interprocessor communication times, *SIAM Journal of Computing*, 18, 244, 1989.
85. Mouhamed, M., Lower bound on the number of processors and time for scheduling precedence graphs with communication costs, *IEEE Transactions on Software Engineering*, 16, 1990.
86. Yen, T.-Y. and Wolf, W., Performance estimation for real-time embedded systems, in *Proceedings of the International Conference on Computer Design*, 1995, 64.
87. Landman, P.E. and Rabaey, J.M., Activity-sensitive architectural power analysis, *IEEE Transactions on CAD*, 15, 571, 1996.

88. Wadekar, S.A., Parker, A.C., and Ravikumar, C.P., FREEDOM: Statistical behavioral estimation of system energy and power, in *Proceedings of the Eleventh International Conference on VLSI Design*, 1998, 30.
89. Brand, D. and C. Visweswariah, Inaccuracies in power estimation during logic synthesis, in *Proceedings of the European Design Automation Conference (EURO-DAC)*, 1996, 388.
90. Mehra, R. and Rabaey, J., Behavioral level power estimation and exploration, in *Proceedings of the First International Workshop on Low Power Design*, 1994, 197.
91. Liu, D. and Svensson, C., Power consumption estimation in CMOS VLSI chips, *IEEE Journal of Solid-State Circuits*, 29, 663, 1994.
92. Landman, P.E. and Rabaey, J.M., Activity-sensitive architectural power analysis, *IEEE Transactions on Computer-Aided Design*, 15, 571, 1996.
93. Zeng, B. and Neuvo, Y., Analysis of floating point roundoff errors using dummy multiplier coefficient sensitivities, *IEEE Transactions on Circuits and Systems*, 38, 590, 1991.
94. Catthoor, F., Vandewalle, J., and De Mann, H., Simulated annealing based optimization of coefficient and data word lengths in digital filters, *International Journal of Circuit Theory Applications*, 16, 371, 1988.
95. Grzeszczak, A., Mandal, M.K., Panchanathan, S., and Yeap, T., VLSI implementation of discrete wavelet transform, *IEEE Transactions on VLSI Systems*, 4, 421, 1996.
96. Sung, W. and Kum, Ki-II., Simulation-based word-length optimization method for fixed-point digital signal processing systems, *IEEE Transactions on Signal Processing*, 43, 3087, 1995.
97. Benini, L. and De Micheli, G., Networks on chip: A new paradigm for systems on chip design, in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2002, 418–419.
98. Raghavan, D., Extending the design space for networks on chip, MS thesis, Department of Electrical Engineering, University of Southern California, Los Angeles, CA, May 2004.

12

Performance Modeling and Analysis Using VHDL and SystemC

12.1 Introduction 12-1
Multilevel Modeling
12.2 ADEPT Design Environment 12-9
Token Implementation • ADEPT Handshaking and Token Passing Mechanism • Module Categories • ADEPTs
12.3 Simple Example of an ADEPT Performance Model 12-18
Three-Computer System • Simulation Results
12.4 Mixed-Level Modeling 12-23
Mixed-Level Modeling Taxonomy • Interface for Mixed-Level Modeling with FSMD Components • Interface for Mixed-Level Modeling with Complex Sequential Components
12.5 Performance and Mixed-Level Modeling Using SystemC 12-42
SystemC Background • SystemC Performance Models • Processor Model • Channels • Shared Bus Model • Fully Connected Model • Crossbar Model • SystemC Performance Modeling Examples • Mixed-Level Processor Model • Mixed-Level Examples • Random Data Generation • Mixed-Level Example Summary
12.6 Conclusions 12-64
References 12-64

Robert H. Klenke
Virginia Commonwealth University

Jonathan A. Andrews
Virginia Commonwealth University

James H. Aylor
University of Virginia

12.1 Introduction

It has been noted by the digital design community that the greatest potential for additional cost and iteration cycle time savings is through improvements in tools and techniques that support the early stages of the design process [1]. As shown in Figure 12.1, decisions made during the initial phases of a product's development cycle determine up to 80% of its total cost. The result is that accurate, fast analysis tools must be available to the designer at the early stages of the design process to help make these decisions. Design alternatives must be effectively evaluated at this level with respect to multiple metrics, such as performance, dependability, and testability. This analysis capability will allow a larger portion of the design space to be explored yielding higher quality as well as lower cost designs.

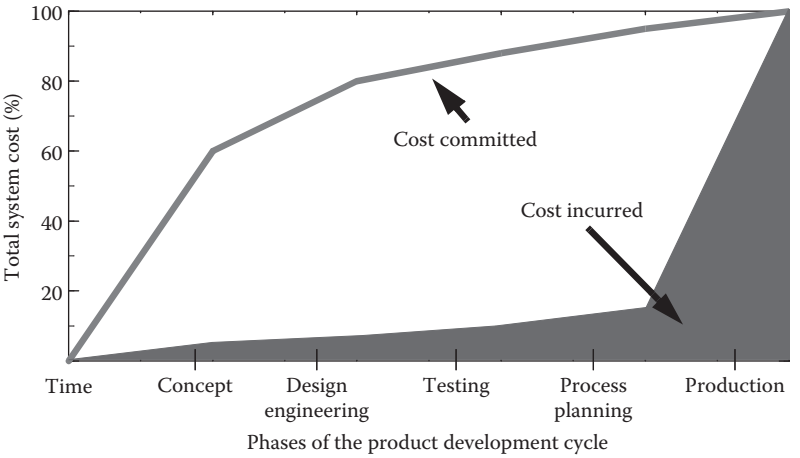


FIGURE 12.1 Product costs over the development cycle.

In 1979, Hill and vanCleemput [2] of the SABLE environment identified three or four stages or phases of design, and noted that each traditionally uses their own simulator. While the languages used have changed and some of the phases can be simulated together these phases still exist in most industry design flows: first an initial high-level simulation, then an intermediate level roughly at the instruction set simulation level, and then a gate-level phase. The authors add a potential fourth stage to deal with custom integrated circuit (IC) simulation. While acknowledging the potential for more efficient simulation at a given level due to potential optimization, they point out five key disadvantages to having different simulators and languages for the various levels. They are

1. Design effort is multiplied by the necessity of learning several simulator systems and recoding a design in each.
2. Possibility of error is increased as more human manipulation enters the system.
3. Each simulator operates at just one level of abstraction. Because it is impossible to simulate an entire computer at a low level of abstraction, only small fragments can be simulated at any one time.
4. Each fragment needs to be driven by a supply of realistic data and its output needs to be interpreted. Often, writing the software to serve these needs is more effortful than developing the design itself.
5. As the design becomes increasingly fragmented to simulate, it becomes difficult for any designer to see how his own particular piece fits into the overall function of the system.

While SABLE was developed almost three decades ago, it is worth noting that today designers are still struggling to address the same basic issues expressed above. Much of the industry has at least two or three design phases requiring separate models and simulators, which still results in a multiplication of design effort. This multiplication of design effort still increases the possibility of error due to inherently fallible human manipulation. While it may not be impossible to simulate an entire computer at one low level of abstraction, it is still impractical and inefficient. Thus, designs are often still fragmented for development and simulation, these fragments still need realistic data, and generating it is still time-consuming. Designers working on a fragment often have difficulty seeing how his or her piece fits into overall function, and this often leads to differing assumptions between fragments and expensive design revision.

There are a number of current tools and techniques that support analysis of these metrics at the system level to varying degrees. A major problem with these tools is that they are not integrated into the engineering design environment in which the system will ultimately be implemented. This problem leads

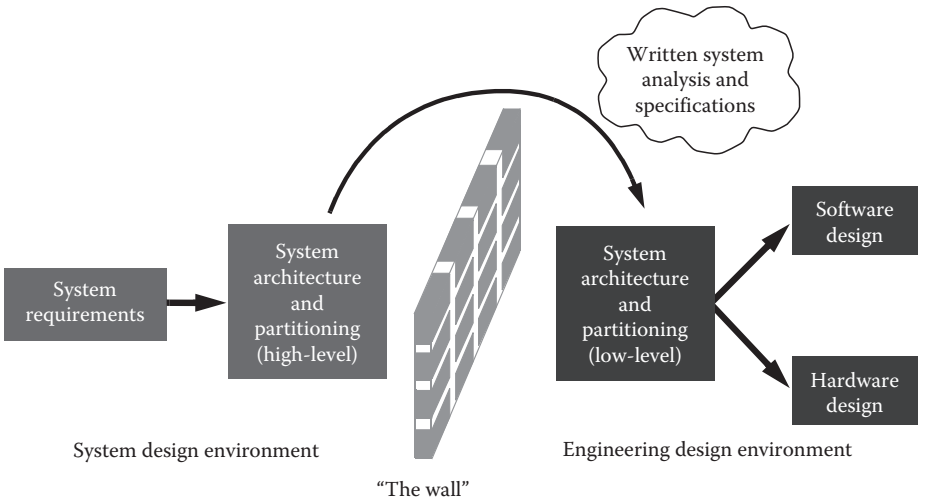


FIGURE 12.2 Disconnect between system-level design environments and engineering design environments.

to a major disconnect in the design process. Once the system-level model is developed and analyzed, the resulting high-level design is specified on paper and thrown ''over the wall'' for implementation by the engineering design team, as illustrated in Figure 12.2. As a result, the engineering design team has to often interpret this specification to implement the system, which often leads to design errors. It also has to develop their own initial ''high-level'' model from which to begin the design process in a top-down manner. Additionally, there is no automated mechanism by which feedback on design assumptions and estimations can be provided to the system design team by the engineering design team.

For systems that contain significant portions of both application-specific hardware and software executing on embedded processors, design alternatives for competing system architectures and hardware/software (HW/SW) partitioning strategies must be effectively and efficiently evaluated using high-level performance models. Additionally, the selected hardware and software system architecture must be refined in an integrated manner from the high-level models to an actual implementation to avoid implementation mistakes and the associated high redesign costs. Unfortunately, most existing design environments lack the ability to model and design a system's hardware and software in the same environment. A similar wall to that between the system design environment and the engineering design environment exists between the hardware and the software design environments. This results in a design path as shown in Figure 12.3, where the hardware and software design process begins with a common system requirement and specification, but proceeds through a separate and isolated design process until final system integration. At this point, assumptions on both sides may prove to be drastically wrong resulting in incorrect system function and poor system performance.

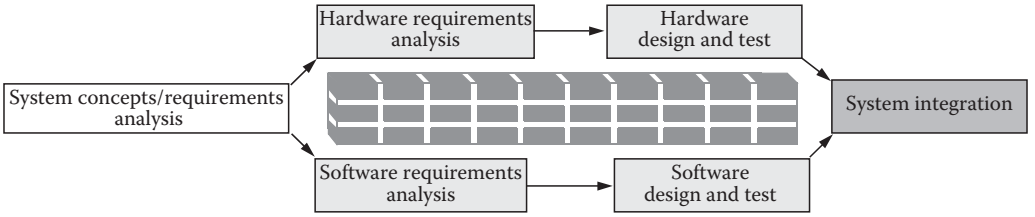


FIGURE 12.3 Current hardware/software system development methodology.

A unified, cooperative approach in which the hardware and software options can be considered together is required to increase the quality and decrease the design time for complex HW/SW systems. This approach is called *hardware/software codesign*, or simply *codesign* [3–5]. Codesign leads to more efficient implementations and improves overall system performance, reliability, and cost-effectiveness [5]. Also, because decisions regarding the implementation of functionality in software can impact hardware design (and vice versa), problems can be detected and changes made earlier in the development process [6].

Codesign can especially benefit the design of *embedded systems* [7], which contain hardware and software tailored for a particular application. As the complexity of these systems increases, the issue of providing design approaches that scale up to more complicated systems becomes of greater concern. A detailed description of a system can approach the complexity of the system itself [8], and the amount of detail present can make analysis intractable. Therefore, decomposition techniques and abstractions are necessary to manage this complexity.

What is needed is a design environment in which the capability for performance modeling of HW/SW systems at a high level of abstraction is fully integrated into the engineering design environment. To completely eliminate the ''over the wall'' problem and the resulting model discontinuity, this environment must support the incremental refinement of the abstract system-level performance model into an implementation-level model. Using this environment, a design methodology based on incremental refinement can be developed.

The design methodology illustrated in Figure 12.4 was proposed by Lockheed Martin Advanced Technology Laboratory as a new way to design systems [9]. This methodology uses the level of the risk of not meeting the design specifications as the metric for driving the design process. In this spiral-based design methodology, there are two iteration cycles. The major cycles (or spirals), denoted as Cycle 1, Cycle 2, . . . , Cycle *N* in the figure, correspond to the design iterations where major architectural changes are made in response to some specification metrics and the system as a whole is refined and more design detail is added to the model. Consistent with the new paradigm of system design, these iterations will actually produce virtual or simulation-based prototypes. A virtual prototype is simply a simulatable model of the system with stimuli described at a given level of design detail or design abstraction that describes the system's operation. Novel to this approach are the mini spirals. The mini spiral cycles, denoted by the levels on the figure labeled Systems, Architecture, and Detailed design, correspond to the refinement of only those portions of the design that are deemed to be ''high risk.'' High risk is obviously defined by the designer but is most often the situation where if one

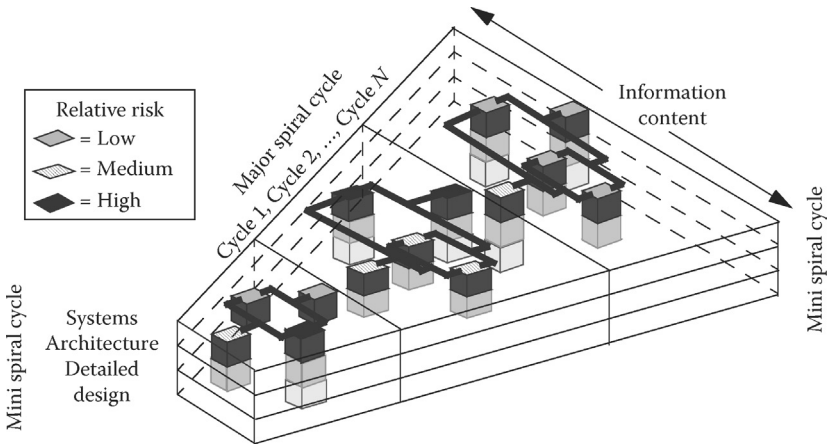


FIGURE 12.4 RDEIM.

or more of these components fail to meet their individual specifications, the system will fail to meet its specifications. The way to minimize the risk is to refine these components to possibly make an implementation so that the actual performance is known. Unlike the major cycles where the entire design is refined, the key to the mini spirals is the fact that only the critical portions of design are refined. For obvious reasons, the resulting models have been denoted as ''Risk Driven Expanding Information Models'' (RDEIMs).

The key to being able to implement this design approach is to be able to evaluate the overall design with portions of the system having been refined to a detailed level, while the rest of the system model remains at the abstract level. For example, in the first major cycle of Figure 12.4 the element with the highest relative risk is fully implemented (detailed design level) while the other elements are described at more abstract levels (system level or architectural level). If the simulation of the model shown in the first major cycle detects that the overall system will not meet its performance requirements, then the ''high risk'' processing element could be replaced by two similar elements operating in parallel. This result is shown in the second major cycle and at this point, another element of the system may become the new ''bottleneck,'' i.e., the highest relative risk, and it will be refined in a similar manner.

Implied in the RDEIM approach is a solution to the ''over the wall'' problem including hardware/software codesign. The proposed solution is to fully integrate performance modeling into the design process.

Obviously, one of the major capabilities necessary to implement a top-down design methodology such as the RDEIM is the ability to cosimulate system models which contain some components that are modeled at an abstract performance level (uninterpreted models) and some that are modeled at a detailed behavioral level (interpreted models). This capability to model and cosimulate uninterpreted models and interpreted models is called *mixed-level modeling* (sometimes referred to as hybrid modeling). Mixed-level modeling requires the development of interfaces that can resolve the differences between uninterpreted models that, by design, do not contain a representation of all of the data or timing information of the final implementation, and interpreted models which require most, or possibly all, data values and timing relationships to be specified. Techniques for systematic development of these mixed-level modeling interfaces and resolution of these differences in abstraction is the focus of some of the latest work in mixed-level modeling.

In addition to the problem of mixed-level modeling interfaces, another issue that may have to be solved is that of different modeling languages being used at different levels of design abstraction. While VHDL, and to some extent various extensions to Verilog, can be used to model at the system level, many designers constructing these types of models prefer to use a language with a more programming language-like syntax. As a result of this and other factors, the SystemC language [10] was developed. SystemC is a library extension to C++ that builds key concepts from existing hardware description languages (HDLs)—primarily a timing model that allows simulation of concurrent events—into an intrinsically object oriented HDL. Its use of objects, pointers, and other standard C++ items makes it a versatile language. Particularly the ability to describe and use complex data types, and the concepts of interfaces and channels make it much more intuitive for describing abstract behaviors than existing HDLs such as VHDL or Verilog. Since SystemC is C++, existing C or C++ code that models behavior can be imported with little or no modification.

12.1.1 Multilevel Modeling

The need for multilevel modeling was recognized almost three decades ago. Multilevel modeling implies that representations at different levels of detail coexist within a model [8,11,12]. Until the early 1990s, the term multilevel modeling was used for integrating behavioral or functional models with lower level models. The objective was to provide a continuous design path from functional models down to implementation. This objective was achieved and the VLSI industry utilizes it today. An example is the tool called Droid, developed by Texas Instruments [13].

The mixed-level modeling approach, as described in this chapter, is a specific type of multilevel modeling which integrates *performance* and *behavioral* models. Thus, only related research on multilevel modeling systems that spans both the performance and the functional/behavioral domains will be described.

Although behavioral or functional modeling is typically well understood by the design community, performance modeling is a foreign topic to most designers. Performance modeling, also called uninterpreted modeling, is utilized in the very early stages of the design process in evaluating such metrics as throughput and utilization. Performance models are also used to identify bottlenecks within a system and are often associated with the job of a system engineer. The term ''uninterpreted modeling'' reflects the view that performance models lack value-oriented data and functional (input/output) transformations. However, in some instances, this information is necessary to allow adequate analysis to be performed.

A variety of techniques have been employed for performance modeling. The most common techniques are Petri nets [14–16] and queuing models [17,18]. A combination of these techniques, such as a mixture of Petri nets and queuing models [19], has been utilized to provide more powerful modeling capabilities. All of these models have mathematical foundations. However, models of complex systems constructed using these approaches can quickly become unwieldy and difficult to analyze.

Examples of a Petri net and a queuing model are shown in Figure 12.5. A queuing model consists of queues and servers. Jobs (or customers) arrive at a specific arrival rate and are placed in a queue for service. These jobs are removed from the queue to be processed by a server at a particular service rate. Typically, the arrival and service rates are expressed using probability distributions. There is a queuing discipline, such as first-come-first-serve, which determines the order in which jobs are to be serviced. Once they are serviced, the jobs depart and arrive at another queue or simply leave the system. The number of jobs in the queues represents the model's state. Queueing models have been used successfully for modeling many complex systems. However, one of the major disadvantages of queuing models is their inability to model synchronization between processes.

As a system modeling paradigm, Petri nets overcome this disadvantage of queuing models. Petri nets consist of places, transitions, arcs, and a marking. The places are equivalent to conditions and hold tokens, which represent information. Thus, the presence of a token in the place of a Petri net corresponds to a particular condition being true. Transitions are associated with events, and the ''firing'' of a transition indicates that some event has occurred. A marking consists of a particular placement of tokens within the places of a Petri net and represents the state of the net. When a transition fires, tokens are removed from the input places and are added to the output places, changing the marking (the state) of the net and allowing the dynamic behavior of a Petri net to be modeled.

Petri nets can be used for performance analysis by associating a time with the transitions. Timed and stochastic Petri nets contain deterministic and probabilistic delays, respectively. Normally, these Petri nets are uninterpreted, since no *interpretation* (values or value transformations) are associated with the tokens or transitions. However, values or value transformations can be associated with the various elements of Petri net models as described below.

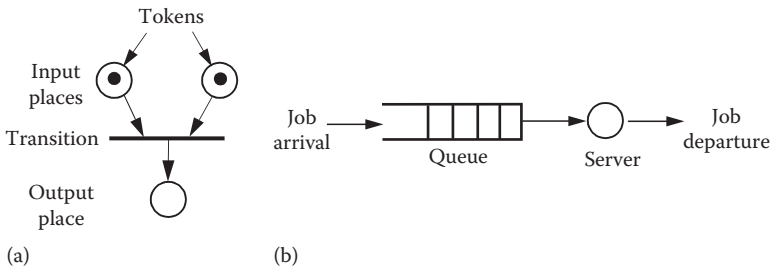


FIGURE 12.5 (a) Petri net and (b) queuing model.

Petri nets that have values associated with tokens are known as colored Petri nets (CPNs). In the colored Petri nets, each token has an attached ''color,'' indicating the identity of the token. The net is similar to the basic definition of the Petri net except that a functional dependency is specified between the color of the token and the transition firing action. In addition, the color of the token produced by a transition may be different from the color of the tokens on the input places. Colored Petri nets have an increased ability to efficiently model real systems with small nets which are equivalent to much larger plain Petri nets due to their increased descriptive powers.

Numerous multilevel modeling systems exist based on these two performance modeling techniques. Architecture Design and Assessment System (ADAS) is a set of tools specifically targeted for high-level design [20]. ADAS models both hardware and software using directed graphs based on timed Petri nets. The flow of information is quantified by identifying discrete units of information called tokens. ADAS supports two levels of modeling. The more abstract level is a dataflow description and is used for performance estimation. The less abstract level is defined as a behavioral level but still uses tokens which carry data structures with them. The functionality is embedded into the models using C or Ada programs. The capability of generating high-level VHDL models from the C or Ada models is provided. These high-level VHDL models can be further refined and developed in a VHDL environment but the refined models cannot be integrated back into the ADAS performance model. The flow of information in these high-level VHDL models is still represented by tokens. Therefore, implementation-level components cannot be integrated into an ADAS performance model. Another limitation is that all input values to the behavioral node must be contained within the token data structure.

Scientific and engineering software (SES)/Workbench is a design specification modeling and simulation tool [21]. It is used to construct and evaluate proposed system designs and to analyze their performance. A graphical interface is used to create a structural model which is then converted into a specific simulatable description (SES/sim). SES/Workbench enables the transition across domains of interpretation by using a *user node*, in which C-language and SES/sim statements can be executed. Therefore, SES/Workbench has similar limitations to ADAS; the inability to simulate a multilevel model when input values of behavioral nodes are not fully specified and the inadequacy of simulating components described as implementation-level HDLs (the capability of integrating VHDL models has been introduced later in the next paragraph). In addition, multiple simulation languages (both SES/sim and C) are required for multilevel models.

The Reveal interactor is a tool developed by Redwood Design Automation [22]. A model constructed in Reveal is aimed at the functional verification of RTL VHDL and Verilog descriptions and, therefore, does not include a separate transaction-based performance modeling capability. However, Reveal can work in conjunction with SES/Workbench. By mixing models created in Reveal and SES/Workbench, a multilevel modeling capability exists. Again, these multilevel models are very limited due to the fact that all the required information at the lower level part of the model must be available within the higher level model.

Integrated Design Automation System (IDAS) is a multilevel design environment which allows for rapid prototyping of systems [23]. Although the behavioral specifications need to be expressed as Ada, C, or FORTRAN programs, IDAS provides the capability of automatically translating VHDL description to Ada. However, the user cannot create abstract models in which certain behavior is unspecified. Also, it does not support classical performance models (such as queuing models and Petri nets) and forces the user to specify a behavioral description very early in the design process.

Transcend claims to integrate multilevel descriptions into a single environment [24,25]. In the more abstract level, T-flow models are used, in which tokens are used to represent flow of data. The capability of integrating VHDL submodels into a T-flow model is provided. However, interfacing between the two models requires a ''C++ like'' language, which map variables to/from VHDL signals, resulting in a heterogeneous simulation environment. Although their approach is geared toward the same objective as mixed-level modeling, the T-flow model must also include all the necessary data to activate the

VHDL submodels. Therefore, the upper-level model cannot be ''too abstract'' and must include lower level details.

Methodology for integrated design and simulation (MIDAS) supports the design of distributed systems via iterative refinement of partially implemented performance specification (PIPS) models [26]. A PIPS model is a partially implemented design where some components exist as simulation models and others as operational subsystems (i.e., implemented components). Although they use the term ''hybrid model'' in this context it refers to a different type of modeling. MIDAS is an ''integrated approach to software design'' [26]. It supports the performance evaluation of software being executed on a given machine. It does not allow the integration of components expressed in an HDL into the model.

The Ptolemy project is an academic research effort being conducted at the University of California at Berkeley [27,28]. Ptolemy, a comprehensive system prototyping tool, is actually constructed of multiple domains. Most domains are geared toward functional verification and have no notion of time. Each domain is used for modeling a different type of system. They also vary in the modeling level (level of abstraction). Ptolemy provides limited capability of mixing domains within one design. The execution of a transition across domains is accomplished with a ''wormhole.'' A wormhole is the mechanism for supporting the simulation of heterogeneous models. Thus, a multilevel modeling and analysis capability is provided. There are two major limitations to this approach compared with the mixed-level modeling approach being described. The first one is the heterogeneity—several description languages. Therefore, translation between simulators is required. The second one is that the interface between domains only translates data. Therefore, all the information required by the receiving domain must be generated by the transmitting domain.

Honeywell Technology Center (HTC) conducted a research effort that specifically addressed the mixed-level modeling problem [29]. This research had its basis in the UVa mixed-level modeling effort. The investigators at HTC developed a performance modeling library (PML) [30,31] and added a partial mixed-level modeling capability to this environment. The PML is used for performance models at a relatively low level of abstraction. Therefore, it assumes that all the information required by the interpreted element is provided by the performance model. In addition, their interface between uninterpreted and interpreted domains allows for bidirectional data flow.

Transaction-level modeling is a model paradigm that combines key concepts from interface-based design [32] with the system-level modeling to develop a refineable model of the system as a whole. Essentially, the system as a whole is represented with the model of computation done in components separated as much as possible from the model of communication between components.

As explained by Cai and Gajski [33],

> In a transaction-level model (TLM), the details of communication among computation components are separated from the details of computational components. Communication is modeled by channels, while transaction requests take place by calling interface functions of these channel models. Unnecessary details of communication and computation are hidden in a TLM and may be added later. TLMs speed up simulation and allow exploring and validating design alternatives at the higher level of abstraction.

There are a number of advantages to such an approach. Of course you get the advantages of a system-level model, but the separation of the modeling of the communication from the modeling of the computation also has a number of advantages. The primary advantage is that it allows the designer to focus on one aspect of the design at a time. The separation limits the complexity that the designer must contend with just as encapsulation and abstraction of component behaviors do. This also allows the designer to develop or refine the communication model of the system in the same way that they would do for the computation model.

To summarize, numerous multilevel modeling efforts exist. However, they are being developed, not addressing the issue of lack of information at the transition between levels of abstraction. The solution of

this problem is essential for true stepwise refinement of the performance models to behavioral models. In addition, integration of performance modeling level and behavioral modeling level was mostly performed by mixing different simulation environments, which results in heterogeneous modeling approach.

12.2 ADEPT Design Environment

A unified end-to-end design environment based solely on VHDL that allows designers to model systems from the abstract level to the gate level as described above has been developed. This environment supports the development of system-level models of digital systems that can be analyzed for multiple metrics like performance and dependability, and can then be used as a starting point for the actual implementation. A tool called Advanced design environment prototype tool (ADEPT) has been developed to implement this environment. ADEPT actually supports both system-level performance and dependability analysis in a common design environment using a collection of predefined library elements. ADEPT also includes the capability to simulate both system- and implementation-level (behavioral) models in a common simulation environment. This capability allows the stepwise refinement of system-level models into implementation-level models.

Two approaches to creating a unified design environment are possible. An evolutionary solution is to provide an environment that ''translates'' data from different models at various points in the design process and creates interfaces for the noncommunicating software tools used to develop these models. With this approach, users must be familiar with several modeling languages and tools. Also, analysis of design alternatives is difficult and is likely to be limited by design time constraints.

A revolutionary approach, the one being developed in ADEPT, is to use a single modeling language and mathematical foundation. This approach uses a common modeling language and simulation environment which decreases the need for translators and multiple models, reducing inconsistencies and the probability of errors in translation. Finally, the existence of a mathematical foundation provides an environment for complex system analysis using analytical approaches.

Simulators for hardware description languages accurately and conveniently represent the physical implementation of digital systems at the circuit, logic, register-transfer, and algorithmic levels. By adding a system-level modeling capability based on extended Petri nets and queuing models to the hardware description language, a single design environment can be used from concept to implementation. The environment would also allow for the mixed simulation of both *uninterpreted* (performance) models and *interpreted* (behavioral) models because of the use of a common modeling language. Although it would be possible to develop the high-level performance model and the detailed behavioral model in two different modeling languages and then develop some sort of ''translator'' or foreign language interface to hook them together, a better approach is to use a single modeling language for both models. A single modeling language that spans numerous design phases is much easier to use, encouraging more design analysis and consequently better designs.

ADEPT implements an end-to-end unified design environment based upon the use of the VHSIC hardware description language (VHDL), IEEE Std. 1076 [34]. VHDL is a natural choice for this single modeling language in that it has high-level language constructs, but unlike other programming languages, it has a built-in timing and concurrency model. VHDL does have some disadvantages in terms of simulation execution time, but techniques have been developed to help address this problem [35].

ADEPT supports the integrated performance and dependability analysis of system-level models and includes the capability to simulate both uninterpreted and interpreted models through mixed-level modeling. ADEPT also has a mathematical basis in Petri nets thus providing the capability for analysis through simulation or analytical approaches [36].

In the ADEPT environment, a system model is constructed by interconnecting a collection of predefined elements called ADEPT modules. The modules model the information flow, both data and control, through a system. Each ADEPT module has a VHDL behavioral description and a corresponding mathematical description in the form of a CPN based on Jensen's CPN model [37]. The modules

communicate by exchanging *tokens*, which represent the presence of information, using a fully interlocked, four-state handshaking protocol [38]. The basic ADEPT modules are intended to be building blocks from which useful modeling functionality can be constructed. In addition, custom modules can be developed by the user if required and incorporated into a system model as long as the handshaking protocol is adhered to. Finally, some libraries of application-specific, high-level modeling modules such as Multiprocessor Communications Network Modeling Library [39] have been developed and included in ADEPT.

The following sections discuss the VHDL implementation of the token data type and transfer mechanism used in ADEPT, and the modules provided in the standard ADEPT modeling library.

12.2.1 Token Implementation

The modules defined in this chapter have been implemented in VHDL, and use a modified version of the token passing mechanism defined by Hady [38]. Signals used to transport tokens must be of the type *token*, which has been defined as a record with two fields. The first field, labeled STATUS, is used to implement the token passing mechanism. The second field, labeled COLOR, is an array of integers that is used to hold user-defined color information in the model. The ADEPT allow the user to select from a predefined number of color field options. The tools then automatically link in the proper VHDL design library so that the VHDL descriptions of the primitive modules operate on the defined color field. The default structure of the *data-type* token used in the examples discussed in this document is shown in Figure 12.6.

There are two types of outputs and two types of inputs in the basic ADEPT modules. *Independent* outputs are connected to *control* inputs, and the resulting connection is referred to as a *control-type* signal. *Dependent* outputs are connected to *data* inputs, and the resulting connection is referred to as a *data-type* signal. To make the descriptions more intuitive, outputs are often referred to as the ''source side'' of a signal, and inputs are referred to as the ''sink side'' of a signal.

```
type handshake is (removed, acked, released, present);
type token_fields is (status,
                      tag1, tag2, tag3, tag4, tag5,
                      tag6, tag7, tag8, tag9, tag10,
                      tag11, tag12, tag13, tag14, tag15,
                      boole1, boole2, boole3,
                      fault, module_info,
                      index, act_time, color);
type color_type is array (token_fields range tag1 to act_time) of integer;

type token is
     record
        status    : handshake;
        color     : color_type;
     end record;

type op_fields is (add, sub, mul, div);
type cmp_fields is (lt, le, gt, ge, eq, ne);
type tkf_array is array (1 to 25) of token_fields;-- used in file_read

type token_vector is array (integer range <>) of token;

--  resolution function for token
function protocol (input : token_vector) return token;
subtype token_res is protocol token;
```

FIGURE 12.6 Token type definition.

Tokens on *independent* outputs may be written over by the next token, so the ''writing'' process is *independent* of the previous value on the signal. In contrast, new tokens may not be placed on *dependent* outputs until the previous token has been removed, so the ''writing'' process is *dependent* on the previous value on the signal. Data-type signals use the four-step handshaking process to ensure that tokens do not get overwritten, but no handshaking occurs on *control-type* signals.

The STATUS field of a token signal can take on four values. Signals connecting *independent* outputs to *control* inputs (control-type signals) make use of only two of the four values. *Independent* outputs place a value PRESENT on the status field to indicate that a token is present. Since *control* inputs only copy the token but do not remove it, the *independent* output only needs to change the value of the status field to RELEASED to indicate that the token is no longer present on the signal, and the *control* input can no longer consider the signal to contain valid information. The signals connecting *dependent* outputs to *data* inputs (data-type signals) need all four values (representing a fully interlocked handshaking scheme) to ensure that a dependent output does not overwrite a token before the data input to which it is connected has read and removed it. This distinction is important since a token on control-type signals represents the presence or absence of a condition in the network while tokens on data-type signals, in contrast, represent information or data that cannot be lost or overwritten. In addition, fanout is not permitted on *dependent* outputs and is permitted on *independent* outputs.

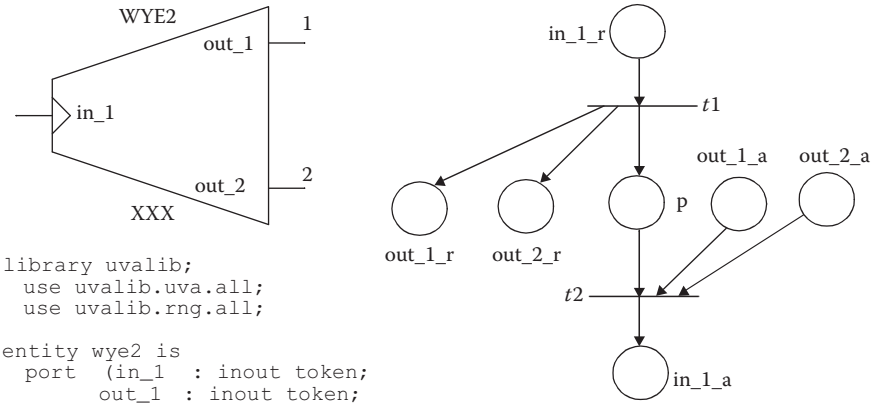
The signals used in the implementation are of type *token*. The token passing mechanism for data-type signals is implemented by defining a VHDL bus resolution function. This function is called each time a signal associated with a dependent output and a data input changes value. The function essentially looks at the value of the STATUS field at the ports associated with the signal and decides the final value of the signal. At the beginning of simulation the STATUS field is initialized to REMOVED. This value on a signal corresponds to an idle link in the nets defined by Dennis [40]. This state of a signal indicates that a signal is free and a request token may be placed on it. Upon seeing the REMOVED value, the requesting module may set the value of the signal to PRESENT. This operation corresponds to the ready signal in Dennis' definition. The requested module acknowledges a ready signal (PRESENT) by setting the signal value to ACKED, after the requested operation has been completed. Upon sensing the value of ACKED on the signal, the requesting module sets the value of the signal to RELEASED, which in turn causes the requested module to place the value of REMOVED on the signal. This action concludes one request/acknowledge cycle, and the next request/acknowledge cycle may begin.

In terms of Petri Nets, the models described here can be described as one-safe Petri nets since, at any given time, no place in the net can contain more than one token. The correspondence between the signal values and transitions in a Petri net may be defined, in general, in the following manner:

1. *PRESENT*. A token arrives at the input place of a transition.
2. *ACKED*. The output place of a transition is empty.
3. *RELEASED*. The transition has fired.
4. *REMOVED*. The token has been transferred from the input place to the output place.

The modules to be described in this chapter may be defined in terms of Petri nets. As an example, a Petri net description of the Wye module is shown in Figure 12.7.

The function of the Wye module is to copy the input token to both outputs. The input token is not acknowledged until both output tokens have been acknowledged. In the Petri net of Figure 12.7, the ''r'' and ''a'' labels correspond to ''ready'' and ''acknowledge,'' respectively. When a token arrives at the place labeled ''0r,'' the top transition is enabled and a token is placed in the ''1r,'' ''2r,'' and center places. The first two places correspond to a token being placed on the module outputs. Once the output tokens are acknowledged (corresponding to tokens arriving at the ''1a'' and ''2a'' places), the lower transition is enabled and a token is placed in ''0a,'' corresponding to the input token being acknowledged. The module is then ready for the next input token. Note that since this module does not manipulate the color fields, no color notation appears on the net.




```
library uvalib;
  use uvalib.uva.all;
  use uvalib.rng.all;

entity wye2 is
  port  (in_1  : inout token;
         out_1  : inout token;
         out_2  : inout token);
end wye2;

architecture ar_wye2 of wye2 is
begin
  pr_wye2 : process (in_1, out_1, out_2)
  begin
    if token_present (in_1)
      and token_removed (out_1)
      and token_removed (out_2) then
      out_1 <= in_1;
      out_2 <= in_1;
    end if;

    if token_acked (out_1) and token_acked (out_2) then
     ack_token (in_1);
     release_token (out_1);
     release_token (out_2);
    end if;

    if token_released (in_1) then
      remove_token (in_1);
    end if;
  end process pr_wye2;
end ar_wye2;
```

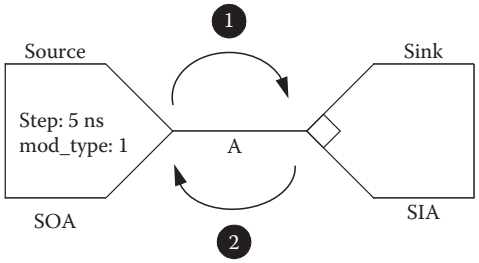
FIGURE 12.7 Wye module ADEPT symbol, its behavioral VHDL description, and its CPN representation.

The specific CPN description used is based on the work of Jensen [37]. The complete CPN descriptions of each of the ADEPT building blocks can be found in Ref. [41].

12.2.2 ADEPT Handshaking and Token Passing Mechanism

Recall that the ADEPT standard token has a *status* field which can take on four values: *PRESENT, ACKED, RELEASED,* and *REMOVED*. These values reflect which stage of the token passing protocol is currently in progress. Several examples will now be presented to show how the handshaking and token passing occurs between ADEPT modules. Figure 12.8 illustrates the handshaking process between a Source module connected to a Sink module.

Figure 12.8a describes the ''simplified'' event sequence. Here, we can think of the four-step handshaking as consisting of simply a forward propagation of the token and a backward propagation of the token acknowledgment. Thinking of the handshaking in this simplified manner, Table A in Figure 12.8 shows the simplified event sequence and times. At time 0 ns, the Source module places a token on A, corresponding to Event 1. The token is immediately acknowledged by the Sink module, corresponding

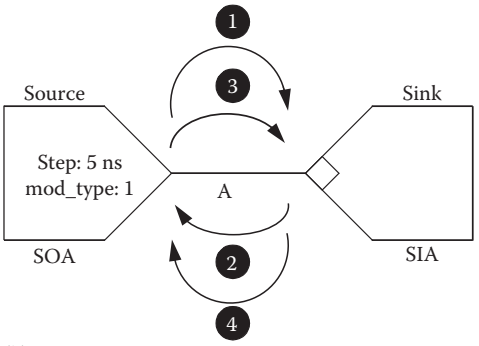


Time between new tokens on A = step + path delay = 5 ns

(a)

TABLE A Simplified Event Sequence

Event	Time (ns)	Description
1	0	Source module places token on signal A
2	0	Sink module acknowledges token on A
1	5	Source module places next token on signal A



(b)

TABLE B Detailed Event Sequence

Event	Time (ns)	Delta	Description	Resolved Signal A
1	0	1	Source module places token on A	*PRESENT*
2	0	2	Sink module acknowledges token on A	*ACKED*
3	0	3	Source module release token on A	*RELEASED*
4	0	4	Sink module removes token on A	*REMOVED*
1	5	1	Source module places token on A	*PRESENT*

FIGURE 12.8 Two module handshaking examples.

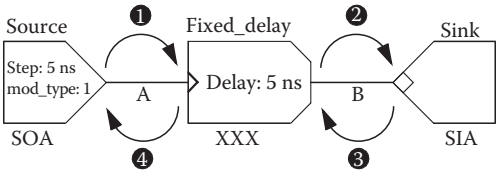
to Event 2. Since the Source module has a *step* generic of 5 ns, and there is no other delay along the path to the Sink, the next token will be output by the Source module at time 5 ns.

Figure 12.8b details the entire handshaking process. Table B in Figure 12.8 lists the detailed event sequence for this example. Since handshaking occurs in zero simulation time, the events are listed by *delta cycles* within each simulation time. The function of delta cycles in VHDL is to provide a means for ordering and synchronizing events (such as handshaking) which occur in zero time. The actual event sequence for this example consists of four steps, as shown in Figure 12.8b. Event 1 is repeated at time 5 ns, which starts the handshaking process over again.

12.2.2.1 Three-Module Example

To illustrate how tokens are passed through intermediate modules, a three-module example will now be examined. Consider the case where a Fixed Delay module is placed between a Source and Sink. This situation is illustrated in Figure 12.9.

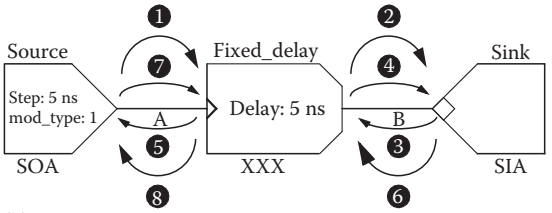
Figure 12.9a shows the simplified event sequence, where we can think of the four-step handshaking as consisting of simply a forward propagation of the token and a backward propagation of the token acknowledgment. Table A in Figure 12.9 lists this simplified event sequence. Notice that since now there is a path delay from the Source to the Sink, the time between new tokens from the Source is *step* + *path_delay* = 5 + 5 = 10 ns. At time 0 ns, the Source module places the first token on Signal A (Event 1). This token is read by the Delay module and placed on Signal B after a delay of 5 ns (Event 2). The Sink module then immediately acknowledges the token on Signal B (Event 3). The Delay module then passes



(a)

TABLE A Simplified Event Sequence

Event	Time (ns)	Description
1	0	Source module places token on signal A
2	5	Delay module places token on signal B
3	5	Sink module acknowledges token on signal B
4	5	Delay module acknowledges token on signal A
1	10	Source module places next token on signal A



(b)

TABLE B Detailed Event Sequence

Event	Time (ns)	Delta	Description	Resolved Signal A	Resolved Signal B
1	0	1	Source module places token on A	*PRESENT*	*REMOVED*
2	5	1	Delay module places token on B	—	*PRESENT*
3	5	2	Sink module acknowledges token on B	—	*ACKED*
4	5	3	Delay module releases token on B	—	*RELEASED*
5			Delay module acknowledges token on A	*ACKED*	—
6	5	4	Sink module removes token on B	—	*REMOVED*
7			Source module relases token on A	*RELEASED*	—
8	5	5	Delay module removes token on A	*REMOVED*	—
1	10	1	Source module places token on A	*PRESENT*	—

FIGURE 12.9 Three-module handshaking examples.

this acknowledgment back through to Signal A (Event 4). At time 10 ns, the Source module places the next token on Signal A, starting the process over again.

Figure 12.9b shows the detailed event sequence for this example. Since there are two signals, there are a total of eight detailed events. Table B in Figure 12.9 lists the detailed event sequence for this example. Again since the handshaking occurs in zero simulation time, the events are listed by delta cycles within each simulation time. Table B lists the resolved values on both Signal A and Signal B, where a value of "—" indicates no change in value. Notice that the sequence of events on each signal proceeds in this order: place the token (*PRESENT*) by the "source side," acknowledge the token (*ACKED*) by the "sink side," release the token (*RELEASED*) by the "source side," and finally remove the token (*REMOVED*) by the "sink side." The important concept to note in this example is the ordering of events. Notice that the Delay module releases the token on its output (Event 4) before it acknowledges the token on its input (Event 5), even though the two events occur in the same simulation time. These actions trigger the Sink module to then remove the token on Signal B (Event 6) and the Source module to release the token on Signal A (Event 7), both in the next delta cycle. Thus Signal B is ready for another token (removed) a full delta cycle before Signal A is ready.

