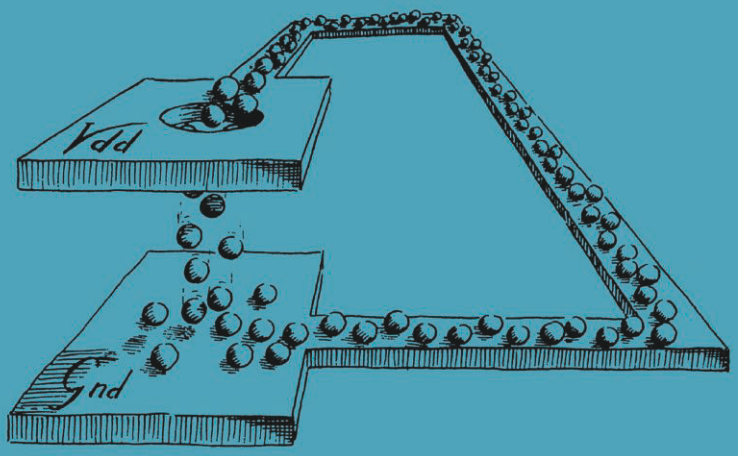
Dynamic Power Management

Design Techniques and CAD Tools

Luca Benini
Giovanni De Micheli



Springer Science+Business Media, LLC

INFINEON 1017

DYNAMIC POWER MANAGEMENT
Design Techniques and CAD Tools

DYNAMIC POWER MANAGEMENT
Design Techniques and CAD Tools

by

Luca Benini
Stanford University

and

Giovanni De Micheli
Stanford University

SPRINGER SCIENCE+BUSINESS MEDIA, LLC

ISBN 978-1-4613-7491-6 ISBN 978-1-4615-5455-4 (eBook)
DOI 10.1007/978-1-4615-5455-4

Library of Congress Cataloging-in-Publication Data

A C.I.P. Catalogue record for this book is available
from the Library of Congress.

Copyright © 1998 by Springer Science+Business Media New York
Originally published by Kluwer Academic Publishers in 1998
Softcover reprint of the hardcover 1st edition 1998
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, mechanical, photo-copying, recording, or otherwise, without the prior written permission of the publisher, Springer Science+Business Media, LLC.

Printed on acid-free paper.

To Natasha, Micol,
Marie-Madeleine, Andrea,
Gabrielle

Contents

Preface xi

Acknowledgments xiii

1. INTRODUCTION 1
   1.1 Motivation 2
   1.2 Sources of power consumption 4
   1.3 Design techniques for low power 8
      1.3.1 Power minimization by frequency reduction 9
      1.3.2 Power minimization by voltage scaling 10
      1.3.3 Power optimization by capacitance reduction 14
      1.3.4 Power optimization by switching activity reduction 16
      1.3.5 Revolutionary approaches 18
   1.4 CAD techniques for low power 20
      1.4.1 Circuit-level techniques 21
      1.4.2 Logic-level techniques 25
      1.4.3 Behavioral-level techniques 30
      1.4.4 System-level techniques 33
   1.5 Power management 37

2. BACKGROUND 41
   2.1 Introduction 42
   2.2 Boolean algebra and finite-state machines 42
      2.2.1 Boolean algebra 42
      2.2.2 Discrete functions 47
      2.2.3 Finite-state machines 47
   2.3 Implicit representation of discrete functions 50
      2.3.1 Binary decision diagrams 51
      2.3.2 Algebraic decision diagrams 54
   2.4 Markov analysis of finite-state machines 56

2.4.1 Explicit methods 59
2.4.2 Implicit methods 61
2.5 Conclusion 63

3. FUNDAMENTAL CONCEPTS 65
3.1 Introduction 66
3.2 Fundamentals 66
3.2.1 External idleness 68
3.2.2 Internal idleness 70
3.2.3 Taxonomy 72
3.2.4 Granularity 73
3.2.5 Input statistics 75
3.3 Power management circuitry 76
3.3.1 Power supply shutdown 76
3.3.2 Clock gating 78
3.3.3 Enabled flip-flops 81
3.4 Conclusion 83

4. POWER MANAGEMENT FOR CONTROL UNITS 85
4.1 Introduction 86
4.2 Top-down synthesis 87
4.2.1 Gated-clock FSMs 88
4.2.2 Mealy and Moore machines 89
4.2.3 Problem formulation 90
4.2.4 Locally-Moore machines 91
4.2.5 Optimal activation function 93
4.2.6 The overall procedure 95
4.2.7 Experimental results 96
4.2.8 Summary 101
4.3 Top-down decomposition 102
4.3.1 Previous work on FSM decomposition 103
4.3.2 Interacting FSM structure 104
4.3.3 Clock gating 108
4.3.4 Partitioning 110
4.3.5 Partitioning algorithm 114
4.3.6 Experimental results 117
4.3.7 Summary 121
4.4 Conclusion 121

5. POWER MANAGEMENT FOR FUNCTIONAL UNITS 123
5.1 Introduction 124
5.2 Clock distribution design 126
5.3 Logic-level techniques 129

5.3.1 Exploiting internal idleness 130
5.3.2 Exploiting external idleness 138
5.3.3 Experimental results 140
5.3.4 Summary 143
5.4 RTL techniques 144
5.4.1 Detection of hold conditions 144
5.4.2 Unobservability detection 148
5.4.3 Glitch minimization 152
5.4.4 Summary 155
5.5 Behavioral-level techniques 155
5.5.1 Operand isolation and controller re-specification 156
5.5.2 Memory segmentation 160
5.5.3 Scheduling 162
5.5.4 Summary 163
5.6 Conclusion 164

6. POWER MANAGEMENT FOR SYSTEMS 167
6.1 Introduction 168
6.2 Operating system-based power management 170
6.2.1 Operating system support 173
6.2.2 Device driver support 174
6.2.3 Application support 176
6.2.4 The *OnNow* architecture 180
6.3 Advanced configuration and power management 180
6.3.1 ACPI state definitions 183
6.3.2 Device power management 187
6.3.3 Battery management 191
6.3.4 Thermal management 192
6.4 Modeling power management 194
6.4.1 Model definition 196
6.4.2 Power states and transitions 197
6.4.3 System behavior 199
6.4.4 Stochastic simulation 202
6.5 The policy optimization problem 205
6.5.1 Disk power management 206
6.6 Conclusion 210

7. CONCLUSION 211
7.1 Summary 212
7.2 Prospects 214

References 215

Index 229

Preface

There has been a growing interest in techniques for designing integrated circuits and systems with low-power consumption in the last few years. Research in this field is motivated by two major markets: portable electronics and high-performance systems. In the former case, low-power circuits are needed to provide a reasonable operation time to battery-operated devices. In the latter case, environmental factors, such as heat dissipation, may pose a practical limit to the use of high-performance processors if power consumption is not controlled and limited.

Low-power consumption in integrated circuits and systems can be achieved through the combination of different techniques, including architectural design choices, logic and physical design, choice of circuit families and implementation technology. Most power saving is achieved by ingenious architectural organization, even though computer-aided design (CAD) techniques have been shown to be effective in synthesizing low-power circuits.

This book presents design techniques and synthesis algorithms for circuits and systems with reduced power consumption. In addressing the multi-faceted aspects of low-power design, we have chosen *dynamic power management* as a leitmotiv, because we think that most power savings can stem from using power management in the system architectural design as well as for the detailed logic design of its hardware component. Therefore this book is not merely a collection of synthesis algorithms. Power minimization algorithms are described in the context of power management, and other techniques, such as software-controlled power management, are also described.

Dynamic power management is a design methodology aiming at controlling performance and power levels of digital circuits and systems, with the goal of extending the autonomous operation time of battery-powered systems, pro-

viding graceful performance degradation when supply energy is limited, and adapting power dissipation to satisfy environmental constraints.

This book addresses design techniques and computer-aided design solutions for power management, which are organized in an order related to their applicability to control-units, macro-blocks, digital circuits and electronic systems, respectively. All approaches are based on the principle of exploiting idleness of circuits, systems, or portions thereof. They involve both detection of idleness conditions and the freezing of power-consuming activities in the idle components.

Sequential circuits, such as control-units, display often a large degree of idleness due to their reactive nature. Techniques based on clock gating and/or on decomposition have been shown to be successful in reducing their power consumption. These methods have been extended to cope with power management in networks representing either data path or control. The power reduction solutions merge seamlessly with circuit synthesis, which is used routinely for circuit design.

The book describes also some approaches to system-level power management, including Microsoft's *OnNow* architecture and the *Advanced Configuration and Power Management* standard proposed by Intel, Microsoft and Toshiba. These approaches migrate power management to the software layer running on hardware platforms, thus providing a flexible and self-configurable solution to adapting the power/performance tradeoff to the needs of mobile (and fixed) computing and communication.

Acknowledgments

We gratefully acknowledge several people who helped to make this book a reality with their work and advise. We are deeply in debt to our co-workers in the Digital Systems CAD Group at Stanford. Thanks to Claudionor Coelho, Hema Kapadia, Giuseppe Paleologo, Polly Siegel, Frederick Vermeulen and Patrick Vuillod. We thank the friends at the Politecnico di Torino: Massimo Poncino and Riccardo Scarsi. We thank the friends at the Università di Bologna: Michele Favalli and Bruno Riccò. We also thank the friends at Synopsys, especially Ivailo Neldechev, Jim Sproch and Bernd Wurth. Special thanks to Alessandro Bogliolo, who also drew the cover of this book, and Enrico Macii, two good friends and excellent co-workers.

Of fundamental importance was the economic support provided by NSF under contract MIP-942119 and by Toshiba Corp. Our gratitude also goes our parents and to all the friends that made life at Stanford an unforgettable human and professional experience: Jan, Ulrich, Belbo and the friends of the "Associazione Italiani at Stanford". We finally would like to thank our wives. We are grateful for their encouragement and support, their patience and understanding.

DYNAMIC POWER MANAGEMENT
Design Techniques and CAD Tools

1 INTRODUCTION

Olim nescio quid sit otium, quid quies, quid denique illud iners quidem, iucundum tamen, nihil agere, nihil esse.

I do not know what is rest, what is tranquillity, what is that state of doing nothing, being nothing, certainly with no activity.

—Plinius Secundus

1

1.1 MOTIVATION

A rule of thumb for *Very-Large Scale of Integration* (VLSI) engineers known as "Moore's Law" [moor96] states that the number of transistors per chip is expected to double every year. Despite numerous adverse predictions, "Moore's Law" has held for the last 35 years, although recently the rate has slowed to about 1.5 times per year, or to quadrupling every three years [mein95]. In the dominant CMOS technology, the increase in integration comes with numerous beneficial effects. Transistors become faster (the delay of a ring oscillator stage in $0.1\mu m$ technology with $1.0V$ supply voltage is less than $5ps$) and performance increases. Commercially available microprocessors run with clock speed exceeding 500MHz and contain more than 9 million transistors [alpha96, expo96]. Processors with clock frequency close to 1GHz are expected to be announced in the next two to three years.

While top-of-the line microprocessors provide impressive computational power and lead the way in addressing the formidable challenges of *Ultra-Large Scale of Integration* (ULSI) design, less aggressive products target the rapidly expanding market of portable electronic devices for personal communication, automotive systems, biomedical and many other applications. In all these cases, power consumption is a primary concern. In high-performance systems, excessive power dissipation increases failure rate, environmental impact and cost imposed by cooling systems and packaging. Reducing power consumption is obviously the primary concern for portable applications, since battery technology cannot keep the fast pace imposed by Moore's law, and there is large demand for devices with light batteries and long time between recharges.