12.2.2.2 Token Passing Example

To illustrate how tokens propagate in a larger system, consider the model shown in Figure 12.10. This figure shows the simplified event sequence, where again we can think of the four-step handshaking as consisting of simply a forward propagation of the token and a backward propagation of the token acknowledgment. In this system, the Source module provides tokens to the Sequence A (SA) module (Event 1), who first passes them to the Sequence B (SB) module (Event 2). The SB module first passes the token to the Fixed Delay A (FDA) module (Event 3), who passes it to the Sink A (SIA) module after a delay of 5 ns (Event 4). The SIA module then immediately acknowledges the token (Event 5), which causes the FDA module to pass the acknowledgment back to the SB module (Event 6). At this point the SB module places a copy of the token on its second output (Event 7), where it is passed through the

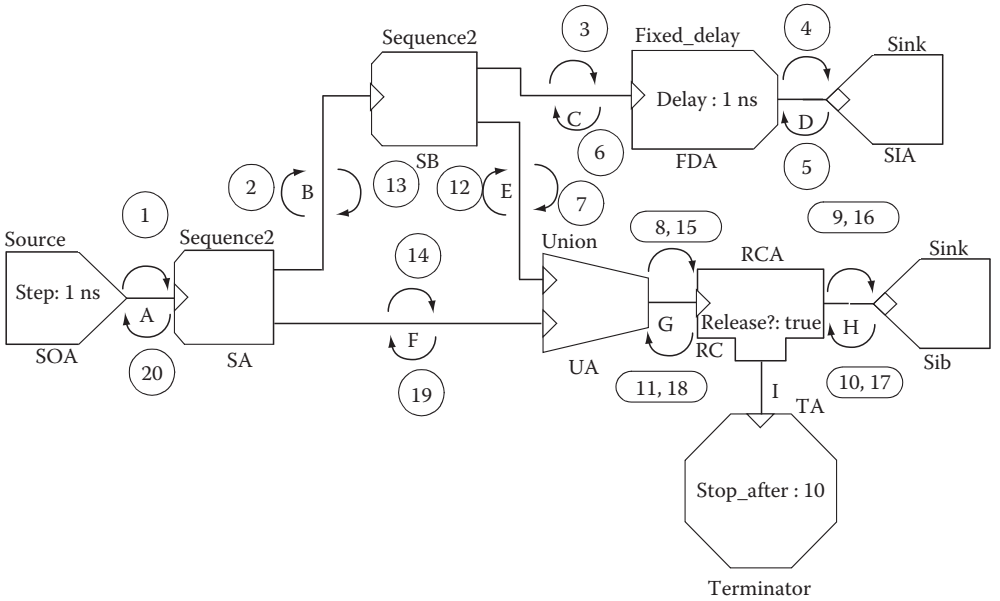


FIGURE 12.10 Token passing example showing simplified event sequence.

Union A and onto its output (Event 8). The Read Color A module then places the token on its out_1 output (Event 9) and simultaneously places the token on its *independent* output. The Sink B module then acknowledges the token on Signal H (Event 10). Upon seeing this acknowledgment, the Read Color A module releases the token on its *independent* output since the Read Color module has no ''memory'' (the *release?* generic equals *true*). The acknowledgment is then propagated back to the SB module (Events 11 and 12). At this point, the SB module acknowledges its input token (Event 13), freeing the SA module to place a token on its second output (Event 14). This token is then passed through the Union A, Read Color A, and Sink module (Events 15 and 16) and the acknowledgment is returned (Events 17–19). The SA module can then acknowledge the token on its input (Event 20), allowing the Source module to generate the next token 5 ns (*step*) later.

If we examine the activity on Signal I (the *independent* output of the Read Color A module), we see that it goes present after Event 8, released after Event 10, present after Event 15, and released again after Event 17. Consider the operation of the Terminator module attached to this signal. Since the Terminator module halts simulation after the number of active events specified by the *stop_after* generic, if we were to simulate this example, the simulation would halt after Event 15 (the second active event).

12.2.3 Module Categories

The ADEPT primitive modules may be divided into six categories: *control modules*, *color modules*, *delay modules*, *fault modules*, *miscellaneous parts modules*, and *mixed-level modules*.

As the name implies, the *control modules* are used to control the flow of tokens through the model. As such, the control modules form the major portion of the performance model. The control modules operate only on the STATUS field of a signal and do not alter the color of a token on a signal. Further, no elapsed simulation time results from the activation of the control modules since they do not have any delay associated with them. There are several control modules whose names end in a numeral, such as ''union2.'' These modules are part of a family of modules that have the same function, but differ in the number of inputs or outputs that they possess. For example, there are eight ''union'' modules, ''union2,'' ''union3,'' ''union4,'' . . . , ''union8.'' With the exception of the Switch, Queue, and logical modules, the control modules have been adapted from those proposed by Dennis [40].

The control modules described above process colored tokens but do not alter the color fields of the tokens passing through them. Manipulation of the color field of a token is reserved to the *color modules*. These color modules permit various operations on the color fields such as allowing the user to read and write the color fields of the tokens. The color modules also permit the user to compare the color information carried by the tokens and to control the flow of the tokens based on the result of the comparisons. The use of these color modules enables high-level modeling of systems at different levels of detail. These modules permit the designer to add more detail or information to the model by placing and manipulating information placed on the color fields of tokens flowing through the model. The color fields of these tokens can be set and read by these modules to represent such things as destination or source node addresses, data length, data type (e.g., ''digital'' or ''analog''), or any type of such information that the designer feels is important to the design.

The *delay modules* facilitate the description of data path delays at the conceptual or block level of a design. As an example, the Fixed Delay module, or the FD module, may be used in conjunction with the control modules to model the delay in the control structure of a design which is independent on the type or size of an operation being performed. In contrast, the Data-Dependent Delay module may be used to model processes in which the time to complete a particular operation is dependent on the amount or type of data being processed. The input to the Data-Dependent Delay module is contained on one of the color fields of the incoming token.

The *fault modules* are used to represent the presence of faults and errors in a system model. The modules allow the user to model fault injection, fault/error detection, and error correction processes. The *miscellaneous parts* category contains modules that perform ''convenience'' functions in ADEPT.

Examples include the Collector, Terminator, and Monitor modules. The Collector module is used to write input activation times to a file, and the Terminator module is used to halt simulation after a specified number of events have occurred. The Monitor module is a data-collection device that can be connected across other modules to gather statistical information during a VHDL simulation.

The *mixed-level modules* support mixed-level modeling in the ADEPT environment by defining the interfaces around the interpreted and uninterpreted components in a system model. The functions of these modules and their use in creating mixed-level models are described in more detail in Sections 12.4.2 and 12.4.3.

The complete functionality of each primitive module is defined and the generics associated with each of the modules are described in the *ADEPT Library Reference Manual* [42]. The standard ADEPT module symbol convention is also explained in more detail in the *ADEPT Library Reference Manual.*

In addition to the primitive modules, there are libraries of more complex modules included in ADEPT. In general, these libraries contain modules for modeling systems in a specific applications area. These libraries are also discussed in more detail in the *ADEPT Library Reference Manual.*

12.2.4 ADEPTs

The ADEPT system is currently available on Sun platforms using Mentor Graphics' *Design Architect* as the front end schematic capture system, or on Windows PCs using OrCAD's *Capture* as the front end schematic capture system. The overall architecture of the ADEPT system is shown in Figure 12.11.

The schematic front end is used to graphically construct the system model from a library of ADEPT module symbols. Once the schematic of the model has been constructed, the schematic capture system's netlist generation capability is used to generate an electronic design interchange format (EDIF) 2.0.0 netlist of the model. Once the EDIF netlist of the model is generated, the ADEPT software is used to translate the model into a structural VHDL description consisting of interconnections of ADEPT

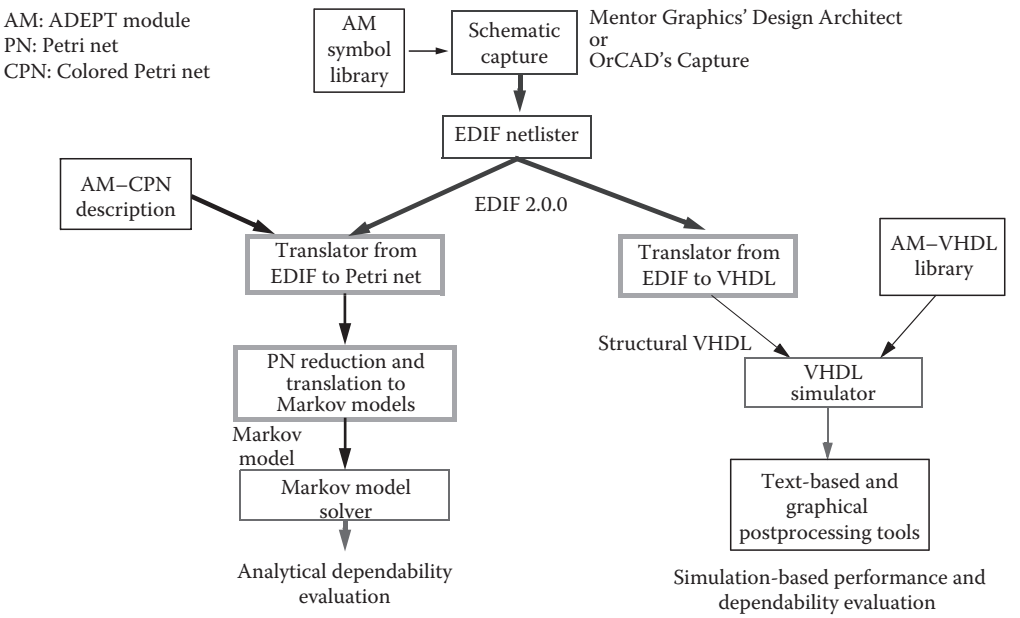


FIGURE 12.11 ADEPT design flow.

modules. The user can then simulate the structural VHDL that is generated using the compiled VHDL behavioral descriptions of the ADEPT modules to obtain performance and dependability measures.

In addition to VHDL simulation, a path exists that allows the CPN description of the system model to be constructed from the CPN descriptions of the ADEPT modules. This CPN description can then be translated into a Markov model using well-known techniques and then solved using commercial tools to obtain reliability, availability, and safety information.

12.3 Simple Example of an ADEPT Performance Model

This section presents a simple example of the usage of the primitive building blocks for performance modeling. The example is a three-computer system wherein the three-computers share a common bus. The example also presents simulation results and system performance evaluations.

12.3.1 Three-Computer System

This section discusses a simple example to illustrate how the modules discussed previously may be interconnected to model and evaluate the performance of a complete system. The system to be modeled consists of three computers communicating over a common bus, as shown in Figure 12.12. Each block representing a computer can be thought to contain its own processor, memory, and peripheral devices. The actual ADEPT schematic for the three-computer system is shown in Figure 12.13.

Computer C1 contains some sensors and preprocessing capabilities. It collects data from the environment, converts it into a more compact form and then sends it to computer C2 via the bus. The computer C2 further processes the data and passes it to computer C3 where the data are appropriately utilized. It is assumed that data are transferred in packets and each packet of data is of varying length. In the example described here, computers C1 and C2 receive packets whose sizes are uniformly distributed between 0 and 100. The packet size of computer C3 is uniformly distributed between 0 and 500. The external environment in this example is modeled by a Source module in C1 and a Sink module in C3.

C1 has an output queue, C2 has both an input queue and an output queue, while C3 has one input queue. All queues in this example are assumed to be of length 8. If the input queues of C2 or C3 are full the corresponding Q_free signal is released (value = *RELEASED*). This interconnection prevents the

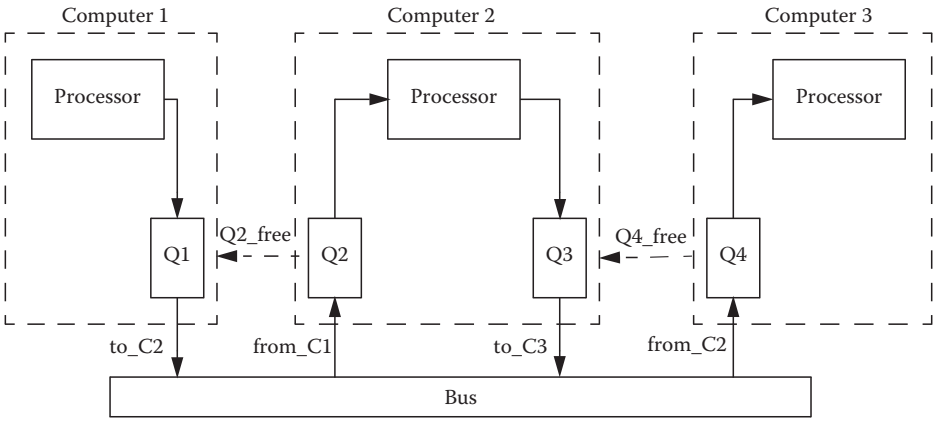


FIGURE 12.12 Three-computer system.

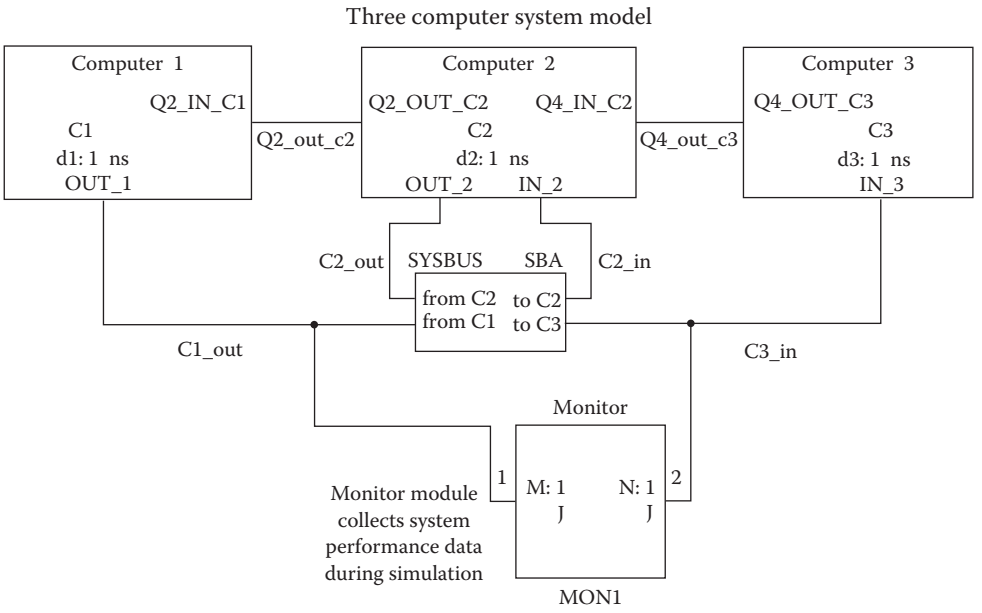


FIGURE 12.13 Three-computer system ADEPT schematic.

computer writing into the corresponding queue from placing data on the bus when the queue is full. This technique not only prevents the bus from being unnecessarily held up but also eliminates the possibility of a deadlock.

The ADEPT model of Computer 1 (C1) is shown in Figure 12.14. The Source A (SOA) module along with the Sequence A (SA), Set_Color A (SCA) and Random A (RA) modules generate tokens whose *tag1* field is set according to a distribution which is representative of the varying data sizes that C1 receives. The Data-Dependent Delay A (DDA) models the processing time of C1 which is directly proportional to the packet size. The *unit_step* delay for the DDA module is passed down as a generic *d1**1 ns. As soon as a token, representing one packet of data, appears at the output of the DDA module, the Sequence B (SB), Set_Color B (SCB), and Random B (RB) modules together set the *tag1* field of the token to represent the packet size after processing. The Constant A (COA) and Set_Color C (SCC) modules set the *tag2* field of the token to 2. This coloring indicates to the bus arbitration unit that the token is to be transferred to C2. The token is then placed in the output Queue (Q1) of C1 and the token is acknowledged. This acknowledge signal is passed back to the Source A module which then produces the next token. The Fixed Delay (FDA and FDB) modules represent the read and write times associated with the Queue. The Switch A (SWA) element is controlled by the Q_free signal from C2 and prevents a token from the output queue of C1 from being placed on the bus if the incoming Q_free signal is inactive.

Figure 12.15 shows the ADEPT model of Computer 2 (C2). When a token arrives at the *data* input of C2 the token is placed at the input of the Queue (Q2). The *control* output of the Queue becomes the Q2_Free output of C2. The remaining modules perform the same function as in C1 except that the tag2 field of the tokens output by C2 is set to 3 which indicates to the bus arbitration unit that the token is to be sent to C3. The DDA module represents the relative processing speed of Computer C2. The *unit_step* delay for the DDA module is passed down as a generic *d2**1 ns.

The modules defining Computer 3 (C3) are shown in Figure 12.16. A token arriving at the input is placed in the queue. The DDA element reads one token at a time and provides the delay associated with

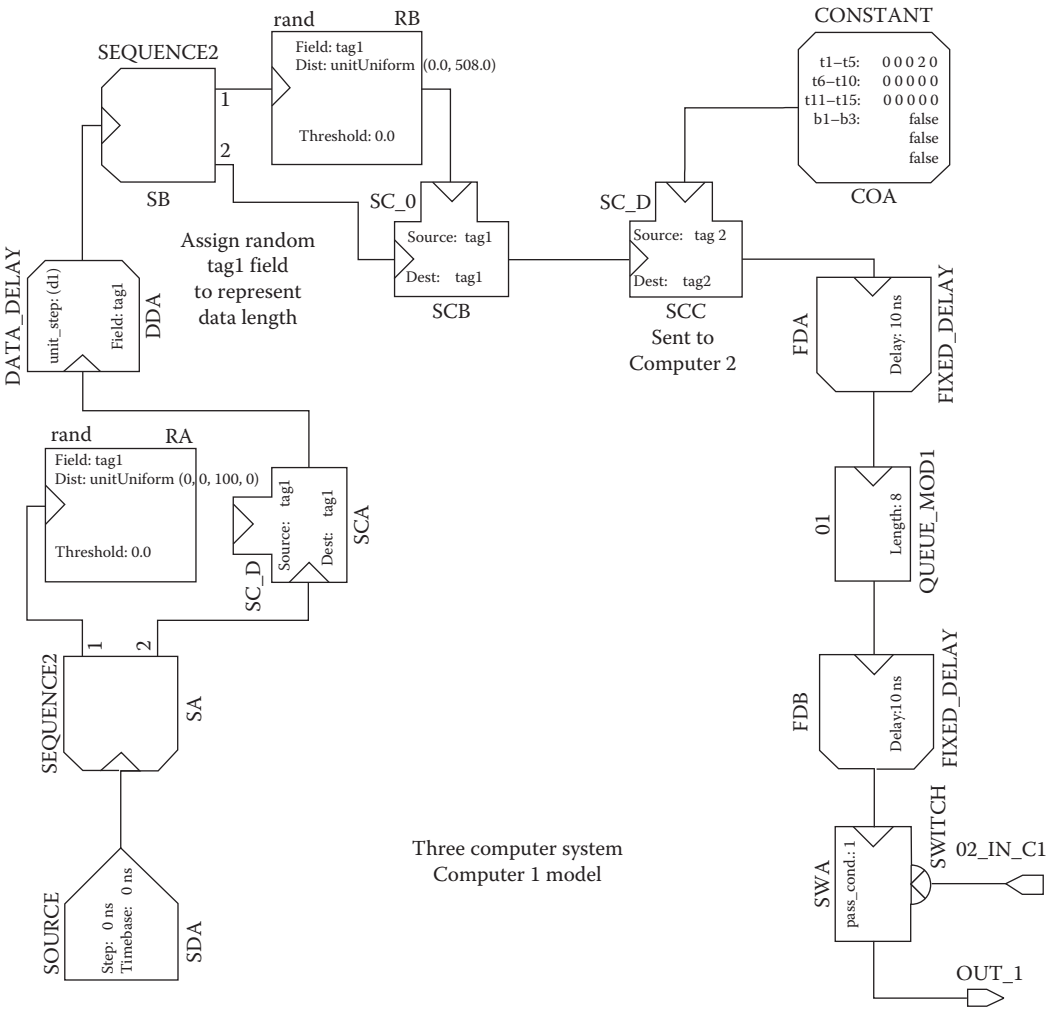


FIGURE 12.14 ADEPT model of Computer 1.

the processing time of C3 before the Sink A (SIA) module removes the token. The *unit_step* delay for the DDA module is passed down as a generic *d3**1 ns.

The bus is shown in Figure 12.17. Arbitration is provided to ensure that both C1 and C2 do not write onto the bus at the same time. The Arbiter A (AA) element provides the required arbitration. Since the output of C2 is connected to the IN_1(1) input of the AA element, it has a higher priority over C1. The Union A (UA) element passes a token present at either of its inputs to its output. The output of the UA element is connected to the Data-Dependent Delay A (DDA) element, which models the packet transfer delay associated with moving packets over the bus. The delay through the bus is dependent on the size of the packet of information being transferred (size stored on the *tag1* field). Note that in this analysis, the bus delay was set to 0. The *independent* output of the Read_Color A(RCA) element is connected to the *control* input of the Decider A (DA) module. The base of the DA element is set to 2. Since the *tag2* field of the token is set to 2 or 3 depending on whether it originated from C1 or C2, the first output of the Decider A module is connected to C2 and the second output is connected to C3. This technique ensures that the tokens are passed on to the correct destination, C2 or C3.

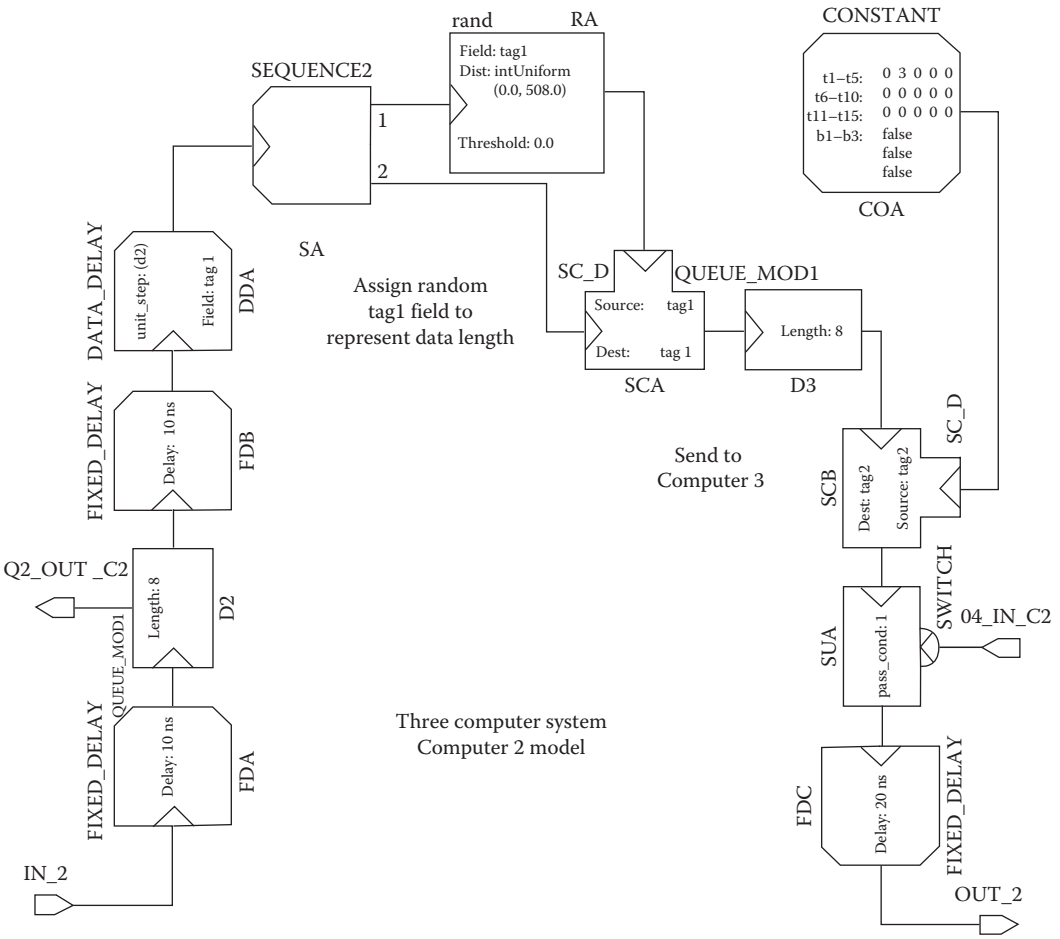


FIGURE 12.15 ADEPT model of Computer 2.

12.3.2 Simulation Results

This section presents simulation results that illustrate the queuing model capabilities of the ADEPT system. The model enables the study of the overall speed of the system in terms of the number of packets of information transferred in a given time. It also allows the study of the average number of tokens present in each queue during simulation, and the effect of varying system parameters on the number of items in the queues and overall throughput of the system. The generic unit_step delay of the DD elements (*d1*, *d2*, and *d3*) associated with the three computers is representative of the processing speed of the computers. The optimal relative speeds of the computers may also be deduced by simulation of this model. Figure 12.18 shows graphs of the number of items in each queue versus simulation time. These graphs were generated using the BAARS postsimulation analysis tool. The upper graph shows the queue lengths when *d1*, *d2*, and *d3* was set to 5, 5, and 2 ns. The lower graph shows the queue lengths when *d1*, *d2*, and *d3* was set to 5, 4, and 1 ns. In the first case, because the processing time of Computer 3 was so much longer than Computer 1 or 2, the queue in Computer 3 became full at ~4000 ns of simulation time. The filling of the queue in Computer 3 delayed items coming out of Computers 1 and 2 thus causing their queues to also become full. In the second case, the processing time ratios were such that Computer 3 could keep up with the incoming tokens and the queues never got completely full.

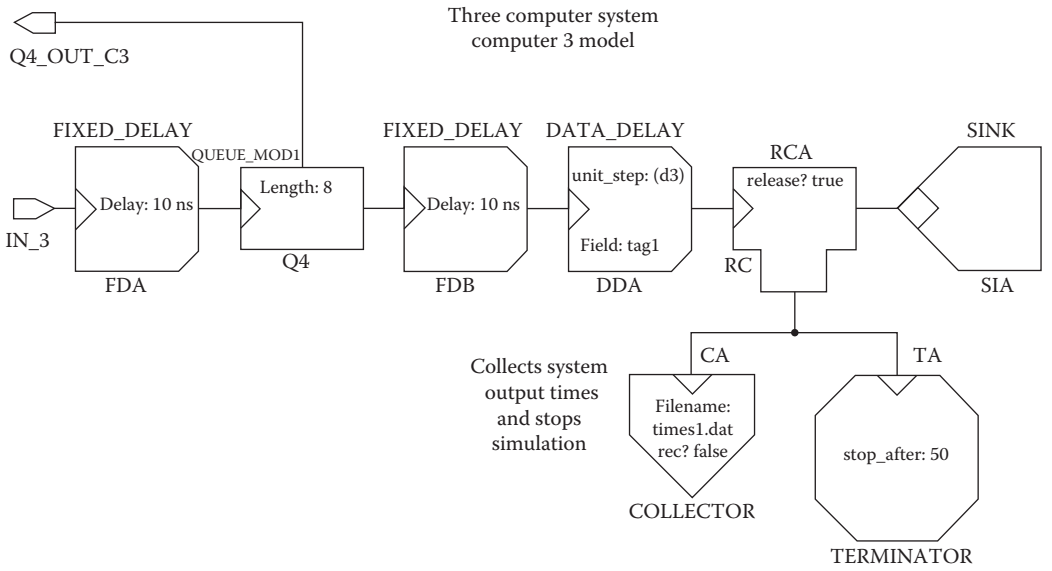


FIGURE 12.16 ADEPT model of Computer 3.

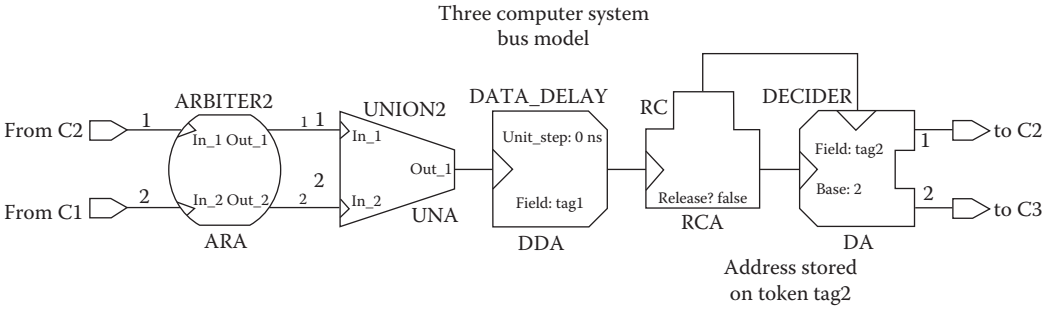


FIGURE 12.17 ADEPT model of system bus.

Table 12.1 summarizes the effect of relative speeds of the computers on the number of packets transferred. Since the size of the packets received by C3 is uniformly distributed between 0 and 500 while the size of the packets received by C1 and C2 is uniformly distributed between 0 and 100, it is intuitively obvious that the overall throughput of the system is largely determined by the speed of Computer C3. The results do indicate this behavior. For example, when the relative instruction execution times for C1, C2, and C3 are 5, 5, and 2, respectively, a total of 197 packets are transferred. By increasing the instruction execution time of C2 by one time unit and decreasing the instruction execution time of C3 by one time unit, it is seen that a total of 321 packets are transferred, an increase of slightly over 60%.

This example has illustrated the use of the various ADEPT modules to model a complete system. Note how the interconnections between modules describing a component of the system are similar to a flow chart describing the behavior of the component. This example also demonstrated that complex systems at varying levels of abstraction and interpretation can easily be modeled using the VHDL-based ADEPTs.

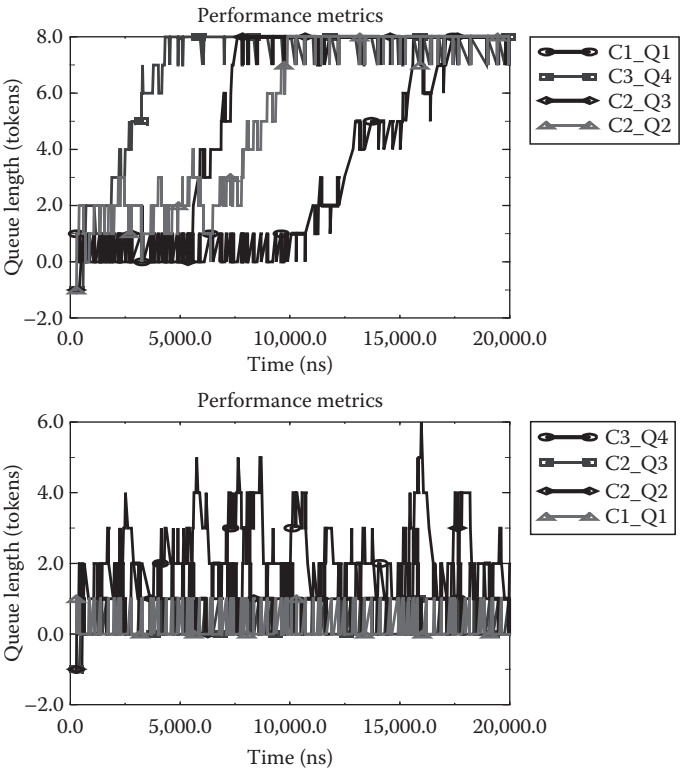


FIGURE 12.18 Queue lengths versus simulation time for various processing delay times.

TABLE 12.1 System Throughput versus Computer Delay Times

C1 Delay (ns)	C2 Delay (ns)	C3 Delay (ns)	Packets per 100,000 Time Units (ns)
8	8	3	131
9	10	2	194
5	5	2	197
3	5	2	197
2	2	2	197
5	6	1	321
5	4	1	322
4	3	1	384
3	2	1	385
2	2	1	386

12.4 Mixed-Level Modeling

As described earlier, performance (uninterpreted) modeling has been previously used primarily by systems engineers when performing design analysis at the early stages of the design process. Although most of the design detail is not included in these models since this detail does not yet exist, techniques such as token coloring can be used to include that design detail that is necessary for an accurate model.

However, most of the detail, especially very low-level information such as word widths and bit encoding, are not present. In fact, many additional design decisions must be made before an actual implementation could ever be constructed. In performance models constructed in ADEPT, abstract tokens are used to represent this information (data and control) and its flow in the system. An illustration of such a performance model with its analysis was given the three-computer system described in Section 12.3.1. In contrast, behavioral (interpreted) models can be thought of as including much more design detail often to the point that an implementation could be constructed. Therefore, data values and system timing are available and variables typically take on integer, real, or bit values. For example, a synthesizable model of a carry-lookahead adder would be one extreme of a behavioral model.

It should be obvious that the two examples described above are extremes of the two types of models. Extensive design detail can be housed in the tokens of a performance model and information in a behavioral model can be encapsulated in a very abstract type. However, there is always a difference in the abstraction level of the two modeling types. Therefore, to develop a model that can include both performance and behavioral models, interfaces between the different levels of design abstraction, called *mixed-level interfaces,* must be included in the overall model to resolve differences between these two modeling domains.

The mixed-level interface is divided into two primary parts which function together; the part that handles the transition from the uninterpreted domain to the interpreted domain (U/I) and the part that handles the transition from the interpreted domain to the uninterpreted domain (I/U). The general structure of a mixed-level model is shown in Figure 12.19.

In addition to the tokens-to-values (U/I) and values-to-tokens (I/U) conversion processes, the mixed-level interface must resolve the differences in design detail that naturally exist at the interface between uninterpreted and interpreted elements. These differences in detail appear as differences in data and timing abstraction across the interface. The differences in timing abstraction across the interface arise because the components at different levels model timing events at different granularities. For example, the passing of a token that represents a packet of data being transferred across a network in

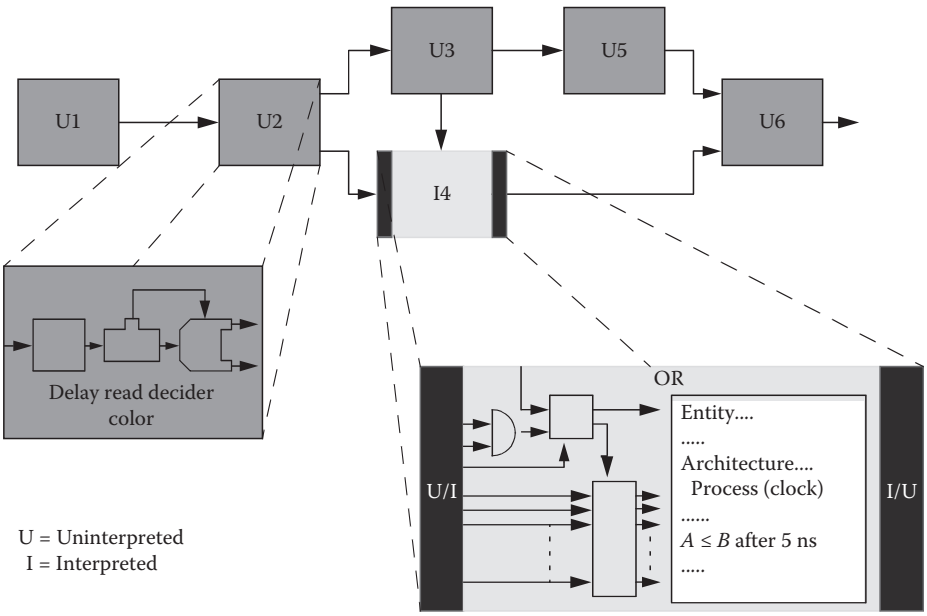


FIGURE 12.19 General structure of a mixed-level model.

a performance model may correspond to hundreds or thousands of bus cycles for a model at the behavioral level.

The differences in data abstraction across the interface are due to the fact that typically, performance models do not include full functional details whereas behavioral models require full functional data (in terms of values on their inputs) to be present before they will execute correctly. For example, a performance level modeling token arriving at the inputs to the mixed-level interface for a behavioral floating point coprocessor model may contain information about the operation the token represents, but may not contain the data upon which the operation is to take place. In this case, the mixed-level interface must generate the data required by the behavioral model and do it in a way that a meaningful performance metric, such as best- or worst-case delays, is obtained.

12.4.1 Mixed-Level Modeling Taxonomy

The functions that the mixed-level interface must perform and the most efficient structure of the interface are affected by several attributes of the system being modeled and the model itself. To partition the mixed-level modeling space and better define the specific solutions, a taxonomy of mixed-level model classes has been developed [43]. The classes of mixed-level modeling are defined by those model attributes which fundamentally alter the development and the implementation of the mixed-level interface. The mixed-level modeling space is partitioned according to three major characteristics:

1. Evaluation *objective* of the mixed-level model
2. *Timing mechanism* of the uninterpreted model
3. *Nature* of the interpreted element

For a given mixed-level model, these three characteristics can be viewed as attributes of the mixed-level model and the analysis effort. Figure 12.20 summarizes the taxonomy of mixed-level models.

Mixed-level modeling objectives. The structure and the functionality of the mixed-level interface are strongly influenced by the objective that the analysis of the mixed-level model will be targeted toward. For the purposes of this work, these objectives were broken down into two major categories:

1. *Performance analysis* and *timing verification*. To analyze the performance of the system (as defined previously) and verify that the specific components under consideration meet system timing constraints. Note that other metrics, such as power consumption, could be analyzed using mixed-level models, but these were outside the scope of this classification.
2. *Functional verification*. To verify that the function (input-to-output value transformation) of the interpreted component is correct within the context of the system model.

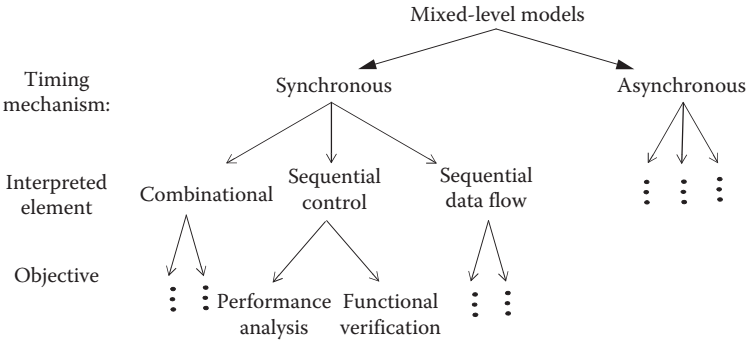


FIGURE 12.20 Mixed-level modeling categories.

Timing mechanisms. Typically, performance (uninterpreted) models are asynchronous in nature and the flow of tokens depends on the handshaking protocol. However, in modeling a system that is globally synchronous (all elements are synchronized to a global clock) a mechanism to synchronize the flow of tokens across the model can be introduced. This synchronization of the performance model will require different mixed-level modeling approaches. Thus the two types of system models that affect the mixed-level interface are

1. *Asynchronous models*. Tokens on independent signal paths within the system model move asynchronously with respect to each other and arrive at the interface at different times.
2. *Synchronous models*. The flow of tokens in the system model is synchronized by some global mechanism and they arrive at the interface at the same time, according to that mechanism.

Interpreted component. The mixed-level modeling technique strongly depends upon the type of the interpreted component that is introduced into the performance model. It is natural to partition interpreted models into those that model combinational elements and sequential elements. Techniques for constructing mixed-level interfaces for models of combinational interpreted elements have been developed previously [44]. In the combinational element case, the techniques for resolving the timing across the interface were more straightforward because of the asynchronous nature of the combinational elements, and the data-abstraction problem was solved using a methodology similar to the one presented here. However, using sequential interpreted elements in mixed-level models requires more complex interfaces to solve the synchronization problem and different specific techniques for solving the data-abstraction problem. Further research into the problem of mixed-level modeling with sequential elements suggested that interpreted components be broken down into three classifications as described below:

1. *Combinational elements*. Unclocked (with no states) elements
2. *Sequential control elements* (SCEs). Clocked elements (with states) that are used for controlling data flow, e.g., a control unit or a controller
3. *Sequential data flow elements* (SDEs). Elements that include datapath elements *and* clocked elements that control the data flow, e.g., control unit and datapath

The major reason for partitioning the sequential elements into SDEs and SCEs is based on the timing attributes of these elements. In a SCE control input values are read every cycle and control output values (that control a datapath) are generated every cycle. In contrast, SDEs have data inputs and may have some control inputs but the output data are usually generated several clock cycles later. This difference in the timing attributes will dictate a different technique for mixed-level modeling. Because the solution for the timing and data-abstraction problem for SDEs is more complex, and the solution for SCEs can be derived from the solution for SDEs, developing a solution for SDEs was the focus of this work.

12.4.2 Interface for Mixed-Level Modeling with FSMD Components

This section describes the interface structure and operation for sequential interpreted elements that can be described as finite state machines with datapaths (FSMDs) [45]. They consist of a finite state machine (FSM) used as a control unit, and a datapath, as shown in Figure 12.21. System models are quite often naturally partitioned to blocks which adhere to the FSMD structure. Each of these FSMDs processes the data and have some processing delays associated with them. These FSMDs indicate the completion of a processing task to the rest of the system either by asserting a set of control outputs or by the appearance of valid data on their outputs. This characteristic of being able to determine when the data-processing task of the FSMD is completed is a key property in the methodology of constructing mixed-level models with FSMD interpreted components.

The functions performed by the U/I operator include placing the proper values on the inputs to the FSMD interpreted model, and generating a clock signal for the FSMD interpreted model. The required

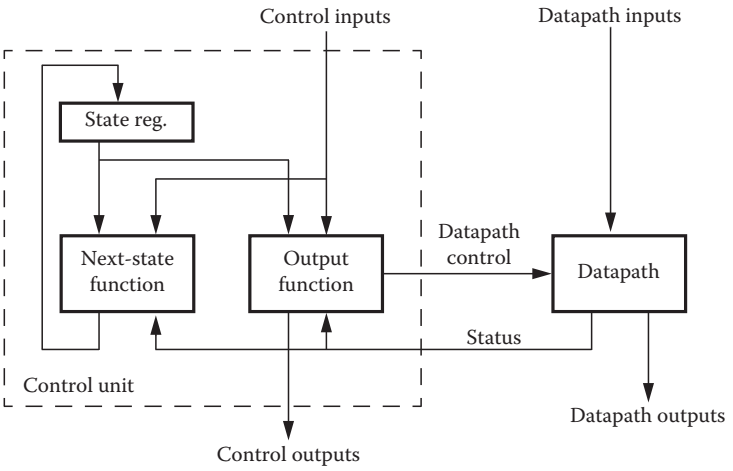


FIGURE 12.21 Generic FSMD block diagram.

values to be placed on the inputs to the FSMD interpreted model are either contained on the incoming token's information, or "color" fields, are supplied by the modeler via some outside source, or are derived using the techniques described in Section 12.4.2.1. The clock signal is either generated locally if the system-level model is globally asynchronous, or converted from the global clock into the proper format if the system-level model is globally synchronous. The functions performed by the I/U operator include releasing the tokens back into the performance model at the appropriate time and, if required, coloring them with new values according to the output signals of the interpreted element. The structure of these two parts of the mixed-level interface is shown in Figure 12.22. The U/I operator is composed of the following blocks: a Driver, an Activator, and a Clock_Generator. The I/U operator is composed of an Output_Condition_Detector, a Colorer, and a Sequential_Releaser.

In the U/I operator, the Activator is used to detect the arrival of a new token (packet of data) to the interpreted element, inform the Driver of a new token arrival, and to drive control input/s of the

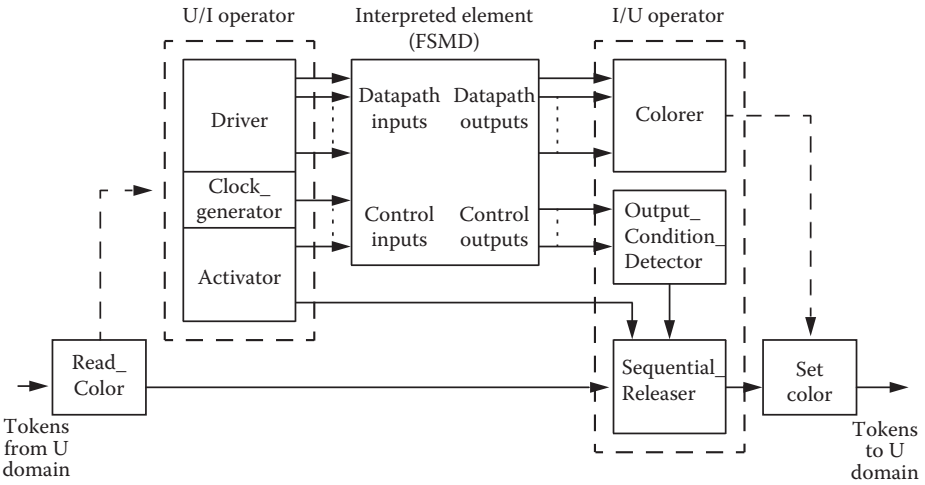


FIGURE 12.22 Mixed-level element structure.

interpreted element. The Activator's output is also connected to the I/U operator for use in gathering information on the delay through the interpreted element. The Driver reads information from the token's color fields and drives the proper input signals to the interpreted element according to predefined assignment properties. The Clock_Generator generates the clock signal according to the overall type of system-level model, either synchronous or asynchronous.

In the I/U operator, the Output_Condition_Detector detects the completion of the interpreted element data-processing operation, as discussed earlier, by comparing the element's outputs to predefined properties. The Colorer samples the datapath outputs and maps them to color fields according to predefined binding properties. The Sequential_Releaser, which "holds" the original token, releases it back to the uninterpreted model upon receiving the signal from the Output_Condition_Detector. The information carried by the token is then updated by the Colorer and the token flows back to the uninterpreted part of the model.

Given this structure, the operation of the mixed-level model can be described using the general example in Figure 12.22. Upon arrival of a new token to the mixed-level interface (U/I operator), the Read_Color module triggers the Activator component and passes the token to the Sequential_Releaser, where it is stored until the interpreted component is finished with its operation. Once triggered by the Read_Color module, the Activator notifies the Driver to start the data-conversion operation on a new packet of data. In the case of a globally asynchronous system-level model, the Activator will notify the Clock_Generator to start generating a clock signal. Since the interpreted element is a sequential machine, the Driver may need to drive sequences of values onto the inputs of the interpreted element. This sequence of values is supplied to the interpreted element, while the original token is held by the Sequential_Releaser. This token is released back to the uninterpreted model only when the Output_Condition_Detector indicates the completion of the interpreted element operation. The Output_Condition_Detector is parameterized to recognize the completion of data processing by the particular FSMD interpreted component. Once the Output_Condition_Detector recognizes the completion of data processing, it signals the Sequential_Releaser to release the token into the performance model. Once the token is released, it passes through the Colorer which maps the output data of the interpreted element onto color fields of the token. The new color information on the token may be used by the uninterpreted model for such things as delays through other parts of the model or for routing decisions.

12.4.2.1 Finding Values for the "Unknown Inputs" in an FSMD-Based Mixed-Level Model

As described previously, because of the abstract nature of a performance model, it may not be possible to derive values for all of the inputs to the interpreted component from the data present in the performance model. Typically, the more abstract the performance model (i.e., the earlier in the design cycle), the higher the percentage of input values that will be unknown. In some cases, particularly during the very early stages of the design process, it is possible that the abstract performance model will not provide any information to the interpreted element, other than the fact that new data has arrived. In this case, the data-abstraction gap will be large. This section describes an analytical technique developed to determine values for these "unknown inputs" such that a meaningful performance metric—best- or worst-case delay—can be derived from the mixed-level model.

If some (or all) inputs are not known from the performance model, some criteria for deriving the values on the unknown inputs must be made. Choosing a criterion is based on the objective of the mixed-level model. For the objective of timing verification, delays (number of clock cycles) through the interpreted element are of interest. The most common criterion in such cases is the worst-case processing delay. In some cases, best-case delay may be desired. If the number of unknown inputs is small, an exhaustive search for worst/best case may be practical. Therefore, it is desirable to minimize the number of unknown inputs which can affect the delay through the interpreted element. The methods for achieving this objective are described conceptually in Section 12.4.2.1.1. By utilizing these methods,

the number of unknown inputs is likely to be reduced but unknown inputs will not be eliminated completely. In this case, the performance metrics of best- and worst-case delay can be provided by a ''traversal'' of the state graph of the SDE component. Section 12.4.2.1.2 describes the traversal method developed for determining the delays through a sequential element.

Note that in the algorithms described below, the function of the SDE component is represented by the state transition graph (STG) or state table. These two representations are essentially equivalent and are easily generated from a behavioral VHDL description of the SDE, either by hand, or using some of the automated techniques that have been developed for formal verification.

12.4.2.1.1 Reducing the Number of Unknown Inputs

Although it is not essential, reducing the number of unknown inputs can simplify the simulation of a mixed-level model significantly. Since the FSMD elements being considered have output signals that can be monitored to determine the completion of data processing as discussed previously, other outputs may not be significant for performance analysis. Therefore, the ''nonsignificant'' (insignificant) outputs can be considered as ''don't cares.'' By determining which inputs do not affect the values on the significant outputs, it is possible to minimize the number of unknown delay affecting inputs (DAIs).

The major steps in the DAI detection algorithm are as follows:

Step 1. Select the ''insignificant'' outputs (in terms of temporal performance).
Step 2. In the state transition graph (STG) of the machine, replace all values for these outputs with a ''don't-care'' to generate the modified state machine.
Step 3. Minimize the modified state machine and generate the corresponding state table.
Step 4. Find the inputs which do not alter the flow in the modified state machine by detecting identical columns in the state table, and combining them by implicit input enumeration.

This method is best illustrated by an example. Consider the state machine which is represented by the STG shown in Figure 12.23. This simple example is a state machine with two inputs, X_1 and X_2, and two outputs, Y_1 and Y_2. This machine cannot be reduced, i.e., it is a minimal state machine. Assume that this machine is the control unit of an FSMD block and that the control output Y_1 is the output which indicates the completion of the ''data processing'' when its value is 1. Thus, output Y_2 is an ''insignificant output'' in terms of delay in accordance with Step 1. Therefore, a ''don't-care'' value is assigned to Y_2 as per Step 2. The modified STG is shown in Figure 12.24. As per Step 3, the modified STG is then reduced. In this example, states A, C, and E are equivalent (can be replaced by a single state, K) and the minimal

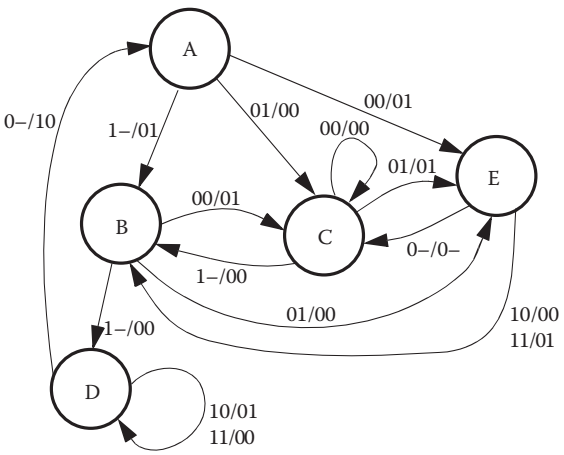


FIGURE 12.23 STG of a two-inputs two-outputs state machine.

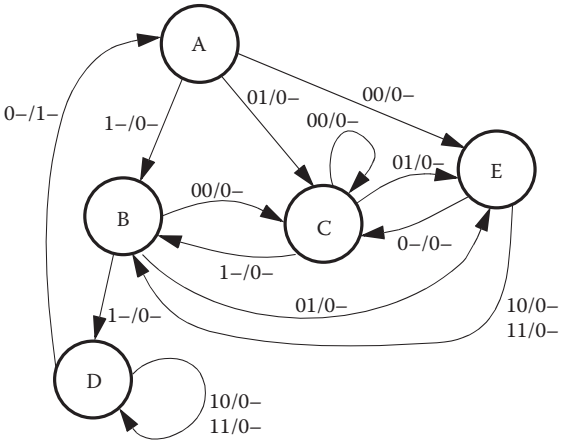


FIGURE 12.24 STG with "significant" output values only.

machine consists of three states, K, B, and D. This minimal machine is described by the state table shown in Table 12.2.

All possible input combinations appear explicitly in Table 12.2. However, it can be seen that the first two columns of the table are identical (i.e., the same next state and output value for all possible present states). Similarly, the last two columns of the table are identical. Therefore, in accordance with Step 4, these columns can be combined yielding the reduced state table shown in Table 12.3.

The reduced state table reveals that the minimal machine does not depend on the value of input X_2. Therefore, the conclusion is that input X_2 is not a DAI. This result implies that by knowing only the value of input X_1, the number of clock cycles (transitions in the original STG) required to reach the condition that output $Y_1 = 1$ can be determined regardless of the values of X_2. This is the case for any given initial state. It is important to emphasize that the paths in the *original* STG and their lengths are those which must be considered during the traversal process and that the modified state machine is used only for the purpose of detecting non-DAIs. The machine which is actually being traversed during the mixed-level simulation is the original state machine with all its functionality.

TABLE 12.2 Next State and Output Y_1 of the Minimal Machine

PS\X_1X_2	00	01	10	11
K = (ACE)	K, 0	K, 0	B, 0	B, 0
B	K, 0	K, 0	D, 0	D, 0
D	K, 1	K, 1	D, 0	D, 0

TABLE 12.3 Minimal Machine with Implicit Input Enumeration

PS\X_1X_2	0–	1–
K = (ACE)	K, 0	B, 0
B	K, 0	D, 0
D	K, 1	D, 0

To demonstrate the meaning of an input which is not a DAI, consider the original state machine represented by the graph in Figure 12.23 and assume that the initial state is A. Consider, for example, one possible sequence of values on input X_1 to be 0, 1, 0, 0, 1, 1, 0. By applying this input sequence, the sequence of values on output Y_1 is 0, 0, 0, 0, 0, 0, 1, regardless of the values applied to input X_2. Therefore, two input sequences which differ only in the values of the non-DAI input X_2 will produce the same sequence of values on the "significant" output Y_1. For example, the sequence $X_1X_2 = 00, 10, 00, 01, 10, 10, 01$ will drive the machine from state A to E, B, C, E, B, D, and back to A, and the sequence of values on output Y_1 will be as above. Another input sequence, $X_1X_2 = 01, 11, 01, 00, 11, 11, 00$, in which the values of X_1 are identical to the previous sequence, will drive the machine from state A to C, B, E, C, B, D, and back to A, while the sequence of values on output Y_1 is identical to the previous case. Therefore, the two input sequences will drive the machine via different states but will produce an identical sequence of values on the "significant" output (which also implies that the two paths in the STG have an equal length).

12.4.2.1.2 Traversing the STG for Best and Worst Delays

After extracting all possibilities for minimizing the number of unknown inputs, a method for determining values for those unknown inputs which are DAIs is required. The method developed for mixed-level modeling is based on the traversal of the original STG of the sequential interpreted element. As explained earlier, some combination of output values may signify the completion of processing the data. The search algorithm will look for a minimum or maximum number of state transitions (clock cycles) between the starting state of the machine and the state (Moore machine) or transition (state and input combination—Mealy machine), which generates the output values which signify the completion of data processing. Once this path is found, the input values for the unknown DAIs that will cause the SDE to follow this state transition path will be read from the STG and applied to the interpreted component in the simulation to generate the required best- or worst-case delay. Since the state machine is represented by the STG, this search is equivalent to finding the longest or shortest path between two nodes in a directed graph (digraph).

The search for the shortest path utilizes a well-known algorithm. Search algorithms exist for both single-source shortest path and all-pairs shortest path. One of the first and most commonly used algorithms is Dijkstra's algorithm [46], which finds the shortest path from a specified node to any other node in the graph. The search for all-pairs shortest path is also a well-investigated problem. One such algorithm by Floyd [47] is based on work by Warshall [48]. Its computation complexity is $O(n^3)$ when n is the number of nodes in the graph, which makes it quite practical for moderate-sized graphs. The implementation of this algorithm is based on Boolean matrix multiplication and the actual realization of all-pairs shortest paths can be stored in an $n \times n$ matrix. Utilizing this algorithm required some enhancements to make it applicable to mixed-level modeling. For example, if some of the inputs to the interpreted element are known (from the performance model), then the path should include transitions that include these known input values.

In contrast, the search for the longest path is a more complex task. It is an NP-complete problem that has not attracted significant attention. Since most digraphs contain cycles, the cycles need to be handled during the search to prevent a path from containing an infinite number of cycles. One possible restriction that makes sense for many state machines that might be used in mixed-level modeling is to construct a path that will not include a node more than once. Given a digraph $G(V, E)$ that consists of a set of vertices (or nodes) $V = \{v_1, v_2, \dots\}$ and a set of edges (or arcs) $E = \{e_1, e_2, \dots\}$, a *simple path* between two vertices v_{init} and v_{fin} is a sequence of alternating vertices and edges $P = v_{\text{init}}, e_n, v_m, e_{n+1}, v_{m+1}, e_{n+2}, \dots, v_{\text{fin}}$ in which each vertex does not appear more than once. Although an arbitrary choice was made to implement the search allowing each vertex to appear in the path only once, the same algorithm could be easily modified to allow the appearance of each vertex a maximum of N times.

Given an initial node and a final node, the search algorithm developed for this application starts from the initial node and adds nodes to the path in a depth-first-search (DFS) fashion, until the final node is

reached. At this point, the algorithm backtracks and continues looking for a longer path. However, since the digraph may be cyclic, the algorithm must avoid the possibility of increasing the path due to a repeated cycle, which may produce an infinite path.

The underlying approach for avoiding repeated cycles in the algorithm *dynamically* eliminates the cycles while searching for the longest simple path. Let u be the node that the algorithm just added to the path. All the in-arcs to node u can be eliminated from the digraph at this stage of the path construction. The justification for this dynamic modification of the graph is that, while continuing in this path, the simple path cannot include u again. While searching forward, more nodes are being added to the path and more arcs can be removed temporarily from the graph. At this stage, two things may happen (1) either the last node being added to the path is the final node or (2) the last node has no out-arcs in the dynamically modified graph. These two cases are treated in the same way except that in the first case the new path is checked to see if it is longer than the longest one found so far. If it is, the longest path is updated. However, in both cases the algorithm needs to backtrack.

Backtracking is performed by removing the last node from the path, hence decreasing the path length by one. During the process of backtracking, the in-arcs to a node being removed from the path must be returned to the current set of arcs. This process will enable the algorithm to add this node when constructing a new path. At the same time, whenever a node is removed from the path, the arc that was used to reach that node is marked in the dynamic graph. This process will eliminate the possibility that the algorithm repeats a path that was already traversed. Therefore, by dynamically eliminating and returning arcs from/to the graph, a cyclic digraph can be treated as if it does not contain cycles. The process of reconnecting nodes, i.e., arcs being returned to the dynamic graph, requires that the original graph be maintained. A more detailed description of this search algorithm can be found in Ref. [49].

The restriction that a longest path does not include any *node* in the graph more than once may not be appropriate for all cases. In some mixed-level modeling cases, a more realistic restriction might be that the longest path does not include any *transition* (arc) more than once. A longest-path with no repeated arcs may include a node multiple times as long as it is reached via different arcs. In the case of more than one transition that meets the condition on the output combination, a search for the longest path should check all paths between the initial state and all of these transitions. However, such a path should include any of these transitions only once, and it should be the last one in the path.

Performing a search with the restriction that no arc is contained in the path more than once requires maintaining information on arcs being added or removed from the path. Maintaining this information during the search makes the algorithm and its implementation much more complicated relative to the search for the longest path with no repeated nodes. Therefore, a novel approach to this problem was developed. The approach used is to map the problem to the problem of searching for the longest path with no repeated nodes. This mapping can be accomplished by transforming the digraph to a new digraph, to be referred to as the transformed digraph (or Tdigraph). Given a digraph, $G(V, E)$, which consists of a set of nodes $V = \{v_1, v_2, \ldots, v_k\}$ and a set of arcs $E = \{e_1, e_2, \ldots, e_l\}$ the transformation τ maps $G(V, E)$ into a Tdigraph $TG(N, A)$, where N is its set of nodes and A its set of arcs. The transformation is defined as $\tau(G(V, E)) = TG(N, A)$, and contains the following steps:

Step 1. $\forall(e_i \in E)$ generate a node $n_i \in N$.
Step 2. $\forall(v \in V)$ and $\forall\left(e_p \in d^v_{\text{in}}, e_q \in d^v_{\text{out}}\right)$ generate an arc $a \in A$ such that a: $n_p \rightarrow n_q$.

The first step is used to create a node in the Tdigraph for each arc in the original digraph. This one-to-one mapping defines the set of nodes in the Tdigraph to be $N = \{n_1, n_2, \ldots, n_l\}$, which has the same number of elements found in the set E.

The second step creates the set of arcs, $A = \{a_1, a_2, \ldots, a_u\}$, in the Tdigraph. For each node in the original digraph and for each combination of in-arcs and out-arcs to/from this node, an arc in the Tdigraph is created. For example, given a node v with one in-arc e_i and one out-arc e_j, e_i is mapped to a node n_i, e_j is mapped to a node n_j, and an arc from n_i to n_j is created. In Step 2 of the transformation process, it is guaranteed that all possible connections in the original digraph are preserved as transitions

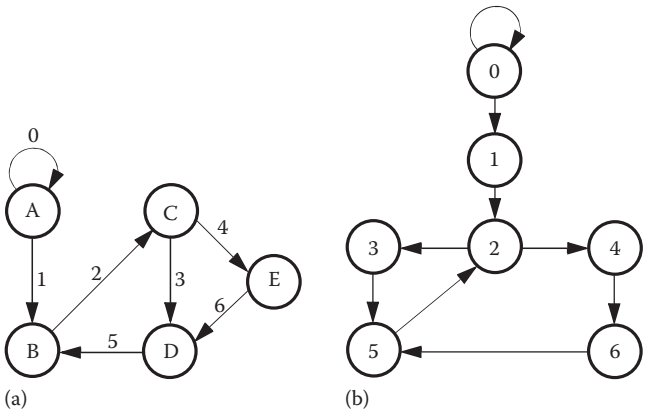


FIGURE 12.25 Transformation of a digraph.

between nodes in the Tdigraph. As a result of this transformation, the restriction on not visiting an arc more than once in the original digraph is equivalent to not visiting a node more than once in the Tdigraph. Therefore, by using this transformation, the problem of searching for the longest path with no repeated arcs in the original digraph is mapped to a search for the longest path with no repeated nodes in the Tdigraph. The algorithm described above can then be used to search the Tdigraph.

This transformation is best illustrated by a simple example. Consider the digraph shown in Figure 12.25a. The arcs in this digraph are labeled by numbers 0–6. The first step of the transformation is to create a node for each arc in the original digraph. Therefore, there will be seven nodes, labeled 0–6, in the Tdigraph as shown in Figure 12.25b. The next step is to create the arcs in the Tdigraph. As an illustration of this step, consider node C in the original digraph. Arc ''2'' is an in-arc to node C while arcs ''3'' and ''4'' are out-arcs from node C. Applying Step 2 results in an arc from node ''2'' to node ''3'' and a second arc from node ''2'' to node ''4'' in the Tdigraph. Considering node B in the original digraph, the Tdigraph will include an arc from node ''1'' to node ''2'' and an arc from node ''5'' to node ''2.'' This process is repeated for all the nodes in the original digraph until the Tdigraph, as shown in Figure 12.25b, is formed. A search algorithm can now be executed to find the longest path with no repeated nodes.

A mixed-level modeling methodology, which is composed of all the methods described above, has been integrated into the ADEPT environment. The steps for minimizing the unknown inputs can be performed prior to simulating the mixed-level model. On the other hand, the search for longest/shortest possible delay must be performed during the simulation itself. This requirement arises because each token may carry different information which may alter the known input values and, therefore, alter the search of the STG. The STG traversal process has been integrated into the ADEPT modeling environment using the following steps: (1) when a token arrives to the mixed-level interface, the simulation is halted and the search for minimum/maximum number of transitions is performed and (2) upon completion of the search, the simulation continues while applying the sequence of inputs found in the search operation. The transfer of information between the VHDL simulator and the search program, which is implemented in C is done by using the simulator's VHDL/C interface. A component called the Stream_Generator has been created that implements the STG search process via this interface.

Since mixed-level models are part of the design process and are constructed by refining a performance model, it is likely that many tokens will carry identical relevant information. This information may be used for selective application of the STG search algorithm, hence increasing the efficiency of the mixed-level model simulation. For example, if several tokens carry exactly the same information (and assuming the same initial state of the FSM), the search is performed only once, and the results can be used for the following identical tokens.

12.4.2.2 Example of Mixed-Level Modeling with an FSMD Component

This section presents an example of the construction of a mixed-level model with an FSMD interpreted component. The example is based on the performance model of an execution unit of a particular processor. This execution unit is composed of an integer unit (IU), a floating-point unit (FPU), and a load-store unit (LSU). These units operate independently although they receive instructions from the same queue (buffer of instructions). If the FPU is busy processing one instruction and the following instruction requires the FPU, it is buffered, waiting for the FPU to be free again. Meanwhile, instructions which require the IU can be consumed and processed by IU at an independent rate. Both the FPU and the IU have the capability of buffering only one instruction. Therefore, if two or more consecutive instructions are waiting for the same unit, the other units cannot receive new instructions (since the second instruction is held in the main queue). One practical performance metric that can be obtained from this model is the time required for the execution unit to process a given sequence of instructions.

Because the FPU was identified as the most complex and time critical portion of the design, a behavioral description of a potential implementation of it was developed. At this point, a mixed-level model, in which the behavioral description of the FPU is introduced into the performance model, was constructed using the interface described above. The mixed-level model is shown in Figure 12.26. The mixed-level interface is constructed around the interpreted block which is the behavioral description of the FPU. This FPU is an FSMD type of element, and the interpreted model consists of a clock cycle-accurate VHDL behavioral description of this component. The inputs to the FPU include the operation to be performed (Add, Sub, Comp, Mul, MulAdd, and Div), the precision of the operation (single or double), and some additional control information. The number of clock cycles required to complete any instruction depends on these inputs.

Figure 12.27 shows the execution unit performance derived from the mixed-level model for three different instruction traces. In this case, only 40% of the inputs have values that are supplied by the abstract performance model, the remainder of the values for the inputs are derived using the techniques described in Section 12.4.2.1. The performance value is normalized by defining unity to be the amount of time required to process a trace according to the initial uninterpreted performance model. In this example, the benefit of the simulation results of the mixed-level model is clear. It provides performance bounds in terms of best- and worst-case delays, for the given implementation of the FPU.

12.4.3 Interface for Mixed-Level Modeling with Complex Sequential Components

A methodology and components for constructing a mixed-level interface involving general sequential interpreted components that can be described as FSMDs was detailed in the previous section. However, many useful mixed-level models can be constructed that include sequential interpreted components that are too complex to be represented as FSMDs, such as microprocessors, complex coprocessors, network interfaces, etc. In these cases, a more ''programmable'' mixed-level interface that is able to deal with the additional complexity in the timing abstraction problem was needed. This section describes the ''watch-and-react'' interface that was created to be a generalized, flexible interface between these complex sequential interpreted components and performance models.

The two main elements in the watch-and-react interface are the *Trigger* and the *Driver*. Figure 12.28 illustrates how the Trigger and Driver are used in a mixed-level interface. Both elements have ports that can connect to signals in the interpreted components of a model. Collectively, these ports are referred to as the *probe*. The primary job of the Trigger is to detect events on the signals attached to its probe, while the primary job of the Driver is to force values onto the signals attached to its probe. The Driver decodes information carried in tokens to determine what values to force onto signals in the interpreted model (the U/I interface) and the Trigger encodes information about events in the interpreted model onto tokens in the performance model (the I/U interface).

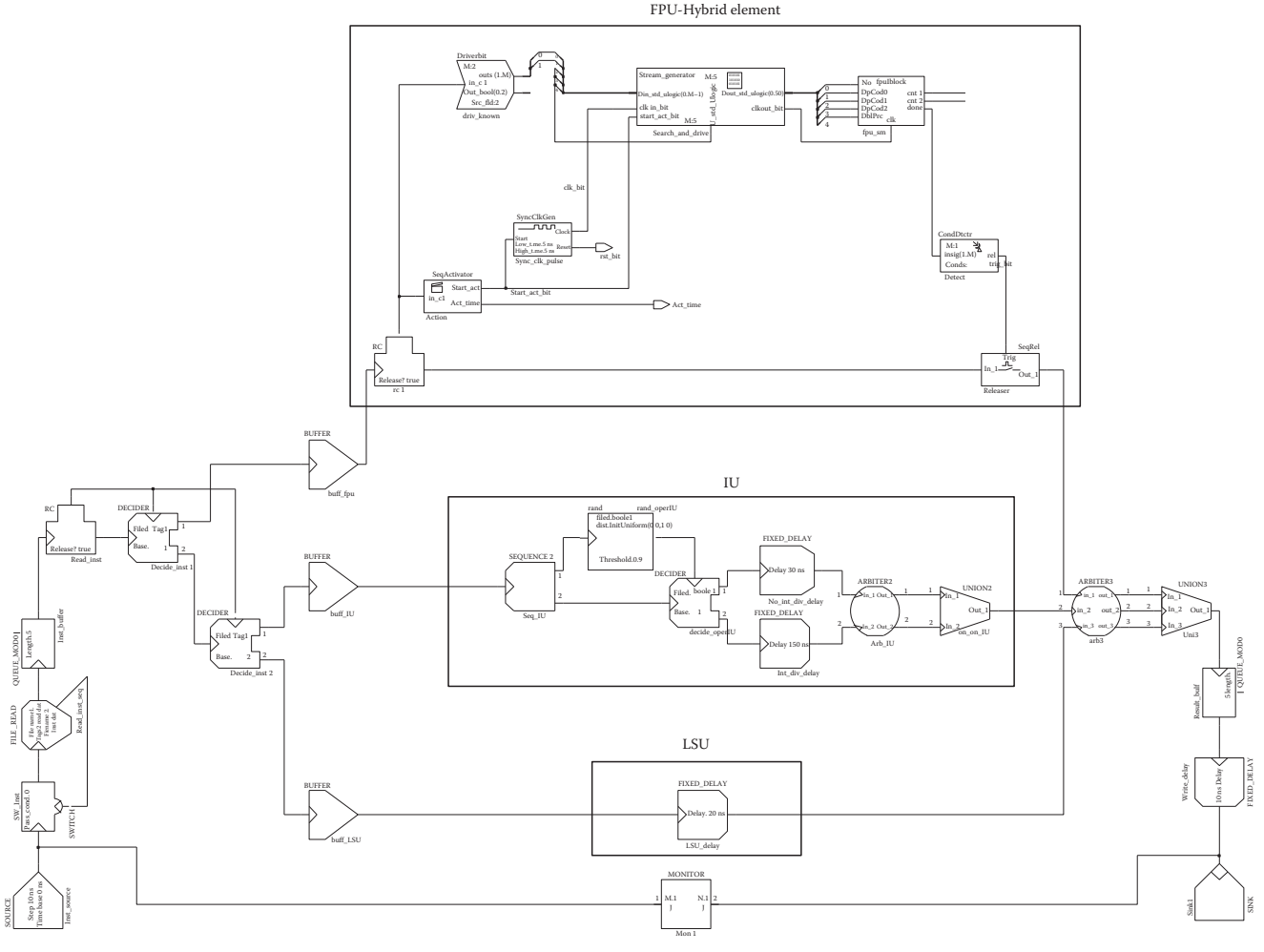


FIGURE 12.26 Mixed-level model with the FPU behavioral description.

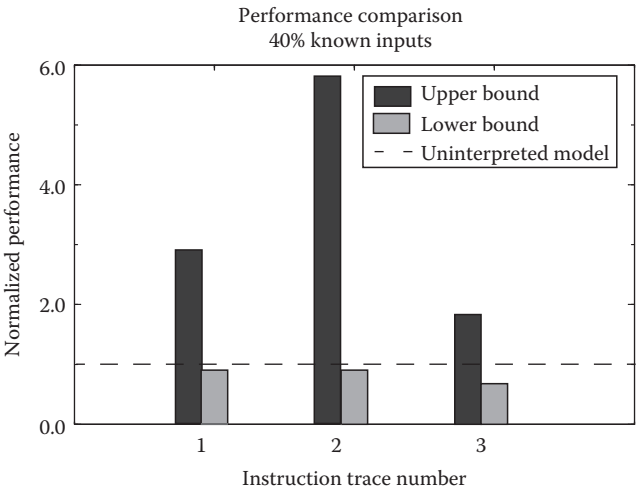


FIGURE 12.27 Performance comparison of the execution unit for three instruction traces.

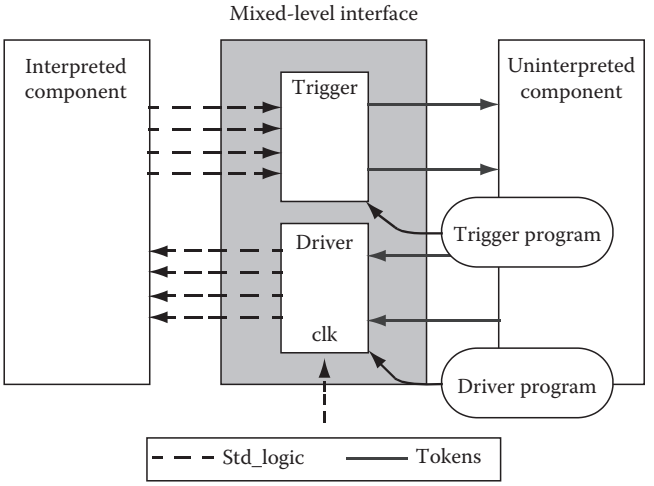


FIGURE 12.28 Watch-and-react mixed-level interface.

The Trigger and Driver were designed to be as generic as possible. A command language was designed that specifies how the Trigger and Driver should behave to allow users to easily customize the behavior of the Trigger and Driver elements without having to modify their VHDL implementation. This command language is interpreted by the Trigger and Driver during simulation. Because the language is interpreted, changes can be made to the Trigger and Driver programs without having to recompile the model, thus minimizing the time required to make changes to the mixed-level model.

The schematic symbol for the Trigger is shown in Figure 12.29. The primary job of the Trigger is to detect events in the interpreted model. The probe on the Trigger is a bus of std_logic signals `probe_size` bits wide, where `probe_size` is a generic on the symbol. There is one token output called `out_event_token` and one token output called `out_color_token`. Tokens generated by the Trigger when events are detected are placed on the `out_event_token` port. The condition number (an integer) of the event that caused the token to be generated is placed on the `condition`

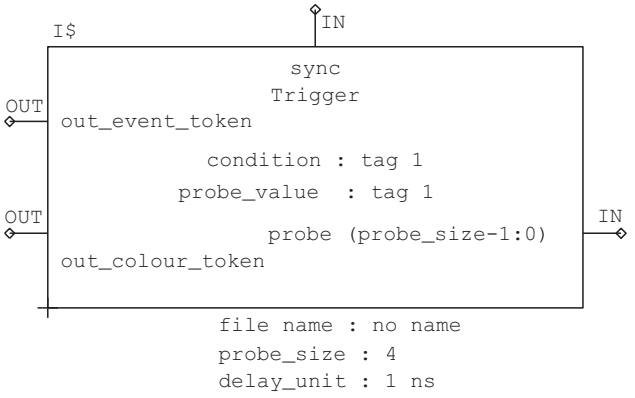


FIGURE 12.29 Schematic symbol for the Trigger.

tag field, which is specified as a generic on the symbol. Also, each time a signal changes on the probe, the color of the token on the `out_color_token` port changes appropriately regardless of whether an event was detected or not. The probe value is placed on the `probe_value` tag field of the `out_color_token` port. The sync port is used to synchronize the actions of the Trigger element with the Driver element as explained below.

The name of the file containing the Trigger's program is specified by the `filename` generic on the symbol. The `delay_unit` generic on the symbol is a multiple that is used to resolve the actual length of an arbitrary number of delay units specified by some of the interface language statements.

The schematic symbol for the Driver element is shown in Figure 12.30. The primary job of the Driver is to create events in the interpreted model by driving values on its probe. These values come from either the command program, or from the values of tag fields on the input tokens to the Driver. The probe on the Driver is a bus of std_logic signals `probe_size` bits wide, where the `probe_size` is a generic on the symbol. There is one token input called `in_event_token`, one token input called `in_color_token`, and a special input for a std_logic type clock signal called `clk`. The `clk` input allows Driver to synchronize its actions with an external interpreted clock source. The `sync` port is used to synchronize the actions of the Driver element with the Trigger element. The `filename` and `delay_unit` generic on the symbol function the same as for the Trigger.

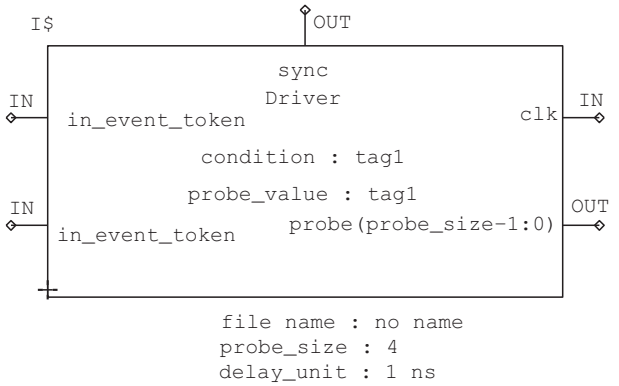


FIGURE 12.30 Schematic symbol for the Driver.

As discussed previously, a command language was developed to allow the user to program the actions of the Trigger and Driver. This command language is read by the Trigger and Driver at the beginning of the simulation. Constructs are available within the command language to allow waiting on events of various signals and driving values or series of values on various signals, either asynchronously, or synchronous with a specified clock signal. In addition, several looping and go-to constructs are available to implement complex behaviors more easily. A summary of the syntax of the command language constructs is shown in Table 12.4.

12.4.3.1 Example of Mixed-Level Modeling with a Complex Sequential Element

This section presents an example which demonstrates how the Trigger and Driver elements can be used to interface an interpreted model of a complex sequential component with an uninterpreted model. In this example, the interpreted model is a microprocessor-based controller and the uninterpreted model is that of a motor control system including a motor controller and a motor. The motor controller periodically asserts the microcontroller's interrupt line. The microcontroller reacts by reading the motor's current speed from a sensor register on the motor controller, calculating the new control information, and writing the control information to the motor controller.

The microcontroller system consists of interpreted models of eight-bit, RISC-like microprocessor, RAM, memory controller, I/O controller, and clock. The memory controller handles read and write requests issued by the processor to the RAM, while the I/O controller handles read and write request issued by the processor to an I/O device. In the system model, the I/O device is the uninterpreted model of a motor controller. A schematic of the model is shown in Figure 12.31.

Three Triggers and two Drivers are used to construct the mixed-level interface for the control system model. One of the Triggers is used to detect when the I/O controller is doing a read or write. The other two Triggers are used to collect auxiliary information about the operation, such as the address on the address bus and data on the data bus. One of the Drivers is used to create a microcontroller interrupt and the other Driver is used to force data onto the data bus when the processor reads from the speed sensor register on the motor controller.

The interrupt Driver's program is listed in Figure 12.32. The program begins by forcing the interrupt line to "Z" and then waiting for a token from the uninterpreted model of the motor controller to arrive. Once a token arrives, the program forces the interrupt line high for 10 clock cycles. This condition is accomplished by using a `for-next` statement with a `wait_on_rclk` as the loop body. After 10 clock cycles, the program jumps to line 10 where the cycle begins again.

The data Driver's program is listed in Figure 12.33. The program begins by also waiting for a token from the uninterpreted model of the motor controller to arrive. If the value on the condition tag field of the token is 1, then "ZZZZZZZZ" is forced onto the data bus. If the condition tag field value is 3, then the value on the `probe_value` tag field of the `in_color_token` input is forced on the data bus. This process is repeated for every token that arrives.

The I/O Trigger's program is listed in Figure 12.34. This Trigger waits until there is a change on the probe. Once there is a change, the program checks to see if the I/O device is being unselected, written to, or read from. If one of the `when` statements matches the probe value, then its corresponding `output` statement is executed. An output of 1 corresponds to the I/O device not being selected. An output of 2 corresponds to the processor writing control information to the motor controller. An output of 3 corresponds to the processor reading the motor's speed from the sensor register on the motor controller.

Figure 12.35 shows the results from the mixed-level model as a plot of the sensor output and the processor's control response. Some random error was introduced to the sensor's output to reflect variations in the motor's load as well as sensor noise. The target speed for the system was 63 ticks per sample time (a measure of the motor's RPM). The system oscillates slightly around the target value because of the randomness introduced into the system.

TABLE 12.4 Trigger and Driver Command Programming Language Constructs

Command	Element	Meaning
– <comment>	Both	Comment—no action
alert_user	Both	Print message in simulation window
delay_for <T>	Both	Delay for <T > time units where the time unit is specified as a generic on the module's symbol
end	Both	End the command program
for <N> <loop body> **next**	Both	Iterate *N* times over the sequence of statement in the loop body
goto <L>	Both	Go to line <L> in the command program
output_sync	Both	Generate a token on the sync output of the module
wait_on_sync	Both	Wait for an occurrence of a token on the sync port of the module
case_probe_is **when** <STD_LOGIC_VAL> <sequence of statements> **when** . . . **end_case**	Trigger	Conditionally execute some sequence of statements depending on the STD_LOGIC value of the probe signal
output <INTEGER_VAL> **after** <T>	Trigger	Generate a token on the Trigger's output with the value of <INTEGER_VAL> on the tag field specified by the generic on the symbol after <T > time units
trigger	Trigger	Must appear as the first statement in the Trigger's command program
wait_on <STD_LOGIC_VAL>	Trigger	Wait until the probe signal takes on the specified STD_LOGIC value
wait_on_probe	Trigger	Wait until there is ANY event on the probe signal
case_token_is **when** <**INTEGER_VAL**> <**sequence of statements**> **when . . .** **end_case**	Driver	Conditionally execute some sequence of statements depending on the integer value of the input token's tag field specified by the generic on the symbol
driver	Driver	Must appear as the first statement in the Driver's command program
dynamic_output_after <T>	Driver	Force the value from the specified tag field of the input token onto the probe after <T> time units
output <STD_LOGIC_VAL> **after** <T>	Driver	Force the specified STD_LOGIC value onto the probe signal after <T> time units
wait_on <INTEGER_VAL>	Driver	Wait until a token with the given integer value on the tag field specified by the generic on the symbol arrives on the in_event_token signal
wait_on_fclk	Driver	Wait until the falling edge of the clock occurs
wait_on_rclk	Driver	Wait until the rising edge of the clock occurs
wait_on_token	Driver	Wait until a token arrives on the in_event_token signal

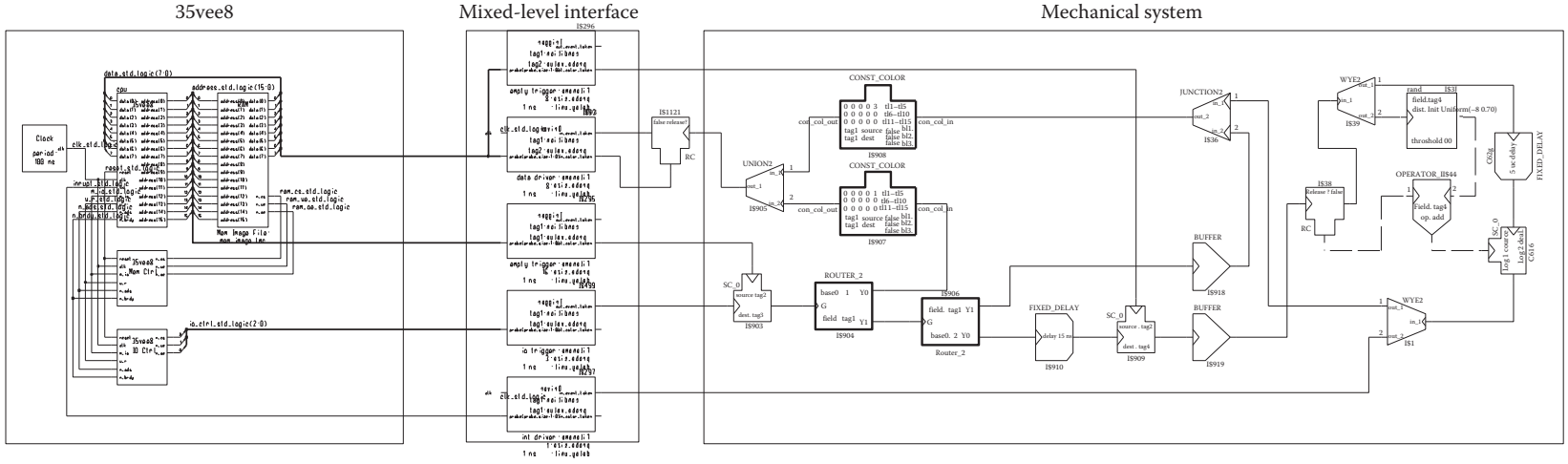


FIGURE 12.31 Schematic of control system model.

```
driver
10 output Z after 0
20 wait_on_token
30 output 1 after 0
40 for 10
50    wait_on_rclk
60 next
70 goto 10
80 end
```

FIGURE 12.32 Interrupt Driver's program for the control system.

```
driver
10 wait_on_token
20 case_token_is
30     when 1
40        -- sensor not selected
50        output ZZZZZZZZ after 0
60     when 3
70        -- sensor selected for reading
80        dynamic_output_after 0
90 end_case
100 goto 10
110 end
```

FIGURE 12.33 Data Driver's program for the control system.

```
trigger
10 wait_on_probe
20 case_probe_is
30     when 111
40        -- sensor not selected
50        output 1 after 0
60     when 001
70        -- sensor selected for writing
80        output 2 after 0
90     when 010
100       -- sensor selected for reading
110       output 3 after 0
120 end_case
130 goto 10
140 end
```

FIGURE 12.34 I/O Trigger's program for the control system.

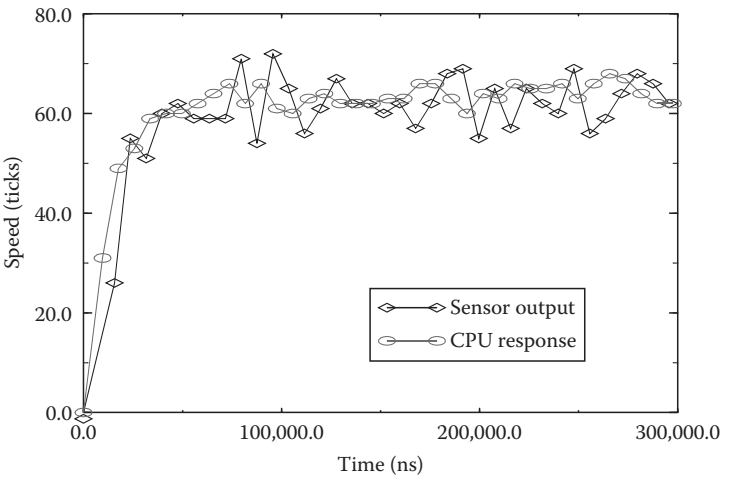


FIGURE 12.35 Sensor and processor outputs for the control system.

12.5 Performance and Mixed-Level Modeling Using SystemC

This section describes a performance-modeling environment capable of mixed-level modeling that is based on the SystemC language [50]. The environment is intended to model the system at the Processor Memory Switch level much like the Honeywell PML environment described earlier. The goal of this work was to show how SystemC could be used to construct a mixed-level modeling capability.

In the SystemC-based PBMT (Performance-Based Modeling Tool), the user begins by describing the functions executed by the system as a task graph. A task graph is a representation of the flow of execution through an application. The nodes in a task graph represent computational tasks, and the edges in a task graph represent the flow of control, or the actual transfer of data, between tasks. An example of a task graph for a simple application is shown in Figure 12.36. Note that the topology shown in the figure, the example application has the opportunity for some tasks, such as Task 2, Task 3, and Task 4, to be executed in parallel if the system architecture upon which the application is to be executed, allows for it.

Once the task graph model is constructed, the user then selects a system architecture on which the application will execute. The system architecture is specified by the number of processors in the architecture, and an interconnect topology used to provide communications between them. The available interconnect topologies include a bus (a single shared communications resource), a crossbar switch (a partially shared communications resource), or fully connected (a completely nonshared communication resource). Note that in this high-level architecture model, what actually constitutes a processor in the system is not specified. That is, a processor is simply modeled as a computational resource and may in implementation be a general-purpose processor (of any clock speed), a special purpose processor like a DSP, or custom hardware for a specific task.

Once the system architecture is specified, all that remains is for the user to specify upon which processor each of the tasks is to execute and what the total execution time for that task on the specific processor will be. This delay value may be either fixed, or dependent on the amount of data that is passed into the task by the previous task in the graph. Once this task-to-processor mapping and delay specification is done, the complete SystemC model is constructed and simulated using either the reference simulator, included as part of the SystemC distribution available in Ref. [10], or the commercial Mentor Graphics ModelSim simulator which includes the capability to cosimulate SystemC models along with Verilog or VHDL models.