The design of electronic circuits with low power dissipation is an old art. Several micro-power design techniques were introduced in the 1970's and commercially exploited in the first low-power applications: electronic wristwatches and implantable units for biomedical applications [bult96]. Although the basic issues are unchanged, the designers of today's low-power systems are faced with a much more complex task: power must be minimized while maintaining high performance. To further complicate the problem, the pressure for fast time-to-market has become extremely high, and it is often unacceptable to completely re-design a system merely to reduce its power dissipation.

The power dissipation of digital VLSI components is only a facet of a much more complex problem. Analog or electro-mechanic components such as displays, RF transmitters or disk drivers are often the main contributors to inflating power budgets. The rapid evolution of CMOS technology seems to indicate that the digital component of electronic devices is likely to remain a primary source of concern. However, well-balanced and successful designs cannot overlook any system component. For example, the power consump-

tion of notebook computers is currently dominated by display power consumption [harr95]. Hence, low-power and high-quality display technology plays a key role in deciding the success of a notebook computer, while digital CMOS circuitry is less critical.

Historically, designers have relied on *Computer-Aided Engineering* (CAE) and *Computer-Aided Design* (CAD) tools to cope with the complexity of the design of electronic systems. Reliable tools for the estimation of critical design metrics, such as area and speed, have been developed in the past. They were followed by the appearance of synthesis tools which have partially automated most of the design tasks, enabling an astonishing increase in designer productivity. Power should be no exception. After a few years of extensive investigation in the academic community, the first analysis and synthesis tools for low power have appeared on the market [keut96]. Nevertheless, the problem is far from solved. On the contrary, the most contradictory claims have been made on the impact and usefulness of CAD tools for low power.

Without a doubt, power optimization is still much more an art than a well understood industrial practice, and CAE support for low power designers is only rudimental. Probably the reason for this situation is that there is only a partial understanding of the fundamental power reduction techniques. In this book we focus on *dynamic power management*, which, in conjunction with technology optimization, has been consistently successful in reducing the power dissipation of electronic systems. The book is not only a collection of ideas and techniques. Our goal is to formulate a general theory and to build a taxonomy for categorizing current and future techniques for dynamic power management. We also perform an in-depth investigation of computer-aided design techniques.

We first describe the rationale and the theoretical foundations of the power reduction approach based on dynamic power management. Then, we explore in greater detail various dynamic power management techniques at multiple levels of abstraction, ranging from logic-level to system-level. Many of the techniques we present are emerging from the research community, while others are fostered by the industrial world, and accepted in the design practice.

Although many approaches are still at the level of hand-crafted solutions that critically depend on designer ingenuity and skills, some degree of computer-aided automation can be achieved. The book contains useful information for the CAD engineer who is interested in designing a new generation of tools for low-power design. We present several algorithmic approaches for the automation of basic and advanced power management techniques, and we attempt to critically assess their impact. Our treatment is purposely informal, in an attempt to increase the accessibility of the material to designers and engineers that focus more on the basic ideas than on the algorithmic details. Nevertheless, extensive bibliographical references are provided for the interested CAD practitioner.

Power management is based on the key concept of *idleness exploitation.* Whenever a device (or a component of a device) is idle, its power dissipation should be reduced as much as possible. A successful power management scheme exploits idleness in an efficient way. First, it detects idleness with high accuracy, second it rapidly shuts down the idle resource and forces it in a state where power dissipation is as low as possible. Whenever the resource exits the idle state, the transition from shut down to fully active should be as fast and power-efficient as possible. Moreover, the hardware support for power management should be compact, power efficient and fast. Obviously, implementing power management schemes involves a complex tradeoff between power, area and speed. For all techniques presented in the book, we will analyze advantages and limitations in an attempt to give the reader deeper understanding and complete information.

This chapter is dedicated to a brief review of the main phenomena that cause power dissipation in CMOS circuits, followed by an overview of the basic power minimization techniques. Our treatment is not exhaustive. Several books have recently been published on low-power design and CAD [chan95, raba96, nebe97, mont97] and they should be referred to for additional information. The last part of the chapter focuses on the main contribution of this book, namely the analysis of power management techniques at all level of abstraction and the study of their applicability.

1.2 SOURCES OF POWER CONSUMPTION

As power dissipation becomes a high-priority cost metric, researchers and designers have increased their efforts in understanding its sources and in minimizing its impact. Power dissipation is not constant during the operation of a digital device. The *peak power* is an important concern. Excessive peak power may cause a circuit to fail because of electro-migration and voltage drops on power and ground lines. Fortunately, correct and reliable circuit operation can be ensured by designing for worst-case conditions. On the other hand, the *time-averaged* power consumption is inversely proportional to the battery lifetime. Hence, minimization of average power consumption is a key issue for the success of numerous electronic products, and it is the primary focus of the following treatment.

The average power dissipation in a CMOS circuit can be described by a simple equation that summarizes the four most important contributions to its final value

$$P_{avg} = P_{dynamic} + P_{short} + P_{leakage} + P_{static} \tag{1.1}$$

The four components are respectively *dynamic* ($P_{dynamic}$), *short-circuit* (P_{short}), *leakage* (P_{lk}) and *static* (P_{static}) power consumption. The partition of P_{avg}

among its component strongly depends on the application and the technology. We analyze each contribution in detail, using a simple combinational static CMOS gate as a motivating example. Dynamic circuits and sequential gates show similar behavior.

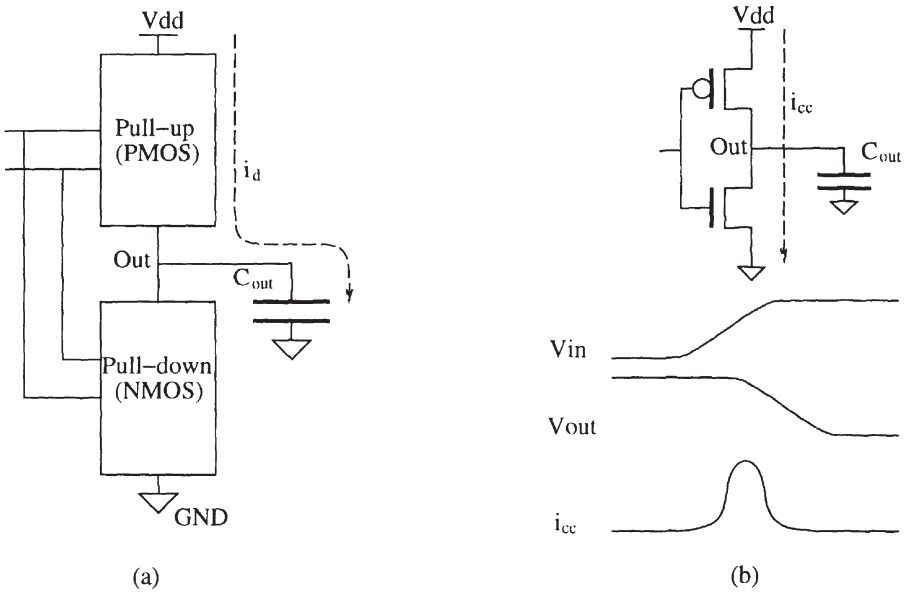


Figure 1.1. CMOS gate structure and power dissipation

Dynamic power consumption, $P_{dynamic}$, is the power consumed during the output switching of a CMOS gate. Figure 1.1 (a) shows the structure of a generic static CMOS gate. The *pull-up* network is generally constructed with PMOS transistors and connects the output node *Out* to the power supply *Vdd*. The *pull-down* network is generally composed of NMOS transistors and connects the output node to the ground node *GND*. In a CMOS gate, the structure of the pull-up and pull-down networks is such that when the circuit is stable (i.e. the output rise or fall transients are exhausted) the output is never connected with a conductive path to both *Vdd* and *GND* at the same time.

When a transition on the inputs causes a change in the conductive state of the pull-up and the pull-down network, electric charge is transferred from the power supply to the output capacitance C_{out} or from the output capacitance to ground. The transition causes power dissipation on the resistive pull-up and pull-down networks. Let us consider a rising output transition. Power is by

definition $P_{dynamic}(t) = dE(t)/dt = i_d(t)v(t)$, where $i_d(t)$ is the current drawn from the supply and $v(t)$ is the supply voltage ($v(t) = Vdd$). The total energy provided by the supply is:

$$E_r = \int_0^{T^r} i_d(t)v(t)dt = V_{dd}\int_0^{V_{dd}} C_{out}dV_{out} = C_{out}V_{dd}^2 \tag{1.2}$$

where T^r is a time interval long enough to allow transient exhaustion. We implicitly assumed that all current provided by V_{dd} is used to charge the output capacitance. We also simplistically assumed that the output capacitance is constant (i.e. it is not voltage-dependent).

At the end of the transition, the output capacitance is charged to V_{dd}, and the energy stored in it is $E_s = 1/2C_{out}V_{dd}^2$. Hence, the total energy dissipated during the output transition is $E_d = C_{out}V_{dd}^2 - 1/2C_{out}V_{dd}^2 = 1/2C_{out}V_{dd}^2$. If we now consider a falling transition, the final value of the output node is 0, and the output capacitance stores no energy. For conservation of energy, the total energy dissipated during a falling transition of the output is again $1/2C_{out}V_{dd}^2$. This simple derivation leads us to the fundamental formula of average dynamic power consumption:

$$P_{dynamic} = K\frac{C_{out}V_{dd}^2}{T} = KC_{out}V_{dd}^2 f \tag{1.3}$$

where T is the clock period of the circuit and $f = 1/T$ is the clock frequency. The factor K is the average number of transitions of the output node in a clock cycle divided by two. Setting $K = 1/2$ is equivalent to assuming that the gate performs a single transition every cycle. Clearly, in any digital circuit the clock cycle is much longer than the time for a gate transition. Hence, a single gate may have multiple transitions in any given clock cycle. On the other hand, the output of a gate may not switch at all during a clock cycle. Equation 1.3 is important mainly because it includes the most important parameters influencing power dissipation, namely supply voltage, capacitance switched, clock frequency and the average number of output transitions per clock cycle.

Figure 1.1 (b) illustrates the origin of the short circuit power dissipation P_{short}. While in deriving $P_{dynamic}$ we assumed that all charge drawn from the power supply is collected by the output capacitance, this is not the case in realistic digital circuits. Since the inputs have finite slope, or, equivalently, the input rise/fall time $t_{r/f}$ is larger than 0, the pull-down and the pull-up are both on for a short period of time. During this time, there is a resistive path between power and ground and some current is drawn from the supply and flows directly to ground. We call this current *short-circuit current*. The total current drawn from V_{dd} is therefore $i(t) = i_d(t) + i_{short}(t)$. The following formula was

proposed to describe the short circuit power dissipation of an inverter with no external load (the analytical derivation of the formula under several simplifying assumptions is carried out in [veen84]):

$$P_{short} = \frac{\beta}{12}(V_{dd} - 2V_T)^3 \tau f \tag{1.4}$$

where β is the gain factor of a MOS transistor, V_T is its threshold voltage and τ is the rise (or fall) of the input of the inverter. The analysis in [veen84] shows that P_{short} depends on the ratio between the transit time of the output and the transit time of the input, the worst case being slow input edges and fast output edges. Although the P_{short} of a single gate is minimized for very fast input edges and slow output edges, the best design point for cascade of gates is when the transit times of all gate outputs are kept roughly constant [veen84].