Figure 12.37 shows the results of executing the task graph of Figure 12.36 on three different system architectures. All of the architectures utilize a single shared bus for communications. The first result is for an architecture with only a single processor. In this case, the obvious result is that all of the tasks execute in sequence on the single processor and the run time is simply the sum of the individual task execution times. The second result is for a three-processor architecture. In this case, after Task 1

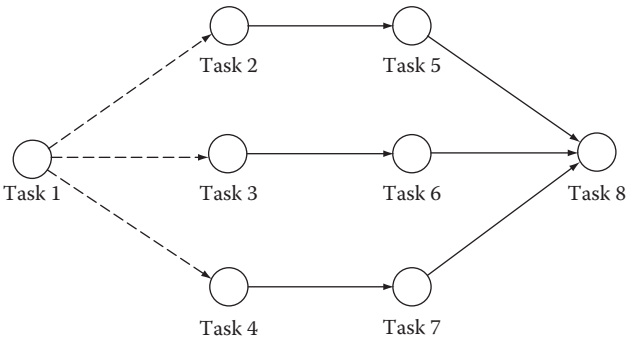


FIGURE 12.36 Example task graph.

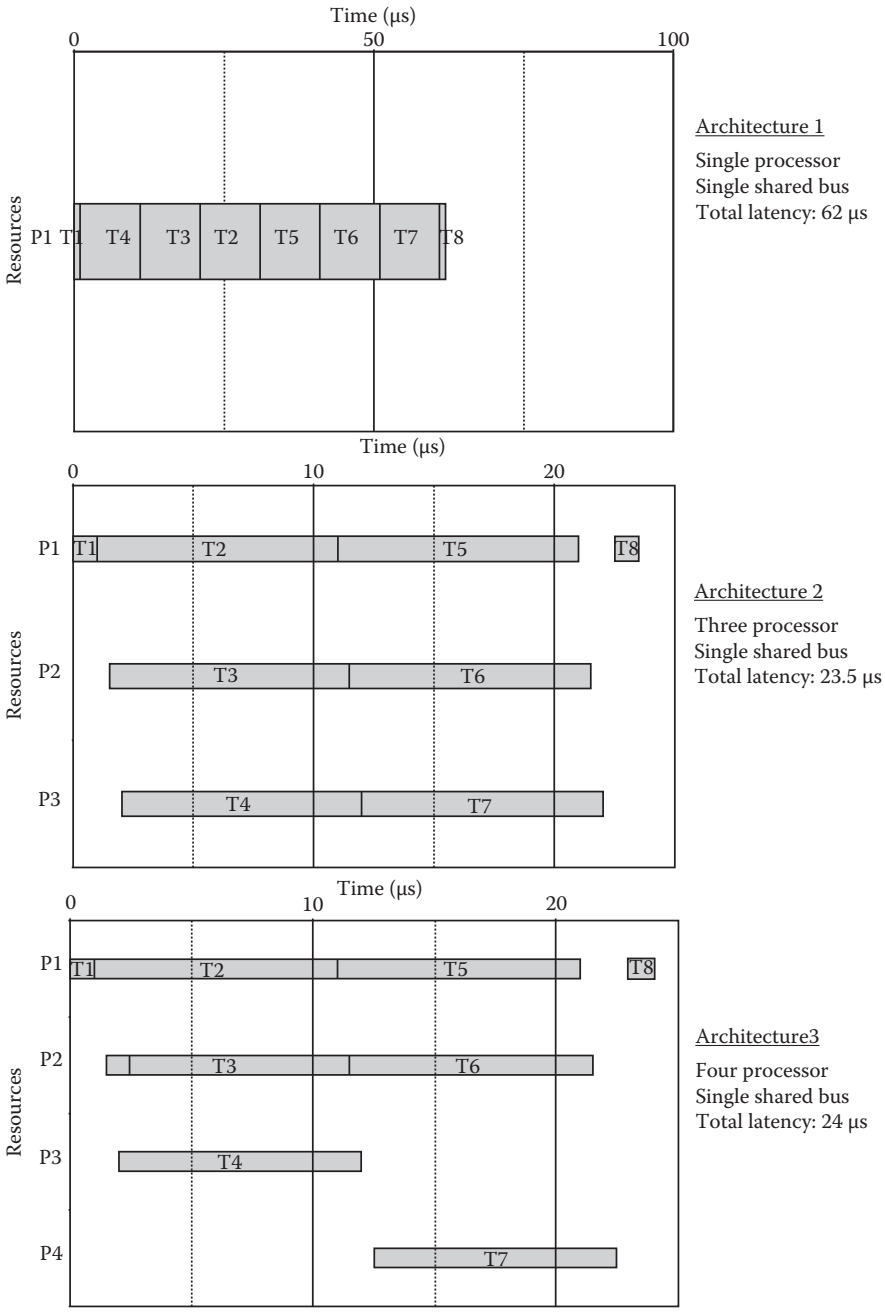


FIGURE 12.37 Parallel example shared bus timelines.

completes, some latency can be seen before Tasks 3 and 4 begin execution. This accounts for the communication time required to send data from the processor that executed Task 1 to the processors that are executing Tasks 3 and 4. In addition, the graph shows that Task 4 begins execution after Task 3 because of the contention for the single shared bus communications resource. Likewise, the latency between the end of execution for Tasks 5–7, and the start of execution for Task 8 accounts for the time required to communicate Tasks 6 and 7's results back to the single processor that is schedule to execute

Task 8. The overall run time for this configuration is much less than the first example because the inherent parallelism in the application is being exploited by the selected architecture. Finally, the third result is for a four-processor architecture. In this simulation, Tasks 1 and 8 are allocated to the fourth processor, separate from the other Tasks 2–7. This results in additional communications time being required using the single bus to transfer all of the data from, and back to, that extra processor. Thus, as can be seen from the graph this architecture actually takes longer than the three processor architecture to execute the application.

12.5.1 SystemC Background

This section describes some of the basic concepts of SystemC and how it is used to model systems at a high level. SystemC is a library extension to C++. Essentially, SystemC uses a set of classes and predefined functions to build a new "language" on top of C++. The basic unit of a SystemC model is the `SC_MODULE` class. The `SC_MODULE` is roughly equivalent to an entity in VHDL. It can have input, output, and bidirectional ports. However, it is a C++ class and as such it can, and does, have member functions. Any member function that is declared as public can be accessed just like the member function of any other class. Every `SC_MODULE` must have at least one public member function, its constructor. In the constructor the module is given a SystemC name, and declares if it has any processes, and does anything else necessary to get the model ready for simulation. There are two types of processes in SystemC, the `SC_THREAD` and the `SC_METHOD`. The behavior of a process must first be declared as a member function, then that member function can be declared to be either an `SC_THREAD` or an `SC_METHOD` in the constructor. The primary differences are how they behave when they reach the end of their definition, and when the scheduler activates them.

An `SC_THREAD` will terminate, and never be activated again, if it reaches the end of its description. Typically `SC_THREAD`s are infinite loops with one or more wait statements to break the execution, and wait on some signal change or other event. An `SC_METHOD` will be activated any time something it is sensitive to changes, and will run once through to the end of its description. If something an `SC_METHOD` is sensitive to changes again, it will run again. `SC_THREAD`s do have the ability to use a form of the wait command that is not available to `SC_METHOD`s. They may use a wait command with no parameters which will cause them to be reactivated and continue execution with the line after the wait statement when something in their sensitivity list has an event.

As indicated by the discussion of processes, SystemC uses an event-driven simulation methodology. The library provides basic signal classes that have a notify-update sequence much like VHDL. As mentioned in the Transaction Level Modeling discussion above, SystemC has a concept of channels. A channel is generally some means of moving information. The basic signal classes provided are base channels. A designer could potentially design some new base channel type that has the same interfaces as the existing base channels. However, building a new base channel is rather involved since its implementation must interact directly with the scheduler to implement the notify/update semantics of a signal. Additionally, such a user-designed base channel would still only have a basic interface, and could not have any internal processes, thus would not provide significant abstraction leveraging for the amount of time required to design it. Instead designers should use what is called a hierarchical channel for most modeling needs. A hierarchical channel is a channel that is made up of a number of elements of base classes. A hierarchical channel can have any number of ports, or methods. The methods that a module could use to access the channel, and convey information through it, are called interface methods. For a module to be able to access an interface method the channel needs to have a defined interface that it inherits. The `sc_interface` class is used as a base class to define such an interface. The file in Figure 12.40 shows one such interface class. Once an interface class has been defined and inherited by the channel, then any module with a port of that type can be bound to the channel's interface. In addition to the base channel classes, the SystemC library provides an event object that can be waited upon using the same syntax as a wait for a signal change event. For more details on the syntax, classes, and functions of SystemC the reader may refer to Ref. [10].

At the time of this writing, there are several options for simulating SystemC models. The two most robust SystemC simulators are the reference simulator, available with the SystemC distribution, and the ModelSim simulator as mentioned above. Because of some internal implementation details which differ between the two simulators, there are minor code differences required in the performance models between the two simulators. The differences and the techniques for enabling the models to be compiled and run in either simulators are described in the sections below.

12.5.2 SystemC Performance Models

The performance models described herein take advantage of the C++ foundations of SystemC to provide a highly parameterizable simulation environment that loads most of the parameters at run time. Figure 12.38 shows the class definition for the top level of the simulation model, and the relevant comments. The top-level entity makes use of a pointer to an object of type `SIM`, declared in the header file *generic.h*, to allow the constructor arguments to be read in from the *top_config.txt* file rather than statically specified in the source code. When the simulation is loaded the SystemC constructor (`SC_CTOR` in the code) will be called. The constructor will create a `streambuf` object, a number of processors variable, a connection type variable, as well as a pointer variable to point to an object of type `SIM`. It will then open the *top_config.txt* file and associate it with the `streambuf` variable. Then it will read an unsigned value into the number of processors variable, then read an unsigned value into the connection type variable. Once those values are read in then the constructor uses the ''new'' command to instantiate a new object of type `SIM`. The ''new'' command allows passing variable constructor arguments to the objects constructor, so the new `SIM` object's constructor builds the object with the value of `num_processors` processors, and the value of `connect_type` interconnect number. If the number of processors or interconnect type was specified via a template argument, or a fixed constructor

```
#ifndef MY_MAIN
#define MY_MAIN

#include <systemc.h>
#include "generic.h"

SC_MODULE(main_sim)
{
   SIM *simulation;

   SC_CTOR(main_sim)
   {
      ifstream topFile;
      unsigned num_processors;
      //1 to 9, could be to 10^13 or memory constraints except for modelsim
namebinding
      unsigned connect_type;   //1,2,3 are valid
      topFile.open("top_config.txt");//this doesn't do error check!
      topFile>>num_processors;
      //get info from file & set num_processors,connect_type
      topFile>>connect_type;// read in the interconnect type to use
      simulation = new SIM("simulation",num_processors,connect_type);
      // systemC name,number of processor ,connection class
   };

};
#endif
```

FIGURE 12.38 Top-level class definition.

```
#include <systemc.h>
#include "generic_main.h"
#ifdef MTI_SYSTEMC
SC_MODULE_EXPORT(main_sim);
#else
//for OSCI reference simulator
int sc_main(int ac, char *av[] )
{
    main_sim my_sim("my_sim");
    sc_start(500, SC_NS);
    sc_close_vcd_trace_file(main_trace);
    return 0;
};
#endif
```

FIGURE 12.39 Top-level C++ file.

value (e.g., "`simulation = new SIM("it,9,1);`") in the source code then the simulation would have to be recompiled every time something was changed.

Notice that all the implementation details of the SystemC modules are declared in header files. This is the recommended way of describing models to compile the SystemC code for the ModelSim simulator. The included guards are also a must for any `SC_MODULE` definitions to ensure they are not included multiple times by the ModelSim SystemC compiler, SCCOM. The actual C++ file that is compiled to generate the simulation models is shown in Figure 12.39. The `SC_MODULE_EXPORT(module_class_name)` is the function used to create the ModelSim model for the specified entity. The `MTI_SYSTEMC` compiler definition is defined specifically for compilation using SCCOM, and allows having a single set of code for both the ModelSim and reference simulator. Any ModelSim specific code can be placed inside a `#ifdef MTI_SYSTEMC` compiler directive statement so that the compiler only uses it when compiling for ModelSim. The `#else` statement prevents the ModelSim compiler from trying to compile the reference simulator-specific portions, and the `#endif` closes out the if-else statement.

The *generic.h* header file defines the class `SIM`. It makes use of the shared base class `my_basic_rw_port` to declare a pointer that will allow assignment of an instantiation of any of the three channel types developed for the environment. The `my_basic_rw_port` base class definition is shown in Figure 12.40. Notice the use of the keyword `virtual`. Since the method is declared as virtual, all

```
#ifndef INTERFACE_TYPE
#define INTERFACE_TYPE
include <systemc.h>
/* this header describes the interface type...*/

class my_basic_rw_port
  : public virtual sc_interface
{
public:
  // basic read/write interface
  //virtual bool read(int address, int data) = 0; //not used anymore!
  virtual bool read(int address, int source_address, int data,
                    int dest_task) = 0;
  virtual bool write(const int source, int address, int data,
                     int dest_task) = 0;
  virtual bool non_blk_write(const int dest_address, int source,
                             int data, int dest_task)=0;
}; // end class
#endif
```

FIGURE 12.40 "interface_type.h."

classes that inherit from this class must either provide, or inherit, an implementation for the `read`, `write`, and `non_blk_write` methods. This common interface is also what allows the type compatibility for pointer assignment used in the *generic.h* file. This is described in more detail in Section 12.5.4.

12.5.3 Processor Model

The processor model has a fairly simple structure. There are three methods in this model, the constructor, and two member functions. The first member function describes the performance only behavior of the processor, the second describes the mixed-level behavior. The mixed-level functionality will be described in detail in Section 12.5.9.

In addition to those methods, the processor model has a number of objects that are members of the class. It has three integer variables for passing command arguments to the interface methods, a pointer for a `command_in` object that opens the command file and parses the model execution commands for the processor model, a `command_type` object that is used to return the commands from the `command_in` object, a signal of enumerated type `action_type` to display the current action, an unsigned signal to display the current task number, and a pointer for a refined computation model. Figure 12.41 graphically shows the objects in the processor model. On the far right is the IO port. It is mapped to the interconnect interface. Next on the right are the three integer variables used to pass information to the interconnect calls. Below the variables are boxes representing the two signals that allow the state of the processor to be viewed in the ModelSim waveform window. Then to the left there are the performance and mixed-level descriptions. One of these member functions will be turned into an `SC_THREAD` and will control the model's behavior during simulation. On the top labeled as refined model, is an outlined box representing the pointer to a refined model object. Below is a storage location (labeled command) that the `command_in` object will return values in. In the bottom left is a dashed box representing a pointer to a `command_in` object. In the top left is a box representing the constructor. The IO port and constructor are both on the edge of the processor model because they are the only ones that interact with other objects in the simulation.

The constructor receives a processor number, and creates a `command_in` object with the appropriate processor number for the processor the current instance represents. The `command_in` object creates a string with the proper processor number in the middle, and uses it to open the processor's command file. It is called by the active behavior to read in the next command once the previous one has executed. Its primary function is to remove command parsing from the processor model. Having it as a separate object makes changing how the commands for the processor are read in or generated a simple matter of including a different implementation of the object. Once the `command_in` object is created and initialized, the processor model constructor then instantiates either the performance only or mixed-level implementation, and opens its log file. It does this by registering the proper member function with

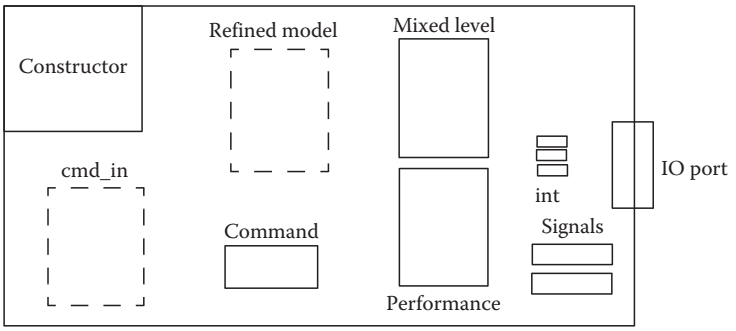


FIGURE 12.41 Graphical representation of processor model.

```
void tlm_behav()
{
   int temp2;
   double temp_time;
   while(1)
   {
      cmd_input->get_command(&this_command);
      switch (this_command.command)
      {
      case 1://ie send
         {...
            break;}
      case 2: //ie receive
         {...
            break;};
      case 0: //ie compute
         {...
            break;};
      case 3: //non_blocking_send
         {...
            break;};
      case 4: //io done
         {...
            break;};
      case 5: //loop/branch
         {...
            break;};
      default: //idle the processor
         {...
            break;};
      };//end of switch
   wait(SC_ZERO_TIME); //to break things up in the text output...
   };//end of infinite while loop
};
```

FIGURE 12.42 Main processor loop.

the simulation kernel as an SC_THREAD. During simulation the processor model essentially reads in a command from a file, then uses a case statement to perform whatever command was read in and repeats until it reaches a done command, an end of file, or some command it does not recognize. Figure 12.42 shows the framework of the main processor loop.

Notice in the source code that the processor has a port of the same type as the base type for the interconnect channels my_basic_rw_ port. This port is bound to the interface of the channel object. If the command read in from the command file is a send or receive command, then the processor uses the port as a pointer to the interface to the channel object and accesses the appropriate interface method to perform the send or receive operation. In the models here the thread in the processor model actually executes all the code in the blocking send and receive methods, so the processor model is incapable of doing anything else until the blocking io function returns.

12.5.4 Channels

The channels used in these models are considered hierarchical channels. They are not any of the predefined SystemC primitive channel types, they are composed of multiple objects, and they contain a number of threads. To allow for a variable number of processors to connect via the channels they have only an interface and no ports. In SystemC, all ports must be bound to something, be it another port, an interface, or a signal. Interfaces however may exist even if nothing is bound to them. So for maximum

flexibility, channels should provide an interface, and any connected modules should have a port of the type of the interface and have that port bound to the interface on the channel.

Since multiple ports can be bound to a single interface, the channel object can have any number of processors bound to it. However, the crossbar and fully connected channels' behavior is determined in part by the number of processors present, so all channels are passed a constructor argument that tells them how many processors are present in the simulation. The channel models are the most extensive models since their behavior is an abstract representation of all of the characteristics of an interconnect topology. They model the arbitration, data transfer, and blocking/nonblocking characteristics of the interconnect without restricting the designer to a particular implementation. Since nonblocking sends are allowed they also implicitly model a sending queue.

All the channel models are based on the `comm_medium` class. The `comm_medium` class provides a logical connection between processors, with signals for the source processor number, the destination processor number, and transaction type, as well as blocking and nonblocking read and write methods, and an arbitration thread. In addition, the `comm_medium` class provides two threads to allow for nonblocking reads and writes. Figure 12.43 shows the members of the `comm_medium` class. The four signals shown in the top left are signals to show the current state of the logical connection in the waveform window. To the right are the two wait queues, one for write request, and one for read request. New transaction requests received via the interface methods are placed into these queues. The Boolean `no_match` variable maintains whether there is currently a match between read operations and received data, the integers below it are used to store the values of the current processor for the sender and receiver, and to store the location in the queue of the current send request being executed, and the current receive request being executed. Below the integers is a dummy variable whose sole purpose is to fix an existing bug in the implementation of the SystemC simulator. In the bottom left of the figure are the member functions of the object. The functions all the way to the left are the functions intended to be accessed by other objects, the remaining four are intended for internal use only, though they are declared as public and thus visible to other objects. The functions with a star after them are registered with the simulation scheduler as `SC_THREAD`s. The top two events in the bottom right of the figure are used by the read and write methods to notify the arbitrator process that there may be new pairs of requests that could be activated to communicate. The event in the very bottom is used to coordinate the execution of two threads when they communicate. In the top right of the figure are two transaction pointers. These

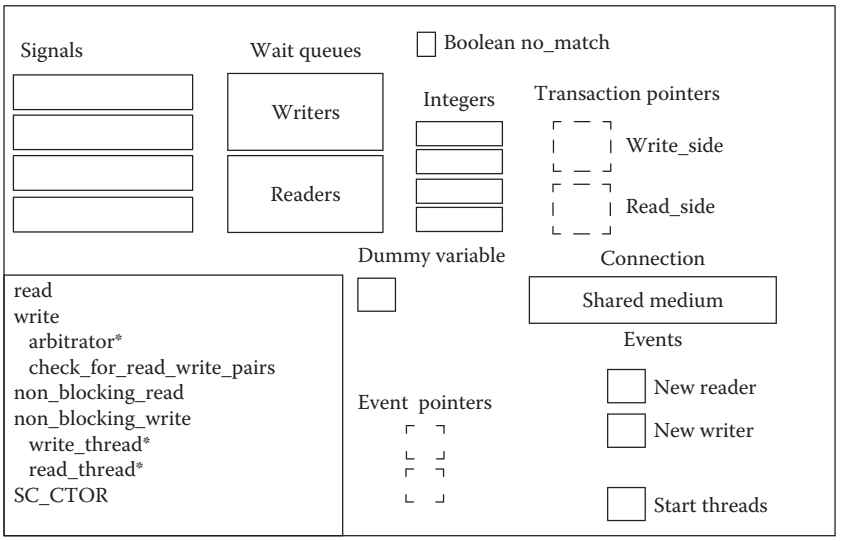


FIGURE 12.43 Graphical representation of ''comm_medium'' class.

pointers are used by the arbitrator to keep track of which two transactions it is dealing with. The event pointers in the bottom center of the figure are also used by the arbitrator. When the arbitrator has selected two transaction requests to communicate it does not actually handle the communication. There is a write thread, and read thread that execute the `transact()` code in each of the two transaction objects. For any blocking transaction objects the calling processor's thread is suspended in the interface call waiting on the transaction to notify it that the transaction is complete. Any nonblocking transaction objects have already had their calling thread return to the processor model code. The arbitrator has no need to check for any potential request pairs until after the two threads have completed their transaction. These two pointers are set to allow the arbitrator to suspend until both threads have completed, rather than wasting simulation resources polling to see if they are done.

While support exists in the `comm_medium` object for nonblocking reads, the channels do not have methods to give access to that functionality to the processor models. This was done on purpose to avoid having to check data dependencies before beginning a computation. This also keeps the simulation simple, and more efficient in terms of simulation time. The functionality was built into the `comm_medium` object because it was easy to do and makes adding nonblocking reads at some future point much easier. The arbitration scheme provided is a longest waiting first scheme. As soon as at least one transaction pair, a matching send and receive, is present the pair with the largest sum of positions in the wait queues is selected to transact next. The crossbar channel uses a variant of the `comm_medium` class. In the crossbar variant there are pointers to the transaction wait queues which allow a single set of queues to be used by all of the logical channels.

The `comm_medium` class also makes use of the class `transaction_wait_queue`, which is a specialized linked list to allow for a large number of waiting transactions without allocating a fixed large amount of memory. The elements of the linked list are of the class transaction_element, which contains all the essential information about a transaction request. The only item of importance from the linked list is the class that actually holds the transaction information. This `transaction_element` class contains all of the information about the transaction request. Figure 12.44 graphically depicts the key elements of the `transaction_element` class.

The integers in the bottom center contain the source processor number, the destination processor number, the destination task's id number, and the size, in nanoseconds, associated with the transaction. The Boolean variables in the top right tell whether the transaction element is a write or read, and whether it is blocking or not. The handshakes object in the upper middle is a set of four events that are used by the transact method to logically ''perform'' the transaction. The complement pointer below the handshakes is a pointer to a `transaction_element` object. To communicate two `transaction_element` objects must be paired up the channel's arbitrator process. It does this pairing by setting a read and a write's compliment pointer to each other. The `activate_me` event in the top left is used by the arbitrator to activate the thread executing the element's side of a transaction.

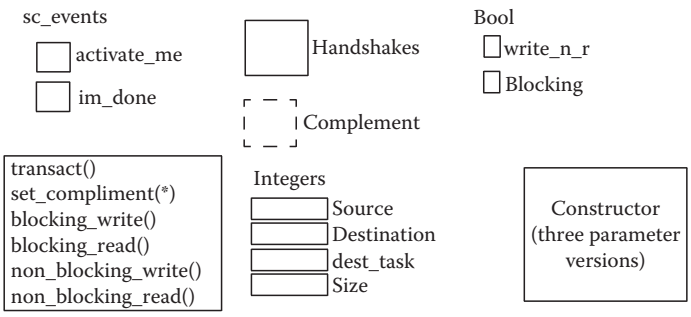


FIGURE 12.44 Graphical representation of ''transaction_element'' class.

While the im_done event is used to notify the arbitrator, and in the case of a blocking transaction the requesting processor's thread, that the transaction is complete. The constructor for this object depicted in the bottom left has three different implementations with different parameter lists, the first constructor implementation that sets all of the transaction values is the one used in the current version of the models. The others were left to maintain backward compatibility with previous versions, and may be useful for future versions. The purpose of the methods displayed in the bottom left of the figure are self-explanatory. The transact method is what controls the actual behavior of a transaction pair once it has been scheduled. The nondebugging parts are repeated below in Figure 12.45. The blocking and nonblocking versions of the read and write routines are the same in this version of the models.

The nondebugging versions of the blocking read and write methods are shown in Figure 12.46. These methods show how the four-way handshaking is implemented. The use of a full four-way handshaking in the channel model is somewhat arbitrary, but it makes incremental refinement of the channel easier.

```
  void transact() //method for creating thread to call (blocking read
or write)
   {
      wait(activate_me); //wait for arbitration process to activate me
      if (blocking)
      {
         if (write_n_r) //then it's a blocking write
            blocking_write(); //call the blocking write routine
         else //it's a blocking read
            blocking_read();  //call the blocking read routine
      }
      else
      {
         if (write_n_r) //then it's a non-blocking write
            non_blocking_write(); //call the non-blocking write routine
         else //it's a non-blocking read
            non_blocking_read();  //call the non-blocking read routine
      };
      im_done.notify(SC_ZERO_TIME);
      return;
   };
```

FIGURE 12.45 Transaction_element transact() method.

```
   void blocking_write()
   {
      compliment->handshakes.start_write.notify(SC_ZERO_TIME);
      wait(handshakes.start_ack);
      compliment->handshakes.write_done.notify(SC_ZERO_TIME);
      wait(handshakes.done_ack);
   };

   void blocking_read()
   {
      wait(handshakes.start_write);

compliment->handshakes.start_ack.notify(size,SC_NS);//SC_ZERO_TIME);
      wait(handshakes.write_done);
      compliment->handshakes.done_ack.notify(SC_ZERO_TIME);
   };
```

FIGURE 12.46 Transaction_element write and read methods.

```
    void blocking_write()
    {
       wait(handshakes.start_ack);
    };

    void blocking_read()
    {

  compliment->handshakes.start_ack.notify(size,SC_NS);//SC_ZERO_TIME);
       wait (size,SC_NS)
    };
```

FIGURE 12.47 Alternative write and read methods.

However, with the abstract behavior described here a single line for the write method and two for the read method would be sufficient. Figure 12.47 shows how the methods could be implemented in this way.

All of the channel models also read in parameter information from the *channel_param.txt* file located in the directory that the simulation is running in. This file contains two lines. The first line is the bus speed in megabytes per second. The second line is the fixed communication overhead per communication transaction. In the top-level channel models the data size parameter passed to read interface method is run through a `data_to_delay` function that return the delay in nanoseconds that the communication should take based on the specified bandwidth and communication overhead.

12.5.5 Shared Bus Model

The shared bus architecture consists of a single logical communication medium, which is an object of class `comm_medium`. The behavior is encapsulated in the `shared_bus_io` class. This class inherits the virtual interface functions from the `my_basic_rw_port` class, and must provide an implementation for them. The implementation for these functions is shown in Figure 12.48. The

```
   inline bool read(int dest_address, int source_address, int data_size, int
dest_task)
   //blocking read function with writer's address
      //read function takes data to be size of data transmission
   {
      int temp_size;
      temp_size = data_to_delay(data_size);
      shared_bus.read(dest_address, source_address, temp_size, dest_task);
      return 1;
   };
   inline bool write(const int dest_address, int source, int data,
                     int dest_task)
      //blocking write function here
   {
      shared_bus.write(dest_address, source, data, dest_task);
      return 1;
   };
   inline bool non_blk_write(const int dest_address, int source, int data,
                             int dest_task)
   //non-blocking write function here
   {
      shared_bus.non_blocking_write(dest_address,source,data,dest_task);
      return 1;
   };
```

FIGURE 12.48 Implementation of inherited virtual interface functions.

functions essentially map the interface functions to the methods of the comm_medium object. The data_to_delay function takes the data_size passed to the read method, and calculates the required transaction time in nanoseconds based on the bandwidth, and then adds the fixed communication overhead that the channel's constructor reads in from the channel parameter file.

12.5.6 Fully Connected Model

The fully connected architecture is built around the same comm_medium object as the Shared Bus model. The fully connected architecture creates a comm_array object which contains all the logical connections, and copies the addresses of the comm_medium objects into a two-dimensional array that it uses to map a processor's request to the overall communication architecture to the appropriate logical connection. The fully connected architecture passes the number of processors to the comm_array object which then instantiates the number of comm_medium objects needed to provide a dedicated shared bus between each pair of processors in the simulation.

12.5.7 Crossbar Model

The crossbar architecture is similar to the fully connected architecture except that it only requires four cross_comm_medium objects since with nine processors only four concurrent connections are allowed. As mentioned earlier, it uses its own version of the basic comm_medium object. This is necessary because the logical connections in the crossbar are not associated with any particular processor, and the communication requests are not associated with any of the logical connections. Just to clarify in the fully connected architecture there is a logical connection between every processor modeled by a comm_medium object. The fully connected architecture model simple directed the requests it received to the correct logical connection. In the fully connected architecture there is only one connection that all requests are intended for, but in the crossbar the number of logical connections is equal to the number of processors divided by two, and every request could potentially communicate over any of them.

12.5.8 SystemC Performance Modeling Examples

This section contains a number of examples of performance models constructed using the SystemC modeling modules described above. The examples are presented as a demonstration that the models execute correctly and also that they demonstrate performance of the system they are intended to model. The first example is a trivial example with a set of four tasks all executing on one processor. Since data communication inside a processor is assumed to take no time, the description should take simulation time equal to the sum of the computation time of all the tasks. The second example is the same four tasks allocated to two processors such that each task must send the data over the communication channel to the next task. This second example should take longer, with three 100 byte sends being sent over the communication channel. The third example is the same task graph description with varying bus parameters. The fourth and fifth examples have bus contention, to show that contention is handled properly. Each example lists the simulated latency, and a timeline showing the simulation results.

12.5.8.1 Single Processor

Figure 12.49 shows the simple sequential task graph for the first example. Here all the tasks are allocated to Processor 0. Each task has a compute value of 10 μs, and each edge has a data value of 100 bytes. Since communication within a processor is assumed to take no time, the latency for this description should be the sum of the compute times, which is 40 μs.

In addition to the processing timeline shown previously, the models generate a text output stream that describes the actions each module is taking at a given simulation time. The text output for this simulation

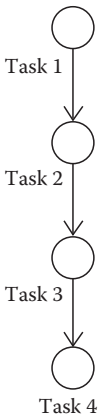


FIGURE 12.49 Single processor example task graph.

is shown below. Note that the final task is completed at 40 μs of simulation time which is exactly as expected.

```
shared bus architecture
0 s proc#0 Task number 1 Computing for 10000 ns
0 μs proc#0 Task number 2 Computing for 10000 ns
20 μs proc#0 Task number 3 Computing for 10000 ns
30 μs proc#0 Task number 4 Computing for 10000 ns
40 μs proc#0 done!
sim done!?
```

12.5.8.2 Dual Processor

The task graph for the second example is shown in Figure 12.50. The graph also shows the allocation of the tasks to two processors. Notice that the sequential tasks are on different processors so the data must be transferred across the communication channels before the computations can begin. Here the communication channel's bandwidth determines how long a communication transaction should take to complete. The length of time is the data size in bytes divided by the bandwidth in megabytes per second. The communication channel can also take into account communication overhead, in nanoseconds, if it is specified. The channel bandwidth and communication overhead are read in from a file. If this file is not present or an item is missing it will take on its default value. The value specified for this example is 100 Mbyte/s for bandwidth and 5 ns for channel overhead.

Since all the communication is of the same size, the expected latency for this example can be determined using the following equation:

$$\text{Computation time} + \text{numtrans} * \left(\frac{\text{Data size (byte)}}{\text{Bandwidth (Mbyte/s)}} + \text{Communication overhead (ns)} \right)$$

which, for this example, evaluates as follows:

$$40\ \mu\text{s} + 3 * \left(\frac{100\ \text{byte}}{10\ \text{Mbyte/s}} + 5\ \text{ns} \right) = 40\ \mu\text{s} + \frac{300}{100 * 10^6}\ \text{s} + 15\ \text{ns} = 40\ \mu\text{s} + \frac{300}{100}\ \mu\text{s} + 15\ \text{ns}$$

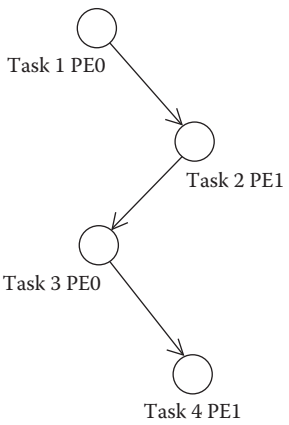


FIGURE 12.50 Dual processor example task graph.

Thus the expected latency is 40 μs for computation plus 3 μs for the actual data transmission, plus 15 ns for communication overhead. That gives a total latency of 43,015 ns. The timeline output for this simulation is shown in Figure 12.51. Note that the final task, Task 4, completes at 43 μs on the graph.

12.5.8.3 Parallel Communications Example

The next example shows the effect of various communications topologies on an application with requirements for simultaneous communications. The task graph for this example is shown in Figure 12.52. Each task (called nodes in this graph) computes for a fixed period of time and then sends data to a second task causing it to begin execution. The tasks are allocated to processors such that after completion of the first set of tasks, all processors attempt to send data to another processor. Because the first tasks all have the same execution time, all the communications become ready to begin at the same time. Thus, if an architecture has parallel

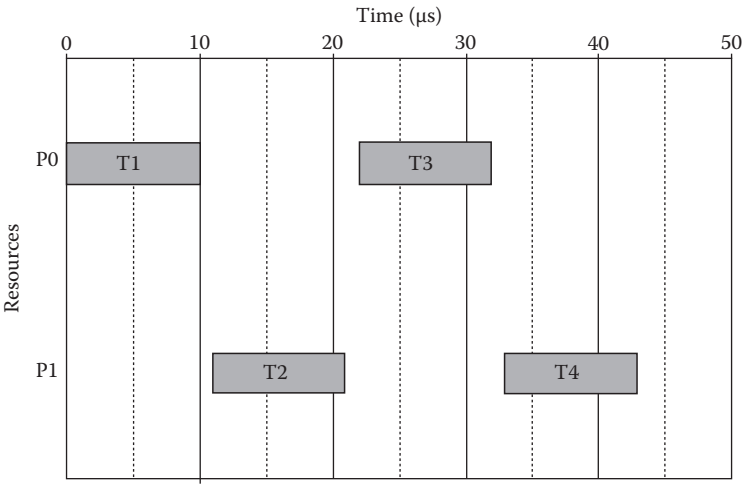


FIGURE 12.51 Dual processor example timeline.

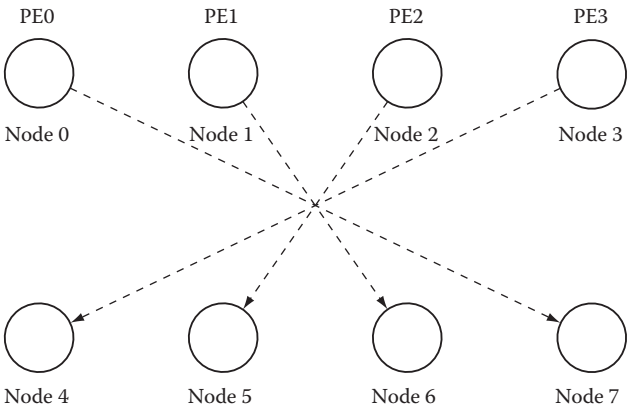


FIGURE 12.52 Bus contention example task graph.

communication paths, this will result in a decrease in the total application run time. The four start tasks (nodes) in this example all compute for 10 μs, then attempt to do a nonblocking send of size 100 byte to a task allocated to another processor. They then move on to start the read required to begin their next task.

For all of the results discussed below, the channel parameters are set to a bus bandwidth of 1 Mbyte/s and a communication overhead of 0 ns.

12.5.8.3.1 Shared Bus Simulation Results

The first set of results is for a system with a single shared bus. On this system Tasks 0–3 all execute in parallel on the four processors. At this point there will be four-way contention for the single system bus. The communications operations will be assigned priority on a first-come-first-serve basis. In the current implementation of SystemC, the task that will get first priority to communicate its data cannot be determined ahead of time, however the tasks will all run in the same order every time the simulation is run.

With a shared bus architecture, the latency is 100 μs for all processors to compute in parallel plus 4*(data size/bandwidth)*1000 (ns), or 400 μs, for each of the four sends to occur in series, plus 100 μs for the last receiver to compute after completing their receive. Thus the overall latency should be 600 μs.

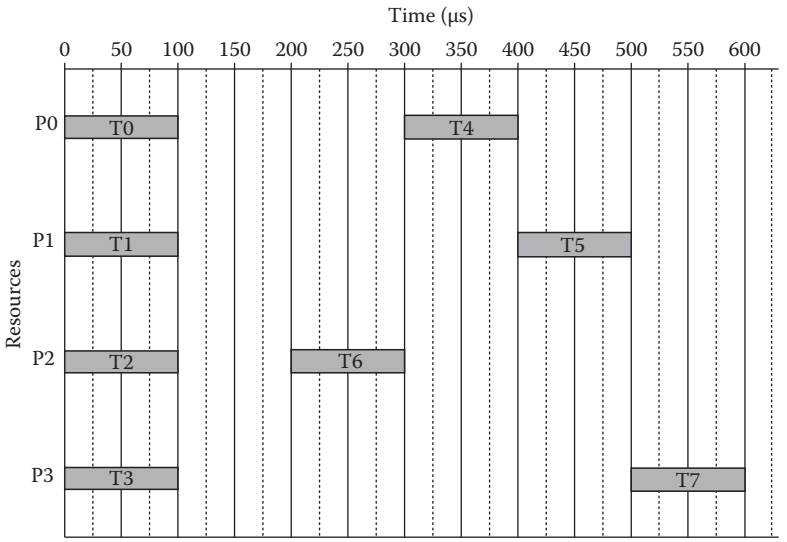


FIGURE 12.53 Shared bus contention example timeline.

The timeline for this example is shown in Figure 12.53. The timeline shows all the tasks beginning their blocking read then having to wait for the arbitrator to select them to communicate across the bus. Once the individual communications have taken place, the destination task, Tasks 4–7, execute. The timeline correctly shows the last task completing execution at 600 μs.

12.5.8.3.2 Fully Connected Simulation Results

In the fully connected architecture there is a dedicated communications channel between each pair of processors. However, in this architecture, it was decided to model a system where a processor cannot both send and receive a message from the same processor at the same time. Because of the connectivity of the task graph for this application, after the first set of tasks execute in parallel, each processor needs to send and receive a message before it can execute the next task. For example, Processor 0 cannot send to Processor 3 and receive from Processor 3 at the same time. Rather it must do one, then the other. Thus, for this example, each channel in the fully connected architecture is effectively a half duplex connection.

During execution the run time is 100 μs for all processors to compute the first four tasks in parallel plus 100 μs for the first set of sends, plus 100 μs for the second set of sends—during which the tasks started by the first set of sends also execute, then finally 100 μs for the last two tasks to compute in parallel. Thus the overall latency for the fully connected architecture should be 400 μs. The timeline for this example is shown in Figure 12.54.

12.5.8.3.3 Crossbar Simulation Results

As mentioned above, the crossbar architecture behaves like a fully connected architecture where the maximum number of connections is limited to the number of processors divided by two. Thus for this four processor example, the crossbar architecture will only allow two communications at a time. This characteristic means that for this example, the crossbar architecture will have the same latency as the fully connected architecture for this example. This result is shown in Figure 12.55.

12.5.8.4 Second Contention Example

This second example expands on the previous example by showing a slightly different set of contention conditions. In the first example, the communication requirements specified by the task graph required

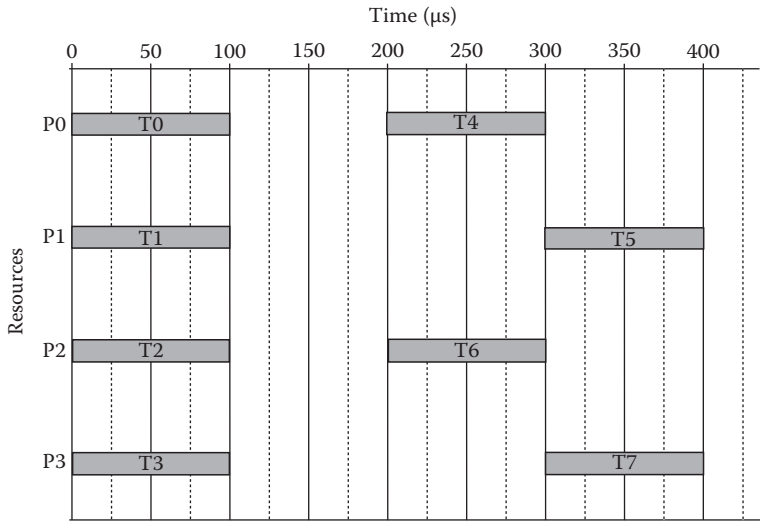


FIGURE 12.54 Fully connected contention example timeline.

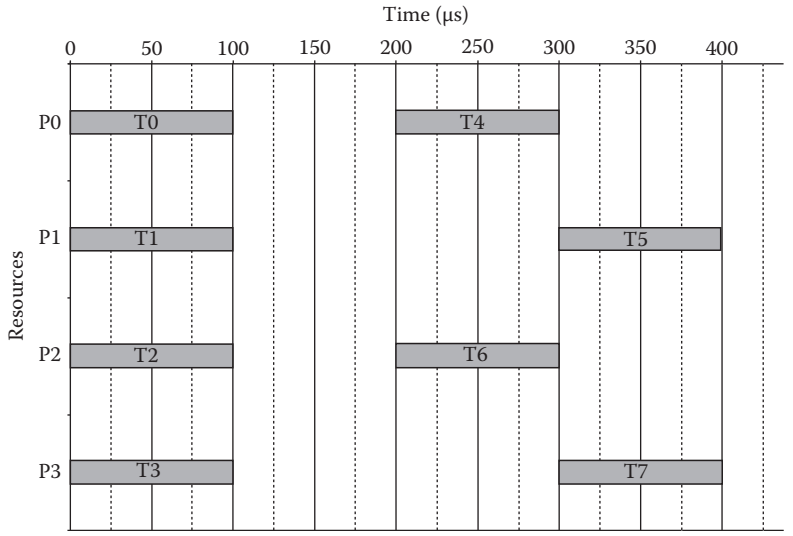


FIGURE 12.55 Crossbar contention example timeline.

the processors to send and receive data from the same processor. This effectively allowed for only two active communication transactions on the fully connected architecture. In this example, as shown in Figure 12.56, the processors will be sending and receiving data from different processors during the communications portion of the application. This set of communication requirements will allow all of the available communication channels to be used concurrently with the fully connected architecture.

Figure 12.57 shows the results for this example for the shared bus, fully connected, and crossbar architectures. Note that in this example, in the fully connected architecture, all of the communication operations occur in parallel which allows the entire application to execute in 300 μs.

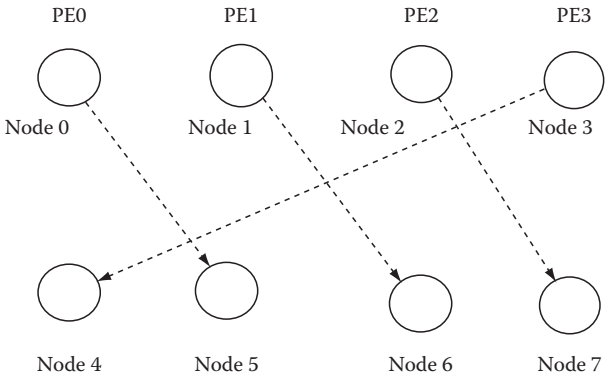


FIGURE 12.56 Second contention example task graph.