Several authors observed that short-circuit power dissipation is usually a small fraction (around 10%) of the total power dissipation in "well-designed" CMOS circuits. The rationale for the observation is that P_{short} becomes sizable when a gate is driven by an excessively loaded driver which generates slow transitions at its input. This situation is generally avoided in circuits designed for high performance. As a consequence, it is reasonable to expect that traditional design techniques for high performance lead to circuits where short-circuit power dissipation is not a major concern.

The third component of the total power dissipation in Equation 1.1 is $P_{leakage}$, the power dissipated by leakage currents. Leakage power is mainly caused by two phenomena: i) *diode leakage* current due to the reverse saturation currents in the diffusion regions of the PMOS and NMOS transistors; ii) *sub-threshold leakage* current of transistors which are nominally off [chan95]. Both currents have an exponential dependence on the voltage: diode leakage depends on the voltage across the source-bulk and drain-bulk junctions of the transistors, while sub-threshold current depends on both the voltage across source and drain and across gate and source.

Diode leakage is an important concern for circuits that are in standby mode for a very large fraction of operation time and it is usually reduced by adopting specialized device technologies with very small reverse saturation current. Sub-threshold leakage is becoming increasingly important because of reductions in power supply. As power supply voltages decrease, the transistor threshold is lowered to keep turned-on transistors well within the conductive region of operation. Consequently, transistors operating in a non-conductive region are turned off only weakly, and conduct some current even in their "OFF" state.

In today's VLSI circuits $P_{leakage}$ is still a small fraction (less than 10%) of the total power dissipation. Reductions in $P_{leakage}$ is achieved mainly through device technology improvements (diffusion region engineering and threshold

control), and by enforcing stricter design rules. It may be possible, however, that specialized design techniques for minimizing $P_{leakage}$ may be required as power supply voltages continue to decrease.

The last component in Equation 1.1 is the static power dissipation, P_{static}, caused by DC current flow from V_{dd} to GND when the pull-up and pull-down are both conducting and the gate output is not transitioning. Correctly designed CMOS circuits do not have static power dissipation, and it is fair to say that the absence of static power in nominal conditions dissipation is probably the most important distinctive characteristic of the CMOS technology. Unfortunately P_{static} may become non-null in faulty circuits. Circuits where $P_{static} \neq 0$ must be detected and discarded because: i) if present, P_{static} becomes the major contributor to the total power dissipation; ii) static current in CMOS circuits is often associated with incorrect or unpredictable functional behavior. As an example of a faulty circuit with $P_{static} \neq 0$ consider the inverter of Figure 1.1 (b) and assume that the gate of the PMOS transistor is connected to GND (i.e., *stuck at zero*). When the input is high, both PMOS and NMOS transistors are conducting and current flows from V_{dd} to GND even if the input is stable.

Summarizing the discussion on the contributions to power dissipation in CMOS circuits, we conclude that the dominant fraction (around 80%) of P_{avg} is attributed to $P_{dynamic}$, the dynamic power dissipation caused by switching of the gate outputs. The reader should refer to the detailed survey by Chandrakasan [chan95] for more information. The vast majority of power reduction techniques concentrate on minimizing the dynamic power dissipation by reducing one or more factors on the right hand side of Equation 1.3. In the next section we will consider each controlling variable in greater detail, and give a brief overview of the techniques currently employed by designers to reduce power dissipation.

1.3 DESIGN TECHNIQUES FOR LOW POWER

We focus in this section on *synchronous CMOS circuits*. Simply stated, we postulate the existence of a clock signal that has no functional value except for its use in synchronization. Such an assumption is implicit in Equation 1.1, where term f represents clock frequency. Assuming the presence of a clock signal with extremely large load and very high switching activity is a questionable choice for low power design. We will elaborate on this observation in the last part of the section.

Practical reasons dictate our choice of focusing on synchronous circuits. First, by far the vast majority of digital designs are synchronous. Second, designers and engineers are well-accustomed to the synchronous design methodol-

ogy and flow. Third, many basic techniques developed for synchronous circuits are applicable to general digital systems.

1.3.1 Power minimization by frequency reduction

The most obvious way to reduce power consumption seems to be to decrease the clock frequency f. Decreasing the clock frequency causes a proportional decrease in power dissipation. However, in digital systems we are interested in performing a given task (e.g., adding a finite stream of numbers). Slowing the clock merely results in a slower computation and not in effective savings for that task. The power consumption over a given period of time is reduced, but the total amount of useful work is reduced as well. In other words, the energy dissipated to complete the task has not changed.

Assume that we want to perform the task with a portable battery-operated system. Assume the system is clocked with a clock period T_1, and the task takes N clock cycles to complete. During each cycle, the system dissipates an average power P_1. If we now decrease the frequency in half, we will dissipate $P_2 = 1/2P_1$, over the original time period, because average power is directly proportional to the clock frequency. However, it now takes a total time of $2NT_1$ to complete the task. As a consequence, the average energy consumed by the system is $E = P_1NT_1$ in both cases. It is true that we consumed less power per cycle with the slower frequency, but we *had to operate the system for a longer time* to execute the same task.

Under the assumption that the total amount of energy provided to complete the task is a constant, decreasing the clock frequency has negative consequences, because it merely increases the time needed to complete the given task. This observation has been often reported in the literature [chan95, burd95].

For portable systems, we are interested in power reduction as a way to maximize battery life. Recent studies [mart96] have shown that the total amount of energy provided by actual batteries is *not* a constant, but depends on the rate of discharge of the battery. Thus, the frequency of operation comes back into play here. According to an empiric equation known as *Peukert's formula* we have [mart96]:

$$C = \frac{\chi}{I^{\alpha}} \tag{1.5}$$

where C is the total energy that can be drawn from a battery (also know as the *energy capacitance*), χ is a technology-dependent constant (a characteristic of the particular type of battery used), I is the average discharge current, and α is a technology-dependent fitting factor. For typical $NiCd$ batteries, for instance, α ranges between 0.1 and 0.3. The most important consequence of Equation 1.5 is that if we decrease the discharge current, we can actually increase the total amount of energy that is provided by the battery. In other words, there may

be some advantage in reducing the clock frequency, because batteries are more proficiently utilized when the discharge current is small.

Although this interesting and somewhat counterintuitive observation may open a new avenue of research for portable systems where clock frequency is reduced (thereby reducing the average current per clock cycle) to maximize the energy capacitance, there are still other important factors that limit the impact of power optimization techniques based on clock frequency reduction. One factor is the constraint on *peak performance*. For many digital systems such as microprocessors, peak performance is used for comparisons against the competitor's products when running benchmark programs and it is related to the user waiting time which is subject to hard limits.

Even if peak performance is not the primary objective, a very large fraction of digital systems are throughput-constrained. In order for the system to meet the design specifications, a given number of computations per second must be performed. This kind of specification is typical of signal processing systems in which the sampling rate is often decided by high-priority system-level constraints. For both peak-performance-constrained and throughput-constrained systems, clock frequency reduction is not a viable alternative for power optimization. Since such systems are the vast majority of VLSI applications implemented today, clock frequency control is used only in conjunction with other techniques to achieve power savings [chan95].

1.3.2 Power minimization by voltage scaling

Voltage scaling is the most effective way to reduce power consumption. This is apparent from Equation 1.3, since $P_{dynamic}$ has quadratic dependence on the power supply voltage. A large body of research has been devoted to voltage scaling for power reduction. The most complete work in the area is the pioneering research of Chandrakasan and Brodersen, summarized in [chan95].

In CMOS, reducing supply voltage causes the circuit to run slower. The delay of a CMOS inverter can be described by the following formula [chan95]:

$$T_d = \frac{C_{out}V_{dd}}{I} = \frac{C_{out}V_{dd}}{\eta(W/L)(V_{dd} - V_t)^2} \tag{1.6}$$

where η is a technology-dependent constant, W and L are respectively the transistor width and length, and V_t is the threshold voltage. Many simplifying assumptions are made in the derivation of Equation 1.6. The most important assumptions are: i) the current through the MOS transistor is well fitted by the quadratic model and ii) during the transient, the device controlling the charge (discharge) of the output capacitance is in saturation.

Unfortunately, deep sub-micron devices such as those used in modern VLSI systems are *velocity saturated* and are not modeled correctly by the simple

quadratic model. A MOS transistor is said to be velocity saturated if no improvement in transit time of the electrons through the conductive channel can be obtained by increasing the drain-source voltage. Equation 1.6 should not be regarded as an accurate analytical model of gate delay (not even for a simple inverter) because it assumes that T_d can be arbitrarily reduced by increasing V_{dd}.

Nevertheless, the equation is important because it contains the variables on which gate delay actually depends, and the nature of their effect is correctly represented. In other words, T_d increases with C_{out} and L/W, and it strongly depends on the voltage supply and the threshold voltage.

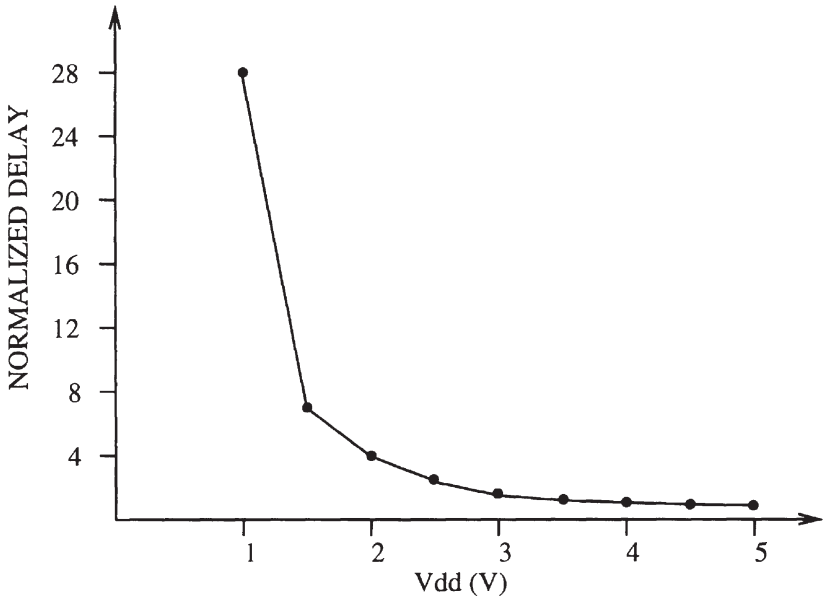


Figure 1.2. Gate delay versus supply voltage

We can study T_d as a function of the voltage supply with all other parameters fixed. A plot depicting the functional dependency of delay from V_{dd} is shown in Figure 1.2. Two main features emerge from the analysis of the plot: i) further increasing the supply voltage above 3V has little impact on performance and ii) the speed decreases abruptly as V_{dd} gets closer to the threshold voltage V_t. The physical phenomena responsible for this behavior are respectively the velocity saturation of the MOS transistors at high V_{ds} [chan95] and the low conductivity of the channel when V_{gs} approximates V_t.

If speed decreases when we decrease the power supply, power decreases as well, and quadratically. Clearly there is no point in increasing the supply voltage beyond velocity saturation, because little, if any, performance advantage can be obtained. This straightforward observation explains the supply voltage reduction observed in CMOS circuits in the last few years. However, if power dissipation is the primary target, we may push power supply reduction in the region where some speed penalty is paid.

In the following discussion we target power reduction of a digital system with throughput constraints and a fixed cycle time T. We assume that the system was originally designed to meet the minimum throughput constraints (i.e. the slowest path of the circuit is matched to the cycle time T). If we lower the power supply, the circuit becomes slower and the computation does not complete within a single clock cycle.

However, the designer can still use voltage scaling to reduce power consumption if design modifications are made to satisfy the throughput constraints. This approach is known as *architecture-driven voltage scaling* [chan95]. The transformations employed in architecture-driven voltage scaling are based on increasing the level of concurrency in the system: more hardware is used and several tasks are performed in parallel. Typical transformations are *pipelining* and *parallelization*. However, pipelining and parallelization result in an area penalty, because they require additional hardware. Increased area in turn implies increased capacitance, which increases power dissipation. However, power decreases quadratically with V_{dd} but increases only linearly with switched capacitance. Thus, there will be a net reduction in power with these techniques.

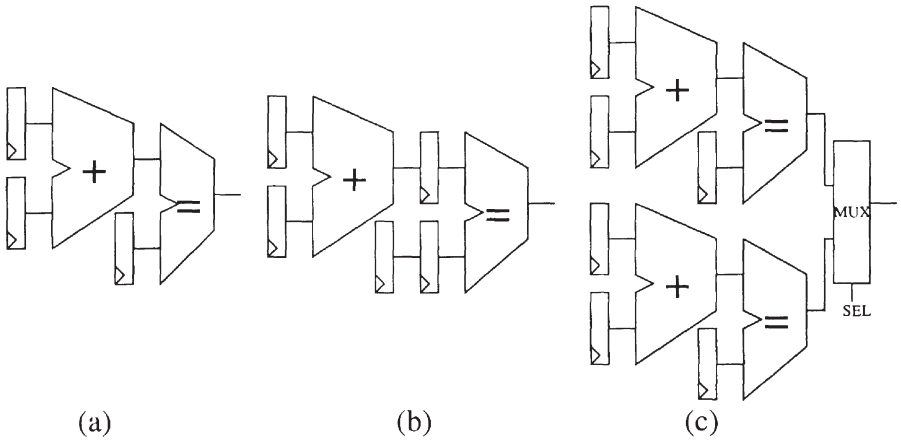


Figure 1.3. Transformation for architecture-driven voltage scaling

Example 1.1 *This example is taken from [chan95]. Consider the add-compare circuit shown in Figure 1.3 (a). The power consumed by the circuit is*

$$P_{ref} = C_{ref} V_{ref}^2 \frac{1}{T}$$

The cycle time is matched to the critical path $T = T_{add} + T_{cmp}$ *and the throughput constraint is* $1/T$ *add-compares per second. Hence, we cannot simply reduce the supply voltage. However, we can pipeline the circuit. The pipelined implementation is shown in Figure 1.3 (b). The critical path becomes* $Max\{T_{add}, T_{cmp}\} < T$. *Now we can lower the power supply until the new critical path matches* T. *Notice that the switched capacitance is increased because of the additional pipeline registers. The power consumed by the pipelined implementation is, for the particular technology and implementation style of the example [chan95]:*

$$P_{pipe} = C_{pipe} V_{pipe}^2 1/T_{pipe} = (1.15 C_{ref}) \cdot (.58 V_{ref})^2 \cdot \frac{1}{T} = .39 P_{ref}$$

Alternatively, we can utilize a parallel architecture, as shown in Figure 1.3 (c). Notice that the two parallel data-paths are clocked respectively on the odd and even clock cycles, and the multiplexer selects alternatively the result of one or the other data-path. In the parallel implementation, every add-compare circuit has $2T - T_{mux}$ *to complete the computation, and we can lower the voltage supply until the critical path matches the available time. The power dissipation of the parallel implementation is:*

$$P_{par} = C_{par} V_{par}^2 1/T_{par} = (2.15 C_{ref}) \cdot (.58 V_{ref})^2 \cdot \frac{1}{2T} = .36 P_{ref}$$

The two transformations can be combined to obtain a parallel and pipelined implementation with even better power savings (and higher hardware overhead). The power dissipation of the combined parallel and pipelined implementation is $P_{parpipe} = .2 P_{ref}$.

Obviously there is a point of diminishing return for architecture-driven voltage scaling. As V_{dd} approaches V_t, the incremental speed reduction paid for an incremental voltage reduction becomes larger. Moreover, the area cost of highly parallel implementation increases more than linearly because of the communications overhead.

Several other voltage scaling techniques for low power have been proposed (Chandrakasan's book contains an excellent survey [chan95]). However the main limitation of all voltage scaling approaches is that they assume the designer has the freedom of choosing the voltage supply for his/her design. Unfortunately, this is almost always impossible. For many real-life systems, the

power supply is part of the specification and not a variable to be optimized. Accurate voltage converters are expensive, and multiple supply voltages complicate board and system-level design and increase overall system cost. Thus, it may not be economically acceptable to arbitrarily control the supply voltage.

Even if low-cost reliable voltage converters became widely available [stra94], there is a more fundamental limitation. Device technologies are designed to perform optimally at a given supply voltage. Since sub-micron devices are velocity saturated, and there is almost no advantage in choosing a high voltage supply, the semiconductor industry is moving to low-voltage technologies (the current standard is $3.3V$ and advanced microprocessors are already operating at $2V$). In current (and future) technologies, voltage scaling will become increasingly difficult because of reduced noise margins and deterioration of device characteristics.

Although voltage scaling has limited applicability to standardized design flows based on mainstream CMOS technology, several innovative CMOS devices and circuits are under active investigation [chan95, usam95, saku96] that allow dynamic control of voltage supply and threshold voltages. If such devices and circuit techniques reach technological maturity, they may replace fixed-voltage fixed-threshold CMOS technology for low power applications. The critical assessment of the viability of these approaches is beyond the scope of this book. We will conservatively assume that standard CMOS will be the mainstream technology for the near future.

1.3.3 Power optimization by capacitance reduction

Equation 1.3 demonstrates that there is a linear dependence of $P_{dynamic}$ on capacitance. The capacitive load C_{out} of a CMOS logic gate G consists mainly of i) gate capacitance of transistors in gates driven by G, ii) capacitance of the wires that connect the gates and iii) parasitic (junction and gate-source or gate-drain) capacitance of the transistors in gate G. In symbols:

$$C_{out} = C_{fo} + C_w + C_p \tag{1.7}$$

where C_{fo} is the input capacitance of fan-out gates, C_w is the wiring capacitance and C_p is the parasitic capacitance.

The fan-out capacitance depends on the number of logic gates driven by G and the size of their transistors. The capacitance of a MOS transistor depends on the technology (more precisely, the gate oxide thickness) and the dimensions of the transistor, $C_g = WL\epsilon_{ox}/t_{ox}$, where ϵ_{ox} is the electric permittivity of the silicon oxide and t_{ox} is the oxide thickness. Since t_{ox} is set by the technology, and it is not under the designer's control, the gate capacitance can be reduced only by shrinking the dimension of the transistors. Another way to reduce the

contribution of C_{fo} is to reduce the number of fan-out gates. In technologies with channel length above $1\mu m$, C_{fo} is by far the most important component in C_{out}. Unfortunately, this is not the case for today's *deep sub-micron* devices, with channel length in the order of $.3\mu m$.

For deep sub-micron technologies, the wiring capacitance C_w is becoming the dominant component of C_{out}. It is extremely hard to estimate C_w accurately. If the circuit is manually laid out, the topology of the wires and their sizing can be decided by the designer (or at least estimated with some accuracy). Unfortunately, state-of-the art technologies have multiple levels of metal and extremely small minimum feature size and therefore wires are very close to each other. Hence, the coupling between wires is becoming the most important factor for determining the wiring capacitance. Accurate modeling of such *cross-talk* capacitance can be achieved only through expensive computations of two and three-dimensional electric field. Even a rough approximation of these effects requires a good deal of engineering ingenuity.

For automatically placed and routed circuits, the situation is even worse, because the designer does not know what the wire's topology and sizing will be. The wiring capacitance can be estimated *after* placement and routing, but it is not clear how the knowledge of the wiring capacitance after placement and routing can be exploited to reduce the impact of C_w. Currently the estimation, and worse yet, the control of wiring capacitance is a problem for which no satisfactory solution is available.

The parasitic capacitance C_p is probably the component causing the least concern, because it is well characterized and constant (to a first approximation) since it depends only on the transistors of the gate itself, and it is relatively small compared to the other two contributions.

In summary, in state-of-the art technologies, approximatively 50% of C_{out} is due to C_{fo}, 40% is due to C_w and 10% is due to C_p. The wiring capacitance already dominates C_{out} for data busses and global control wires, and will become largely dominant in the coming years.

Reducing C_{out} not only decreases power but also reduces area and increases speed. For this reason, techniques for capacitance minimization have been practiced for a long time, in practice since the birth of VLSI technology. Capacitance minimization is *not* the distinctive feature of low-power design, since in CMOS technology power is consumed only when the capacitance is switched. Focusing on reducing power by decreasing C_{out} is a tempting alternative since it allows us to exploit the mature technology for area minimization (capacitance is proportional to active silicon area).

What differentiates power optimization from capacitance minimization is the fact that we do not need to minimize capacitance if it is seldom switched. Although a minimum capacitance (i.e. minimum area) circuit has generally low

power dissipation, a minimum-power circuit does not necessarily have minimum capacitance. Pure capacitance reduction is not generally the most effective way to reduce power dissipation because it is useless in reducing capacitance when there is little activity. Moreover, as we will see later, a slight capacitance increase (i.e. the addition of some redundant circuitry) may lead to remarkable power reduction.

1.3.4 Power optimization by switching activity reduction

We can summarize the previous subsections as follows: the supply voltage V_{dd} is usually not under designer's control; the clock frequency, or more generally, the system throughput is a constraint more than a design variable; and capacitance is important only if switched. What really distinguishes power is its dependence on the switching activity (i.e. factor K in Equation 1.3). More precisely, power minimization techniques should target the reduction of the *effective capacitance*, defined as $C_{eff} = K \cdot C_{out}$. The fundamental equation of dynamic power dissipation can be rewritten as:

$$P_{dynamic} = C_{eff} V_{dd}^2 f \tag{1.8}$$

Equation 1.8 helps clarify our fundamental claim: power minimization is achieved through the reduction of C_{eff}. It is important to reiterate the assumptions behind this claim. First, $P_{dynamic}$ is the dominant factor in power dissipation. Second, V_{dd} is a technology-related parameter that cannot be directly controlled. Third, the performance (in terms of amount of work carried out in given amount of time) of the system is constrained.

The implications of Equation 1.8 have been clear to digital designers for a long time. In surveying the description of commercial chips with low power consumption, it is obvious that once a technology and a supply voltage have been set, power savings come from the careful minimization of the switching activity.

While some switching activity is *functional*, i.e. it is required to propagate and manipulate information, there is a substantial amount of *useless* activity in virtually any digital circuit. Useless switching activity is due to: i) spurious transitions due to unequal propagation delays (also known as *glitches*) and ii) transitions occuring within units that are not taking active part in a computation or whose computation is redundant. For example, the result of an arithmetic operation is useless when an exception is raised that invalidates it.