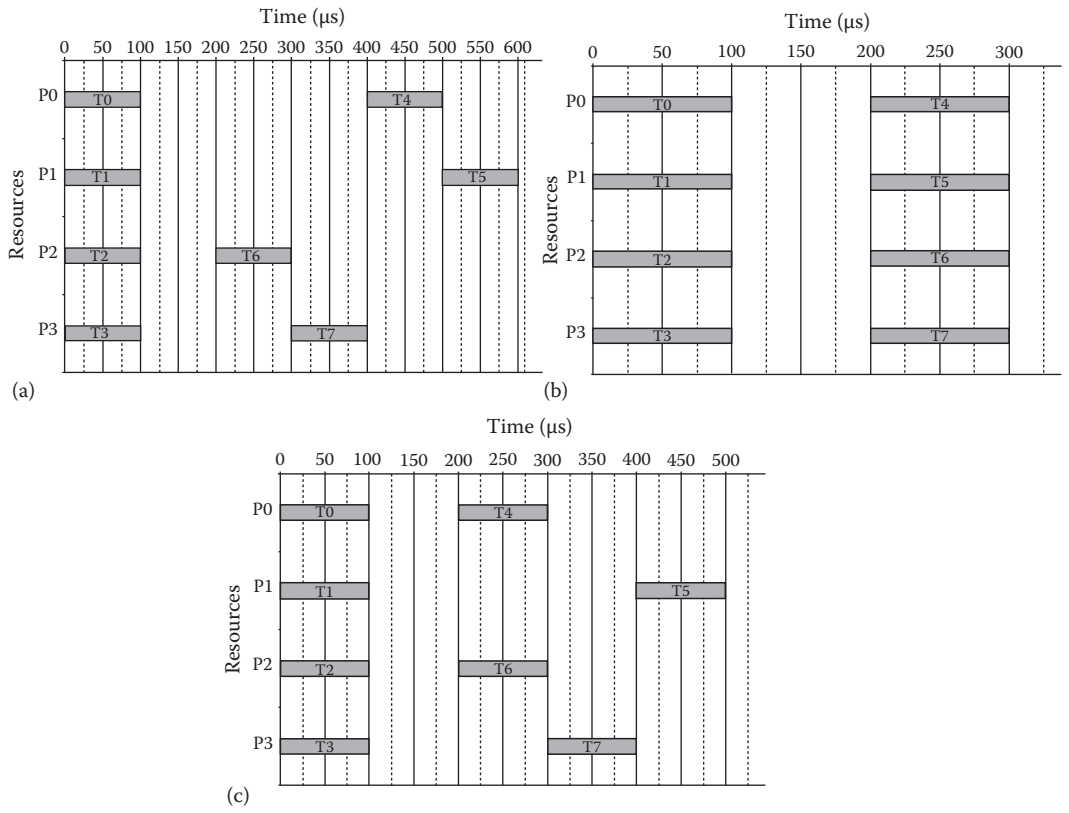


FIGURE 12.57 Second contention example timelines: (a) shared bus, (b) fully connected, and (c) crossbar.

12.5.9 Mixed-Level Processor Model

Although simplistic in nature, the above examples show that the SystemC-based performance modeling methodology can be used to model the execution of different applications on various system architectures. In addition to this capability, as mentioned earlier, the SystemC module for the processor has the

ability to replace the performance only computation delay with a refined computation model that is described in VHDL or Verilog. This mixed-level modeling capability allows the model to be refined to a lower level in a step-wise fashion.

The replacement of an abstract processor model with a refined (RTL or gate-level model) is accomplished by using the ModelSim `SC_FORIEGN_MODULE` syntax. The `SC_FORIEGN_MODULE` provided by ModelSim allows a non-SystemC model that has been compiled for ModelSim to be instantiated by a SystemC model. Incidentally, it also allows an already compiled SystemC module to be loaded in the same manner.

The processor model opens its processor command file and looks at the first line during the execution of its constructor. If the first line is "mixed" then the processor model knows that it should run as a mixed-level model with a refined computation model. If the model is to be a mixed-level one, the constructor of the processor model will then look for a *mixed_processor.txt* text file that specifies the ModelSim path for the refined computation model to use. The relevant part of the constructor that instantiates the refined model is shown in Figure 12.58.

The `refined_computation` object is the one that opens the *mixed_processorX.txt* file, where *X* is the processor number. Once this file is opened, the constructor uses the path contained within it to open the refined model object. In this example, it instantiates an object of class `rng_comp_tb` that is shown in Figure 12.59.

This class is essentially a wrapper for the actual precompiled model VHDL or Verilog model. This class/module definition would map to a VHDL entity definition like the one shown in Figure 12.60. Note that this entity corresponds to what is effectively a test bench for the refined model.

```
        cmd_input = new command_in(proc_num);
        string_temp.assign(cmd_input->get_proc_type());
        if(/*1st_line*/string_temp=="mixed")
        {
           SC_THREAD(mixed_behav);
           //only make the relevant behavior a thread!
           hardware_compute=new
 refined_computation("name",proc_num);
           //may need to add an argument this class...
        }
```

FIGURE 12.58 Refined computation part of processor model constructor.

```
class rng_comp_tb : public sc_foreign_module
{
public:
    sc_in<sc_logic> start;
    sc_out<sc_logic> done;

    rng_comp_tb(sc_module_name nm, const char* hdl_name)
     : sc_foreign_module(nm, hdl_name),
       start("start"),
       done("done")
    {
      //cout<<"Building hardware model\n";
   };

     ~rng_comp_tb()
     {}

};
```

FIGURE 12.59 Code for "rng_comp_tb" class.

```
ENTITY rng_comp_tb IS
   port(
     start : in std_logic;
     done : out std_logic
     );

END rng_comp_tb;
```

FIGURE 12.60 Sample VHDL refined computation entity declaration.

The intent for this interface is for it to be easy to integrate into an existing test bench for the refined model of computation. The start and done signals are active high. So when the start goes high to a logical ''1,'' the refined model should start a ''computation.'' This computation is effectively the test bench applying a set of predefined stimulus waveforms to the refined model. As described below, these stimulus waveforms can be derived for the specific refined model in a number of different ways depending on the objectives for the mixed-level model.

When the refined model has finished its ''computation'' it should raise the done signal. Both signals should be low at the start of the simulation. When the processor model puts the start signal high, the refined model begins its computation. The presumption is that the refined model is something like a test bench, with a model of the actual hardware, or some other more detailed model, instantiated inside of it. The refined model presents any required data to the detailed model, and watches for whatever condition indicates that it has completed the computation. Once the computation is completed, it raises the done signal telling the abstract processor model it is done. The processor model will then lower the start signal, and the refined model will respond by lowering the done signal. Figure 12.61 shows the basic timing diagram for the interface.

Figure 12.62 shows the portion of the processor model that raises and lowers the start signal and waits for the done signal. This code is located in the `refined_computation` object. The `refined_computation` object is instantiated by the processor model's constructor when it reads in from the command file and determines that it should be a mixed-level model.

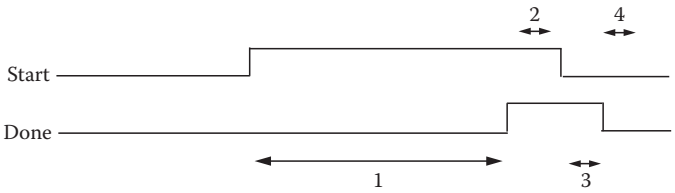


FIGURE 12.61 Refined computation model interface-timing diagram.

```
   bool start_compute()
   {/* basic algorithm is:
      send start signal
      wait for done signal
      return
      */
#ifdef MTI_SYSTEMC
      temp= true; //'1';
      start = temp;
      while (done != temp)  //ie while done != 1
         wait(done.value_changed_event());//wait until event and
loop..
      temp = false;
      start = temp;
#endif
      return 0;
   };
```

FIGURE 12.62 SystemC start/done interface code.

```
-----------------------------------------------------------------------
-- Process to respond to start and create go and done. JA
-----------------------------------------------------------------------
   CompBench: PROCESS (start, tb_done)
   Begin
      if (start='1') then
         if ((tb_done'event) AND (tb_done='1')) then
            go   <= '0';
            done <= '1';
         ELSE
            go   <= '1';
            done <= '0';
         end if;
      end if;
   End process CompBench;
```

FIGURE 12.63 Sample VHDL start/done interface code.

Figure 12.63 shows a portion of sample VHDL test bench that implements the refined computation model's side of the start/done interface. The start signal from the start/done interface activates the process. The process uses the go signal to cause the test bench to perform a computation, and the test bench raises the `tb_done` signal when it is finished with a single computation.

12.5.10 Mixed-Level Examples

The following examples are mixed-level examples where the computation part of the processor is modeled in more detail. Since the rest of the simulation is at a more abstract level and does not have all of the stimuli needed for the refined model, the stimuli need to be created in some way. The following examples focus mainly on different ways of generating the data that the various refined models need to function.

12.5.10.1 Fixed Cycle Length

The first example is one where an abstract processor is replaced with a random number generator. The model for the random number generator is an RTL discrete digital model of a random number generator described in Ref. [51]. The model attempts to describe a number of elements that are extremely sensitive to initial conditions, and thus in reality exhibit more random behavior than can be modeled with a solely digital model. As it is, the model always generates the same nonrepeating sequence of values. For this example, the existing test bench was modified to put the generator through its reset cycle, then through a single random number generation. The only inputs to the refined model are a set of control signals and a clock. When the processor, that it is the refined computation model for, gets a compute command; it will send the start signal to the test bench, which will then go through the reset and generate phases, and signal back with the done signal once a number has been generated. In this particular example the internal signals continue to oscillate between compute commands. Since this model takes a fixed number of cycles using the refined model, and the values generated are not passed elsewhere, this example is less efficient and no more accurate than putting in the actual compute time for the abstract compute command.

12.5.10.2 Variable Cycle Length

The rest of the mixed-level examples presented use an RTL booth multiplier model. The booth multiplier takes a variable number of cycles to complete the binary multiplication. The number of cycles required depends on the numbers being multiplied. The inputs to the model are the two numbers to be multiplied, and a clock, the outputs are the result and a control signal indicating that the multiplication is complete.

There is a slight propagation delay for the result to appear on the output after the done signal appears. If the input clock continues to cycle after the multiplication is complete, then the result will become invalid after a clock cycle. Thus the test bench allows a half cycle to elapse before considering the computation done. Note that the effect of this refined model is that the computational delay is dependent on the data being applied to it for the computation. This delay mechanism is a more accurate representation of how a refined model would be used in, and add additional accuracy to, a system-level performance model. However, as discussed in the section above on VHDL-based mixed-level modeling, the important question is how to generate the data that is input to the refined model in such a way as to accurately represent the performance of the refined component in the real system. Typically, this can be done by either presenting the refined model with random data to develop a statistical representation of average system performance, or presenting the refined model with predefined data. This data can be generated by the designer to represent typical system performance, or to exercise the best-, or worst-case delay scenarios.

12.5.11 Random Data Generation

In this example, the booth multiplier is presented with two random numbers generated by two other entities instantiated in the test bench. Since random number generation is not present in standard VHDL library, ModelSim's mixed language ability is utilized to allow a SystemC random number generator module to be used. The SystemC module uses the C++ `rand()` function to generate a pseudo random number, which is passed back to the VHDL test bench. Since the VHDL test bench is instantiated by the SystemC performance simulation, this is in fact a SystemC–VHDL–SystemC hierarchy. ModelSim's mixed language interface allows the designer to use whatever language is best suited for the task at hand. Here the random number must be passed back to the VHDL test bench as an `sc_logic` vector, which is automatically translated into a VHDL `std_logic_vector` by the simulator. Since both numbers are pseudorandom, the computer will take a variable amount of time to complete. Although it is possible that the use of the `rand()` function will result in repeating sequence of numbers, most simulations will not run long enough for this to be noticeable. If a more random distribution, or a particular type of distribution is desired, a specialized random number generator package can be used in the model. Since most of the computations at the system level, will consist of multiple operations, to make this example more typical, a means of generating a set multiple numbers to be multiplied was needed. While it is possible to generate a fixed size data set this example goes just a little farther and generates a pseudorandom size data set of pseudorandom numbers. The generation of the random size of the data set is done using the same SystemC module that generates the random data itself. A few changes to the test bench needed to be made so that it did not send the done signal back until all of the multiplications were completed. Figure 12.64 shows two waveforms from two different mixed-level simulations, each with two different size sets of random numbers multiplied together.

12.5.11.1 Data Set from File

As described above, when a sample set of data that exercises a specific scenario for the refined model is available, it can be advantageous to use it rather than generating a new set that may or may not be close to the actual data. Using this predefined sample data set ensures accurate performance for that data set, and removes any guesswork as to what a realistic data set might be. Since VHDL has standard file access capabilities that are easy to use, the test bench for this example reads the values directly from the input file, and does not send the done signal until it reaches the end of file. Unfortunately, the current implementation ModelSim used for this example did not allow passing generic information across the language boundary, thus specification of the file to use had to be done in the VHDL code. An extension to this approach would be to read the filename to use from a configuration file similar to what is done to specify the refined model to use in the SystemC performance code. Figure 12.65 shows the waveform of the mixed-level model simulation where the data for the refined component were read in from a file.

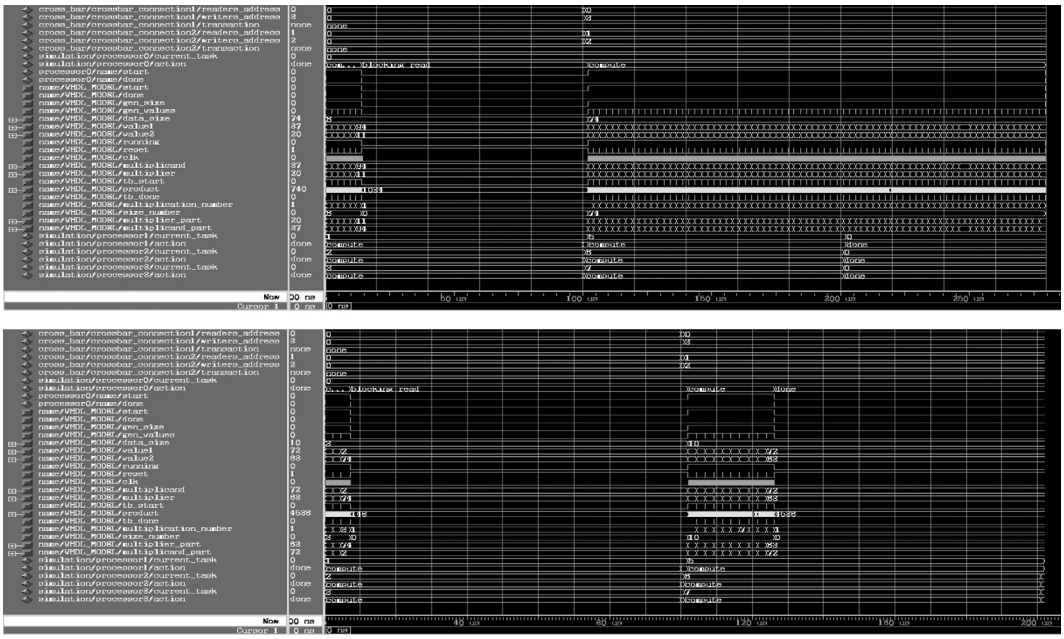


FIGURE 12.64 First- and second-random data set waveforms.

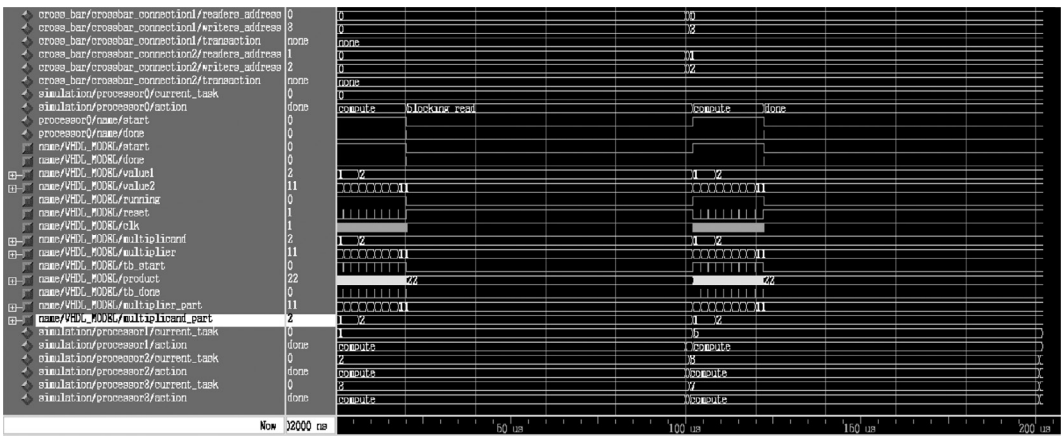


FIGURE 12.65 Data from file example waveform.

12.5.12 Mixed-Level Example Summary

Mixed-level modeling techniques in SystemC-based performance models have been demonstrated. These examples utilized simple pseudorandom data generation, variable size pseudorandom data set generation, reading a data set from single file, and reading multiple data sets from multiple files. Multiple variations and combinations of the above approaches can be used to generate stimuli for a wide variety of refined computation models. Anything from randomly selecting a file from which to read a data set, to using a file to parameterize random data generation is possible depending on whether realistic data are available and on the simulation objective. It is clear to see, in any case, that including refined behavioral

RTL or gate-level components in a performance model in SystemC is fairly easily done and can increase the overall accuracy of the performance model, just as was possible in the VHDL-based performance model.

12.6 Conclusions

Integration of performance modeling into the design process such that it can actually drive the refinement of the design into an implementation has clear advantages in terms of design time and quality. The capability to cosimulate detailed behavioral models and abstract system-level models is vital to the development of a design environment that fully integrates performance modeling into the design process. One methodology and implementation for cosimulating behavioral models of individual components with an abstract performance model of the entire system was presented. This environment results in models that can provide estimates of the performance bounds of a system that converge as the refinement of the overall model increases.

This chapter has only scratched the surface on the possible improvements that performance- or system-level modeling can have on the rapid design of complex VLSI systems. As more and more functionality can be incorporated into the embedded VLSI systems and these systems find their way into safety-critical applications, measures of dependability such as reliability and safety at the system-level are becoming of great interest. Tools and techniques are being developed that can use the performance model from which to derive the desired dependability measures. In addition, behavioral fault simulation and testability analysis are finding their way into the early phases of the design process. In summary, the more attributes of the final implementation that can be determined from the early and often incomplete model, the better the resulting design and shorter the design cycle.

References

1. ASIC and EDA Magazine, January 1993, Reed Business Information, a division of Reed Elsevier Inc., New York, 1993.
2. D. Hill and W. vanCleemput, SABLE: A tool for generating structured, multi-level simulations, *Proceedings of the 16th ACM/IEEE Design Automation Conference*, San Diego, CA, June 25–27, 1979, pp. 272–279.
3. D.W. Franke and M.K. Purvis, Hardware/software codesign: A perspective, *Proceedings of the 13th International Conference on Software Engineering*, Austin, TX, May 13–17, 1991, pp. 344–352.
4. D.W. Franke and M.K. Purvis, Design automation technology for codesign: Status and directions, *International Symposium on Circuits* and *Systems*, San Diego, CA, 1992, pp. 2669–2671.
5. G.A. Frank, D.L. Franke, and W.F. Ingogly, An architecture design and assessment system, *VLSI Design*, 6(8): 30–50, August 1985.
6. C. Terry, Concurrent hardware and software design benefits embedded systems, *EDN*, 148–154, July 1990.
7. S. Napper, Embedded-system design plays catch-up, *IEEE Computer*, 31(8): 118–120, August 1998.
8. F.W. Zurcher and B. Randell, Iterative multi-level modeling—A methodology for computer system design, *Proceedings of IFIP Congress '68*, Edinburgh, UK, August 5–10, 1968, pp. 867–871.
9. Martin Marietta Laboratories, RASSP First Annual Interim Technical Report (CDRL A002), Moorestown, NJ, October 31, 1994.
10. Open SystemC Initiative Website, www.systemC.org.
11. P.A. Wilsey and S. Dasgupta, A formal model of computer architectures for digital system design environments, *IEEE Transaction on Computer-Aided Design*, 9(5): 473–486, May 1990.
12. C.A. Giumale and H.J. Kahn, Information models of VHDL, *Proceedings of the 32nd Design Automation Conference*, San Francisco, CA, 1995, pp. 678–683.

13. P. Kollaritsch, S. Lusky, D. Matzke, D. Smith, and P. Stanford, A unified design representation can work, *Proceedings of the 26th Design Automation Conference*, ACM Press, Las Vegas, NV, June 25–29, 1989, pp. 811–813.
14. J. Peterson, Petri nets, *Computing Surveys*, 9(3): 223–252, September 1997.
15. M.K. Molloy, Performance analysis using stochastic petri nets, *IEEE Transactions on Computers*, C-31(9): 913–917, September 1982.
16. M.A. Holliday and M.K. Vernon, A generalized timed petri net for performance analysis, *IEEE Transactions on Software Engineering*, SE-13(12): 1297–1310, December 1987.
17. L. Kleinrock, *Queuing Systems*, Vol. 1: *Theory*, Wiley, New York, 1975.
18. G.S. Graham, Queuing network models of computer system performance, *Computing Surveys*, 10(3): 219–224, September 1978.
19. G. Balbo, S.C. Bruell, and S. Ghanta, Combining queuing networks and generalized stochastic petri nets for solutions of complex models of system behavior, *IEEE Transactions on Computers*, 37(10): 1251–1268, October 1988.
20. G. Frank, Software/hardware codesign of real-time systems with ADAS, *Electronic Engineering*, 95–102, March 1990.
21. *SES/Workbench User's Manual*, Release 2.0, Scientific and Engineering Software Inc., Austin, TX, January 1991.
22. L. Maliniak, ESDA boosts productivity for high-level design, *Electronic Design*, 41: 125–128, May 27, 1993.
23. Integrated Design Automation System (IDAS), JRS Research Laboratories Inc., Orange, CA, 1988.
24. Application Note for TRANSCEND/VANTAGE Optium Cosimulation, TD Technologies, pp. 1–33, 1993.
25. Application Note for TRANSCEND Structural Ethernet Simulation, TD Technologies, August 1993, pp. 1–13.
26. R. Bargodia and C. Shen, MIDAS: Integrated design and simulation of distributed systems, *IEEE Transactions on Software Engineering*, 17(10): 1042–1058, October 1991.
27. E. Lee et al., Mini Almagest, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, April 1994.
28. E.A. Lee and D.G. Messerschmitt, Synchronous data flow, *Proceedings of the IEEE*, 75(9): 1235–1245, September 1987.
29. VHDL Hybrid Models Requirements, Honeywell Technology Center, Version 1.0, December 27, 1994.
30. Honeywell Technology Center, *VHDL Performance Modeling Interoperability Guideline*, Version Draft, August 1994.
31. F. Rose, T. Steeves, and T. Carpenter, VHDL performance models, *Proceedings of the 1st Annual RASSP Conference*, Arlington, VA, August 1994, pp. 60–70.
32. J.A. Rowsen and A. Sangiovanni-Vincentelli, Interface-based design, *Proceedings of the 34th ACM/IEEE Design Automation Conference (DAC-97)*, ACM Press, Anaheim Convention Center, Anaheim, CA, June 9–13, 1997, pp. 178–183.
33. L. Cai and D. Gajski, Transaction level modeling: An overview, *Proceedings of CODES + ISSS'03*, 2003, pp. 19–24.
34. IEEE, *IEEE Standard VHDL Language Reference Manual*, IEEE Std. 1076–1993, New York, NY, June 6, 1994.
35. A.P. Voss, R.H. Klenke, and J.H. Aylor, The analysis of modeling styles for system level VHDL simulations, *Proceedings of the VHDL International Users Forum*, IEEE Computer Society, Washington, DC, Fall 1995, pp. 1.7–1.13.
36. J.H. Aylor, R. Waxman, B.W. Johnson, and R.D. Williams, The integration of performance and functional modeling in VHDL, in *Performance and Fault Modeling with VHDL*, J.M. Schoen (Ed.), Prentice-Hall, Englewood Cliffs, NJ, 1992, pp. 22–145.

37. K. Jensen, Colored petri nets: A high level language for system design and analysis, in *High-Level Petri Nets: Theory and Application*, K. Jensen and G. Rozenberg (Eds.), Springer, Berlin, Germany, 1991, pp. 44–119.
38. F.T. Hady, A methodology for the uninterpreted modeling of digital systems in VHDL, MSc thesis, Department of Electrical Engineering, University of Virginia, Charlottesville, VA, January 1989.
39. A.P. Voss, Analysis and enhancements of the ADEPT environment, MSc thesis, Department of Electrical Engineering, University of Virginia, May 1996.
40. J.B. Dennis, Modular, asynchronous control structures for a high performance processor, *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, SECTION: Asynchronous Modular Systems*, Association for Computing Machinery, ACM Press, New York, 1970, pp. 55–80.
41. G. Swaminathan, R. Rao, J. Aylor, and B. Johnson, Colored petri net descriptions for the UVa primitive modules, CSIS Technical Report No. 920922.0, University of Virginia, Charlottesville, VA, September 1992.
42. ADEPT Library Reference Manual, CSIS Technical Report No. 960625.0, University of Virginia, Charlottesville, VA, June 6, 1996.
43. M. Meyassed, R. McGraw, J. Aylor, R. Klenke, R. Williams, F. Rose, and J. Shackleton, A framework for the development of hybrid models, *Proceedings of the 2nd Annual RASSP Conference*, Arlington, VA, July 1995, pp. 147–154.
44. R.A. MacDonald, R. Williams, and J. Aylor, An approach to unified performance and functional modeling of complex systems, *IASTED Conference on Modeling and Simulation*, Pittsburg, PA, April 1995.
45. D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic, Dordrecht, the Netherlands, 1992.
46. E.W. Dijkstra, A note on two problems in connection with graphs, *Numerische Mathematik*, 1: 269–271, 1959.
47. R. Floyd, Algorithm 97 (shortest path), *Communications of the ACM*, 5(6): 345, 1962.
48. S. Warshall, A theorem on Boolean matrices, *Journal of the ACM*, 9(1): 11–12, 1962.
49. M. Meyassed, System-level design: Hybrid modeling with sequential interpreted elements, PhD dissertation, Department of Electrical Engineering, University of Virginia, Charlottesville, VA, January 1997.
50. J.J. Hein, J.H. Aylor, and R.H. Klenke, A performance-based design tool for hardware/software systems, *Proceedings of the IEEE Workshop on Rapid System Prototyping*, San Diego, CA, June 2003.
51. S.T. Mitchum and R.H. Klenke, Design and fabrication of a digitally synthesized, digitally controlled, ring oscillator, *Proceedings of the IASTED Circuits Signals and Systems Conference*, Marina Del Rey, CA, October 24–26, 2005.

13

Embedded Computing Systems and Hardware/ Software Codesign

13.1 Introduction 13-1
13.2 Uses of Microprocessors 13-1
13.3 Embedded System Architectures 13-3
13.4 Hardware/Software Codesign 13-6
Models • Cosimulation • Performance Analysis • Hardware/Software Cosynthesis • Design Methodologies

Wayne Wolf
Princeton University

13.1 Introduction

This chapter describes embedded computing systems that make use of microprocessors to implement part of the system's function. It also describes hardware/software codesign, which is the process of designing embedded systems while simultaneously considering the design of its hardware and software elements.

13.2 Uses of Microprocessors

An embedded computing system (or more simply an embedded system) is any system which uses a programmable processor but itself is not a general-purpose computer. Thus, a personal computer is not an embedded computing system (though PCs are often used as platforms for building embedded systems), but a telephone or automobile which includes a CPU is an embedded system. Embedded systems may offer some amount of user programmability—3Com's PalmPilot, for example, allows users to write and download programs even though it is not a general-purpose computer—but embedded systems generally run limited sets of programs. The fact that we know the software that we will run on the hardware allows us to optimize both the software and hardware in ways that are not possible in general-purpose computing systems.

Microprocessors are generally categorized by their word size, since word size is associated both with maximum program size and data resolution. Commercial microprocessors come in many sizes; the term microcontroller is used to denote a microprocessor which comes with some basic on-chip peripheral devices, such as serial input/output (I/O) ports. Four-bit microcontrollers are extremely simple but capable of some basic functions. Eight-bit microcontrollers are workhorse low-end microprocessors.

Sixteen- and 32-bit microprocessors provide significantly more functionality. A 16/32-bit microprocessor may be in the same architectural family as the CPUs used in computer workstations, but microprocessors destined for embedded computing often do not provide memory management hardware. A digital signal processor (DSP) is a microprocessor tuned for signal processing applications. DSPs are often Harvard architectures, meaning that they provide separate data and program memories; Harvard architectures provide higher performance for DSP applications. DSPs may provide integer or floating-point arithmetic.

Microprocessors are used in an incredible variety of products. Furthermore, many products contain multiple microprocessors. Four- and eight-bit microprocessors are often used in appliances: for example, a thermostat may use a microcontroller to provide timed control of room temperature. Automatic cameras often use several eight-bit microprocessors, each responsible for a different aspect of the camera's functionality: exposure, shutter control, etc. High-end microprocessors are used in laser and ink-jet printers to control the rendering of the page. Many printers use two or three microprocessors to handle generation of pixels, control of the print engine, and so forth. Modern automobiles may use close to 100 microprocessors, and even inexpensive automobiles generally contain several. High-end microprocessors are used to control the engine's ignition system—automobiles use sophisticated control algorithms to simultaneously achieve low emissions, high fuel economy, and good performance. Low-end microcontrollers are used in a number of places in the automobile to increase functionality: for example, four-bit microcontrollers are often used to sense whether seat belts are fastened and turn on the seat belt light when necessary.

Microprocessors may replace analog components to provide similar functions, or they may add totally new functionality to a system. They are used in several different ways in embedded systems. One broad application category is signal conditioning, in which the microprocessor or DSP performs some filtering or control function on a digitized input. The conditioned signal may be sent to some other microprocessor for final use. Signal conditioning allows systems to use less-expensive sensors with the application of a relatively inexpensive microprocessor. Beyond signal conditioning, microprocessors may be used for more sophisticated control applications. For example, microprocessors are often used in telephone systems to control signaling functions, such as determining what action to take based on the reception of dial tones, etc. Microprocessors may implement user interfaces; this requires sensing when buttons, knobs, etc. are used, taking appropriate actions, and updating displays. Finally, microprocessors may perform data processing, such as managing the calendar in a personal digital assistant.

There are several reasons why microprocessors make good design components in such a wide variety of application areas. First, digital systems often provide more complex functionality than can be created using analog components. A good example is the user interface of a home audio/video system, which provides more information and is easier use than older, non-microprocessor-controlled systems. Microprocessors also allow related products much more cost-effectively. An entire product family, including models at various price and feature points, can be built around a single microprocessor-based platform. The platform includes both hardware components common to all the family members and software running on the microprocessor to provide functionality. Software elements can easily be turned on or off in various family members. Economies of scale often mean that it is cheaper to put the same hardware in both expensive and cheap models and to turn off features in the inexpensive models rather than to try to optimize the hardware and software configurations of each model separately. Microprocessors also allow design changes to be made much more quickly. Many changes may be possible simply by reprogramming; other features may be made possible by adding memory or other simple hardware changes along with some additional programming. Finally, microprocessors aid in concurrent engineering. After some initial design decisions have been made, hardware and software can be designed in parallel, reducing total design time.

While embedded computing systems traditionally have been fabricated at the board level out of multiple chips, embedded computing systems will play an increasing role in integrated circuit design as well. As VLSI technology moves toward the ability to fabricate chips with billions of transistors, integrated circuits will increasingly incorporate one or several microprocessors executing embedded

software. Using microprocessors as components in integrated circuits increases design productivity, since CPUs can be used as large components which implement a significant part of the system functionality. Single-chip embedded systems can provide much higher performance than board-level equivalents, since chip-to-chip delays are eliminated.

13.3 Embedded System Architectures

Although embedded computing spans a wide range of application areas, from automotive to medical, there are some common principles of design for embedded systems. The application-specific embedded software runs on a hardware platform. An example hardware platform is shown in Figure 13.1. It contains a microprocessor, memory, and I/O devices. When designing on a general-purpose system such as a PC, the hardware platform would be predetermined, but in hardware/software codesign the software and hardware can be designed together to better meet cost and performance requirements.

Depending on the application, various combinations of criteria may be important goals for the system design. Two typical criteria are speed and manufacturing cost. The speed at which computations are made often contributes to the general usability of the system, just as in general-purpose computing. However, performance is also often associated with the satisfaction of deadlines—times at which computations must be completed to ensure the proper operation of the system. If failure to meet a deadline causes a major error, it is termed a hard deadline. And missed deadlines, which result in tolerable but unsatisfactory degradations, are called soft deadlines. Hard deadlines are often (though not always) associated with safety-critical systems. Designing for deadlines is one of the most challenging tasks in embedded system design. Manufacturing cost is often an important criteria for embedded systems. Although the hardware components ultimately determine manufacturing cost, software plays an important role as well. First, the size of the program determines the amount of memory required, and memory is often a significant component of the total component cost. Furthermore, the improper design of software can cause one to require higher-performance, more-expensive hardware components than are really necessary. Efficient utilization of hardware resources requires careful software design. Power consumption is becoming an increasingly important design metric. Power is certainly important in battery-operated devices, but it can be important in wall socket-powered systems as well—lower power consumption means smaller, less-expensive power supplies and cooling and may result in environmental ratings that are advantageous in the marketplace. Once again, power consumption is ultimately determined by the hardware, but software plays a significant role in power characteristics. For example, more efficient use of on-chip caches can reduce the need for off-chip memory access, which consumes much more power than on-chip cache references.

Figure 13.1 shows the hardware architecture of a basic microprocessor system. The system includes the CPU, memory, and some I/O devices, all connected by a bus. This system may consist of multiple chips for high-end microprocessors or a single-chip microcontroller. Typical I/O devices include analog/digital (ADC) and digital/analog (DAC) converters, serial and parallel communication devices, network and bus interfaces, buttons and switches, and various types of display devices. This configuration is a complete, basic, embedded computing hardware platform on which application software can execute.

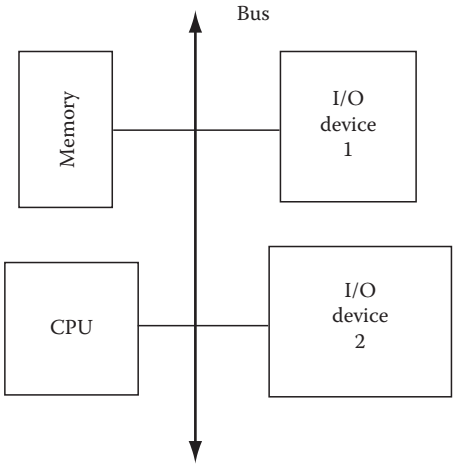


FIGURE 13.1 Hardware structure of a microprocessor system.

The embedded application software includes components for managing I/O devices and for performing the core computational tasks. The basic software techniques for communicating with I/O devices are polling and interrupt-driven. In a polled system, the program checks each device's status register to determine if it is ready to perform I/O. Polling allows the CPU to determine the order in which I/O operations are completed, which may be important for ensuring that certain device requests are satisfied at the proper rate. However, polling also means that a device may not be serviced in time if the CPU's program does not check it frequently enough. Interrupt-driven I/O allows a device to change the flow of control on the CPU and call a device driver to handle the pending I/O operation. An interrupt system may provide both prioritized interrupts to allow some devices to take precedence over others and vectored interrupts to allow devices to specify which driver should handle their request.

Device drivers, whether polled or interrupt-driven, will typically perform basic device-specific functions and hand-off data to the core routines for processing. Those routines may perform relatively simple tasks, such as transducing data from one device to another, or may perform more sophisticated algorithms such as control. Those core routines often will initiate output operations based on their computations on the input operations.

Input and output may occur either periodically or aperiodically. Sampled data is a common example of periodic I/O, while user interfaces provide a common source of aperiodic I/O events. The nature of the I/O transactions affects both the device drivers and the core computational code. Code which operates on periodic data is generally driven by a timer which initiates the code at the start of the period. Periodic operations are often characterized by their periods and the deadline for each period. Aperiodic I/O may be detected either by an interrupt or by polling the devices. Aperiodic operations may have deadlines, which are generally measured from the initiating I/O event. Periodic operations can often be thought of as being executed within an infinite loop. Aperiodic operations tend to use more event-driven code, in which various sections of the program are exercised by different aperiodic events, since there is often more than one aperiodic event which can occur.

Embedded computing systems exhibit a great deal of parallelism which can be used to speed up computation. As a result, they often use multiple microprocessors which communicate with each other to perform the required function. In addition to microprocessors, application-specific ICs (ASICs) may be added to accelerate certain critical functions. CPUs and ASICs in general are called processing elements (PEs). An example multiprocessor system built from several PEs along with I/O devices and memory is shown in Figure 13.2.

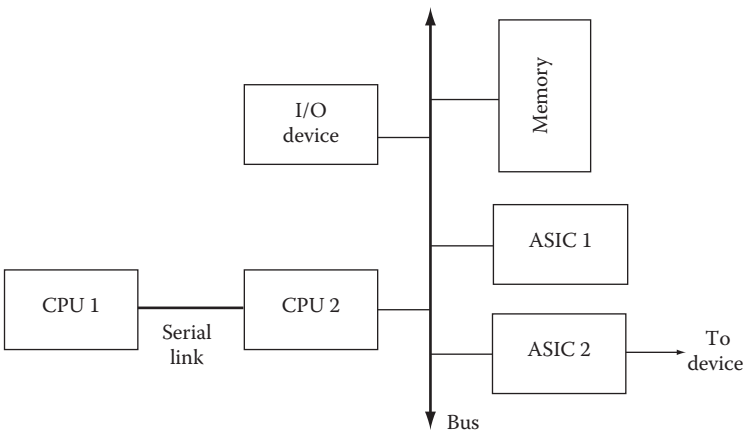


FIGURE 13.2 Heterogeneous embedded multiprocessor.

The choice of several small microprocessors or ASICs rather than one large CPU is primarily determined by cost. Microprocessor cost is a nonlinear function of performance, even within a microprocessor family. Vendors generally supply several versions of a microprocessor which run at different clock rates; chips which run at varying speeds are a natural consequence of the variations in the VLSI manufacturing process. The slowest microprocessors are significantly less expensive than the fastest ones, and the cost increment is larger at the high end of the speed range than at the low end. As a result, it is often cheaper to use several smaller microprocessors to implement a function.

When several microprocessors work together in a system, they may communicate with each other in several different ways. If slow data rates are sufficient, serial data links are commonly used for their low hardware cost. The I^2C bus is a well-known example of a serial bus used to build multi-microprocessor embedded systems; the CAN bus is widely used in automobiles. High-speed serial links can achieve moderately high performance and are often used to link multiple DSPs in high-speed signal processing systems. Parallel data links provide the highest performance thanks to their sheer data width. High-speed busses such as PCI can be used to link several processors.

The software for an embedded multiprocessing system is often built around processes. A process, as in a general-purpose computing system, is an instantiation of a program with its own state. Since problems complex enough to require multiprocessors often run sophisticated algorithms and I/O systems, dividing the system into processes helps manage design complexity. A real-time operating system (RTOS) is an operating system specifically designed for embedded, and specifically real-time applications. The RTOS manages the processes and device drivers in the system, determining when each executes on the CPU. This function is termed scheduling. The partitioning of the software between application code which executes core algorithms and an RTOS which schedules the times to which those core algorithms are executed is a fundamental design principle in computing systems in general and is especially important for real-time operation.

There are a number of techniques which can be used to schedule processes in an embedded system—that is, to determine which process runs next on a particular CPU. Most RTOSs use process priorities in some form to determine the schedule. A process may be in any one of three states: currently executing (there can obviously be only one executing process on each CPU), ready to execute, or waiting. A process may not be able to execute until, for example, its data has arrived. Once its data arrives, it moves from waiting to ready. The scheduler chooses among the ready processes to determine which process runs next. In general, the RTOS's scheduler chooses the highest-priority ready process to run next; variations between scheduling methods depend in large part on the ways in which priorities are determined. Unlike general-purpose operating systems, RTOSs generally allow a process to run until it is preempted by a higher-priority process. General-purpose operating systems often perform time-slicing operations to maintain fair access of all the users on the system, but time-slicing does not allow the control required for meeting deadlines.

A fundamental result in real-time scheduling is known as rate-monotonic scheduling. This technique schedules a set of processes which run independently on a single CPU. Each process has its own period, with the deadline happening at the end of each period. There can be arbitrary relationships between the periods of the processes. It is assumed that data does not in general arrive at the beginning of the period, so there are no assumptions about when a process goes from waiting to ready within a period. This scheduling policy uses static priorities—the priorities for the processes are assigned before execution begins and do not change. It can be shown that the optimal priority assignment is based on period—the shorter the period, the higher the priority. This priority assignment ensures that all processes will meet their deadlines on every period. It can also be shown that at most, 69% of the CPU is used by this scheduling policy. The remaining cycles are spent waiting for activities to happen—since data arrival times are not known, it is not possible to utilize 100% of the CPU cycles.

Another well-known, real-time scheduling technique is earliest deadline first (EDF). This is a dynamic priority scheme—process priorities change during execution. EDF sets priorities based on the impending deadlines, with the process whose deadline is closest in the future having the highest priority.

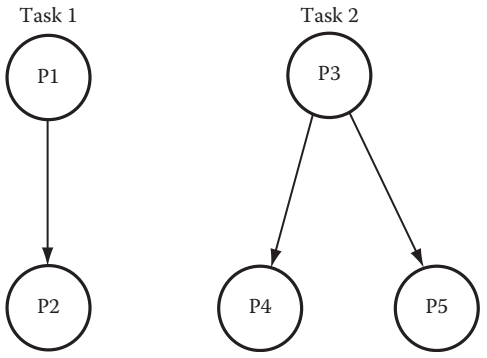


FIGURE 13.3 Task graph with two tasks and data dependencies between processes.

Clearly, the rate of change of process priorities depends on the periods and deadlines. EDF can be shown to be able to utilize 100% of the CPU, but it does not guarantee that all deadlines will be met. Since priorities are dynamic, it is not possible in general to analyze whether the system will be overloaded at some point.

Processes may be specified with data dependencies, as shown in Figure 13.3, to create a task graph. An arc in the data dependency graph specifies that one process feeds data to another. The sink process cannot become ready until all the source processes have delivered their data. Processes which have no data dependency path between them are in separate tasks. Each task can run at its own rate. Data dependencies allow schedulers to make more efficient use of CPU resources. Since the source and sink processes of a data dependency cannot execute simultaneously, we can use that information to eliminate some combinations of processes which may want to run at the same time. Narrowing the scope of process conflicts allows us to more accurately predict how the CPU will be used.

A real-time operating system is often designed to have a small memory footprint, since embedded systems are more cost-sensitive than general-purpose computers. RTOSs are also designed to be more responsive in two different ways. First, they allow greater control over the order of execution of processes, which is critical for ensuring that deadlines are met. Second, they are designed to have lower context-switching overhead, since that overhead eats into the time available for meeting deadlines. The kernel of an RTOS is the basic set of functions that is always resident in memory. A basic RTOS may have an extremely small kernel of only a few hundred instructions. Such microkernels often provide only basic context-switching and scheduling facilities. More complex RTOSs may provide high-end operating system functions such as file systems and network support; many high-end RTOSs are POSIX (a Unix standard) compliant. While running such a high-end operating system requires more hardware resources, the extra features are useful in a number of situations. For example, a controller for a machine on a manufacturing line may use a network interface to talk to other machines on the factory floor or the factory coordination unit; it may also use the file system to access a database for the manufacturing process.

13.4 Hardware/Software Codesign

Hardware/software codesign refers to any methodology which takes into account both hardware and software during the design of an embedded computing system. When the hardware and software are designed together, the designer has more opportunities to optimize the system by making trade-offs between the hardware and software components. Good system designers intuitively perform codesign, but codesign methods are increasingly being embodied in computer-aided design (CAD) tools. We will discuss several aspects of codesign and codesign tools, including models of the design, cosimulation, performance analysis, and various methods for architectural cosynthesis. We will conclude with a look at design methodologies that make use of these phases of codesign.

13.4.1 Models

In designing embedded computing systems, we make use of several different types of models at different points in the design process. We need to model basic functionality. We must also capture nonfunctional requirements: speed, weight, power consumption, manufacturing cost, etc.

In the earliest stages of design, the task graph is an important modeling tool. The task graph does not capture all aspects of functionality, but it does describe the various rates at which computations must be performed and the expected degrees of parallelism available. This level of detail is often enough to make some important architectural decisions. A useful adjunct to the task graph are the technology description tables, which describe how processes can be implemented on the available components. One of the technology description tables describes basic properties of the processing elements, such as cost and basic power dissipation. A separate table describes how the processes may be implemented on the components, giving execution time (and perhaps other function-specific parameters like precise power consumption) on a processing element of that type. The technology description is more complex when ASICs can be used as processing elements, since many different ASICs at differing price/performance points can be designed for a given functionality, but the basic data still applies.

A more detailed description is given by either high-level language code (C, etc.) for software or hardware description language code (VHDL, Verilog, etc.) for software components. These should not be viewed as specifications—they are, in fact, quite detailed implementations. However, they do provide a level of abstraction above assembly language and gates and so can be valuable for analyzing performance, size, etc. The control-data flow graph (CDFG) is a typical representation of a high-level language: a flowchart-like structure describes the program's control, while data flow graphs describe the behavior within expressions and basic blocks.

13.4.2 Cosimulation

Simulation is an important tool for design verification. The simulation of a complete embedded system entails modeling both the underlying hardware platform and the software executing on the CPUs. Some of the hardware must be simulated at a very fine level of detail—for example, busses and I/O devices may require gate-level simulation. On the other hand, the software can and should be executed at a higher level of abstraction. While it would be possible to simulate software execution by running a gate-level simulation of the CPU and modeling the program as residing in the memory of the simulated CPU, this would be unacceptably slow.

We can gain significant performance advantages by running different parts of the simulation at different levels of detail: elements of the hardware can be simulated in great detail, while software execution can be modeled much more directly. Basic functionality aspects of a high-level language program can be simulated by compiling the software on the computer on which the simulation executes, allowing those parts of the program to run at the native computer speed. Aspects of the program which deal with the hardware platform must interface to the section of the simulator which deals with the hardware. Those sections of the program are replaced by stubs which interface to the simulator. This style of simulation is a multirate simulation system, since the hardware and software simulation sections run at different rates: a single instruction in the software simulation will correspond to several clock cycles in the hardware simulation. The main jobs of the simulator are to keep the various sections of the simulation synchronized and to manage communication between the hardware and software components of the simulation.

13.4.3 Performance Analysis

Since performance is an important design goal in most embedded systems, both for overall throughput and for meeting deadlines, the analysis of the system to determine its speed of operation is an important element of any codesign methodology. System performance—the time it takes to execute a particular aspect of the system's functionality—clearly depends both on the software being executed and the underlying hardware platform. While simulation is an important tool for performance analysis, it is not sufficient, since simulation does not determine the worst-case delays. Since the execution times of most programs are data-dependent, it is necessary to give the simulation of the program the proper set

of inputs to observe worst-case delay. The number of possible input combinations makes it unlikely that one will find those worst-case inputs without the sort of analysis that is at the heart of performance analysis.

In general, performance analysis must be done at several different levels of abstraction. Given a single program, one can place an upper bound on the worst-case execution time of the program. However, since many embedded systems consist of multiple processes and device drivers, it is necessary to analyze how these programs interact with each other, a phase which makes use of the results of single-program performance analysis.

Determining the worst-case execution time of a single program can be broken into two subproblems: determining the longest execution path through the program and determining the execution time of that program. Since there is at least a rough correlation between the number of operations and the actual execution time, we can determine the longest execution path without detailed knowledge of the instructions being executed—the longest path depends primarily on the structure of conditionals and loops. One way to find the longest path through the program is to model the program as a control-flow graph and use network flow algorithms to solve the resulting system.

Once the longest path has been found, we need to look at the instructions executed along that path to determine the actual execution time. A simple model of the processor would assume that each instruction has a fixed execution time, independent of other factors such as the data values being operated on, surrounding instructions, or the path of execution. In fact, such simple models do not give adequate results for modern high-speed microprocessors. One problem is that in pipelined processors, the execution time of an instruction may depend on the sequence of instructions executed before it. An even greater cause of performance variations is caching, since the same instruction sequence can have variable execution times, depending on whether the code is in the cache. Since cache miss penalties are often 5X or 10X, the cost of mischaracterizing cache performance is significant. Assuming that the cache is never present gives a conservative estimate of worst-case execution time, but one that is so over-conservative that it distorts the entire design. Since the performance penalty for ignoring the cache is so large, it results in using a much faster, more expensive processor than is really necessary. The effects of caching can be taken into account during the path analysis of the program—path analysis can determine bound how often an instruction present in the cache.

There are two major effects which must be taken into account when analyzing multiple-process systems. The first is the effect of scheduling multiple processes and device drivers on a single CPU. This analysis is performed by a scheduling algorithm, which determines bounds on when programs can execute. Ratemonotonic analysis is the simplest form of scheduling analysis—the utilization factor given by ratemonotonic analysis tells one an upper limit on the amount of active CPU time. However, if data dependencies between processes are known, or some knowledge of the arrival times of data is known, then a more accurate performance estimate can be computed. If the system includes multiple processing elements, more sophisticated scheduling algorithms must be used, since the data arrival time for a process on one processing element may be determined by the time at which that datum is computed on another processing element.

The second effect which must be taken into account is interactions between processes in the cache. When several programs on a CPU share a cache, or when several processing elements share a second-level cache, the cache state depends on the behavior of all the programs. For example, when one process is suspended by the operating system and another process starts running, that process may knock the first program out of the cache. When the first process resumes execution, it will initially run more slowly, an effect which cannot be taken into account by analyzing the programs independently. This analysis clearly depends in part on the system schedule, since the interactions between processes depends on the order in which the processes execute. But the system scheduling analysis must also keep track of the cache state—which parts of which programs are in the cache at the start of execution of each process. Good accuracy can be obtained with a simple model which assumes that a program is either in the cache or out of it,

without considering individual instructions; higher accuracy comes from breaking a process into several subprocesses for analysis, each of which can have its own cache state.

13.4.4 Hardware/Software Cosynthesis

Hardware/software cosynthesis tries to simultaneously design the hardware and software for an embedded computing system, given design requirements such as performance as well as a description of the functionality. Cosynthesis generally concentrates on architectural design rather than detailed component design—it concentrates on determining such major factors as the number and types of processing elements required and the ways in which software processes interact.

The most basic style of cosynthesis is known as hardware/software partitioning. As shown in Figure 13.4, this algorithm maps the given functionality onto a template architecture consisting of a CPU and one or more ASICs communicating via the microprocessor bus. The functionality is usually specified as a single program. The partitioning algorithm breaks that program into pieces and allocates pieces either to the CPU or ASICs for execution. Hardware/software partitioning assumes that total system performance is dominated by a relatively small part of the application, so that implementing a small fraction of the application in the ASIC leads to large performance gains. Less performance-critical sections of the application are relegated to the CPU.

The first problem to be solved is how to break the application program into pieces; common techniques include determining where I/O operations occur and concentrating on the basic blocks of inner loops. Once the application code is partitioned, various allocations of those components must be evaluated. Given an allocation of program components to the CPU or ASICs, performance analysis techniques can be used to determine the total system performance; performance analysis should take into account the time required to transfer necessary data into the ASIC and to extract the results of the computation from the ASIC. Since the total number of allocations is large, heuristics must be used to search the design space. In addition, the cost of the implementation must be determined. Since the CPU's cost is known in advance, that cost is determined by the ASIC cost, which varies as to the amount of hardware required to implement the desired function. High-level synthesis can be used to estimate both the performance and hardware cost of an ASIC which will be synthesized from a portion of the application program.

Basic, cosynthesis heuristics start from extreme initial solutions: We can either put all program components into the CPU, creating an implementation which is minimal cost but probably does not meet performance requirements, or put all program elements in the ASIC, which gives a maximal-performance, maximal-expense implementation. Given this initial solution, heuristics select which program component to move to the other side of the partition to either reduce hardware cost or increase performance, as desired. More sophisticated heuristics try to construct a solution by estimating how critical a component will be to overall system performance and choosing a CPU or ASIC implementation accordingly. Iterative improvement strategies may move components across the partition boundary to improve the design.

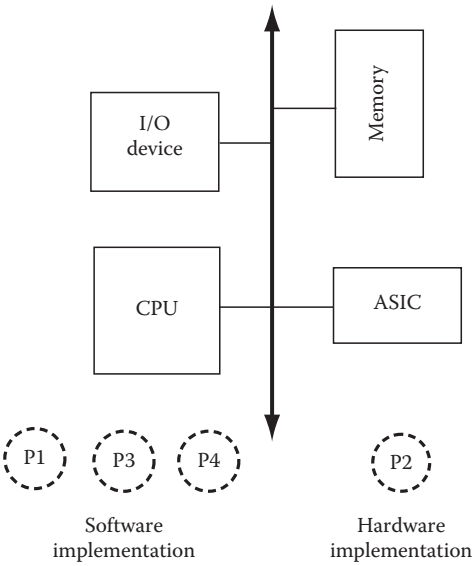


FIGURE 13.4 Hardware/software partitioning.

However, many embedded systems do not strictly follow the one CPU, one bus, *n* ASIC architectural template. These more general architectures are known as distributed embedded systems. Techniques for designing distributed embedded systems rest on the foundations of hardware/software partitioning, but they are generally more complicated, since there are more free variables. For example, since the number and types of CPUs is not known in advance, the cosynthesis algorithm must select them. If the number of busses or other communication links is not known in advance, those must be selected as well. Unfortunately, these decisions are all closely related. For example, the number of CPUs and ASICs required depends on the system schedule. The system schedule, in turn, depends on the execution time of each of the components on the available hardware elements. But those execution times depend on the processing elements available, which is what we are trying to determine in the first place. Cosynthesis algorithms generally try to fix several designs and vary only one or a few, then check the results of a design decision on the other parameters. For example, the algorithm may fix the hardware architecture and try to move processes to other processing elements to make more efficient use of the available hardware. Given that new configuration of processes, it may then try to reduce the cost of the hardware by eliminating unused processing elements or replacing a faster, more expensive processing element with a slower, cheaper one.

Since the memory hierarchy is a significant contributor to overall system performance, the design of the caching system is an important aspect of distributed system cosynthesis. In a board-level system with existing microprocessors, the sizes of second-level caches is under designer control, even if the first-level cache is incorporated on the microprocessor and therefore fixed in size. In a single-chip embedded system, the designer has control over the sizes of all the caches. Cosynthesis can determine hardware elements such as the placement of caches in the hardware architecture and the size of each cache. It can also determine software attributes such as the placement of each program in the cache. The placement of a program in the cache is determined by the addresses used by the program—by relocating the program, the cache behavior of the program can be changed. Memory system design requires calculating the cache state when constructing the system schedule and using the cache state as one of the factors to determine how to modify the design.

13.4.5 Design Methodologies

A codesign methodology tries to take into account aspects of hardware and software during all phases of design. At some point in the design process, the hardware and software components are well-specified and can be designed relatively independently. But it is important to consider the characteristics of both the hardware and software components early in design. It is also important to properly test the system once the hardware and software components are assembled into a complete system.

Cosynthesis can be used as a design planning tool, even if it is not used to generate a complete system architectural design. Because cosynthesis can evaluate a large number of designs very quickly, it can determine the feasibility of a proposed system much faster than a human designer. This allows the designer to experiment with what-if scenarios, such as adding new features or speculating on the effects of lower component costs in the future. Many cosynthesis algorithms can be applied without having a complete program to use as a specification. If the system can be specified to the level of processes with some estimate of the computation time required for each process, then useful information about architectural feasibility can be generated by cosynthesis.

Cosimulation plays a major role once subsystem designs are available. It does not have to wait until all components are complete, since stubs may be created to provide minimal functionality for incomplete components. The ability to simulate the software before completing the hardware is a major boon to software development and can substantially reduce development time.

14

Design Automation Technology Roadmap

14.1 Introduction 14-1
14.2 Design Automation: Historical Perspective 14-3
The 1960s: The Beginnings of Design Automation • The 1970s: The Awaking of Verification • The 1980s: Birth of the Industry • The 1990s: The Age of Integration
14.3 The Future 14-27
International Technology Roadmap for Semiconductors • EDA Impact
14.4 Summary 14-38
References 14-38

Donald R. Cottrell
Silicon Integration Initiative, Inc.

14.1 Introduction

No invention in the modern age has been as pervasive as the semiconductor and nothing has been more important to its technological advancement than has electronic design automation (EDA). EDA began in the 1960s both for the design of electronic computers and because of them. It was the advent of the computer that made possible the development of specialized programs that perform the complex management, design, and analysis operations associated with electronics and electronic systems. At the same time, it was the design, management, and manufacture of the thousands (now tens of millions) of devices that make up a single electronic assembly that made EDA an absolute requirement to fuel the semiconductor progression. Today, EDA programs are used in electronic packages for all business markets from computers to games, telephones to aerospace guidance systems, and toasters to automobiles. Across these markets, EDA supports many different package types such as integrated circuit (IC) chips, multichip modules (MCMs), printed circuit boards (PCBs), and entire electronic system assemblies.

No electronic circuit package is as challenging to EDA as the IC. The growth in complexity of ICs has placed tremendous demands on EDA. Mainstream EDA applications such as simulation, layout, and test generation have had to improve their speed and capacity characteristics with this ever-increasing growth in the number of circuits to be processed. New types of design and analysis applications, new methodologies, and new design rules have been necessary to keep pace. Yet, even with the technological breakthroughs that have been made in EDA across the past four decades, it is still having difficulty keeping up with the breakthroughs being made in the semiconductor technology progression that it fuels. Decrease in size and spacing of features on the chip is causing the number of design elements per chip to increase at a tremendous rate. The decrease in feature size and spacing coupled with the increase in operating frequency is causing additional levels of complexity to be approximated in the models used by design and analysis programs.

In the period from 1970 to the present semiconductor advances such as the following have had a great impact on EDA technology (Figure 14.1):

- IC integration has grown from tens of transistors on a chip, to beyond tens of millions.
- Feature size on production ICs has shrunk from 10 μm to 90 nm and smaller.
- On-chip clock frequency has increased from a few megahertz to many gigahertz.

Playing an essential part in the advancement of EDA have been advances in computer architectures that run the EDA applications. These advances have included the following:

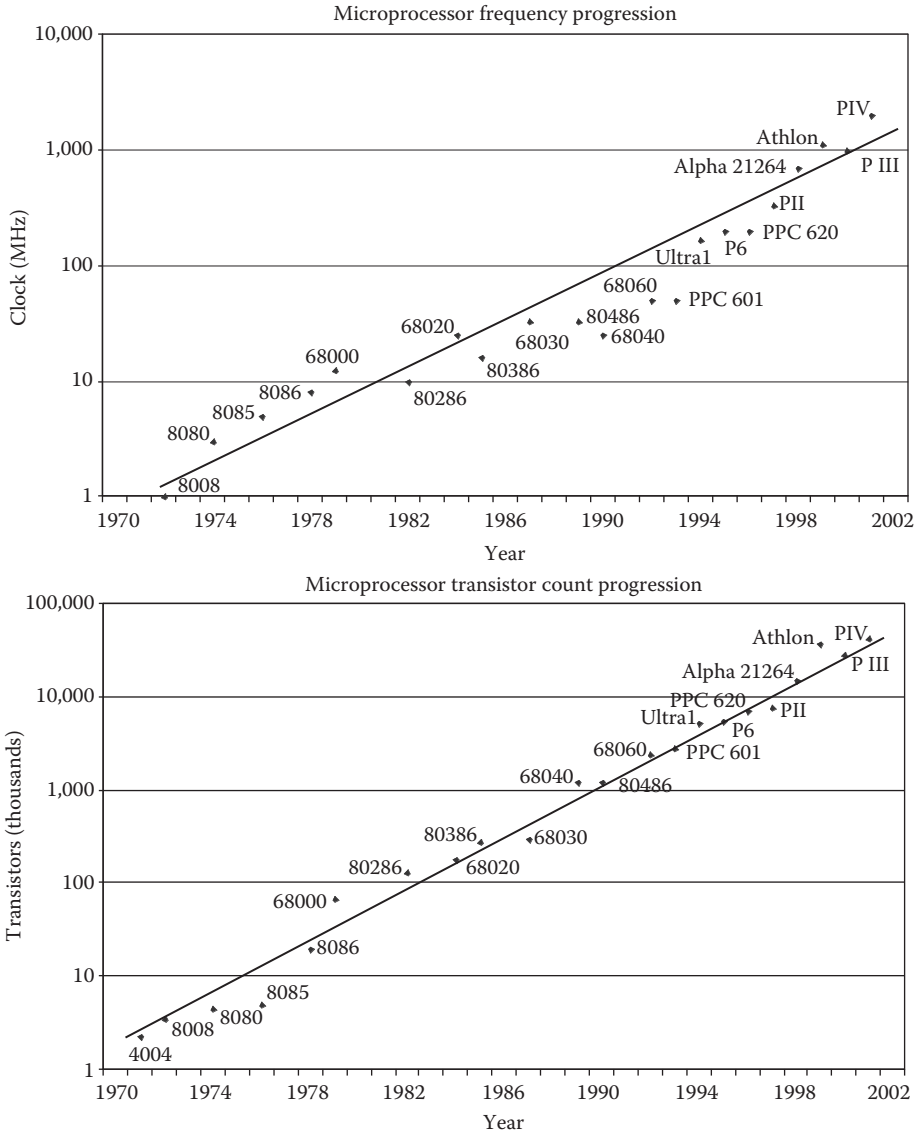


FIGURE 14.1 Microprocessor development.

- Computer CPU speed: from less than a million instructions per second (MIPS) of shared mainframe to hundreds of MIPS on a dedicated workstation
- Computer memory: from <32 KB to >500 GB
- Data archive: from voluminous reels of (rather) slow-speed tape to virtually limitless amounts of high-speed electronic storage

However, these major improvements in computing power alone would not have been sufficient to meet the EDA needs of semiconductor advancement. Major advances have also been made to fundamental EDA algorithms, and entirely new design techniques and design paradigms have been invented and developed to support semiconductor advancement. This chapter will trace the more notable advancements made in EDA across its history for electronics design and discuss important semiconductor technology trends predicted across the next decade along and the impact they will have on EDA for the future. It is important to understand these trends and projections, because if the EDA systems cannot keep pace with the semiconductor projections, then these projections cannot be realized. Although it may be possible to build foundries that can manufacture ultradeep submicron wafers and even to acquire the billions of dollars of capital required for each, without the necessary EDA support these factories will never be fully utilized. SEMATECH reports that chip design productivity has increased at a compounded rate somewhere between 21% and 30%, while Moore's Law predicts the number of transistors on a chip to increase at a compound rate of 56%. This means that at some time, bringing online new foundries that can produce smaller, denser chips may reach a point of diminishing returns, because the ability to design and yield chips with that many transistors may not be possible.

14.2 Design Automation: Historical Perspective

14.2.1 The 1960s: The Beginnings of Design Automation

Early entries into design automation were made in the areas of design (description) records, PCB wiring, and manufacturing test generation. A commercial EDA industry did not exist and developments were made within companies with the need, such as IBM [1] and Bell Labs, on mainframe computers such as the IBM 7090. The 7090 had addressable 36-bit words and a limit of 32,000 words of main storage (magnetic cores). This was far less than the typical 256+ MB of RAM on the average notebook PC, and certainly no match for a high-function workstation with gigabytes of main store. Though computer limitations continue to be an on-going challenge for EDA development, the limitation of the computers of the 1960s was particularly acute.

In retrospect, the limitation of computers in the 1960s was a blessing for the development of design automation. Because of these limitations, design automation developers were forced to invent highly creative algorithms that operated on very compact data structures. Many of the fundamental concepts developed during this period are still in use within commercial EDA systems today. Some notable advances during this period were

- Fundamental ''stuck-at'' model for manufacturing test and a formal algebra for the generation of tests and diagnosis of faults
- Parallel fault simulation, which provided simulation of many fault conditions in parallel with the good-machine (nonfaulty circuit) to reduce fault-simulation run times
- Three-valued algebra for simulation which yields accurate results using simple delay models, even in the presence of race conditions within the design.
- Development of fundamental algorithms for the placement and wiring of components
- Checking of designs against prescribed electrical design requirements (rules)

Also, there was development of fundamental heuristics for placement and wire routing, and for divide-and-conquer concepts supporting both. One such concept was the hierarchical division of a wiring image

into cells, globally routing between cells, and then performing detailed routing within cells, possibly subdividing them further. Many of these fundamental concepts are still applied today, although the complexities of physical design (PD) of today's large-scale integration (LSI) are vastly more complex.

The 1960s represented the awakening of design automation and provided the proof of its value and need for electronics design. It would not be until the end of this decade when the explosion of the number of circuits designed on a chip would occur and the term LSI would be coined. EDA development in the 1960s was primarily focused on printed circuit assemblies, but the fundamental concepts developed for design entry, test generation, and PD provided the basics for EDA in the LSI era.

14.2.1.1 Design Entry

Before the use of computers in electronics design, the design schematic was a hard-copy drawing. This drawing was a draftsman's rendering of the notes and sketches provided by the circuit designer. The drawings provided the basis for manufacturing and repair operations in the field. As automation developed, it became desirable to store these drawings on storage media usable by computers so that the creation of input to the automated processes could, itself, be automated. So, the need to record the design of electronic products and assemblies in computers was recognized in the late 1950s and early 1960s. In the early days of design automation, the electronics designer would develop the design using paper and pencil and then transcribe it to a form suitable for keyed entry to a computer. Once keyed into the computer, the design could be rendered in a number of different formats to support the manufacturing and field operations. It was soon recognized that these computerized representations of the circuit design drawing could also drive design processes such as the routing of printed circuit traces or the generation of manufacturing test patterns. Finally, from there, it was but a short step to the use of computers to generate data in the form required to drive automated manufacturing and test equipment.

Early design entry methods involved the keying of the design description onto punch cards that were read into the computer and saved on a persistent storage device. This became known as the design's database, and is the start of the design automation system. From the database, schematic diagrams and logic diagrams were rendered for use in engineering, manufacturing, and field support. This was typically a two-step process, whereby the designer drew the schematic by hand and then submitted it to another for conversion to the transcription records, keypunch, and entry to the computer. Once in the computer, the formal automated drawings were generated, printed, and returned to the designer. Although this process seems archaic by today's standards, it did result in a permanent record of the design in computer readable format. This could be used for many forms of records management, engineering change history, and as input to design, analysis, and manufacturing automation that would soon follow.

With the introduction of the alphanumeric terminal, the keypunch was replaced as the window into the computer. With this, new design description languages were developed, and the role of the transcription operator began to move back to the designer. Although these description languages still represented the design at the device or gate level, they were free format and keyword oriented and design engineers were willing to use them. The design engineer now had the tools to enter design descriptions directly into the computer thus eliminating the inherent inefficiencies of the ''middleman.'' Thus, a paradigm shift began to evolve in the method by which design was entered to the EDA system. Introduction of the direct access storage devices (disks) in the late 1960s also improved the entry process as well as the entire design system by providing online, high-speed direct access to the entire design or any portion of it. This was also important to the acceptance of design entry by the designer as the task was still viewed as a necessary overhead rather than a natural part of the design task. It was necessary to get access to the other evolving design automation tools, but typically, the real design thought process took place with pencil and paper techniques. Therefore, any change that made the entry process faster and easier was eagerly accepted.

The next shift occurred in the later part of the 1970s with the introduction of graphics terminals. With these, the designer could enter design into the database in schematic form. This form of design entry was a novelty at first, but not a clear performance improvement. In fact, until the introduction of the

workstation with dedicated graphics support, graphic entry was detrimental to design productivity in many cases. Negative effects such as less than effective transaction speed, time lost in making the schematic esthetically pleasing and the low level of detail all added to less than obvious advances. In contrast, use of the graphics display to view the design and make design changes proved extremely effective and was a great improvement over the red-lined hard-copy prints. For this reason, use of computer graphics represented a major advance and this style of design entry took off with the introduction of the workstation in the 1980s. In fact, the graphic editor's glitz and capability was often a major selling point in the decision to use one EDA system over another. Further, to be considered a commercially viable system, graphics entry was required. Nevertheless, as the density of ICs grew, graphics entry of schematics would begin to yield to the productivity advantages of (alphanumeric) description languages. As EDA design and analysis tool technology advanced, entry of design at the register-transfer level (RTL) would become commonplace and today the design engineer is able to represent his design ideas at many levels of abstraction and throughout the design process. System-level and RTL design languages have been introduced and the designer is able to verify design intent much earlier in the design cycle.

By the 1990s and after the introduction of synthesis automation, design entry using RTL descriptions was the generally accepted approach for entry of design, although schematic entry remained the accepted method for PCB design and many elements of custom ICs. There is no doubt that the introduction of graphics into the design automation system represents a major advance and a major paradigm shift. The use of graphics to visualize design details, wiring congestion, and timing diagrams is of major importance. The use of graphics to perform edit functions is standard operating procedure.

Large system design, often, entails control circuitry, dataflow, and functional modules. Classically, these systems span across several chips and boards and employ several styles of entry for the different physical packages. These may include

- Schematics—graphic
- RTL and behavioral level languages—alphanumeric
- Timing diagrams—graphic
- State diagrams—alphanumeric
- Flowcharts—graphic

Today, these entry techniques can be found in different EDA tools and each is particularly effective for different types of design problems. Schematics are effective for the design of ''glue'' logic that interconnects functional design elements such as modules on a PCB and for custom IC circuitry. Behavioral languages are useful for system-level design, and particularly effective for dataflow behavior. Timing diagrams lend themselves well to describe the functional operations of ''black-box'' components at their I/Os without needing to describe their internal circuitry. State diagrams are a convenient way to express the logical operation of combinational circuits. Flowcharts are effective for describing the operations of control logic, much like use of flowcharts for specification of software program flow. With technology advances, the IC is engulfing more and more of the entire system and all of these forms of design description may be prevalent on a single chip. It is even expected that the design of ''black-box'' functions will be available from multiple sources to be embedded onto the chip similar to the use of modules on a PCB. Thus, it is likely that future EDA systems will support a mixture of design description forms to allow the designer to represent sections of the design in a manner most effective to each. After all, design is described in many forms by the designer outside the design system.

14.2.1.2 Test Generation

Testing of manufactured electronic subassemblies entails the use of special test hardware that can provide stimulus (test signals) to selected (input) pins of the part under test and measure for specified responses on selected (output) pins. If the measured response matches the specified response, then the part under test has passed that test successfully. If some other response is measured, then the part has failed that test

and the presence of a defect is indicated. Manufacturing test is the successive application test patterns that cause some measurable point on the part under test to be sensitized to the presence of a manufacturing defect. That is, some measurable point on the part under test will result in a certain value if the fault is present and a different one if no fault were present. The collection of test patterns causes all (or almost all) possible manufacturing failures to render a different output response than would the nondefective part. For static DC testers, each stimulus is applied and after the part under test settles to a steady state, the specified outputs are measured and compared with the expected results for a nondefective part.

To bound the test generation problem, a model was developed to represent possible defects at the abstract gate level. This model characterizes the effects of defects as stuck-at values. This model is fundamental to most of the development in test generation and is still in use today. It characterizes defects as causing either a stuck-at-one or a stuck-at-zero condition at pins on a logic gate. It assumes hard faults (i.e., if present, a fault remains throughout the test) and that only one fault occurs at a time. Thus, this model became known as the single stuck-at fault model. Stuck-at fault testing assumes that the symptom of any manufacturing defect can be characterized by the presence of a stuck-at fault some place within the circuit and that it can be observed at some point on the unit under test. By testing for the presence of all possible stuck-at faults that can occur, all possible manufacturing hard-defects in the logic devices can thus be tested.

The stuck-at fault models for NAND and NOR gates are shown in Figure 14.2. For the NAND gate, the presence of an input stuck-at-one defect can be sensitized (made detectable) at the gate's output pin by setting the good-machine state for that input to 0, and setting the other input values to 1. If a zero is observed at the gate's output node, then a fault (stuck-at-one) on the input pin set to 0 is detected. A stuck-at-zero condition on any specific input to the NAND gate is not distinguishable from a stuck-at-zero on any other input to the gate, thus is not part of the model. However, a stuck-at-one is modeled for the gate's output to account for such a fault or a stuck-at-one fault on the gate's output circuitry. Similarly, a stuck-at-zero is modeled on the gate's output.

The fault model for the NOR gate is similar except that here input faults are modeled as stuck-at-zero, as the stuck-at-one defect cannot be isolated to a particular input.

Later in time, additional development would attack defects not detectable with this stuck-at fault model; for example, bridging faults where nodes are shorted together, and delay faults where the output response does not occur within the required time. The stuck-at fault model cannot detect these fault types and they became important as the development of CMOS progressed. Significant work was performed in both these areas beginning in the 1970s, but it did not have the impact on test generation development that the stuck-at fault model did.

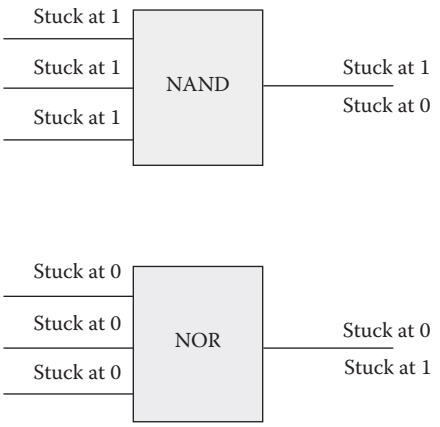


FIGURE 14.2 Stuck-at fault models.

The creation of the fault model was very important to test generation as it established a realistic set of objectives to be met by automated test set generation that could be achieved in a realistic amount of computational time. A formal algebra was developed by Roth [2] called the D-ALG that formalized an approach to test generation and fault diagnosis.

The test generation program could choose a fault based on the instances of gates within the design and the fault models for the gates. It could then trace back from that fault to the input pins of the design and, using the D-ALG calculus, it could find a set of input states that would sensitize the fault. Then, it could trace forward from that fault to the design's output pins, sensitizing the path (causing the good-machine value to be the opposite of the stuck-at fault value along that path) to at least one observable pin (Figure 14.3).

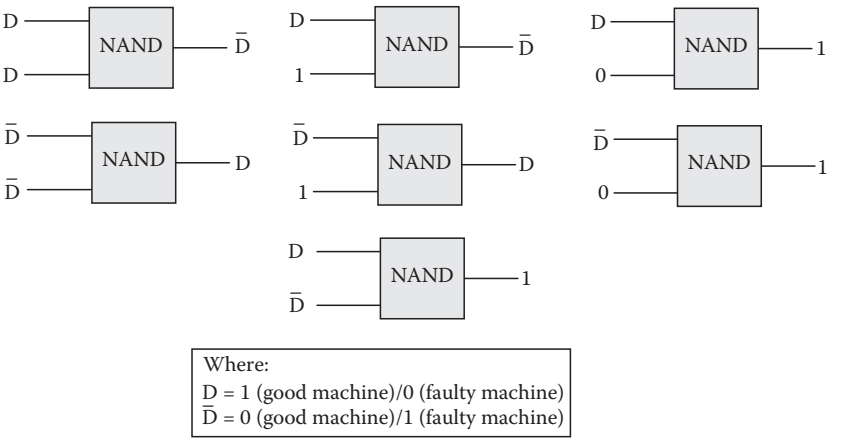


FIGURE 14.3 D-ALG calculus.

The use of functional patterns as the test patterns in lieu of heuristic stuck-at test generation was another approach for manufacturing test. However, this required an extreme number of tests to be applied and provided no measurable objective to be met (i.e., 100% of all possible stuck-at faults) thus, depended on the experience and skill of the designer to create quality tests. The use of fault models and automatic test generation produced a minimum set of tests and greatly reduced intensive manual labor.

The exhaustive method to assure coverage of all the possible detectable defects would be to apply all possible input states to the design under test and compare the measured output states with the simulated good-machine states. For a design with n input pins, however, this may theoretically require the simulation of 2^n input patterns for combinatorial logic and at least 2^{n+m} for sequential logic (where m is the number of independent storage elements). For even a relatively small number of input pins, this amount of simulation would not be possible even on today's computers, and the time to apply this number of patterns at the tester would be grossly prohibitive.

The amount of time that a part resides on the tester is critical in the semiconductor business, as it impacts a number of parts that can be produced in a given amount of time and the capital cost for testers. Thus, the number of test patterns that need to be applied at the tester should be kept to a minimum. Early work in test generation attacked this problem in two ways. First, when a test pattern was generated for a specific fault, it was simulated against all possible faults one at a time. In many cases, the application of a test pattern that is targeted for one specific fault will also detect several other faults at the same time. The presence of a specific fault may be detectable on one output pin, for example, but at the same time additional faults may be observable at the other output pins. The use of fault simulation (discussed in Section 14.2.1.3) detected these cases and provided a mechanism to mark as tested those faults that were "accidentally" covered. This meant that the test generation algorithm did not have to generate a specific test for those faults. Second, schemes were developed to merge test patterns together into a smaller test pattern set to minimize the number of patterns required for detection of all faults. This is possible when two adjacent test patterns require the application of specific values on different input pins, each allowing all the other input pins to be at a do not-care state. In these cases, the multiple test patterns can be merged into one. With successive analysis in this way, all pairs of test patterns (pattern n with $n+1$, or the merger of m and $m+1$ with pattern $m+2$) are analyzed and merged into a reduced set of patterns.

Sequential design elements severely complicate test generation, as they require the analysis of previous states and the application of sequences of patterns. Early work in test generation broke feedback nets, inserted a lumped delay on them, and analyzed the design as a combinatorial problem using a Huffman model. Later work attempted to identify the sequential elements within the design using sophisticated topological analysis and then used a Huffman model to analyze each unique element. State tables for each

unique sequential element were generated and saved for later use as lookup tables in the test generation process. The use of three-value simulation within the analysis reduced the analysis time as well as guaranteed that the results were always accurate. Huffman analysis required 2^x simulations (where x is the number of feedback nets) to determine if critical hazards existed in the sequential elements. Using three-valued simulation [3] (all value transitions go through an X state), this was reduced to a maximum of 2^x simulations.

This lumped delay model did not account for the distribution of delays in the actual design, thus it often caused pessimistic results. Often simulated results yielded do not-know (X-state) conditions when a more accurate model could yield a known state. This made it difficult to generate patterns that would detect all faults in the model. As the level of integration increased, the problems associated with automatic test generation for arbitrary sequential circuits became unwieldy. This necessitated that the designer be called back into the problem of test generation most often to develop tests that would detect those that were missed by the test generation program. New approaches that ranged from random pattern generation to advanced algorithms and heuristics, which use a combination of different approaches, were developed. However, by the mid-1970s the need to design-for-test was becoming evident to many companies.

During the mid-1970s the concept of scan design such as IBM's Level Sensitive Scan Design (LSSD), was developed [4]. Scan design provides the ability to externally control and observe internal state variables of the design. This is accomplished by the use of special scan latches into the design at the points to be controlled and observed. These latches are controllable and can accept one of the two different data inputs depending on an external control setting. One of these data inputs is the node within the IC to be controlled/observed and the other is used as the test input. These latches are connected into a shift register chain that has its stage-0 test input and stage-n output connected to externally accessible pins on the IC. Under normal conditions, signals from within the design are passed through individual latches via the data input/output. Under test conditions, test vectors can be scanned, under clock control, onto the scan register's externally accessible input pin and into the scan latches via their test data input. Once scanned into the register, the test vector is applied to the internal nodes. Similarly, internal state values within the design are captured in individual scan latches and scanned through the register out to its observable output pin.

The development of scan design was important for two reasons. First, LSSD allowed for external control and observability of all sequential elements within the design. With specific design-for-test rules, this reduced the problem of test generation to one of a set of combinatorial circuits. Rigid design rules such as the following assure this, and checking programs and scan chain generation algorithms were implemented to assure that they were adhered to before entering test generation:

- All internal storage elements implemented in hazard-free polarity-hold latches
- Absence of any global feedback loops
- Latches may not be controlled by the same clock that controls the latches feeding them
- Externally accessible clocks control all shift register latches

Second, scan design allowed for external control and observability at otherwise nonprobable points between modules on an MCM or PCB. During the 1980s an industry standard was developed, called Boundary Scan (IEEE 1149.1 Joint Test Action Group) [5], which uses scan design techniques to provide control and observability for all chip or module I/Os from pins of their next level package (MCM or PCD, respectively).

Scan design requires additional real estate in the IC design and has a level of performance overhead. However, with the achievable transistor density levels on today's ICs and the test and diagnosis benefits accrued, these penalties are easily justified in all but the most performance critical designs (Figure 14.4).

During the 1980s, IC density allowed for the design of built-in self-test (BIST) circuitry on the chip itself. Operating at hardware speed, it became feasible to generate complete exhaustive tests and large numbers of random tests never possible with software and stored program testers. BISTs are generated

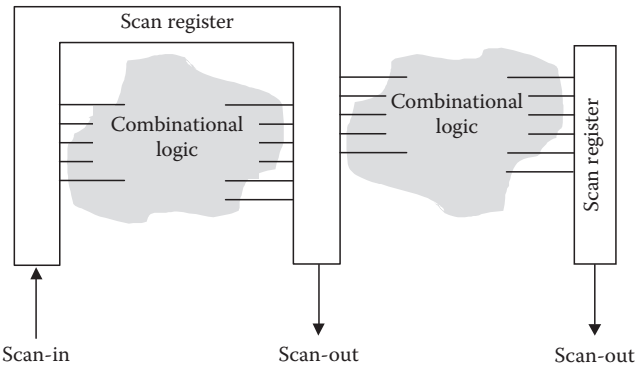


FIGURE 14.4 Scan design.

on the tester by the device under test, reducing the management of data transfer from design to manufacturing. A binary counter is used or linear feedback shift registers (LFSR) are used to generate the patterns for exhaustive or random tests, respectively. In the latter case, a pseudorandom bit sequence is formed by the exclusive-OR of the bits on the LFSR and this result is then fed back into the LFSR input. Thus, a pseudorandom bit sequence whose sequence length is based on the number of LFSR stages and the initial LFSR state can be generated. The design can be simulated to determine the good-machine state conditions for the generated test patterns, and these simulated results are compared with the actual device-under-test results observed at the tester.

BIST techniques have become common for the test of on-chip RAM and ROS. BIST is also used for logic sections of chips either with fault-simulated weighted random test patterns or good machine-simulated exhaustive patterns. In the latter case, logic is partitioned into electrically isolated regions with a smaller number of inputs to reduce the number of test patterns. Partitioning of the design reduces the number of exhaustive tests from 2^n (where n is the total number of inputs) to

$$\sum_{i=1}^{m} 2^{n^i}$$

where

m is the number of partitions
n^i ($n^i < n$) is the number of inputs on each logic partition

Since the use of BIST implies an extremely large number of tests, the simulated data transfer and test measurement time is reduced greatly by the use of a compressed signature to represent the expected and actual test results. Thus, only a comparison of the simulated signatures for each BIST region needs to be made with the signatures derived by the on-chip hardware, rather than the results of each individual test pattern. This is accomplished by feeding the output bit sequence to a single input serial input LFSR after exclusive-OR with the pseudorandom pattern generated by that LFSR. In this way, a unique pattern can be observed for a sequence of test results, which is a function of the good-machine response and a pseudorandom number (Figure 14.5).

Today, testing of ICs typically consists of combinations of different test strategies. These may include stored program stuck-at fault tests, BIST, delay (or at-speed) test (testing for signal arrival times in addition to state), and I_{DDQ} tests (direct drain quiescent current testing checks for CMOS defects that cause excessive current leakage). The latter two test techniques are used to detect defect conditions not identified by stuck-at fault tests such as shorts (bridging faults) between signal nets or gate oxide defects

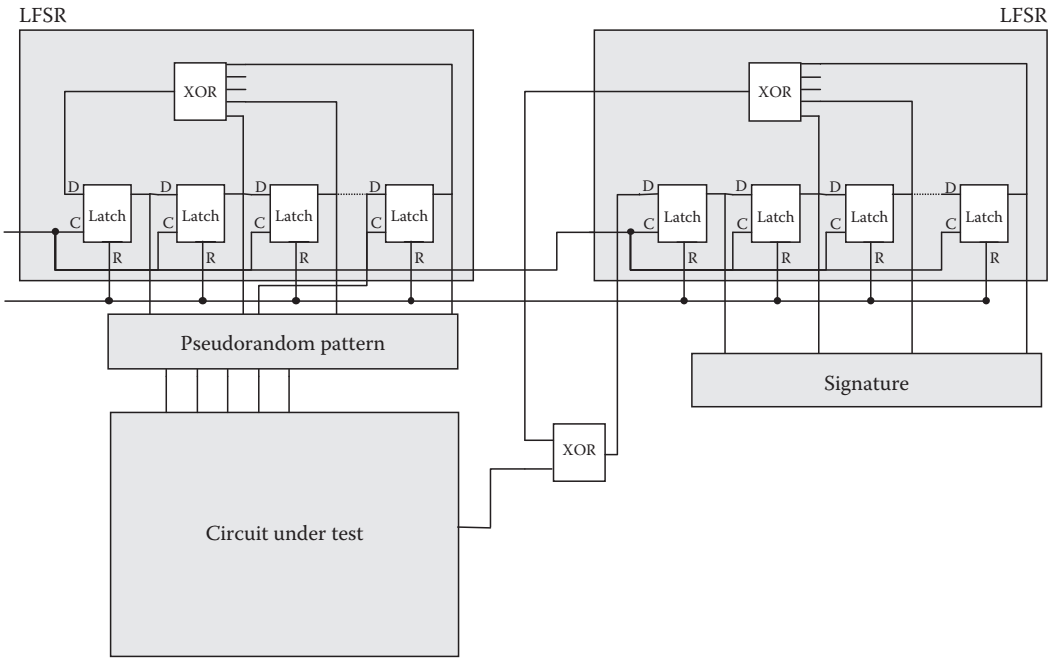


FIGURE 14.5 BIST logic.

that cause incorrect device operation and parametric variability resulting from process variations in the wafer foundry. However, the stuck-at fault model and scan techniques have been and will continue to be fundamental ingredients for the manufacturing test recipe.

14.2.1.3 Fault Simulation

Fault simulation is used to predict the state of a design at observable points when in the presence of a defect. This is used for manufacturing test and for field diagnostic pattern generation. Early work in fault simulation relied on the stuck-at fault model and performed successive simulations of the design with each single fault independent of any other fault; thus, a single stuck-at fault model was assumed. Because even these early designs consisted of thousands of faults, it was too time consuming to simulate each fault serially and it was necessary to create high-speed fault simulation algorithms.

For manufacturing test, fault simulation took advantage of three-valued zero-delay simulation. The simulation model of the design was levelized and compiled into an executable program. Levelization assured that driver gates were simulated before the gates receiving the signals; thus, allowing a state resolution in a single simulation pass. Feedback loops were cut and the *X*-transition of three-valued simulation resolved race conditions. The inherent instruction set of the host computer (e.g., AND, OR, and XOR.) allowed the use of a minimum set of instructions to simulate a gate's function.

The parallel-fault simulation algorithm that allowed many faults to be simulated in parallel was developed during the 1960s. Using the 32-bit word length of the IBM 7090 computer architecture, for example, simulation of 31 faults in a single pass (using the last bit for the good-machine) was possible. For each gate in the design, two host machine words were assigned to represent its good-machine state and the state for 31 single stuck-at faults. The first bit position of the first word was set to 1 or 0 representing the good-machine state for the node, and each of the successive bit position was set to the 1/0 state that would occur if the simulated faults were present (each bit-position representing a single fault). The corresponding bit position in the second word was set to 0 if it was a known state

and one if it was an *X*-state. The entire fault list was divided into *n* partitions of 31 faults and each partition was then simulated against all input patterns. In this way, the run time for simulation is a function of

$$\sum_{i=1}^{n=F/31} Patterns$$

where *F* is the total number of single stuck-at faults in the design.

Specific faults are injected within the word representing their location within the design by the insertion of a mask that is AND'd or OR'd at the appropriate word. For stuck-at-one conditions the mask contains a 1-bit in the position representing the fault (and 0-bit at the others) and it is OR'd to the gate's memory location. For stuck-at-zero faults the mask contains a 0-bit at the positions representing the fault (and 1-bit at the others) and it is AND'd with the gate's memory location.

As the level of integration increased, so did the number of faults. The simulation speed improvement realized from parallel-fault simulation was limited to the number of faults simulated in parallel and because of the algorithm overhead, it did not scale with the increase in faults.

Deductive-fault simulation was developed early in the 1970s and required only one simulation pass per test pattern. This is accomplished by simulating only the good-machine behavior and using deductive techniques to determine each fault that is detectable along the simulated paths. Note here that fault detection became the principal goal and fault isolation (for repair purposes) was ignored, as by now the challenge was to isolate from a wafer bad chips that were not repairable. Because lists of faults detectable at every point along the simulated path need to be kept, this algorithm requires extensive memory, far more than the parallel-fault simulation one. However, with increasing memory on host computers and the inherent increase in fault simulation speed, this technique won the favor of many fault simulators.