Thus, the key to designing low-power VLSI systems is to minimize the amount of switching activity needed to carry out a given task within its perfor-

mance constraints. There are many examples of effective applications of this idea.

- Dynamic power management. The *Nap* and *doze* modes of operation in portable computers [elli91, harr95, debn95, slat95] are examples of this approach to the reduction of useless switching activity. Power is reduced by stopping the clock or shutting down the power supply of parts of the system that are not required to carry out the current task. The same principle can be applied at a finer granularity: the clock distribution to a unit in a chip can be disabled at run time if the unit is not needed for a given computation. The clock is enabled as soon as the unit is required.

- Algorithmic transformations for signal processing tasks [meng95, chan95b, mehr96]. Reducing the number of operations needed to carry out a given computation may not be always useful in terms of performance (if the operations are parallelizable), but it is often useful for reducing power. In this case the functional switching activity is reduced.

- Communication protocol design [mang95]. Communication protocols can be modified to improve the activity patterns. For example, a communication protocol for pagers where a paging request may come at any time is less power efficient than a *synchronous protocol* where a paging request may come only in a periodic, short time window (called a *slot*). In the first case the receiving unit must be continuously turned on, while in the second case it must be on only during the time slots where some incoming data is expected.

- Memory allocation [pand96, gebo97] and bus encoding [stan95, meh96a, beni97b] techniques to minimize the power dissipated in memories and system busses. Again power is reduced by minimizing the cost of communication. Bus encoding reduces the switching activity on highly-loaded system busses, while power-conscious memory allocation tends to maximize the usage of local memory (registers, caches) that is inherently more power efficient than remote memory (main memory, disk).

Notice that in many cases the reduction in C_{eff} comes with a concomitant increase of C_{out}. For example, the addition of the circuitry for controlling sleep modes in a microprocessor marginally increases the area, and consequently the total capacitance. However, the C_{out} increase is more than paid off by the reduction of the switching activity. Whenever investigating some power reduction technique that implies some area overhead, we should always consider the trade off between increasing C_{out} and decreasing K, to guarantee an overall decrease in C_{eff}.

Design techniques that reduce the useful switching activity are inherently harder to apply than those targeting useless switching. Reducing useful switching is usually accomplished through algorithm redesign and optimization, a task that largely relies on human skills. In this book we will investigate in detail techniques based on dynamic power management for the reduction (or the elimination) of useless switching activity and their impact on design methodologies and flows.

1.3.5 Revolutionary approaches

Before concluding the section we briefly mention techniques for power reduction that adopt a more radical approach by removing one or more of the assumptions leading to the conclusions of Subsection 1.3.4. We call these approaches *revolutionary* to contrast with the *evolutionary* nature of the power optimization strategies previously discussed. We consider two revolutionary approaches that have been proposed in the literature in the last few years, namely asynchronous circuits and adiabatic circuits.

Asynchronous circuits [birt95] are an interesting alternative to standard synchronous circuits for low-power design. Synchronous circuits define one or more clock signals that are used to synchronize the sequential elements. Although a common clock simplifies the interface of sub-modules in complex systems, clocking circuitry is hard to design and power consuming. The C_{eff} of the clock is almost invariably the largest of the chip, since it has large switching activity and capacitance. Asynchronous design techniques eliminate the need for clock signals. Units interface through handshake signals which are activated only when necessary and do not require global synchronization. Several asynchronous chips have been designed targeting low power [niel94, mars94, gars96].

It is often claimed that asynchronous circuits are inherently more power efficient than synchronous circuits because they eliminate global synchronization signals that do not actually perform any "useful computation". Furthermore, as the size and clock speed of VLSI circuits increases, new clocking paradigms are emerging that have much in common with asynchronous circuits: sub-units of a large digital systems are clocked at different speeds and require handshaking to communicate among them.

Unfortunately, asynchronous circuits have not yet become a mainstream technology, mainly because of the lack of commercially-available computer-aided design tools to help engineers design large chips and the overhead of local handshaking signals that erode the claimed power savings. The few large asynchronous designs described in the literature have not incontrovertibly proven that there are substantial advantages with respect to synchronous designs, mainly because they were compared to functionally equivalent syn-

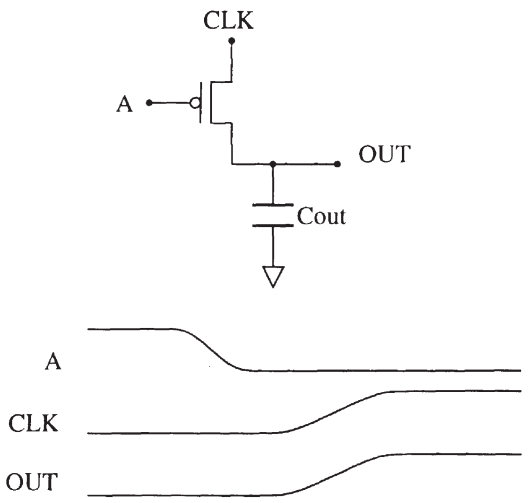


Figure 1.4. Simple adiabatic inverter

chronous designs that were not optimized for power. Although we believe that asynchronous design methodologies hold great promise, their final success as a revolutionary design style for low power is yet to be realized.

Design techniques typical of asynchronous design have been employed within the realm of synchronous circuits: a typical example is *clock gating*. With clock gating, power is reduced by stopping the clock of idle units. Clock gating can be seen as a specialized use of asynchronous techniques because the clock becomes an activation signal that is provided to a sub-system only when its computation is required. It is likely that asynchronous design techniques will be integrated in the mainstream synchronous paradigm in an evolutionary fashion.

Adiabatic computation has been proposed as a low-power design technique [benn88, atha94, denk94]. The principles of adiabatic computation are rooted in a simple physical principle: since power is $p(t) = i(t)v(t)$, little power is dissipated (and little heat is transferred to the environment) if the charge transfer needed to perform computation is performed at $v(t) \approx 0$. Obviously, if $v(t) = 0$ no charge is transferred. If we keep $v(t)$ small and change it slowly, we can transfer charge (i.e. perform useful computation) with minimal power dissipation. Key to the practical applicability of adiabatic computation is the quantitative meaning of the word "slowly". For charge transfer to be adiabatic, the transfer time must be $T_t >> \tau$, where τ is the RC time constant of the circuit performing the computation.

The simplest adiabatic circuit is the *adiabatic inverter* [denk94] shown in Figure 1.4. When the input A switches, the CLK line is low. The transistor is turned on, but no power is dissipated through it, because the voltage across its drain and source is zero. When the transient on A is exhausted, the CLK line is raised with a slow transition. In this case, "slow" means that the rise time of CLK T_r has to be $T_r >> C_{out}R$, where R is the equivalent resistance of the transistor and C_{out} is the load capacitance. Since T_r is slow, the output voltage V_{out} tracks the clock waveform and the voltage across the source and drain of the transistor is always very close to zero ($V_{ds} \approx 0$). Since power is dissipated on the transistor resistance, $P = VI = V_{ds}^2/R \approx 0$. Theoretically, $lim_{T_r \to \infty} P = 0$. In practice $T_r \approx 10C_{out}R$ is sufficient for the circuit to operate adiabatically, with negligible power dissipation. Notice that, at the end of the transition on CLK, OUT is equal to A', thus the circuit behaves as an inverter.

Several *adiabatic logic families* have been proposed [raba96] and actually implemented in silicon, showing extremely low power dissipation. Unfortunately there are numerous practical and theoretical objections to the concept of adiabatic circuits. Probably the most convincing one has been proposed by Indermaur and Horowitz [inde94]. In their paper, the authors claim that adiabatic circuits should be compared to voltage-scaled CMOS circuits with similar performance, and show that the operation frequencies at which adiabatic circuits become more power-efficient than voltage-scaled CMOS are extremely low.

Nevertheless, many papers have been presented where adiabatic circuits are implemented successfully within standard CMOS systems [raba96]. It appears that adiabatic techniques may help in designing critical sub-units and save some power, but it is unlikely that fully adiabatic designs will ever become a practical alternative to mainstream CMOS.

1.4 CAD TECHNIQUES FOR LOW POWER

Designers faced with the challenges of tight power constraints optimize power dissipation following the basic principles outlined in the previous section. Designing for low power is at least as difficult as designing for maximum speed or minimum area. Power dissipation is a pattern-dependent cost function, unlike area, which is constant with respect to input patterns. Delay is pattern-dependent as well, but as far as delay is concerned, we are interested in reducing its worst case value (i.e. the *critical path delay*) and relatively simple and fast estimates of the worst case can be obtained. The main source of difficulty in low-power design is the dependence of power on the switching activity of the internal and output nodes, which in turn depends on the input statistics.

Designers must rely on automatic optimization techniques, to speed up design turnaround time. Low-power designs are no exception. Since power be-

comes increasingly important as a design evaluation metric, a new generation of computer-aided design tools targeting power minimization is urgently needed by designers. In the last few years, significant research and development effort has been undertaken in academia and industry targeting the creation of a new generation of CAD tools for low power. As a result, hundreds of papers and several books have been published on the subject.

We will not attempt to provide a complete categorization of all computer-aided techniques for power estimation and optimization. This is a formidable task and it has been addressed by several authors [raba96, chan95b, nebe97]. Instead, we will focus on key ideas and try to provide some insight on why they may be successful in practice or, on the contrary, why they are unlikely to have a major impact. In our overview, we will move from the circuit level of abstraction up to the system level.

1.4.1 Circuit-level techniques

Power estimation at the circuit level can be performed with traditional circuit simulation tools such as SPICE and its many commercial enhancements. Circuit-level simulators must be integrated by accurate netlist extraction tools in order to achieve a high level of accuracy. In state-of-the-art sub-micron technology, netlist extraction is a challenging task, but worst-case delay estimation is the critical issue that drives CAD vendors towards ever increasing levels of accuracy in netlist extraction. Average power estimation imposes less stringent accuracy constraints on the extraction process, hence it can rely on technology developed for delay estimation.

Efficiency and input selection is the fundamental issue in circuit-level power simulation. Average power estimation requires: i) the selection of a set of input vectors which represents the typical input excitation for the circuit and ii) a fast simulation engine that can estimate the power dissipation for large circuits and long streams of input patterns. The efficiency problem is addressed by fast circuit simulators that achieve speedups of more than two orders of magnitude with respect to SPICE. The increase in simulation speed is achieved at the expense of a marginal decrease in accuracy and a more severe restriction on the class of circuits that can be handled. Efficient power simulators are commercially available and widely used [huan95]. They are probably the most successful CAD tools for low-power design.

Input pattern generation and selection is still based on the designer's intuition and skill. In the last few years, a large amount of research has been devoted to this topic. Probably the most established technique is known as *Monte Carlo simulation* [burc93]. Monte Carlo techniques are exploited in many fields (device simulation, test generation, etc.). Monte Carlo simulation is based on

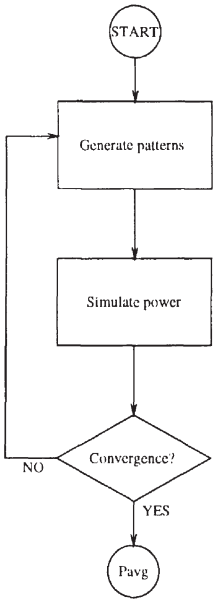


Figure 1.5. Flow of the basic Monte Carlo algorithm

an iterative process where random input patterns are generated and the target circuit is simulated until a statistical convergence test on the estimate of the average power dissipation has been satisfied (Figure 1.5). The main advantage of the Monte Carlo technique is its generality. Any simulator can be used to provide power estimates and the implementation of the statistical convergence test is straightforward.

The most serious limitation of Monte Carlo simulation is the pattern generation process. In almost all practical cases, the input patterns provided to a circuit during normal operation are not randomly generated. On the contrary, they must obey to a precise set of conventions (for example, initialization sequences or programming sequences). If the input patterns are not carefully chosen, the likelihood of bringing the circuit to a state which is never reached during normal operation is extremely high. As a consequence, the power estimate could be completely inaccurate. Many extensions and enhancements of the basic Monte Carlo techniques have been proposed in recent years to address this limitation [najm95]. Although some good progress has been made, Monte Carlo techniques are still not mature enough to provide reliable power estimates for large sequential circuits. In the majority of commercial designs, input pattern selection is still a hand-crafted or partially automated process.

Circuit-level optimizations for low power have been extensively investigated. Probably the most well-known techniques are: transistor sizing, clock tree optimization, low-power placement, and transistor reordering. Transistor sizing tools leverage non-linear optimization techniques such as convex optimization [came85]. Refer to [cong97] for an overview.

An interesting type of transistor sizing tool is build around a circuit simulator. Its structure is conceptually simple, and can be summarized as follows. The initial specification is a transistor-level netlist where some geometric device features (typically, transistor widths) are parameterized. We can represent the set of parameterized features by a vector $\mathbf{w} = [w_1, w_2, ..., w_n]^T$. The optimization engine searches for the values of the parameters that optimize a figure of merit $F(\mathbf{w})$ (in our case, power dissipation). Usually, there are constraints on the ranges of allowable values for the components of $\mathbf{w}$.

For a given value of $\mathbf{w}$, $\mathbf{w}^0$, the value of the figure of merit $F(\mathbf{w}^0)$ is computed by running a circuit-level simulation. Usually, it is relatively straightforward to compute the gradient of the figure of merit ($\nabla F(\mathbf{w}^0)$) as well. This information is exploited by general purpose non-linear optimization algorithms such as the Newton-Ralphson method that iteratively compute a succession of values $\mathbf{w}^n$. Convergence of $\mathbf{w}^n$ to the point of global minimum for F is guaranteed only if F is convex. Even if F is non-convex (and this is the case when complex device models are used in circuit simulation), it is often "well-behaved": iterative methods converge to good, albeit local, minima.

The optimization procedure terminates upon convergence and returns the optimal value $\mathbf{w}^*$ of device parameters which minimize power dissipation. Convergence speed and solution quality can be improved if the designer chooses a good initial guess for $\mathbf{w}^0$. Solid engineering intuition and skill are still required for successfully exploiting transistor sizing tools, but several research projects have reported encouraging results [cong97]. Recently, commercial transistor sizing tools for low power have been announced. The main limitation of transistor sizing is its limited scope of applicability: it can be used only in flows where the designer can control the size of the transistors. In many cases this is not possible: the atomic elements are cells belonging to a pre-defined and pre-optimized library. Thus, transistor sizes can take very few values, and transistor sizing has limited impact for library-based design flows.

Clock tree optimization is performed with similar techniques. In this case, the parameters to be optimized are: i) transistor sizes of clock drivers, ii) wire sizes of the clock tree, and iii) structure of the clock tree itself. Under several simplifying assumptions, clock tree optimization can be cast as a constrained convex optimization problem which can be efficiently solved by non-linear optimization techniques. Optimization of the clock distribution network is key for low-power operation in high-performance circuits, since the capacitance driven

by clock drivers is by far the largest on a chip, and it is switched twice every cycle. Several researchers have focused on clock distribution optimization for low power and have reported promising results [cong97].

Unfortunately, the most critical issue in clock design is skew control. Clock distribution is often over-designed to minimize the risk of timing violation due to clock skew in worst-case operating conditions. Functional correctness has clearly higher priority than reduced power dissipation, and aggressive low-power clocking styles have little chances of success if they reduce the reliability of the clock distribution network. Hence, the impact of clock tree optimization is limited by the need of meeting strict timing-related design constraints.

Placement algorithms for low power have received some attention as well [cong97]. The fundamental intuition in these approaches is that the wire length of connections with high switching activity should be minimized by careful gate placement. The effective switched capacitance is thereby reduced thanks to the minimization of the parasitic wiring capacitance. Placement for low power leverages pre-existing placement algorithms by specifying a different cost function. In traditional placement algorithms, capacitance reduction is the objective, while low-power placement targets the reduction of the product $C_{eff} = K \cdot C_{out}$. Iterative heuristic algorithms, such as simulated annealing or recursive bi-partitioning, are employed to minimize the objective function. Although placement for low power does not require extensive algorithmic enhancements to traditional tools, it has not been very successful in commercial applications. The most plausible explanation of this fact is that the results published by researchers show marginal power savings.

Transistor reordering rules for reducing power dissipation have been proposed by several researchers [cong97]. The target is a reduction of the power dissipated by CMOS gates on internal parasitic capacitances. Additionally, transistor reordering can be helpful in reducing the power consumed during spurious signal transitions (glitches). Current technology trends have much reduced the potential impact of transistor reordering for the simple reason that gate output capacitance is dominated by wiring and it is generally much larger than internal capacitance. Nevertheless, *input reordering* (where inputs signals are assigned to the pins of a gate so as to reduce its internal power consumption) is applied at the logic level during technology mapping. Input reordering is slightly less powerful than transistor reordering, but it has the key advantage that it can be applied on library-based designs.

Example 1.2 *Consider the pull-down network of Figure 1.6 (a). Assume that the inputs are A, B, C and D, with probability of being high respectively* $p_A = .1$ $p_B = .4$, $p_C = .4$ $p_D = .8$. *Moreover, assume that the inputs are mutually independent. In order to avoid the charging of the internal capacitances as much as possible, we want to assign input low probability to pins that are toward*

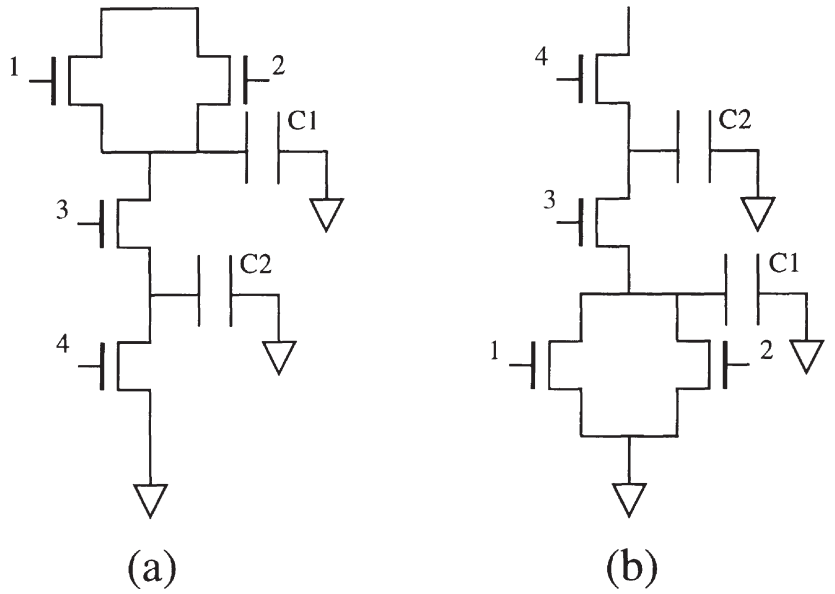


Figure 1.6. (a) A pull-down network. (b) Equivalent network after transistor reordering

the top of the pull-down chain. Assume for instance that $C1 = 20fF$ *and* $C2 = 40fF$. *The assignment:* $A \rightarrow 1$, $B \rightarrow 2$, $C \rightarrow 3$, $D \rightarrow 4$ *leads to an equivalent charging capacitance* $C_{eq} = (.1 + .4) * (20 + .4 * 40) = 18$. *The assignment* $A \rightarrow 3$ $B \rightarrow 1$, $C \rightarrow 2$. $D \rightarrow 4$ *leads to an equivalent charging capacitance* $C'_{eq} = (.4 + .4) * (20 + .1 * 40) = 19.2$. *Thus, the first assignment is preferable.*

If we allow transistor reordering, we can construct the equivalent pull-down network shown in Figure 1.6 (b). In this case, the optimal assignment is $A \rightarrow 4$, $B \rightarrow 2$, $C \rightarrow 3$, $D \rightarrow 1$ *which produces an equivalent capacitance* $C''_{eq} = .1*(40+ .4 * 20) = 4.8$. *Unfortunately, transistor reordering requires the modification of the pull-down network. This may not be allowed in a library-based design flow, while simple input reordering is always possible.*

1.4.2 Logic-level techniques

Logic-level power estimation and optimization has been one of the most extensively researched area in the CAD community. For static CMOS technology, power is modeled at the logic level since it depends on signal switching. Needless to say, some accuracy is lost (leakage power and short circuit power are not

accurately modeled), but power estimates based on switching activity usually correlate well with the actual power dissipation. Moreover, the accuracy of logic-level power simulation can be much improved for library-based designs if the library cells are pre-characterized and annotated. A few advanced logic-level power simulation engines have been developed. They claim marginal accuracy losses with respect to fast circuit level simulators accompanied by speedups by more than two orders of magnitude [najm95].

Unfortunately, the speed of accurate logic power simulation is still insufficient for usage within the inner loop of iterative optimization algorithms. In many cases only good relative accuracy is required to choose between alternative implementations or to evaluate the impact of a local optimization. Two classes of power estimation approaches have been developed for this purpose. Both classes share the common assumption that power can be simply computed as the average number of signal transitions on circuit nodes weighted by node capacitances. The first class is represented by logic simulators augmented by Monte Carlo criteria to automatically detect convergence of average power estimates. The main limitation of such tools is that convergence can be slow when power estimates are required on a node-by-node basis. The second class is known as *probabilistic simulation*. It attempts to obviate to the slow convergence problem by avoiding simulation. In contrast, switching probabilities for internal nodes are directly computed starting from the switching probabilities of the primary inputs.

Several probabilistic methods for the computation of the switching probability of internal nodes have been proposed, spanning the trade-off between accuracy and speed [najm95]. Unfortunately, all most efficient approaches neglect spurious switching activity (also known as glitches) caused by unequal signal arrival times at the input of logic gates. Moreover, probabilistic simulation for sequential circuits is quite inaccurate because it models approximatively the signal correlations introduced by state feedback. The limited accuracy of probabilistic techniques can be tolerated when they are used for evaluating the effect of local transformation of logic-level netlists. In other words, the *local* and *relative* accuracy of probabilistic simulation is satisfactory, but their *global* and *absolute* accuracy is not.

The latest generation of gate-level power estimation engines employs a hybrid approach. Standard logic power simulation is run at the beginning of an optimization phase and on the final optimized implementation. The effect of local transformations during optimization is efficiently estimated by probabilistic simulation. As a final note on the topic, it is important to observe that, despite the vast amount of research activity, the accuracy of logic-level power simulation engines is not completely validated, and remarkable improvements are to be expected in the near future, as new tools gain industrial acceptance.