Concurrent-fault simulation refined the deductive algorithm by recognizing the fact that paths in the design quickly become insensitive to the presence of most faults, particularly after some initial set of test patterns is simulated (an observation made in the 1970s was that a high percentage of faults is detected in a low percentage of the initial test patterns, even if these patterns are randomly generated). The concurrent-fault simulation algorithm simulates the good machine and concurrently simulates a number of faulty machines. Once it is determined that a particular faulty machine state is the same as the good-machine one, simulation for that fault ceases. Since on logic paths most faults will become insensitive rather close to the point where the fault is located, the amount of simulation for these faulty machines was kept small. This algorithm required even more memory, particularly for the early test patterns; however, host machine architectures of the late 1970s were supporting, what then appeared as, massive amounts of addressable memory.

With the introduction of scan design in which all sequential elements are controllable from the tester, the simulated problem is reduced to that of a combinatorial circuit whose state is deterministic based on any single test pattern, and is not dependent on previous patterns or states. Parallel-pattern fault simulation was developed in the late 1970s to take advantage of this, by simulating multiple test patterns in parallel against a single fault. A performance advantage is achieved because compiled simulation could again be utilized as opposed to the more costly event-based approach. In addition, because faults not detected by the initial test patterns are typically only detectable by a few patterns and for these a sensitized path often disappears within a close proximity to the fault's location, simulation of many patterns does not require a complete pass across the design.

Because of the increasing number of devices on chips, the test generation and fault simulation problem continued to face severe challenges. With the evolution of BIST however, the use of fault simulation was relaxed. With BIST, only the good-machine behavior needs to be simulated and compared with the actual results at the tester.

14.2.1.4 Physical Design

As with test generation, PD has evolved from early work on PCBs. PD automation programs place devices and generate the physical interconnect routing for the nets that connect them into logic paths assuring that electrical and physical constraints are met. The challenge for PD has become ever greater since its early development for PCBs where the goal was simply to place components and route nets typically looking for the shortest path. Any nets that could not be auto-routed were routed (embedded) manually, or as a last resort with nonprinted (yellow) wires. As the problem moved on to the ICs, the ability to use nonprinted wires to finish routing was no more. Now, all interconnects had to be printed circuits and anything less than a 100% solution is unacceptable. Further, as the IC densities increased, so did the number of nets which necessitated the invention of smarter wiring programs and heuristics. Even a small number of incomplete (overflow) routes became too complex of a task for manual solutions.

As IC device sizes shrank and the gate delays decreased, the delay caused by interconnect wiring also became an important factor for a valid solution. No longer was any wiring solution a correct solution. Complexity increased by the need to find wiring solutions that fall within acceptable timing limits. Thus, the wiring lengths and thickness needed to be considered. As IC features become packed closer together cross-coupled capacitance (cross talk) effects between them is also an important consideration and for the future, wiring considerations will expand into a three-dimensional space. PD solutions must consider these complex factors and still achieve a 100% solution that meets the designer-specified timing for IC designs that contain hundreds of millions of nets.

Because of these increasing demands on the PD, major paradigm changes have taken place in the design methodology. In the early days, there was a clear separation of logic design and PD. The logic designer was responsible for creating a netlist that correctly represented the logic behavior desired. Timing was a function of the drive capability of the driving circuit and the number of receivers. Different power levels for drivers could be chosen by the logic designer to match the timing requirements based on the driven circuits. The delay imposed by the time-of-flight along interconnects and owing to the parasitics on the interconnect was insignificant. Therefore, the logic designer could hand off the PD to another, more adept at using the PD programs and manually embedding overflow wires. As the semiconductor technology progressed, however, there needed to be more interactions between the logic designer and the physical designer, as the interconnect delays became a more dominant factor across signal paths. The logic designer had to give certain timing constraints to the physical designer and if these could not be met, the design was often passed back to the logic designer. The logic designer, in turn, then had to choose different driver gates or a different logical architecture to meet his design specification. In many cases, the pair had to become a team or there was a merger of the two previously distinct operations into one ''IC designer.''

This same progression of merging logic design and PD into one operational responsibility has also begun at the EDA system architecture level. In the 1960s and 1970s, front-end (design) programs were separate from back-end (physical) programs. Most often they were developed by different EDA development teams and designs were transferred between them by means of data files. Beginning in the 1980s, the data transferred between the front-end programs and the back-end ones included specific design constraints that must be met by the PD programs—the most common being a specific amount of allowed delay across an interconnect or signal path. Moreover, as the number of constraints that must be met by the PD programs increases so does the challenge to achieve a 100% solution. Nonetheless, many of the fundamental wiring heuristics and algorithms used by PD today spawned from work done in the 1960s for PCBs.

Early placement algorithms were developed to minimize the total length of the interconnect wiring using Steiner trees and Manhattan wiring graphs. In addition, during these early years, algorithms were developed to analyze wiring congestion that would occur as a result of placement choices and minimize it to give routing a chance to succeed. Later work in the 1960s led to algorithms that performed a hierarchical division of the wiring image and performed global wiring between these subdivisions

(then called cells) before routing within the cells [6]. This divide-and-conquer approach simplified the problem and led to quicker and more complete results. Min-cut placement algorithms often used today are a derivative of this divide-and-conquer approach. The image is divided into partitions and the placements of these partitions are swapped to minimize interconnect length between them and possible wiring congestion. Once a global solution for the cells is found, placement is performed within them using the same objectives. Many current placement algorithms are based on these early techniques, although they now need to consider more physical and electrical constraints.

Development of new and more efficient routing algorithms progressed. The Lee algorithm [7] finds a solution by emitting a ''wave'' from both the source and target points to be wired. This wave is actually an ordered identification of available channel positions—where the available positions adjacent to the source or destination are numbered 1, and the available positions adjacent to them are numbered 2, etc. Successive moves and sequential identification is made (out in all directions as would a wave) until the source and destination moves meet (the waves collide). Then a backtrace is performed from the intersecting position in reverse sequential order along the numbered track positions back to the source and destination. At points where a choice is available (i.e., there are two adjacent points with the same order number), the one which does not require a change in direction is chosen.

The Hightower line-probe technique [8], also developed during this period and speeded up routing, by use of emanating lines rather than waves. This algorithm emanated a line from both the source and destination points, toward each other. When either line encounters an obstacle, then another line is emanated from a point just missing the edge of the obstacle on the original line at a right angle to the original line, and toward the target or source. Thus, the process is much like walking blindly in an orthogonal line toward the target and changing direction only after bumping into a wall. This process continues until the lines intersect at which time the path is complete.

In today's ICs, the challenge of completed wiring that meets all constraints is of crucial importance. Unlike test generation, which can be considered successful when a very high percentage of the faults are detected by the test patterns, 100% complete is the only acceptable answer for PD. Further, all of the interconnects must fall within the required electrical and physical constraints. Nothing $<100\%$ is acceptable! Today these constraints include timing, power consumption, noise, and yield and this list will become more complex as IC feature sizes and spacing are reduced further.

14.2.2 The 1970s: The Awaking of Verification

Before the introduction of large-scale integrated (LSI) circuits in the 1970s, it was common practice to build prototype hardware to verify the design correctness. PCB packages containing discrete components, single gates, and small-scale integrated modules facilitated engineering rework of real hardware within the verification cycle. Prototype PCBs were built and engineering used test stimulus drivers and oscilloscopes to determine whether the correct output conditions resulted from input stimuli. As design errors were detected, they were repaired on the PCB prototype, validated, and recorded for later engineering into the production version of the design. Use of the wrong logic function within the design could easily be repaired by replacing components in error with the correct ones. Incorrect connections could easily be repaired by cutting a printed circuit and replacing it with a discrete wire. Thus, design verification (DV) was a sort of trial-and-error process using real hardware.

The introduction of LSI circuits drastically changed the DV paradigm. Although the use of software simulation to verify system design correctness began during the 1960s, it was not until the advent of the LSI circuits that this concept became widely accepted. With large-scale integration, it became impossible to use prototype hardware or to repair a faulty design after it was manufactured. The 1970s are best represented by a quantum leap into verification before manufacture through the use of software modeling (simulation). This represented a major paradigm shift in electronics design and was a difficult change for some to accept. DV on hardware prototypes resulted in a tangible result that could be touched and held. It was a convenient tangible, which could be shown by management to represent real progress.

Completed DV against a software model did not produce the same level of touch and feel. Further, since the use of computer models was a relatively new concept, it met with the distrust of many. However, the introduction of LSI circuits demanded this change and today, software DV is a commonplace practice used in all levels of electronic components, subassemblies, and systems.

Early simulators simulated gate-level models of the design with two-valued (1 and 0) simulation. Since gates had nearly equal delays and the interconnect delay was insignificant by comparison, the use of a unit of delay for each gate with no delay assigned to interconnects was common. Later simulators exploited the use of three-valued simulation (1, 0, and unknown) to resolve race conditions and identify oscillations within the design more quickly. With the emergence of LSI circuits, however, these simple models had to become more complex and, additionally, simulators had to become more flexible and faster—much faster. In the first half of the 1970s, important advances were made to DV simulators that included

- Use of abstract (with respect to the gate-level) models to improve simulation performance and enable verification throughout the design cycle (not just at the end)
- More accurate representations of gate and interconnect delays to enhance the simulated accuracy

In the latter half of the decade, significant contributions were made to facilitate separation of function verification from timing verification and formal approaches for verification. However, simulation remains a fundamental DV tool and the challenge to make simulators faster and more flexible continues even today.

14.2.2.1 Simulation

Though widely used for DV, simulation has a couple of inherent problems. First, unlike test generation or PD, there is no precise measure of completeness. Test generation has the stuck-at fault model and PD has a finite list of nets that must be routed. However, there is no equivalent metric to determine when verification of the design is complete, or when enough simulation is done. During the 1970s research began to develop a metric for verification completeness, but to this day none has been generally accepted. Use is made of minimum criteria such as all nets must switch in both directions, and statistical models using random patterns. Recent work in formal verification is applying algorithmic approaches to validate coverage of paths and branches within the model. However, the generally accepted goal is to ''do more.''

Second, it is a hard and time-consuming task to create effective simulation vectors to verify a complex design. DV simulators typically support a rich high-level language for the stimulus generation, but still require the thought, experience, and ingenuity of the DV engineer to develop and debug these programs. Therefore, it is desirable to simulate the portion of the design being verified in as much of the total system environment as possible and have the simulation create functional stimulus for the portion to be verified.

Finally, it is tedious to validate the simulated results for correctness, making it desirable to simulate the full system environment where it is easier to validate results. Ideally, the owner of an application-specific integrated circuit (ASIC) chip being designed for a computer could simulate that chip within a model of all of the computer's hardware, the microcode, the operating system, and use example software programs as the ultimate simulation experiment. To even approach this goal, however, simulator technology must continuously strive for rapid and more rapid techniques.

Early DV simulators often used compiled models (where the design is represented directly as a computer program), but this technique gave way to interpretative event-driven simulation. Compiled simulators have the advantage of higher speeds because host machine instructions are compiled in-line to represent the design to be verified and are directly executed with minimum simulator overhead. Event-based simulators require additional overhead to manage the simulation operations, but provide a level of flexibility and generality not possible with the compiled model. This flexibility was necessary to provide for simulation of timing characteristics as well as function, and to handle general sequential designs. Therefore, this approach was generally adopted for DV simulators.

14.2.2.1.1 Event-Based Simulation

There are four main concepts to an event-based simulator:

- Netlist, which provides the list of blocks (gates at first, but any complex function later), connections between blocks, and delay characteristics of the blocks.
- Event time queues, which are lists of events that need to be executed (blocks that need to be simulated) at specific points in (simulation) time. Event queues contain two types of events—update and calculate. Update-events change the specified node to the specified value, then schedule calculate-events for the blocks driven from that node. Calculate-events call the simulation behavior for the specified block and, on return from the behavior routine, schedule update-events to change the states on the output nodes to the new values at the appropriate (simulation) time
- Block Simulation Behavior (the instructions that will compute that block's output(s) states when there is a change to its input(s) state—possibly also scheduling some portion of the block's behavior to be simulated at a later time).
- Value list—the current state of each node in the design.

Simulation begins by scheduling update-events in appropriate time queues for the pattern sequence to be simulated. After the update-events are stored, the first-time event queue is traversed and, one-by-one, each update-event in the queue is executed. Update-events update the node in the value list and, if the value new value is different from the current value (originally set to unknown), schedule calculate-events for blocks driven from the updated node. These calculate-events are saved back in time queues based on delay specifications for later execution—which will be discussed later. After all update-events are executed and removed from the queue, the simulator selects calculate-events in the queue sequentially, interprets its function, and passes control to the appropriate block simulation behavior for calculation.

The execution of a calculate-event causes simulation of a block to take place. This is accomplished by passing control to the simulation behavior of the block with pointers to the current state-values on its inputs (in the value list). When complete, the simulation routine of the block will pass control back to the simulator with the new state condition for its output(s). The simulator then schedules the block output(s) value update by placing an update-event for it in the appropriate time queue. The decision of which time queue the update-events are scheduled within is based on what the delay value is for the block.

Once all calculate-events are executed and removed from the queue, the cycle begins again at the next time queue and the process of executing update-events followed by calculate-events for the current time queue repeats. This cycle repeats until there are no more events or until some specified maximum simulation time. To keep update-events separated from calculate-events within the linked-list queues, it is common to add update-events at the top of the linked-list and calculate-events at the bottom, updating the chain-links accordingly.

Because it is not possible to predetermine how many events will reside in a queue at any time, it is common to create these as linked lists of dynamically allocated memory. Additionally, time queues are linked since the required number of time queues cannot be determined in advance and similar to events, new time queues can be inserted into that chained list as required (Figure 14.6).

A number of techniques have been developed to make queue management fast and efficient as they are the heart of the simulator. Because of its generality, the event-based algorithm was the clear choice for DV. One advantage of event-based simulation over the compiled approach is that it easily supports the simulation of delay. Delays can be simulated for blocks and nets and even early simulators typically supported a very complete delay model, even before sophisticated delay calculators were available to take advantage of it.

Treatment of delay has had to evolve and expand since the beginnings of software DV. Unit delay was the first model used—where each gate in the design is assigned one unit of delay and interconnect delays are zero. This was a crude approximation, but allowed for high-speed simulations because of the

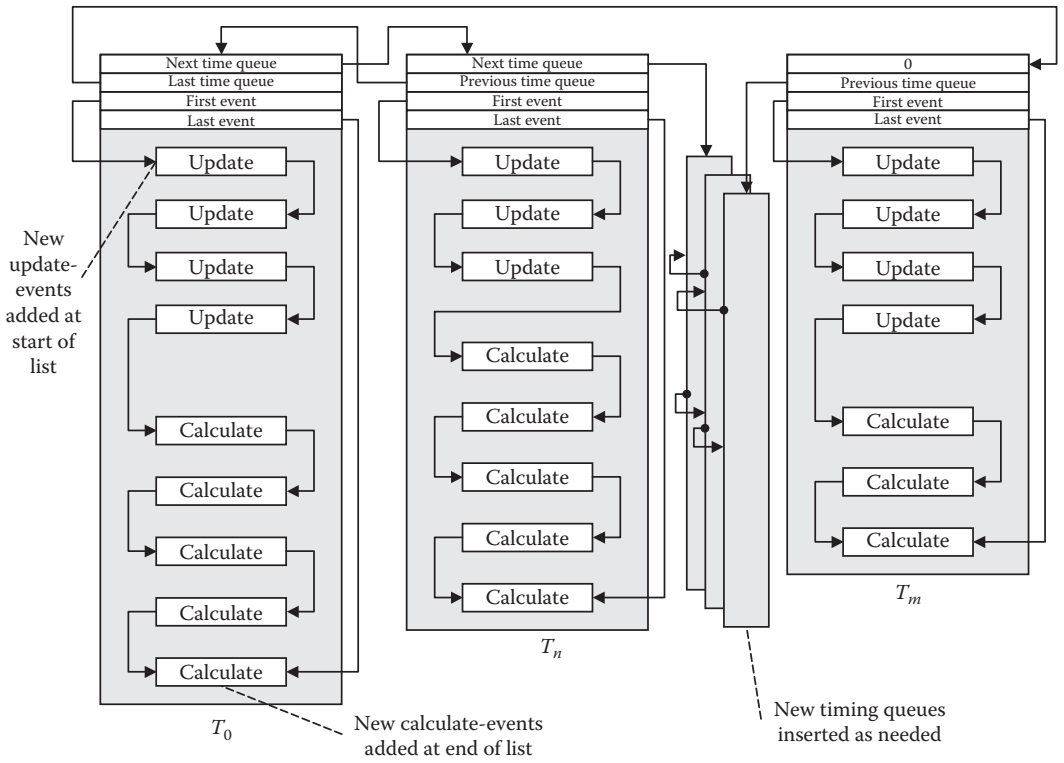


FIGURE 14.6 Event queues.

simplifying assumptions and it worked reasonably well. As the IC evolved, however, the unit delay model was replaced by a lumped delay model in which, each gate could be assigned a unique value for delay—actually, a rise-delay and a fall-delay. This was assigned by the technologist based on some average load assumption. At this time, also, the beginnings of development of delay calculators began. These early calculators used simple equations, adding the number of gates driven by the gate being calculated, and then adding the additional delay to the intrinsic delay values of that gate. As interconnect wiring became a factor in the timing of the circuit the pin-to-pin delay came into use. Though delay values used for simulation in the 1970s was crude by today's norm, the delay model Figure 14.7 was rich and supported specification and simulation for

- Intrinsic block delay (T_{block})—the time required for the block output to change state relative to the time that a controlling input to that block changed state
- Interconnect delay (T_{int})—the time required for a specific receiver pin in a net to change state relative to the time that the driver pin changed state
- Input–output delay (T_{io})—the time required for a specific block output to change state relative to the time the state changed on a specific input to that block

Today's ICs, however, require delay calculation based on very accurate distributed RC models for interconnects, as these delays have become more significant with respect to gate delays. Future ICs will require even more precise modeling for delays, as will be discussed later in this chapter, and consider inductance (L) in addition to the RC parasitics. Additionally, transmission line models will be necessary for the analysis of certain critical global interconnects. However, the delay model defined in the early years of DV and the capabilities of the event-based simulation algorithm stand ready to meet this challenge.

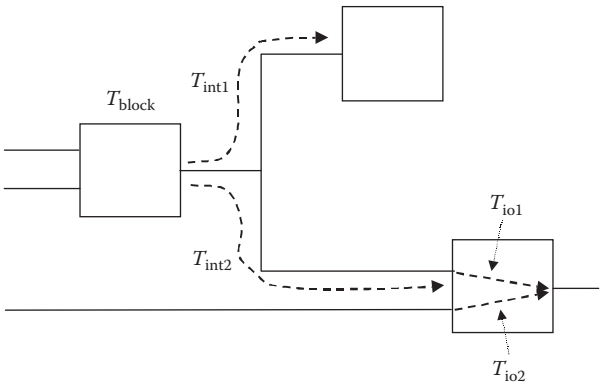


FIGURE 14.7 Simulation delay types.

Another significant advantage of the event-based simulation is that it easily supports simulation of blocks in the netlist at different levels of abstraction. Recall that one of the fundamental components of event simulation is the block simulation behavior. A calculation-event for a block passes control to the subroutine that simulates the behavior using the block's input states. For gate-level simulation, these behavior subroutines are quite simple—AND, OR, NAND, and NOR. However, since the behavior is actually a software program, it can be arbitrarily complex as well. Realizing this, early work took place to develop description languages that could be coded by designers and compiled into block simulation behaviors to represent arbitrary sections of a design as a single (netlist) block. For example, a block simulation behavior might look like the following:

```
  FUNCTION (CNOR);
/* Complimentary output NOR function with 3 inputs. Input delay=2,
   intrinsic delay for true output=10, intrinsic delay for complement
   output=12*/
  INPUT (a,b,c); /*Declare for inputs a, b, c*/
  OUTPUT (x,y);
  DECLARE input1, input2, input3; /*Declare storage for inputs*/
  DECLARE out, cout;              /*Declare storage for outputs*/

  GET a,input1, b,input2, c,input3;

  DELAY 2, Entry1;
    Entry1: out=input1|input2|input3;
    cout=¬out;

    DELAY 10, (x=out); /*Schedule this update-event for 10 time units
                         later */
    DELAY 12 (y=cout); /*Schedule this update-event for 12 time units
                         later */
    END Entry1;

  ENDCNOR;
```

The sophisticated use of these behavioral commands supported the direct representation not only for simple gates, but also for complex functions such as registers, MUXs, RAM, and ROS. In doing so, simulation performance was improved because what was treated before as a complex interconnection of gates could now be simulated as a single block.

By generalizing the simulator system, block simulation routines could be loaded from a library at simulation run time only when required (by the use of a block with that function in the netlist), and dynamically linked with the simulator control program. This meant that the block simulation behaviors could be developed by anyone, compiled independently from the simulator, stored in a library of simulation behavioral models, and used as needed by a simulation program. This is common practice in DV simulators today, but this concept in the early 1970s represented a significant breakthrough and supported a major paradigm shift in design—namely, the concept of top-down design and verification.

With top-down design, the designer no longer had to design the gate-level netlist before verification could take place. High-level (abstract) models could be written to represent the behavior of sections of the design not yet complete in detail, and they could be used in conjunction with gate-level descriptions of the completed parts to perform full system verification. Now, simulation performance could be improved by use of abstracted high-speed behavior models in place of detailed gate-level descriptions for portions of the overall design. Now, concurrent design and verification could take place across design teams. Now, there was a formal method to support design reuse of system elements without the need to expose internal design details for reusable elements. In the extreme case, the system designer could write a single functional model for the entire system and verify it with simulation. The design could then be partitioned into subsystem elements and each could be described with an abstract behavioral model before being handed off for detailed design. During the detailed design phase, individual designers could verify their subsystem in the full system context even before the other subsystems were completed. Also, the behavioral model concept was particularly valuable for generation of the functional patterns to be simulated as they could now be generated by the simulation of the other system components. Thus, verification of the design no longer had to wait until the end, it could now be a continuous process throughout the design.

Throughout the period, improvements were made to DV simulators to improve performance and in the formulation and capability of behavioral description languages. In addition, designers found increasingly novel ways to use behavioral models to describe full systems containing nondigital system elements and peripherals such as I/O devices. Late in the 1970s and during the 1980s two important formal languages for describing system behavior were developed:

- VHDL [9] (very high-speed integrated circuit [VHSIC] high-level description language), sponsored by DARPA
- Verilog [10], a commercially developed RTL description language

These two languages are now accepted industry-standard design description languages.

14.2.2.1.2 Compiled Simulation

Synchronous design such as that used for scan-based design which emerged in the 1970s provides for a clean separation of timing verification and functional verification of combinatorial circuits. Because of this, the use of compiled simulation returned for high-speed verification for designs meeting constraints. A simulation technique called cycle simulation that was developed yielded a major performance advantage over event-based simulation. Cycle-simulation treats the combinational sections of a scan-based design as compiled zero-delay models moving data between them at clock-cycle boundaries. The simulator executes each combinatorial section at each simulated cycle by passing control to the compiled routine for it along with its input states. The resulting state values at the outputs of these sections are assumed to be correctly captured in their respective latch positions. That is, the clock circuitry and path delays are assumed correct, and are not simulated during this phase of verification. The (latched) output values are used as the input states to the combinatorial sections they drive at the next cycle, and the process repeats for each simulated cycle. Each simulation pass across the compiled models represents one cycle of the design's system clock, starting with an input state and resulting in an output state. To assure that only one pass is required, the gates or RTL statements for the combinational sections are levelized

before compilation into the host-machine instructions. This assures that value updates occur before calculations and only one pass across a section of the model is required to achieve the correct state response at the outputs.

Simulation performance was greatly improved with cycle simulation because of the compiled model and because the clock circuitry did not have to be simulated repeatably with each simulated-machine cycle. Cycle simulation did not replace event simulation even for constrained synchronous designs as the clock circuitry needs to be verified; however, it proved to be effective for many large systems and these early developments provided the foundations for modern compiled simulators used today.

14.2.2.1.3 Hardware Simulators

During the 1980s, research and development took place on custom hardware simulators and accelerators. Special-purpose hardware-simulators use massively parallel instruction processors with much customized instruction sets to simulate gates. These provide simulation speeds that are orders of magnitude faster than the software simulation on general-purpose computers. However, they are expensive to build, lack the flexibility of software simulators, and the hardware technology they are built in soon becomes outdated (although general parallel architectures may allow the incremental addition of additional processors). Hardware accelerators use custom hardware to simulate portions of design in conjunction with the software simulator. These are more flexible, but still have the inherent problems that their big brothers have. Nonetheless, use of custom hardware to tackle the simulation performance demands has gained acceptance in many companies and they are commercially available today using both gate-level and hardware description language (HDL) design descriptions.

14.2.2.2 Timing Analysis

The practice of divide and conquer in DV started in the 1970s with the introduction of the behavioral model. Another divide-and-conquer style born in the 1970s, which achieved wide popularity in the 1990s, is to separate verification of the design's function from its timing. With the invention of scan design, it became possible to verify logic as a combinational circuit using high-speed compiled cycle simulators. Development of path tracing algorithms to verify timing resulted in a technique to verify timing without simulation; thus providing a complete solution which is not dependent on completeness of any input stimulus as required by simulation. For this reason alone, this technique coined static timing analysis (STA) was a major contribution to EDA—one that became key to the notion of ''signoff'' to the wafer foundry.

STA is used to analyze projected versus required timing along signal paths from primary inputs to latches, latches to latches, latches to primary outputs, and primary inputs to primary outputs. This is done without the use of simulation, but by summing the min–max delays along each path. At each gate, the STA program computes the min–max time in which that gate will change in state based on the min–max arrival times of its input signals. STA tools do not simulate the gate function, they only add its contribution to the path delay, although the choice of using rise or fall times for the gate is based on whether it has a complementary output, or not. Because the circuitry between the latches is combinational, only one pass needs to be made across the design. The addition can be based on the minimum rise or fall delay for gates or both, providing a min–max analysis. The designer specifies the required arrival times for paths at the latches or primary outputs, and the STA program compares these with the actual arrival times. The difference between the required arrival and the actual arrival is defined as slack. The STA tool computes the slack at the termination of each path, sorts them numerically, and provides a report. The designer then verifies the design correctness by analysis of all negative slacks.

Engineering judgment is applied during the analysis, including the elimination of false-path conditions. A false-path is a signal transition that will never occur in the real operation of the design. Since the STA tool does not simulate the behavior of the circuit, it cannot automatically eliminate all false-paths. Through knowledge of the signal polarity, STA can eliminate false-paths caused by the fan-out and reconvergence of certain signals. However, other forms of false-paths are only identifiable by the designer.

14.2.2.3 Formal Verification

Development of formal methods to verify the equivalence of two different representations of a design began during the 1970s. Boolean verification analyzes a circuit against a known good reference and provides a mathematical proof of equivalence. An RTL model, for example, of the reference circuit is verified using standard simulation techniques. The Boolean verification program then compiles the known-good reference design into a canonical NAND–NOR equivalent circuit. This equivalent circuit is compared with the gate-level hardware design using sophisticated theorem provers to determine equivalence. To reduce processing times, formal verification tools may preprocess the two circuits to create an overlapping set of smaller logic *cones* to be analyzed. These *cones* are simply the set of logic traversed by backtracing across the circuit from an output node (latch positions or primary outputs) to the controlling input nodes (latch positions or primary inputs). User controls specify the nodes that are supposed to be logically equivalent between the two circuits.

Early work in this field explored the use of test generation algorithms to prove equivalence. Two *cones* to be compared can be represented as $F_{\text{cone1}} = f(a, b, c, X, Y, Z, \dots)$ and $F_{\text{cone2}} = f(d, e, f, \dots, X', Y', Z', \dots)$. F_{cone1} and F_{cone2} are user defined output nodes to be compared for equivalence. The terms a, b, c and d, e, f represent the set of input nodes for the function and X, Y, Z are the computational subfunctions. User inputs define the equivalence between a, b, c and d, e, f. If F_{cone1} and F_{cone2} are functionally equivalent, then the value of G must be 0 for all possible input states. If the two cones are equivalent then, use of D-ALG test generation techniques for $G = F_{\text{cone1}}$ XOR F_{cone2} will be unable to derive a test for the stuck-at-zero fault on the output of G. Similarly, the use of random pattern generation and simulation can be applied to prove equivalence between *cones* by observing the value of G for all input states.

Research during the 1980s provided improvements in Boolean equivalence checking techniques and models (such as binary decision diagrams), and modern Boolean equivalence checking programs may employ a number of mathematical and simulation algorithms to optimize the overall processing. However, Boolean equivalence checking methods require the existence of a reference design against which equivalence is proved. This implies there must be a complete validation of the reference design against the design specification. Validating the reference design has typically been a job for simulation and, thus, is vulnerable to the problems of assuring coverage and completeness of the simulation experiment. Consequently, formal methods to validate the correctness of functional-level models have become an important topic in EDA research. Modern design validation tools use a combination of techniques to validate the correctness of a design model. These typically include techniques used in software development to measure completeness of simulation test cases such as

- Checking for coverage of all instructions (in the model)
- Checking to assure that all possible branch conditions (in the model) were exercised

They may also provide more formal approaches to validation such as

- Checking (the model) against designer-asserted conditions (or constraints) that must be met
- Techniques that construct a proof that the intended functions are realized by the design

These concepts and techniques continue to be the subject of research and will gain more importance as the size of IC designs stretches the limits of simulation-based techniques.

14.2.2.4 Verification Methodologies

With the use of structured design techniques, the design to be verified (Figure 14.8) can be treated as a set of combinational designs. With the arsenal of verification concepts that began to emerge during this period, the user had many verification advantages not previously available. Without structured design, delay simulation of the entire design was required. Use of functional models intermixed with gate-level descriptions of subsections of the design provided major improvements, but was still very costly and time consuming. Further, to be safe, the practical designer would always attempt to delay simulate the entire design at the gate level.

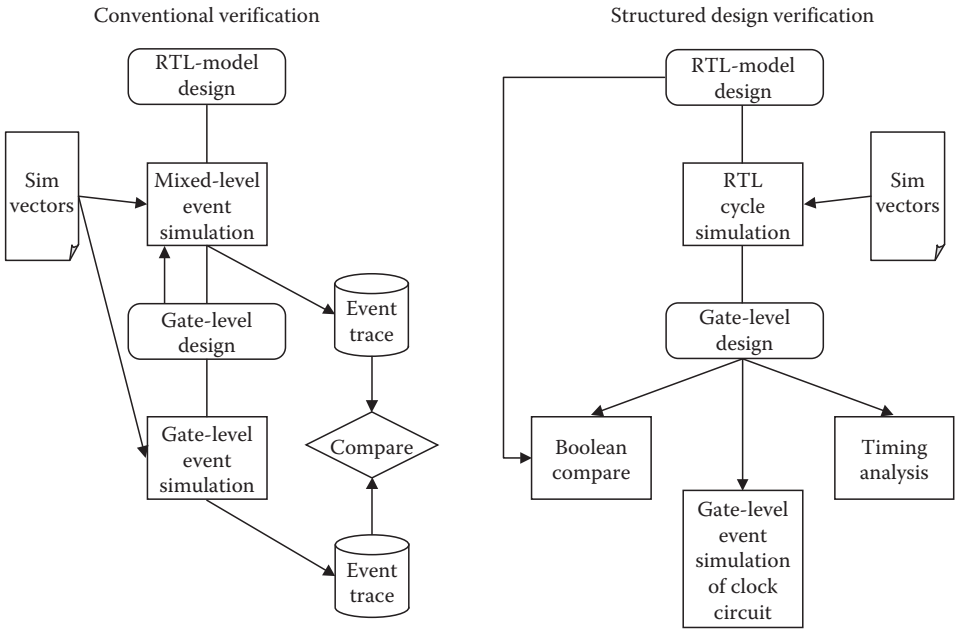


FIGURE 14.8 Verification methodology.

With structured design techniques, the designer could do the massive simulations at the RTL focusing of the logic function only, regardless of the timing. Cycle simulation improved the simulation performance by 1 to 2 orders of magnitude by eliminating repetitive simulations of the clock circuitry and use of compiled (or table lookup) simulation. For some, the use of massively parallel hardware simulators offered even greater simulation speeds—the equivalent of hundreds of millions events per second. Verification of the logic function of the gate-level design could be accomplished by formally proving its equivalence with the simulated RTL model. STA provided the basis to verify the timing of all data paths in a rigorous and methodical manner. Functional and timing verification of the clock circuitry could be accomplished with the standard delay-simulation techniques using event simulation. Collectively, these tools and techniques provided major design productivity improvements. However, the IC area and performance overhead required for these structured design approaches limited the number of designs taking advantage of them. During the 1980s as the commercial EDA business developed, these new verification tools and techniques remained as in-house tools in a few companies. Commercial availability did not emerge until the 1990s when the densities and complexities of ICs began to demand the change.

14.2.3 The 1980s: Birth of the Industry

Up to the 1980s design automation was, for the most part, developed in-house by a small number of large companies for their own proprietary use. Almost all EDA tools operated against large mainframe computers using company-specific interfaces. High-level description languages were unique and most often proprietary, technology rule formats were proprietary, and user interfaces were unique. As semiconductor foundries made their manufacturing lines available for customer-specific chip designs; however, the need for access to EDA tools grew. With the expansion of the ASIC, the need for commercially available EDA tools exploded. Suddenly, a number of commercial EDA companies began to emerge and electronics design companies had the choice of developing tools in-house or

purchasing them from a variety of vendors. The EDA challenge now often became that of integrating tools from multiple suppliers into a homogeneous system. Therefore, one major paradigm shift of the 1980s was the beginnings of EDA standards to provide the means to transfer designs from one EDA design system to another or from a design system to manufacturing. VHDL and Verilog matured to become IEEE industry-standard HDLs. Electronic data interchange format (EDIF) [11] was developed as an industry standard for the exchange of netlist and GDSII (developed in the 1970s at Calma) became a standard interface for transferring mask pattern data. (In 2003, SEMI introduced a new mask pattern exchange standard, OASIS, that compacts mask pattern data to one-tenth or less of the bytes used by GDSII.)

A second paradigm shift of the 1980s was the introduction of the interactive workstation as the platform for EDA tools and systems. Although some may view it as a step backward, the ''arcade'' graphics capabilities of this new hardware and its scalability caught the attention of design managers and made it a clear choice over the mainframe wherever possible. For a time, it appeared that many of the advances made in alphanumeric HDLs were about to yield to the pizzazz of graphical schematic editors. Nevertheless, although the graphics pizzazz may have dictated the purchase of one solution over another, the dedicated processing, and the ability to incrementally add compute power made the move from the mainframe to the workstation inevitable. Early commercial EDA entries such as from Daisy (Logician) and Valid (SCALD) were developed on custom hardware using commercial microprocessors from Intel and Motorola. This soon gave way, however, to commercially available workstations using RISC-based microprocessors, and UNIX became the de facto operating system standard. During the period, there was a rush of redevelopment of many of the fundamental algorithms for EDA onto the workstation. However, with the development of commercial products and with the new power of high-function graphics, these applications were vastly improved along the way. Experience and new learning streamlined many of the fundamental algorithms. The high-function graphic display provided the basis for enhanced user interfaces to the applications and high-performance high-memory workstations allowed application speeds to compete with the mainframe. The commercial ASIC business provided focus on the need for technology libraries from multiple manufacturers. Finally, there was significant exploration into custom EDA hardware such as hardware simulators and accelerators and parallel processing techniques.

From an IC design perspective however, the major paradigm shift of the 1980s was synthesis. With the introduction of synthesis, automation could be used to reduce an HDL description of the design to the final hardware representation. This provided major productivity improvements for ASIC design, as chip designers could work at the HDL level and use automation to create the details. Also, there was a much higher probability that the synthesis-generated design would be correct than for manually created schematics. The transgression from the early days of IC design to the 1980s is similar to what occurred earlier in software. Early computer programming was done at the machine language level. This could provide optimum program performance and efficiency, but at the maximum labor cost. Programming in machine instructions proved too inefficient for the vast majority of software programs, thus the advent of assembly languages. Assembly language programming offers a productivity advantage over machine instructions because the assembler abstracts up several of the complexities of machine language. Thus, the software designer works with less complexity, using the assembler to add the necessary details and build the final machine instructions. The introduction of functional-level program languages (such as FORTRAN then, and C++ now) provided even more productivity improvements by providing a set of programming statements for functions that would otherwise require many machine or assembler instructions to implement. Thus, the level at which the programmer could now work was even higher, allowing him/her to construct programs with far fewer statements. The analog in IC design is the progression of transistor-level design (machine level), to gate-level design (assembler level), to HDL-based design. Synthesis provided the basis for HDL-based design and its inherent productivity improvements, and major changes to IC design methodologies.

14.2.3.1 Synthesis

Fundamentally, synthesis is a three-step process:

- Compile an HDL description of a design into an equivalent NAND–NOR description
- Optimize the NAND–NOR description based on design targets
- Map the resulting NAND–NOR description to the technology building blocks (cells) supported for the wafer foundry (process) to be used

Although work on synthesis techniques has occurred on and off since the beginnings of design automation and back to Transistor–Transistor Logic (TTL)-based designs, it was not until gate array style ICs reached a significant density threshold that it found production use. In the late 1970s and 1980s, considerable research and development in industry and universities took place on the high-speed algorithms and heuristics for synthesis. In the early 1980s, IBM exploited the use of synthesis on the ICs used in the 3090 and AS400 computers. These computers used chips that were designed from qualified sets of predesigned functions interconnected by personalized wiring. These functions (now called cells) represented the basic building blocks for each specific IC family. The computers used a high number of uniquely personalized chip designs so it was advantageous to design at the HDL level and use automation to compile to the equivalent cell-level detail. The result was significant improvements to the overall design productivity.

Today, synthesis is a fundamental cornerstone of the design methodology (Figure 14.9) for ASICs supporting both VHDL and Verilog as the standard input. As the complexities of IC design have increased, so have the challenges for synthesis. Early synthesis had relatively few constraints to be observed in its optimization phase. Based on user controls, the design was optimized for minimum area, minimum fan-out (minimum power), or maximum fan-out (maximum performance). This was accomplished by applying a series of logic reduction algorithms (transforms) that provide different types of reduction and refinement [12]. Cell behavior could also be represented in primitive-logic equivalent form and a topological analysis could find matches between the design gate patterns and equivalent cells.

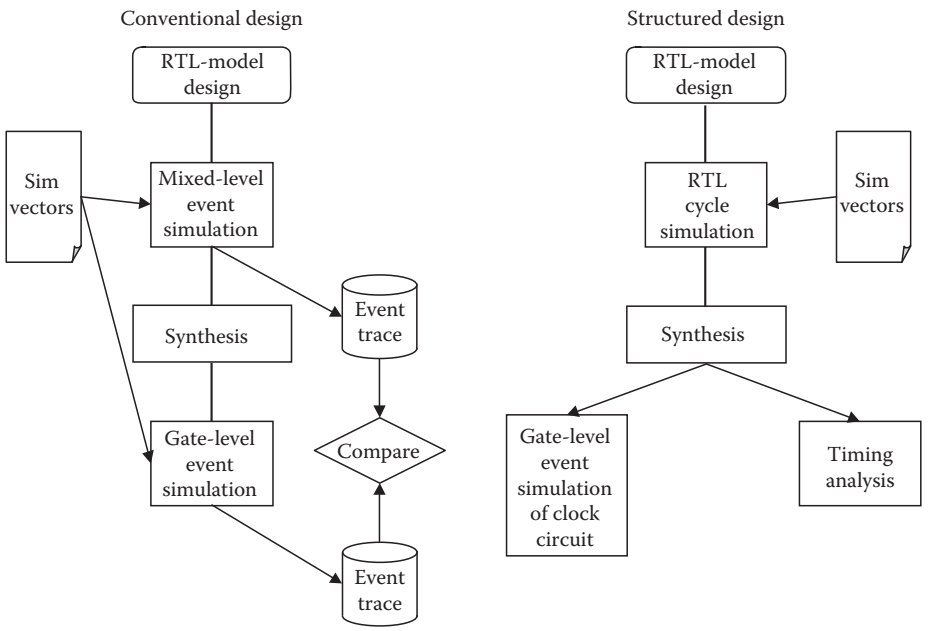


FIGURE 14.9 Synthesis methodology.

The mapping phase would then select the appropriate cells based on function and simple electrical and physical parameters for each cell in the library such as drive strength and area. Previously, synthesis was not overly concerned with signal delays in the generated design nor for other electrical constraints that the design may be required to meet. As IC feature sizes decreased, however, these constraints became critical in synthesis and automation became required to consider them in its solution.

The design input to modern synthesis tools is not the functional HDL design alone, but now includes constraints such as the maximum allowed delay along a path between two points in the design. This complicates the synthesis decision process as it must now generate a solution that meets the required function with a set of cells and interconnections that will fall within the required timing constraints. Therefore, additional trade-offs must be made between the optimization and mapping phases, and the effects of interconnection penalty need to be considered. Additionally, synthesis must now have technology characteristics available to it such as delay for cells and wiring. To determine this delay it is necessary to understand the total load seen by a driver cell as a result of the input capacitance of the cells it drives and the RC parasitics on the interconnect wiring. However, the actual wiring of the completed IC is not yet available (since routing has not taken place) and estimates for interconnect parasitics must be made. These are often based on wire load models that are created from empirical analysis of other chip designs in the technology. These wire load models are tables that provide an estimate of interconnect length and parasitic values based on the fan-out and net density.

As interconnect delay became a more dominant portion of path delay it became necessary to refine the wire load estimations based on more than total net count for the chip. More and more passes through the synthesis-layout design loop became necessary to find a final solution that achieved the required path delays. The need for a new design tool in the flow became apparent as this estimation of interconnect delay was too coarse. The reason is that different regions on a typical chip have different levels of real estate (block and net) congestion. This means that wire load estimates differ across different regions on the chip because the average amount of wiring length for a net with a given pin count differs with the region. Conversely, the actual placement of cells on the chip effect the congestion and therefore, the resulting wire load models. Consequently, a new tool, called floor planning, was inserted in the flow between synthesis and layout.

14.2.3.2 Floor Planning

The purpose of floor planning is to provide a high-level plan for the placement of functions on the chip, which is used to both refine the synthesis delay estimations and direct the final layout. In effect, it inserts the designer into the synthesis-layout loop by providing a number of automation and analysis functions used to effectively partition and place functional elements on the chip. As time progressed, the number of analysis and checking functions integrated into the floor planner grew to a point where today, it is more generally thought of as design planning [13]. The initial purpose of the floor planner was to provide the ability to partition the chip design functions and develop a placement of these partitions that optimized the wiring between them. Optimization of the partitioning and placement may be based on factors such as minimum wiring or minimum timing across a signal path. A typical scenario for the use of a design planner is as follows:

- Run synthesis on the RTL description and map to the cell level.
- Run the design planner against the cell-level design with constraints such as path delays and net priorities.
  - Partition and floor plan the design, either automatically or manually, with the design planner.
  - Create wire load models based on the congestion within the physical partitions and empirical data.
- Rerun synthesis to optimize the design at the cell level using the partitioning, global route, and wire load model data.

User-directed graphics and autoplacement-wiring software algorithms are used to place the partitions and route the interpartition (global) nets. One key to the planner is the tight coupling of partitioning and routing capability with electrical analysis tools such as power analysis, delay calculation, and timing analysis. Checking of the validity of a design change (such as placement) can be made immediately on that portion of the design that changed. Another key is that it needs to be reasonably tightly connected to synthesis, as it is typical to pass through the synthesis-planner-synthesis loop a number of times before reaching a successful plan for the final layout tools.

After a pass in the floor planner, more knowledge is available to synthesis. With this new knowledge, the optimization and mapping phases can be rerun against the previous results to produce a refined solution. The important piece of knowledge now available is the gate density within different functions in the design (or regions on the chip). With this, the synthesis tool can be selective about which wire load table to use and develop a more accurate estimate of the delay resulting from a specific solution choice.

Floor planning has been a significant enhancement to the design process. In addition to floor planning, modern design planners include support for clock tree, design power, bus design, I/O assignment, and a wealth of electrical analysis tools. As will be discussed later, semiconductor technology trends will place even more importance on the design planner and dictate an ever-tightening integration between it with both synthesis and PD. Today, synthesis, design planning, and layout are three discrete steps in the typical design flow. Communication between these steps is accomplished by file-based interchange of the design data and constraints. As designs become more dense and with the growing complexities of electrical analysis required, it will become necessary to integrate these three steps into one tight process as are the functions within the modern design planners.