On the optimization side, a successful research effort has targeted the adaptation of traditional optimization techniques to the new cost function. In particular all *technology independent* techniques (i.e. techniques that are applied to abstract representation of the functional behavior of the target circuits) based on the algebraic model [dmc94] have been revisited and modified to take power dissipation into account [iman96]. The main challenge in adapting the traditional approaches to the new cost metric is probably the fact that power cannot be predicted without taking into account its dependence on input pattern distribution. In contrast, area is pattern-independent and speed, although pattern-dependent, can be often safely approximated in a pattern-independent fashion.

Unfortunately, the power savings claimed by technology-independent techniques are often regarded with some suspicion by designers. The main reason for the lack of confidence is that optimization relies on power estimation techniques which are neither technically stable nor very accurate. It is often the case that the power savings achieved by technology independent techniques are below the uncertainty level of the power estimates. On the other hand, experimental evidence [iman96] shows that technology independent power optimization can indeed lead to final implementations with reduced power and it is reasonable to expect that some of the techniques presented in the literature will be implemented in commercial synthesis tools.

The last phase of the logic synthesis process is called *technology-dependent*, because it assumes the existence of a technology library of gate primitives. The final product of the technology dependent phase is a *mapped netlist*: an interconnection of gate primitives belonging to the technology library. The process of translating a technology-independent specification into a mapped netlist is known as *technology mapping* (or library binding). Technology mapping for low-power has been extensively studied [tsui93, tiwa93].

Again, well-established algorithms targeting speed and area have been revisited and adapted to the power cost metric. Compared to technology-independent techniques, technology mapping can rely on much more accurate power estimates. Hence it produces more reliable results. Low-power mapping techniques have been implemented in commercial tools [keut96]. Two are the main weaknesses of library binding for low power. On one side, it is applied at a very late stage of the synthesis process, thus it has a relatively small impact. On the other side, it is still quite prone to inaccuracies, especially for sub-micron technologies, because it does not accurately model the effect of wiring capacitances.

To obviate to the second limitation, a new class of logic optimization techniques has been recently investigated. We call these approaches *remapping* [rofl96, vuill97, iman97]. Once the circuit has been mapped, placed

and routed, wiring capacitances can be extracted from the layout and back-annotated on the logic network. Based on such accurate information, re-mapping techniques apply local transformations which target nodes (i.e. connection) with high effective capacitance. The key idea is to attempt to eliminate the high-power wires/gates, and to replace them with equivalent combinations of gates and wires that dissipate less power. Re-mapping techniques have reported promising results, and are particularly attractive because the uncertainty on the quality of results is very small. No commercial tools based on re-mapping are currently available, because the technique is relatively new, but a commercial application of this interesting approach is to be expected in the near future.

Re-mapping can locally change the functionality of a network. Several algorithms have been proposed where local functionality is not changed, but gates can be replaced by functionally equivalent cells that have different driving strength, internal delay or input load. Post-mapping optimization based on this type of transformations is called *re-sizing* [coud96]. Resizing can be highly effective for low power. The power consumption of a network is reduced by slowing down non-critical paths. Cells that belong to non-critical paths are replaced with functionally equivalent cells of smaller size which are slower and less power-consuming. The speed of the circuit is not changed by resizing; hence it is a very useful technique that does not involve any trade-off between power and delay.

The optimization techniques outlined so far deal with the combinational component of logic-level descriptions. Several algorithms targeting sequential circuits have been developed as well. These approaches focus on the concurrent optimization of both combinational logic and state-holding elements (latches and flip/flops). For sequential circuits specified as finite-state machines (FSMs) using state tables or equivalent formats, an interesting optimization problem is *state assignment*. State assignment algorithms assign binary codes to symbolic state names in a FSM specification trying to optimize a cost metric. It is important to note that the choice of state codes influences not only the power dissipation on the sequential elements, but also that of the combinational logic. This is due to the fact that the next state computation, given a present state and input value, is computed by the combinational logic of the state machine.

The complex (and still unclear) dependence of the area/speed/power of the combinational logic from the state assignment makes the problem an interesting challenge. Indeed, several researchers have worked on state assignment for low power and numerous algorithms have been proposed [roy93, olso94, hahe94, tsui94, beni95]. Unfortunately, the state assignment problem itself and, as a consequence, the algorithms solving it, have limited practical relevance in the power optimization arena. This is due to the fact that only small components of

a digital system are specified as FSMs. Although such components do dissipate power, they are almost negligible when compared the (much larger) fraction of the system which is not specified as a FSM. State assignment for low power is nevertheless an interesting theoretical problem. Its basic mathematical structure is similar to optimization problems such as bus encoding which have much greater practical relevance.

Optimization techniques targeting sequential circuits described in a structural fashion as an interconnection of combinational gates and sequential primitives have potentially a greater impact. In fact, they may target systems whose state set is so large that it cannot even be enumerated in a reasonable time. An example of such techniques is *retiming*.

Similarly to all other techniques described so far, retiming was originally formulated to improve the performance of sequential circuits [dmc94], by moving sequential elements within a synchronous network. The retiming transformation does not change the input-output behavior of the network, but it completely changes the state space. Retiming for low power has been proposed by Monteiro et al. [mont93]. The basic idea is that sequential elements can be moved to positions where their presence reduces the global C_{eff}. If the wire where the flip-flop is moved by retiming has high capacitance and large switching activity due to spurious transitions, the flip-flop will only latch the last value appearing on the wire before the clock edge, and will therefore propagate a maximum of one transition to the high capacitive load.

Obviously, retiming for power has to be constrained. The cycle time of the circuit must not increase because of retiming, and bounds on the maximum increase in the number of registers may be specified as well. The strongest objection to retiming for low power is that it is very difficult to accurately estimate the spurious switching activity in a logic circuit without layout data back-annotation and accurate (and slow) simulation. When a flip-flop is moved, glitches disappear on its output, but other glitches may be generated because the timing of the combinational logic is changed. Consequently, it is hard to estimate not only the opportunity of a retiming move, but also its impact on the power dissipation of the final implementation.

All logic-level optimization techniques can be seamlessly integrated in a traditional synthesis-based design flow, but they share a shortcoming. Since they are applied late in the design process, they produce relatively small power reduction. If a minimum-area speed-constrained implementation exceeds by 20% or less the power budget, logic-level optimizations may actually succeed in producing a final implementation which meets the requirements. This is not the case when larger power reductions are desired. Obviously, one way to obviate to this problem is to optimize power at higher level of abstraction.

1.4.3 Behavioral-level techniques

Designing systems at the behavioral level is a relatively new art made possible by the appearance and widespread diffusion of *hardware description languages* (HDLs). Hardware description languages, such as VHDL and Verilog HDL, allow designers to describe the abstract behavior of a system using a specification style similar to that of high-level software programming languages. While the increased abstraction improves the designer's productivity, it is much harder to estimate the impact of design choices and to compare design alternatives.

Power estimation at the behavioral level is one of the most active research areas, and the availability of robust and accurate behavioral power estimation tools is critical for the success of design flows based on behavioral specification. The pioneering work by Landman and Rabaey [land95, land96] represents the first attempt to develop a methodology which is general enough to model systems of practical relevance and accurate enough to be useful.

Landman and Rabaey focused on digital signal processing systems described in behavioral *register-transfer* (RT) style. The key feature of a RT description is that it matches the logic-level implementation at the clock cycle boundaries. A RT description is an interconnection of basic blocks (also called *units*) which contain executable functional models. The power estimation methodology associates to each unit a *pattern dependent* model of its internal power consumption. Pattern dependence is modeled by expressing power as a function $P(sw_i, sw_o)$, where sw_i and sw_o are respectively the input and output switching probabilities. The switching probabilities can be obtained by running fast RT simulation. Power is then estimated by computing P for each resource and summing the power contributions.

The main limitations of the work by Landman and Rabaey are that power models must be provided by the designer and that a numeric data representation scheme is assumed for all data communication. Several successive works have addressed these limitations [hsie96, meht96, bogl97]. Commercial RT power estimation tools have been developed and are available [keut96]. Obviously, estimation accuracy at the behavioral level is much reduced compared to circuit or gate level power estimation. Fortunately, what is really important at high level of abstraction is relative accuracy, which is needed to compare alternatives and to explore the design space.

RT level descriptions are much more abstract than logic level ones, but they are still too detailed to allow the exploration of complex trade-off between throughput, area and power. Although the interface of a system may be constrained by interface requirements, there are multiple degrees of freedom in deciding how the computation to be performed within the system can be distributed over time. The exploration and exploitation of these degrees

of freedom is the main task of *behavioral synthesis* (also known as *high-level* synthesis).

During behavioral synthesis, the specification is represented by a graph (called *control-data-flow* or *sequencing graph* [dmc94]) where vertices are operations and edges represent functional or control dependencies between operations. The mapping of this abstract structure to hardware is done in two steps: *scheduling* and *resource binding.*

During scheduling, operations are assigned to the clock cycles in which they will be executed. During resource binding the operations are mapped to actual hardware resources (such as adders, multipliers, etc.). If the result of an operation is to be used in a clock cycle following its computation, it must be stored in a sequential element (a register). *Register binding* is the part of the resource binding step where the outputs of operations are assigned to registers. The last part of resource binding is *steering logic generation*, where the hardware connections among units or between units and registers are created. Steering logic consists of multiplexers, decoders, encoders and data busses. A good overview of behavioral synthesis can be found in [dmc94].

There are two basic approaches to high-level power minimization. One approach attempts to minimize the *switching activity* of the circuit, because accurate estimation of the capacitance is not available at this level of abstraction. In other words, this approach attempts to minimize the average activity factor K of Equation 1.3. The second approach tries to minimize $C_{eff} = KC_{out}$, by taking the active area into account as well. This approach relies on behavioral power estimation tools that provide accurate information on C_{out}.

We first outline the synthesis flow followed by techniques in the first approach. To collect information about the switching activity of the initial design, the behavioral description of the design is simulated, prior to scheduling and resource binding. Each edge of the sequencing graph is associated with a unique identifier called a *variable.* Information on the switching activity of all variables is collected. More precisely, an *activity matrix* is constructed. An element of the matrix is the average number of bit differences between a variable and all other variables.

The sequencing graph is then scheduled, usually with constraints on resource usage, latency and throughput. After scheduling, resource binding is performed targeting the minimization of switching activity. The key idea is to assign hardware resources trying to minimize the average functional switching activity at their inputs and outputs, since it is experimentally observed that low input and output activity is accompanied by low internal power dissipation in register and data-path units. Moreover, low switching on the input and outputs implies that the capacitance associated with the steering logic is switched less frequently.

Raghunatan and Jha [ragh94] focus on binding of data-path units to minimize their input-output switching activity, while Chang and Pedram [chang95] focus on register binding. Mussol and Cortadella [muss95] modify the scheduling algorithm as well in order to approximatively take into account switching activity.

The reduction of the switching activity in the steering logic is targeted in the papers by Dasgupta and Karri [dasg95] and by Raghunatan et al. [ragh96a, ragh96b]. In the first paper, the authors target functional switching activity in bus-based systems: exploiting the same intuition as in the resource binding case, they assign data transfers to the bus in a sequence that minimize the average number of transitions between values output on the bus in successive cycles. By contrast, Ragunathan et al. target the useless switching activity on the steering logic when it is not used to communicate data required for the computation. Such switching activity is doubly harmful because it dissipates power in both the steering logic and the units connected to it.