14.2.4 The 1990s: The Age of Integration

Before EDA tools were commercially available, software architecture standards could be established to assure smooth integration of the tools into complex EDA systems supporting the entire design flow. Choice of languages, database and file exchange formats, host hardware and operating systems, and user interface conventions could be made solely by the developing company. Moreover, the need for comprehensive EDA systems supporting the entire flow through design and manufacturing release in a managed process is of paramount importance. Design of ICs with hundreds of thousands of gates (growing to hundreds of millions) across a design team requires gigabytes of data contained within thousands of files. All these data must be managed and controlled to assure its correctness and consistency with other design components. Design steps must be carried out and completed in a methodical manner to assure correctness of the design before release to manufacturing. The EDA system that encompasses all of this must be efficient, effective, and manageable. This was always a challenge for EDA system integrators, but the use of EDA tools from multiple external vendors compounded the problem. EDA vendors may develop their tools to different user interfaces, database and file formats, computer hardware, and operating systems. In turn, it is the EDA system integrator who must find a way to connect them in a homogeneous flow that is manageable, yet provides the necessary design efficiency across the tools. By the late 1980s it was said that the cost to integrate an EDA tool into the in-house environment was often over twice that of the tool itself. Consequently, the CAD Framework Initiative (CFI) was formed whose charter was to establish software standards for critical interfaces necessary for EDA tool integration. Thus began the era of the EDA Framework.

The EDA Framework [14] is a layer of software function between the EDA tool and the operating system, database, or user. For this software layer CFI defined a set of functional elements, each of which had a standard interface and provided a unique function in support of communication between different tools. The goal was to provide EDA system integrators with the ability to choose EDA tools from different vendors and be able to plug them into a homogeneous system to provide an effective system operation across the entire design flow. The framework components included

- Design information access—a formal description of design elements and a standard set of software functions to access each, called an application program interface (API)
- Design data management—a model and API to provide applications the means to manage concurrent access and configuration of design data
- Intertool communication (ITC)—an API to communicate control information between applications and a standard set of controls (called messages)
- Tool encapsulation specification (TES)—a standard for specifying the necessary information elements required to integrate a tool into a system such as input files and output files
- Extension language (EL)—specification of a high-level language that can be used external to the EDA tools to access information about the design and perform operations on it
- Session management—a standard interface used by EDA tools to initiate and terminate their operation and report errata
- Methodology management—APIs used by tools so that they could be integrated into software systems to aid the design methodology management
- User interface—a set of APIs to provide consistency in the graphical user interface (GUI) across tools

Each of these functional areas needs to be considered for integration of EDA tools and to provide interoperability between them. In its first release, CFI published specifications for design representation (DR) and access, ITC, TES, and EL. Additionally, the use of MOTIF and X-Windows as specified as the GUI toolkit. However, for many practical, business, and human nature reasons, these standards did not achieve widespread adoption across the commercial EDA industry and the ''plug-and-play'' motto of CFI was not realized. However, the early work at CFI did serve to formalize the science of integration and communication, and a number of the information models developed for the necessary components are used today, but often, slightly modified and within a proprietary interface.

As learned, it was not only expensive to change EDA tools to use a different internal interface or data structures, but it was also not considered good business to allow tools to be integrated into commercial system offerings from competitors. For a period, several framework offerings became available, but these quickly became the product end in themselves rather than a means to an end. These products were marketed as a means to bring outside EDA tools into the environment, but few vendors modified their tools to support this by adopting the CFI specifications. Instead, to meet the need for tool integration, the commercial industry focused on data flow only and chose an approach that provides communication between tools by standard data file formats and sequential data translation. Consequently, EDA flows of the 1990s were typically a disparate set of heterogeneous tools stitched together by the industry-standard data files. These files are created from one tool and translated by another tool (within the design flow) to its internal data structures. Many of these formats evolved from proprietary formats developed within commercial companies that were later transferred or licensed for use across the industry. Common EDA file formats that evolved were

- Electronic data interchange format (EDIF)—netlist exchange (EIA → IEC)
- Physical data exchange format (PDEF)—floor plan exchange (Synopsys → IEEE)
- Standard delay file (SDF)—delay data exchange (Cadence Design Systems → IEEE)
- Standard parasitic exchange file (SPEF)—parasitic data exchange (Cadence Design Systems → IEEE)
- Library exchange file (LEF)—physical characteristics information for a circuit family (Cadence → OpenEDA)
- Data exchange format (DEF)—physical data exchange (Cadence → OpenEDA)
- Library data format (.lib)—ASIC cell characterization (Synopsys)
- ASIC library format (ALF)—ASIC cell characterization (Open Verilog International)
- VISIC high-level design language (VHDL)—behavioral design description language (IEEE)
- Verilog—RTL design description language (Cadence Design Systems → IEEE)

These "standard" formats have provided the desired engineering and business autonomy necessary to get adoption and their use across the industry has improved the ability to integrate tools into systems drastically compared with the 1980s. It has proved to be an effective method for the design of today's commodity ICs above 0.25 μm. However, the inherent ambiguities, translation requirements, rapidly increasing file sizes (owing to increasing IC density), and the fundamental nonincremental sequential nature of design flows based on them will soon give way to IC technology advances.

14.3 The Future

It may be said that the semiconductor industry is the most predictable industry on earth. Whether it was an astute prediction or the cause, the prediction of Gordon Moore (now known as Moore's Law) has held steadily over the past two decades. Every 18 months, the number of transistors or bits on an IC doubles. To accomplish this, feature sizes are shrinking, spacing between features is shrinking, and chip die sizes are increasing. In addition, the fundamental physics that govern the electrical properties of these devices and between them has been documented in electrical engineering textbooks for some time. However, there are points within this progression where paradigm shifts occur in the elemental assumptions and models required to characterize these circuits, and where fundamental EDA design and analysis must change. Earlier, we discussed paradigm shifts in DV because of the inability to repair. We also discussed shifts resulting from the number of elements in a design versus the necessary tool performance and designer productivity. Through the past three decades, we have seen shifts in test, verification, design abstraction, and design methodologies. When the 0.25 μm node was passed, another paradigm shift was required. Feature packing, decreased rise times, increased clock frequencies, increased die sizes, and the explosion of the number of switching transistors are all interacting to place new demands of models, tools, and EDA systems. After that and including the present, new challenges came about in design tools, rules, systems, and schools for problematic areas of delay, signal integrity, power, test, and manufacturability.

14.3.1 International Technology Roadmap for Semiconductors

SEMATECH periodically publishes a report called the *International Technology Roadmap for Semiconductors* [15] (ITRS). This report is jointly sponsored by the European Semiconductor Industry Association, Japan Electronics and Information Technology Industries Association, Korea Semiconductor Industry Association, Taiwan Semiconductor Industry Association, and the Semiconductor Industry Association. The objective of the ITRS "is to ensure advancements in the performance of integrated circuits." Thus ITRS characterizes planned semiconductor directions and advances across the next decade and the necessary technology advancements in areas including

- Design and test
- Process integration devices and structures
- Front-end (manufacturing) processes
- Lithography
- Interconnect
- Factory integration
- Assembly and packaging
- Environment, safety, and health
- Defect reduction
- Metrology
- Modeling and simulation

ITRS provides a wealth of information on the semiconductor future as well as the problematic areas that will arise and areas where inventive new technology will be required. Some relevant ITRS-projected

TABLE 14.1 2004 ITRS Microprocessor Roadmap

Characteristic	2001	2004	2007	2010	2013	2016
Transistor gate length (nm)	90	53	35	25	18	13
Chip size (mm^2)	310	310	310	310	310	310
Million transistors/mm^2	0.89	1.78	3.57	7.14	14.27	28.54
Million transistors/chip	276	553	1106	2212	4424	8848
Wire pitch (nm), metal	350	210	152	104	76	54
Clock frequency (GHz) (on-chip)	1.684	4.171	9.285	15.079	22.980	39.683
Supply voltage (V) (low power)	1.1	0.9	0.8	0.7	0.6	0.5
Maximum power (W) (high-performance)	130	158	189	198	198	198

semiconductor advancements for MPU/ASIC ICs are shown in Table 14.1 and compared with previous ITRS numbers for year 2001. The reader is encouraged to study the technology characteristics in the ITRS, as the following is just a summary of certain points for the purpose of illustration.

To understand some of the implications of the ITRS data, it is convenient to review the effects of scaling on a number of electrical parameters [16]. *Gate delay* is a function of the gate capacitance and transistor resistance. The gate capacitance ($C = k\varepsilon_o A/T_{gox}$) is a function of the gate area (A), and the gate-oxide thickness (T_{gox}). Assuming ideal scaling (all features scale down in direct proportion to the change in transistor size), gate capacitance scales in direct proportion to transistor-size scaling (ΔSize) since

$$\Delta C_{gate} \propto \left(\Delta W_{gate} \Delta L_{gate}\right)/\Delta T_{gox}$$

therefore

$$\Delta C_{gate} \propto \Delta \text{Size}$$

The transistor resistance (R_{tr}) is proportional to supply voltage and inversely proportional to current both of which scale down proportionally, so the R_{tr} is constant. Thus, the gate delay (D_{gate}) decreases in direct proportion to ΔSize since

$$D_{gate} = R_{tr} C_{gate},$$

therefore

$$\Delta D_{gate} \propto 1^{*}\Delta \text{Size} = \Delta \text{Size}$$

Thus, gate delays scale down in proportion to feature size. However, the effects of interconnect delay must be considered to understand the impact of scaling on timing. For $R_{int} < R_{tr}$ interconnect effects have little affect on signal path timing. As R_{int} increases, however, delay analysis must also take into account the interconnect parasitics.

Interconnect delay is a function of the *RC*-time constant, which is a result of resistance and distributed capacitance of the interconnect. Assuming ideal scaling, the interconnects' cross-sectional area scales in direct proportion to ΔSize.

Interconnect resistance is proportional to the length and inversely proportional to the cross-sectional area of the interconnect. Since

$$R_{int} \propto L_{int}/(W_{int} H_{int}),$$

therefore

$$\Delta R_{\text{int}} \propto L_{\text{int}}/\Delta\text{Size}^2$$

where, L_{int}, H_{int}, and W_{int} are the length, height, and width of the interconnect, respectively.

The self-capacitance ($C_{\text{self}} = k\varepsilon_o A/t$) of the interconnect is proportional to its plate area ($A = W_{\text{int}}\, L_{\text{int}}$) and inversely proportional to dielectric layer thickness (t). Assuming ideal scaling for t,

$$\Delta C_{\text{self}} \propto \Delta L_{\text{int}}\Delta\text{Size}/\Delta\text{Size}$$

therefore

$$\Delta C_{\text{self}} \propto \Delta L_{\text{int}}$$

The resulting interconnect RC constant, therefore, scales as

$$\Delta R_{\text{int}}\Delta C_{\text{self}} \propto \left(\Delta L_{\text{int}}/\Delta\text{Size}^2\right)(\Delta L_{\text{int}}) = \Delta L_{\text{int}}^2/\Delta\text{Size}^2$$

For *local nets* the interconnect length (L_{int}) scales with ΔSize thus, the RC constant remains relatively constant

$$\Delta(R_{\text{int}}C_{\text{self}})_{\text{local}} \propto \Delta L_{\text{int}}^2/\Delta\text{Size}^2 = \Delta\text{Size}^2/\Delta\text{Size}^2 = 1$$

Local interconnect delay does not scale down in proportion to gate delay on successive technology generations. Therefore, its relative impact on the overall timing across a signal path is increasing in importance.

For *global nets*, the interconnect length does not scale down with the feature size. Here, the die size (S_{die}) is the more predominant determinant. Therefore, the RC constant for global nets scales as

$$\Delta(R_{\text{int}}C_{\text{self}})_{\text{gobal}} \propto \Delta L_{\text{int}}^2/\Delta\text{Size}^2 = 1/\Delta\text{Size}^2$$

Global interconnect delay increases in indirect proportion to gate delay for successive technology generations. Thus, global interconnect routing and delay analysis has become a critically important area for EDA.

For complete analysis on interconnect delay, the cross-coupling effect of closely spaced features on the chip must also be considered. Capacitance results whenever two conducting bodies are charged to different electric potentials and may be viewed as the reluctance of voltage to build or decay quickly in response to injected power. The self-capacitance between the interconnect surfaces and ground is only a part of the overall capacitance to be considered. Another important consideration is the mutual capacitance between interconnects in close proximity. To get an accurate measure of the total capacitance along any interconnect it is necessary to consider all conducting bodies within a sufficiently close proximity. This includes not only the ground plane, but also adjacent interconnects as there is a mutual capacitance between these. This mutual capacitance is a function of interconnect sidewall area, the distance between it and adjacent interconnects, and the dielectric constant of the dielectric separating them.

Figure 14.10 depicts three equally spaced adjacent interconnect wires on a signal plane. An approximate formula for the capacitance [17] of the center wire is

$$C = \text{self-capacitance} + \text{mutual capacitance} = C_{\text{int-ground}} + 2C_{\text{m}}$$

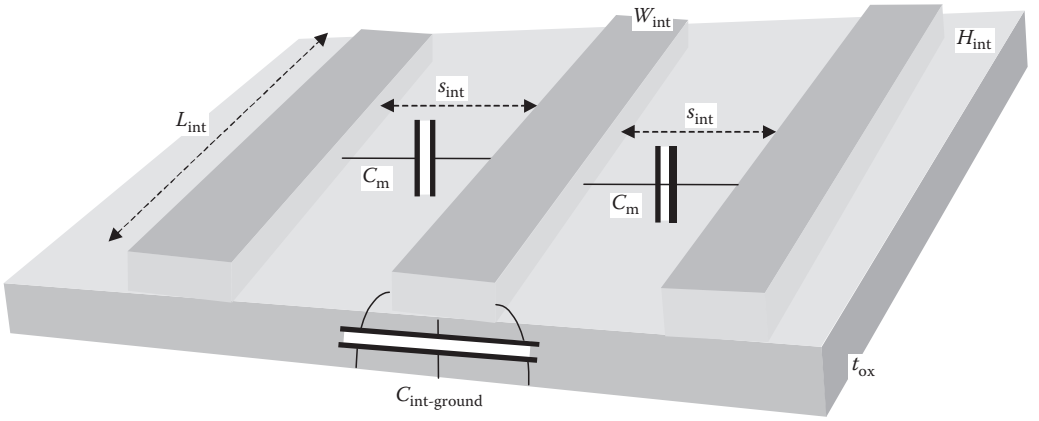


FIGURE 14.10 Mutual capacitance.

The mutual capacitance is a function of the plate area along the sides of the interconnects and the spacing between them and may be simply approximated as

$$C_m = k\varepsilon_o L_{int} H_{int} / S_{int}$$

where S_{int} is the distance between the interconnects.

Therefore (for a constant k-value), the change in mutual capacitance as a function of ideal scaling is given by

$$\Delta C_{m(Local)} \propto \Delta Size^2 / \Delta Size = \Delta Size \quad \text{and} \quad \Delta C_{m(Global)} \propto \Delta Size / \Delta Size = 1$$

Thus, mutual capacitance for local interconnects scales with feature size, while for global interconnects, it does not. Because of increased frequencies, electrical cross talk effects between interconnects must be considered. Further, mutual capacitance between only adjacent interconnects may not be complete. Full analysis may require consideration of all interconnects within a sphere around the interconnect being analyzed or even on wiring layers above or below.

Noise is the introduction of unwanted voltage onto a signal line, and is proportional to mutual capacitance between them and inversely proportional to the rate at which the voltage potential between them changes. Therefore, because the full effect of mutual capacitance is also a function of the voltage potential between interconnects (Miller effect) analysis of the simultaneous signal transitions on each becomes an important consideration for delay. Mutual capacitance injects current into adjacent (victim) signal lines proportional to the rate of change in voltage on the exciting (aggressor) signal line. The amount of induced current can be approximated by the following equation [18]:

$$I_{mC} = C_m \left(dV_{aggressor} / dt \right)$$

The amount of noise induced onto the victim net is proportional to the resistance on the victim net and the mutual capacitance, and inversely proportional to the rise time (τ_r) of the aggressor waveform:

$$V_{mC} = I_m Z_{victim} = C_m \left(\Delta V_{aggressor} / \tau_r \right) R_{victim}$$

Noise can increase or even decrease [19] the signal delay on the victim net. Voltage glitches induced can also give rise to faulty operation in the design (e.g., if the glitch is of sufficient amplitude and duration to cause a latch to go to an unwanted logic state.

Inductance can also cause noise when large current changes occur in very short times according to the following relationship:

$$V_L = L(dI/dt)$$

Sudden changes in large amounts of current can be seen on the power grid resulting from a large number of circuits switching at the same time. The inductance in the power grid and this large current draw over short periods of time (di/dt) results in a self-induced electromagnetic force whose voltage is equal to Ldi/dt. This induced voltage causes the power supply level to drop, resulting in a voltage glitch that is proportional to the switching speed, the number of switching circuits, and the effective inductance in the power grid.

14.3.2 EDA Impact

Scaling effects have impact on EDA beyond delay and timing, and the list of DF*x* (design-for-*x*) topics is increasing beyond design-for-test and design-for-timing (closure). Scaling down V_{dd} decreases power for a transistor, but the massive scaling up of total transistors on the IC dictates extensive analysis of high currents and leakage (design for power). Lithographic limitations (discussed later) now require consideration of diffraction physics (design for manufacturability [DFM]). Further, while the design complexities owing to electrical effects such as these are increasing, the number of circuits on the IC continues to increase. The fundamental challenge for EDA is more analysis on many more circuit elements with no negative effect on design cycle times.

14.3.2.1 EDA System Integration

It is now necessary to use EDA applications with increased accuracy and scope for electrical analysis to assure that designs will meet intended specifications and to integrate EDA design flows in ways that contain design cycle times. EDA systems must now take a data-centric view of the design process as opposed to the tool-centric view of the past. Data must be organized and presented in ways that allow EDA to exploit: abstraction and hierarchy, shared access by design tools, incremental processes (whereby only portions of a design that have changes need be reanalyzed), and concurrent execution of design and analysis tools within the design flow.

EDA systems have evolved or are evolving toward integration and database technology that meets the above requirements to one degree or another. EDA systems have evolved from vendor-specific sets of design and analysis applications supporting data exchange between custom, often proprietary, file formats to systems that support interchange between vendors through industry-standard files. Over the past decade they have moved toward systems of applications communicating through vendor-specific integrated database technology supporting intersystem exchange via the same industry-standard files. Because of the increasing need for new and additional EDA applications companies performing IC design continue to require the ability to use EDA applications from many vendors and develop flows that provide efficient and complete intervendor integration. With increasing feature counts and complexities for successive technology generations, this dictates the need to reduce the overhead and possible data loss caused by file-based interchange (Figure 14.11). In 2000, a multicompany effort under Si2 was initiated to develop an industry-open data model that could be used by commercial EDA companies and for university research and (EDA customer) proprietary EDA development. This effort resulted in the publication of an EDA data model specification, which includes an application program software interface, and development of a production-quality *reference* database that is compliant with that specification. Collectively, this specification and database is called OpenAccess (see www.si2.org) and was made available to the industry in 2003 on a royalty-free basis. OpenAccess has an excellent chance of being the foundation on which the next point of EDA systems' evolution will be founded. At that

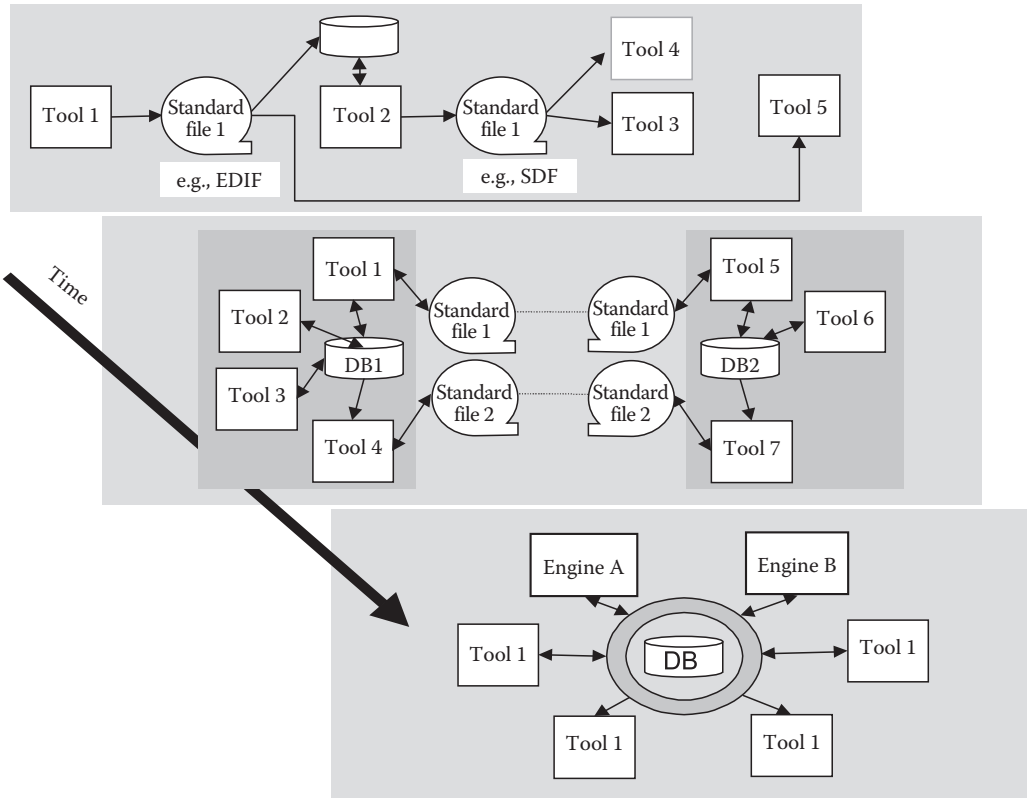


FIGURE 14.11 EDA system architecture progression.

point, highly efficient EDA flows will be built using EDA tools from multiple sources and integrated around a single database meeting the fundamental requirements mentioned above.

14.3.2.2 Delay

Simplifying assumptions and models for delay have provided the foundation for high-speed event-driven delay simulation since its initial use in the 1970s. More accurate waveform simulation such as that provided by SPICE implementations has played a crucial role in the characterization of IC devices during over three decades and continues to do so. SPICE-level simulations are important for characterization and qualification of ASIC cells and custom macros of all levels of complexity. However, the run times required for this level of device modeling are too large to support its use across the entire ICs. SPICE is often used to analyze the most critical signal paths, but SPICE-level simulation on circuits with hundreds of millions of transistors is not practical today. Consequently, simulation and STA at the abstracted gate-level, or higher, is essential for IC timing analysis. However, simplifying models used to characterize delay for discrete-event simulation and STA have become more complex as feature sizes on ICs have shrunk, and the importance of interconnect resistance and cross talk increased. In the future, these models will need to improve even more (Figure 14.12).

When ICs were considerably >1 μm, gate delay was the predominant factor in determination of timing. Very early simulation of TTL used a simple model which assigned a fixed unit of delay to each gate and assumed no delay across the interconnects. Timing was based only on the number of gates through which a signal traversed along its path. As LSI advanced delay was based on a foundry-specified gate delay value (actually a rise delay and a fall delay) that was adjusted based on the capacitive load of

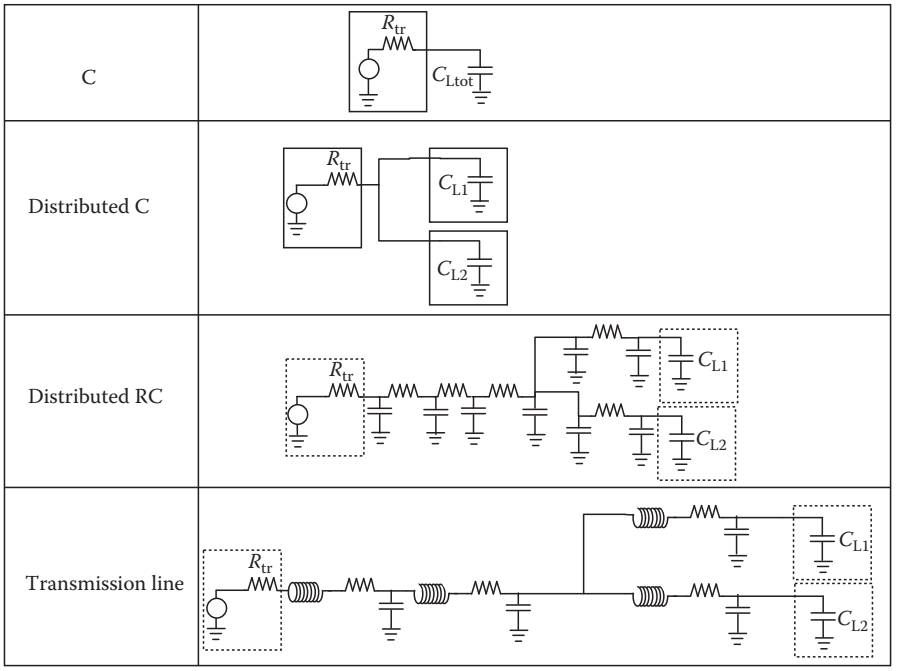


FIGURE 14.12 Delay models.

the gates it fanned out to (receiver gates). At integration levels above 1 μm, the load seen by a gate was dominated by the capacitance of its receivers, so this delay model was sufficiently accurate. As feature sizes crossed below 1 μm, however, the parasitic resistance and capacitance along the interconnects became a significant factor. More precise modeling of the total load effect on gate delay and interconnect delay had to be taken into account. By 0.5 μm the delay attributed to global nets almost equaled that of gates, and by 0.35 μm the delay attributed to short nets equaled the gate delay.

Today, a number of models are used to represent the distributed parasitics along interconnects using lumped-form equations. The well-known π-model, for example, is a popular method to model these distributed RCs. Different degrees of accuracy can be obtained by adding more sections to the equivalent-lumped RC model for interconnect.

As the importance of timing-driven design tools grows, integration of multiply sourced design tools into an EDA flow becomes problematic. As more EDA vendors use custom built-in models for computation of gate and interconnect delays, the calculated values may differ across different design tools. This results in difficult and time-consuming analysis on the designers' part to correlate the different results. In mid-1994, an industry effort began to develop an open architecture supporting common delay-calculation engines [20]. The goal of this effort was to provide a standard language usable by ASIC suppliers to specify the delay calculation models and equations (or table lookups) for their technology families. In this way, the complexities of deciding appropriate models to be used to characterize circuit delay and the resulting calculation expressions could be placed in the hands of the semiconductor supplier rather than across the EDA vendors. These models could then be compiled into a form that could be directly used by any EDA application requiring delay calculation and in a way that would protect the intellectual property contained within them. By allowing the semiconductor supplier to provide a single software engine for the calculation of delay, all applications in the design flow could provide consistency in the computed results. The Delay Calculation Language (DCL) technology, originally developed by IBM Corporation, was contributed to industry as the basis for this. Today,

DCL has been extended to cover power calculation in addition to delay and it has been ratified by the IEEE as an open industry standard (Delay and Power Calculation System [DPCS], IEEE 1481–1999). For a number of technical and business reasons, adoption of this standard has failed to take hold. Although release and adoption of the Synopsys.lib format, used to provide technology parameters used by delay calculation in a standard form, greatly improved this situation, problems with accuracy and consistency of delay calculation remain. To that end, a renewed interest in developing an industry-wide solution came about in 2005 and the Open Modelling Coalition (OMC) has been formed under Si2 to readdress the challenge with industry partners.

At 0.25 μm, cross talk noise resulting from mutual capacitance between signal lines begins to have a significant effect on delay. At this technology generation, it became necessary to consider effects of mutual capacitance between interconnects. Today, sophisticated extraction tools must analyze the proximity of features (including wires, vias, and pads) and properties of dielectrics between them. In addition, EDA design and analysis applications must account for mutual parasitics and cross talk. This is an important element of layout, parasitic extraction, delay calculation, and timing analysis. Further, the development of wafer fabrication techniques such as dielectric air gaps to reduce effective *k*-values will bring further challenges to EDA.

With future technology generations, more sophisticated models that consider inductance may also become necessary. Distributed *RC* models used to characterize delay are very accurate approximations so long as the rise time (t_r) of the signals is much larger than the time-of-flight (t_{of}) across the interconnect wire. As the transistor size scales down, so do the signal rise times. As t_r approaches t_{of}, transmission line analysis may be necessary to accurately model timing. Between these points is a gray area where transmission line analysis may be necessary depending on the criticality of the timing across a particular path. Published papers [21,22] address the question of where this level of analysis is important and the design rules to be considered for time-critical global lines. For future EDA systems, this means more complexities in the design and analysis of circuits. Extraction tools will then need to derive inductance of interconnects so that the effects of the magnetic fields surrounding these lines can be factored into the delay calculations. Effects of mutual inductance also becomes important, particularly along the power lines which will drive large amounts of current in very short times ($e = L\ \mathrm{d}i/\mathrm{d}t$). Design planners and routers will need to be aware of these effects when designing global nets and power bus structures.

14.3.2.3 Test

Manufacturing test of ICs today is becoming a heuristic process involving a number of different strategies in combination to assure coverage and maximize generation costs and throughput at the tester. Today, IC test employs the use of one or more of the following approaches:

- Static stuck-at fault test using stored program testers: The test patterns may be derived algorithmically or from random patterns and the test is based on final steady-state conditions independent of arrival times. The goal of stuck-at fault test is to achieve 100% detection of all stuck-at faults (except for redundancies and certain untestable design situations that can result from the design). Algorithmic test generation for all faults in general sequential circuits is often not possible, but test generation for combinational circuits is. Therefore the use of design-for-test strategies (such as scan design) is becoming more accepted.
- Delay (at-speed) test: Patterns for delay testing may be algorithmically generated or functional patterns derived from DV. The tester measures output pins for the logic state at the specified time after the test pattern is applied.
- BIST: BISTs are generated at the tester by the device under test and output values are captured in scan latches for comparison with simulated good-machine responses. BISTs tests may be exhaustive or random patterns and the expected responses are determined by simulation. BIST requires special circuitry on the device-under-test (LFSR and scan latches) for pattern generation and result

capture. To improve tester times, output responses are typically compressed into a signature and comparison of captured results and simulated results is made only at the end of sequences of tests.

- Quiescent current test (I_{DDQ}): I_{DDQ} measures quiescent current draw (current required to keep transistors at their present state) on the power bus. This form of testing can detect faults not observable by stuck-at fault tests (such as bridging faults) that may cause incorrect system operation.

Complicated semiconductor trends that will affect manufacturing test in the future are the speed at which these future chips operate and the quiescent current required for the large density of transistors on the IC. To perform delay test, for example, the tester must operate at speeds of the device-under-test. That is, if a signal is to be measured at a time n after the application of the test pattern, then the tester must be able to cycle in n units of time. It is possible that the hardware costs required to build testers that operate at frequencies above future IC trends will be prohibitive. I_{DDQ} tests are a very effective means to quickly identify faulty chips as only a small number of patterns are required. Further, these tests find faults not otherwise detected by stuck-at fault test. Electric current measurement techniques are far more precise than voltage measurements. However, the amount of quiescent current draw required for the millions of transistors in future ICs may make it impossible to detect the small amount of excess current resulting from small numbers of faulty transistors. New inventions and techniques may be introduced into the test menu. However, at present it appears that in the future manufacturing test must rely on static stuck-at fault tests and BIST. Therefore, it is expected that more and more application of scan design will be prevalent in future ICs.

14.3.2.4 Design Productivity

The ITRS points out that future ICs will contain hundreds of millions of transistors. Even with the application of the aforementioned architectural features and new and enhanced EDA tools, it is unclear that design productivity (number of good circuits designed per person year) will be able to keep pace with semiconductor technology capability. Designing and managing anything containing over 100,000,000 subelements is almost unthinkable. Doing it right and within the typical 18-month product cycles of today seems impossible. Yet, if this is not accomplished, semiconductor foundries may run at less than full production, and the possibility of obtaining returns against the exorbitant capital expenditures required for new foundries will be low. Ultimately, this could negate the predictions in the ITRS.

Over the history of IC design, a number of paradigm shifts enhanced designer productivity, the most notable being high-level design languages and synthesis. Use of design rules such as LSSD to constrain the problem has also represented productivity advances. Algorithm advances and faster processing computers will necessarily play a crucial role, as the design cycle time is a function of computer resources required. Many of the EDA application algorithms, however, are not linear with respect to design size and processing times can increase as an exponential function of transistors. Therefore exploitation of hierarchy, abstraction, shared, incremental, and concurrent EDA system architectures will play an important role to overall productivity. Even with all of this, there is a major concern that industry will not be able to design a sufficient number of good circuits fast enough. Consequently, there is a major push in the semiconductor industry toward design reuse. There is a major activity in the EDA industry (Virtual Sockets Interface Alliance [VSIA] www.vsia.org) to define the necessary standards, formats, and test rules to make design reuse a reality with ICs.

Design reuse is not new to electronic design or EDA. Early TTL modules were an example of reuse, as are standard cell ASICs and PCB-level MSI modules. With each, the design of the component is done once, then qualified, then reused repeatedly in application-specific designs. Reuse is also common where portions of a previous design are carried forward to a new system or technology node, and where a common logical function used multiple times across the system. However, the ability to embed

semiconductor designs qualified for one process into chips manufactured on another, or for which the logical design was developed by one company to be used by another one results in new and unique challenges. This is the challenge that the VSIA is addressing.

The VSIA has defined the following three different types of reusable property for ICs:

1. Hard macros: These functions have been designed and verified, and have a completed layout. They are characterized by being a technology-fixed design and a mapping of manufacturing processes is required to retarget them to another fabrication line. The most likely business model for hard macros is that they will be available from the semiconductor vendor for use in application-specific designs being committed to that supplier's fabrication line. In other words, hard macros will most likely not be generally portable across foundries except in cases where special business partnerships are established. The complexities of plugging a mask-level design for one process into another process line are a gating factor for further exploitation at present.
2. Firm macros: These can be characterized as reusable parts that have designed down to the cell level through partitioning and floor planning. These are more flexible than hard macros since they are not process dependent and can be retargeted to other technology families for manufacture.
3. Soft macros: These are truly portable design descriptions, but are only completed down through the logical design level. No technology mapping or PD is available.

To achieve reuse at the IC level, it will be necessary to define or otherwise establish a number of standard interfaces for design data. The reason for this is that it will be necessary to integrate design data from other sources into the design and EDA system being used to develop the IC. VSIA was established to determine where standard interfaces or data formats are necessary and choose the right standard. For soft macros these interfaces will need to include behavioral descriptions, simulation models, and timing models. For firm macros the scope will additionally include cell libraries, floor plan information, and global wiring information. For hard macros GDSII may be supplied.

To reuse designs that were developed elsewhere (hereafter called intellectual property [IP] blocks), the first essential requirement is that the functional and electrical characteristics of the available IP blocks be available. Since internal construction of firm and hard IP may be highly sensitive and proprietary, it will become more important to describe the functional and timing characteristics at the I/Os. This will drive the need for high-level description methods such as use of VHDL behavioral models, DCL, I/O Buffer Information Specification (IBIS) models, and dynamic timing diagrams. Further, the need to test these embedded IP blocks will mandate more use of scan-based test techniques such as Joint Test Action Group (JTAG) boundary scan. Standard methods such as these for encapsulating IP blocks will be of paramount importance for broad use across many designs and design systems.

There are risks, however, and reuse of IP will not cover all design needs. Grand schemes for reusable software yielded less than desired results. Extra effort to generalize the IP block design points for broad use, and describe its characteristics in standard formats is compounded by the ability to identify an IP block that fits a particular design need. The design and characterization of reusable IP will need to be robust in timing, power, reliability, and noise in addition to function and cost. Further, the broad distribution of IP information may conflict with business objectives. Moreover, even if broadly successful, use of reusable IP will not negate the fundamental need for major EDA advances in the areas described. First, tools are needed to design the IP. Second, tools are needed to design the millions of circuits that will interconnect IP. Finally, tools and systems will require major advances to accurately design and analyze the electrical interactions between, over, and under IP and application-specific design elements on the IC. Standards are essential, but they are not by themselves sufficient for success. Tools, rules, and systems will be the foundation for future IC design as they have been over the past. Nevertheless, the potential rewards are substantial and there is an absence of any other clear EDA paradigm shift. Consequently, new standards will emerge for design and design systems, and new methods for characterization, distribution, and lookup of IP will become necessary.

14.3.2.5 DFM

14.3.2.5.1 Lithography

Because of the physical properties of diffraction, as printed features shrink, the ability to print them with the required degree of fidelity becomes a challenge for mask making. The scaling effects for lithographic resolution can be generally viewed using the Rayleigh equation for a point light source:

$$\text{Smallest resolvable feature} = K(\lambda/\text{NA})$$

Here

Value of K is a function of the photoresist process used
λ is the wavelength of the coherent light source used for the exposure
NA is the numerical aperture of the lens system

As feature shrink continues, the ITRS projects that design wafer foundry lithographic systems will keep pace by moving to light sources of smaller wavelength, photoresist systems with lower values for K, and immersion lithography. (Immersion lithography increases the maximum achievable numerical aperture ($\text{NA} = I \sin a$) beyond that in air (in air, the refractive index (I) = 1, thus $\text{NA}_{\max} = 1$), because of higher refractive indices for water and other fluids). However, the physics of light diffraction is now affecting the design of photomasks, and this will dramatically increase in importance with future technology generations.

Without attempting to discuss the physics behind them, two complications need to be accounted for in the design of a mask. First, the intensity of the light source passing through the mask and onto the photoresist is nonlinear. That is, the intensity at the edges of small features is less than that in the center. Thus, edges of small features do not print with fidelity. To compensate for this, for example on the corners of lines (which will expose as rounded edges) and line ends (which will be rounded and short), a series of reticle (mask) enhancement techniques are used. These techniques add subresolution features to the pattern data. Some examples are the addition of serifs on corners and hammerheads on line ends to extend as well as square them.

The second complication is the interference of diffracted light between adjacent features. That is, the light used to expose densely packed features may interfere with the exposure of other features in close proximity. This may be corrected for by phase-shift mask techniques or be accounted for by adjusting feature placement (spreading densely packed areas and using scatter bars to fill sparsely packed areas) and biasing feature widths to adjust for the interference.

Though these corrections are necessary for two distinct physical properties, they are generally collectively performed by optical proximity correction (OPC) programs. OPC in today's design systems is generally performed a step after the design is complete and ready for tape-out. The desired design pattern data is topographically analyzed by OPC against a set of rules or models that determine if and what corrections need to be made in the mask pattern to assure wafer fidelity. These corrections result in the incorporation of additional features to be cut into the mask such as serifs, hammerheads, and scatter bars.

With successive technology generations, the number of OPCs for a mask set will increase and cause mask pattern data to increase beyond that of Moore's Law for features. This, in turn, causes increased time (and resulting costs) in capital-intensive mask manufacturing. Therefore, future design systems will need to consider lithography limitations early in the design process (synthesis and PD) rather than at the end, and new OPC methods to optimize the mask pattern volume will be required.

14.3.2.5.2 Yield

Traditionally, wafer yield has been viewed as the responsibility of the wafer foundry alone. Hard defects caused by random particles or incorrect process settings are under control of the foundry manager and defects in feature parametrics were sufficiently bounded by design rules to be insignificant. As features shrink, small variations in print fidelity or process parameters have an increasingly important impact on

feature parasitics and this can be a significant factor in the yield of properly operating chips. Traditionally, parasitic extraction performed on the PD-generated geometry was sufficient to analyze a parasitic effect on, for example, timing. In future, because of lithography limitations, the analysis of parasitics may require extraction based on the simulated geometries that are projected to print on the wafer. Traditionally, EDA design applications can operate within the specific bounds of the design rules to yield good design. Future design applications may require that statistical analysis of their decisions be performed to account for intrachip parametric variations and to provide a design known to be good within a certain degree of probability. The topic of yield will become an important aspect of the next EDA paradigm shift.

Future design system flows will require a closer linkage between the classical EDA flow and Technology Computer Aided Design (TCAD) systems and mask making. In addition, the need for rich models that can be used to accurately predict the impacts of design choices on the results of foundry and mask manufacturing processes will become important. New design tools such as Statistical Static Timing Analysis (SSTA) have become available and are a topic of much discussion. How to incorporate the effects of variability on yield in, for example PD and synthesis will be a topic for future research. High-speed lithographic simulation, capable of full chip analysis, will be an important field for EDA R&D. Finally, database and integration technology to support this increased level of collaboration between IC designers, mask manufacturers, and the fabricating engineers will be critical.

14.4 Summary

Over the past four decades, EDA has become a critical science for electronics design. Where it may have begun as a productivity enhancement, it is now a fundamental requirement for the design and manufacture of electronic components and products. The criticality of EDA for semiconductors is the focus of much attention because of the expanding semiconductor advancements. Fundamental approaches in test, simulation, and PD are being constantly emphasized by these semiconductor advancements. New EDA disciplines are opening up. Fundamental electrical models are becoming more important and at the same time, more complex. New problems in design and IC failure modes will surely surface.

The next decade of EDA will prove to be as challenging and exciting as its past!

References

1. Case, P.W., Correia, M., Gianopulos, W., Heller, W.R., Ofek, H., Raymond, T.C., Simel, R.L., and Stieglitz, C.B., Design automation in IBM, *IBM Journal of Research and Development*, 25(5): 631, 1981.
2. Roth, J.P., Diagnosis of automata failures: A calculus and a method, *IBM Journal of Research and Development*, 10: 278, 1966.
3. Eichelberger, E.B., Hazard detection in combinational and sequential switching circuits, *IBM Journal of Research and Development*, 9: 90, 1965.
4. Eichelberger, E.B. and Williams, T.W., A logic design structure for LSI testability, *Proceedings of the 14th Design Automation Conference*, New Orleans, LA, pp. 462–468, June 1977.
5. IEEE Std 1149.1-1990, *IEEE Standard Test Access Port and Boundary-Scan Architecture*, IEEE, Washington, 1990.
6. Hitchcock, R.B., Cellular wiring and the cellular modeling technique, *Proceedings of the 6th Annual Design Automation Conference*, ACM, New York, pp. 25–41, 1969.
7. Lee, C.Y., An algorithm for path connection and its applications, *IRE Transactions on Electronic Computers*, 10(1): 364–365, 1961.
8. Hightower, D.W., The interconnection problem: A tutorial, *Proceedings of the 10th Annual Design Automation Workshop*, IEEE Press Piscataway, NJ, pp. 1–21, 1973.
9. *1076-1993 IEEE Standard VHDL Language Reference Manual*, IEEE, Washington, 1993.

10. *1364-2005 IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language*, IEEE, Washington, 2006.
11. Electronic Industry Associates, *Electronic Design Interchange Format (EDIF)*, IHS, www.globalihs.com.
12. Stok, L., Kung, D.S., Brand, D., Drumm, A.D., Sullivan, A.J., Reddy, L.N., Heiter, N., et al., BooleDozer: Logic synthesis for ASICs, *IBM Journal of Research and Development*, 40(4): 407, 1996.
13. Sayah, J.Y., Gupta, R., Sherlekar, D.D., Honsinger, P.S., Apte, J.M., Bollinger, S.W., Chen, H.H., et al., Design planning for high-performance ASICs, *IBM Journal of Research and Development*, 40(4): 431, 1996.
14. Barnes, T.J., Harrison, D., Newton, R.A., and Spickelmier, R.L., *Electronic CAD Frameworks*, Kluwer Academic, Dordrecht, the Netherlands, Chapter 10, 1992.
15. Semiconductor Industry Association, *International Technology Roadmap for Semiconductors, Technology Needs, 2001 Edition*, www.sematech.org.
16. Bakoglu, H.B., *Circuits, Interconnections, and Packaging for VLSI*, Addison-Wesley, Reading, MA, Chapters 1–7, 1990.
17. Goel, A.K., *High-Speed VLSI Interconnections, Modeling, Analysis & Simulation*, Wiley, New York, Chapter 2, 1994.
18. Johnson, H.W. and Graham, M., *High-Speed Digital Design, A Handbook of Black Magic*, Prentice-Hall PTR, New York, Chapter 1, 1993.
19. Pandini, D., Scandolaro, P., and Guardiani, C., Network reduction for crosstalk analysis in deep submicron technologies, *ACM/IEEE International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, Austin, TX, pp. 280, December 1997.
20. Cottrell, D.R., Delay calculation standard and deep submicron design, *Integrated Systems Design*, July 1996.
21. Fisher, P.D., Clock cycle estimates for future microprocessor generations, *IEEE 1997 ISIS: International Conference on Innovative Systems in Silicon*, Austin, TX, pp. 61–71, October 1997.
22. Deutsh, A., Kopcsay, G.V., Restle, P., Katopis, G., Becker, W.D., Smith, H., Coteus, P.W., et al., When are transmission-line effects important for on-chip interconnections, *Electronic Components and Technology Conference*, San Jose, CA, pp. 704, 1997.