The techniques targeting minimization of C_{eff} leverage the behavioral power estimation capability of a class of power analysis tools that has been recently developed [land96, sanm96]. The key improvement of these analysis tools with respect to behavioral simulation is that they exploit power information on data-path units, steering logic and registers that have been pre-characterized once for all in a preliminary step.

Behavioral power analysis tools are fast, therefore they can be used in the inner loop of an optimization algorithm. The basic flow of power optimization is the following. First, a scheduling and resource binding is generated, then the analysis tool is used to estimate its power. Alternative solutions are then generated and their power consumption is estimated as well. Finally, the solution with minimum estimated power is chosen.

What differentiates the approaches in this area is the strategy used to generate alternative solutions and the behavioral power estimation tool used. Chandrakasan et al. [chan95b] propose a *transformation-based* approach. The synthesis tool operates a set of power-optimizing transformations on the original specification. Transformations that deeply modify the structure of the circuit are first applied one at a time using heuristic rules. The best resulting circuits are then memorized and become the starting points for a probabilistic algorithm that attempts to further improve power dissipation by local, low-impact transformations. The power analysis tool used to estimate the quality of the generated solutions is *pattern independent.* The power of an implementation is estimated using only information on the switching activities of the variables in the control data-flow graph (collected once for all at the beginning of the synthesis process) and the back-annotated library of hardware component used for synthesis [land96].

In contrast, Kumar et al. [kuma95], propose an approach based on *pattern-dependent* power estimation. A fast simulation with a user-specified input pattern set is performed on a candidate solution, and the power is estimated using the switching activities and the back-annotated library of components. Notice that, in contrast to the approach of Chandrakasan et al., a new simulation is performed for each candidate solution. The candidate solutions are generated from a set of valid solutions that satisfy timing and area constraint using simple enumerative techniques.

Our brief overview of behavioral synthesis techniques for low power shows that the abstractions and transformations are the same used for traditional cost metrics, with the modifications required by a new (and highly complex) cost function. Despite the large amount of research activity, high-level power optimization is not a mature area. First, the lack of a large set of publicly available benchmarks severely limits comparisons and neutral evaluations. Second, the power reductions claimed by several researchers, even on carefully selected benchmarks, are often smaller than what could be expected (usually below 50% with respect to minimal-area, timing-constrained implementations). Arguably, the main reason for the somewhat disappointing results is the inaccuracy of the power model used for driving optimization.

Commercial success of high-level power optimization techniques depends on an essential prerequisite: a generalized adoption of a design methodology based on behavioral synthesis. Currently, the number of systems designed at the behavioral level is a small fraction of those designed at the RT or logic level, hence the market for high-level power optimization tools is relatively small. On the other hand, the rapidly increasing complexity of designs and the fast turn-around time are strong motivators for a paradigm shift toward behavioral-level design.

1.4.4 System-level techniques

Even the most enthusiastic supporters of synthesis for low power admit that there is a significant gap between the power reductions attainable with the optimization approaches summarized in the previous subsections and the results obtained by carefully hand-crafting a system with low power consumption as a primary target. Specifically, it is unlikely that the combined effort of high-level, gate-level and circuit-level optimizations would reduce power by more than factor of two in average. Technology roadmaps and trend analyses [sema97] clearly show that this result is far to be sufficient. Power dissipation is expected to grow at a fast rate and reductions of more than one order of magnitude will soon be requested.

Digital designers have promptly adjusted to the challenges posed by low power design. Designer skills, technology improvements and CAD tools may actually be successful in keeping power consumption under control at the chip level. Unfortunately, software engineers and system architects are often less power-conscious than digital designers. Hence, power-efficient hardware is often employed in a power-inefficient fashion, and architectural choices are often based only on intuition, which, in the case of power, is often deceiving. Ultimately, the frontier of low power design is at the system level.

One of the main difficulties in dealing with system-level power optimizations, is that the scope of the problem itself is not clearly defined. In this book, we call *system* an interconnection of resources which are separately designed. The resources may be heterogeneous (electronic, electro-mechanical, optical, etc.), and system-level design deals with connecting the resources, setting up communication and operating the resources in a functionally correct and efficient fashion. All practical digital devices (cellular phones, computers, digital assistants, etc.) are systems, according to the previous definition. A chip can be a system if its components are designed and optimized one at a time as separate resources.

Although the bulk of research effort on computer-aided power optimization has been focused on the lower level of abstractions, a few research groups have dedicated some attention on system-level CAD tools. We will review here the basic concepts and ideas, and give references to more detailed information. The group lead by Rabaey has focused on signal processing systems based on dedicated ASICS [guer94, mehr96, lids96]. Guerra et al. [guer94] introduced the notions of *algorithmic properties* which correlate well with power dissipation area and speed. The intuition is that the analysis of an algorithm expressed in a high-level programming language can provide information about its hardware implementation. Obviously the information is not accurate in absolute terms, but it is sufficiently informative to help the designer in the choice among alternative and functionally equivalent algorithms.

Mehra et al. [mehr96] carried on this concept and formulated a low-power scheduling procedure which takes into account communication cost. The key intuition is that power in real-life systems mainly dissipated in "communication channels" (busses and steering logic) more than in computational elements. Although Mehra's scheduling algorithms belongs to the class of behavioral-level synthesis techniques, it is based on an abstract (and highly relevant for power optimization) cost metric that could be employed for scheduling at even higher level of abstraction.

On the estimation front, Lidski et al. [lids96] proposed a spreadsheet-based system-level power estimation tool called *Powerplay*. The main purpose of Powerplay is to help the designer in investigating the effect of design choices

on power dissipation. A system is represented by a collection of resources and of usage data. For each resource, a power model must be provided which correlates usage data with power dissipation. The information is organized on a spreadsheet-like data structure (i.e. a matrix) in which columns represent resources, and rows represent usage conditions. Each matrix element represents the power dissipation of a resource under a given usage condition. When the designer wants to evaluate the effect of a change in usage condition (or the impact of a new resource), it merely modifies the corresponding row (column) in the spreadsheet and re-computes the power dissipation. Although the method gives very rough power estimates, it is very helpful in the first phases of the design process, when the designers want to "play" with alternatives (i.e. heuristically explore the design space).

Potkonjak et. al. [hong96, kiro97] explore several techniques for power optimization at the system level. In [hong96] the authors consider the problem of scheduling tasks and disk location assignments for real-time systems to minimize power dissipation. In [kiro97], the scheduling of tasks (with hard real-time constraints) on multiple processors for minimum power is considered. In the first work the key idea is that power can be reduced by first increasing the speed beyond what is barely necessary to meet timing constraints, then by reducing the voltage supply of the components, slowing them down until the timing constraints are barely met again. This is obviously the same concept exploited in power-driven voltage scaling and it suffers the same limitations, namely the practical difficulties and costs involved in the usage of variable supply voltages. However, at the system level the large size and small number of components may actually allow multiple (and adjustable) voltage supplies.

The second work presents a heuristic scheduling algorithm based on the hypothesis that the workload of a given task can be performed by several processors with different performance levels (and power costs). Starting from an initial feasible randomized solution, the algorithm randomly attempts to remove or add components, and accepts the best solution over a set of attempts. The greedy randomized procedure continues until there is improvement. An "optimal" resource binding is returned upon termination. The algorithm then moves to the active scheduling phase where tasks are mapped onto processors in a heuristic fashion, trying to assign tasks on processors that execute them in a maximally power-efficient fashion.

In processor-based systems, a relevant fraction of the total power budged is consumed in memories and busses. There is a power cost for each memory access and bus transaction. Power can be reduced by minimizing the memory accesses, and clustering them in patterns that minimize the cost of bus transactions. These problems are tackled in [pand96] and [gebo97], where memory allocation techniques for reducing power consumption are developed. Internal

registers and memory hierarchy can be exploited to reduce the likelihood of power-expensive accesses to disks or slow and large memories. Accesses can be reordered to minimize the transition activity on memory busses. The average number of transitions on bus lines can be further reduced by power-conscious bus-encoding techniques [stan95, meh96a, beni97b].

Software power minimization and compilation techniques for low power consumption have been studied by Wuytack et al. [wuyt96] and by Tiwari et al. [tiwa94]. Wuytack et al. performed a detailed study on power efficient data-structures. Again the purpose is to minimize the power dissipated in memory accesses, but here complex and general data structures are considered, while the previous memory-related power optimization techniques operate at a lower level of abstraction (i.e. instruction level and word level). Tiwari et al. first proposed an instruction-level power model for general-purpose as well as DSP processors, then exploited the model to find code optimizations which can be automatically applied during compilation.

The results for general-purpose microprocessors are somewhat disappointing because they seem to indicate that power is reduced mainly by reducing the dynamic count of executed instructions. This is exactly the same objective targeted by compilation techniques for high performance. The results seems to indicate that low-power compilation techniques are likely to have marginal impact. However, the conclusion may be an artifact of the excessively simplified instruction-level model proposed by the authors, that does not allow to accurately capture the effect of internal processor memory, pipeline stalls and memory hierarchy.

For DSP processors, low-power compilation techniques based on the instruction-level power model have produced more interesting results. In particular, it was shown in [lee95] that instruction scheduling has sizable impact on global power consumption. This result can be explained by observing that the hardware implementation (and power dissipation) of DSP processors is dominated by the functional units in the data path, hence there is a strong correlation between the executed instruction and the execution unit involved in the computation.

More recently, a refined power estimation method for general-purpose microprocessors has been proposed [hsie97]. Given a microprocessor, its memory systems and the executable of a target application, the method estimates the average power dissipation of the application by running a much shorter (as far a dynamic instruction count is concerned) automatically-generated program on the microprocessor. The technique takes into account the effect of memory hierarchy (cache misses, and miss penalties) as well as the dynamic instruction mix. We conjecture that the new power model is more suitable than simple

instruction-level power costs for formulating aggressive compiler optimizations for low power.

1.5 POWER MANAGEMENT

In the previous section we have purposely ignored a set of techniques based on the key concept of *dynamic power management*. The detailed analysis of dynamic power management, applied at all levels of abstraction, will be the focus of the remaining chapters. This section is dedicated to a comparative analysis based on general observations. As we already pointed out, the majority of techniques described in the previous section can be seen as extensions of classic synthesis algorithms where cost functions have been changed and the algorithmic framework slightly tuned.

Although these "evolutionary" techniques are undoubtedly powerful and relevant, they do not capture an essential characteristic of power consumption for static CMOS circuits. In static CMOS technology, quiescent portions of a system dissipate a minimal amount of power. In other words, power can be reduced even if more hardware is instantiated (i.e. area is increased) when we can guarantee that large circuit blocks are frozen in a quiescent state. This statement warrants further discussion. First, it is important to note that the necessary condition for the statement to be true is the existence of low-power, high performance shutdown mechanisms. The presence of shutdown circuitry is not required for functional correctness, and thus it can be seen as pure functional overhead. Such overhead must be small: the performance of the power-managed system should never deteriorate up to violating performance constraints. Similarly, the power dissipation of the shutdown circuits should be small compared to the power saved in the component that is kept quiescent.

Second, the system implementation must allow shutdown of one or more of its parts. If a system is fully utilized for the entire operation time, shutdown is obviously impossible, and dynamic power management useless. Power management exploits *idleness* caused by system under-utilization. It may be observed that, if the system is underutilized, it is not well designed and could be optimized to eliminate redundancy. This conclusion is almost never true. For almost any conceivable practical system, the utilization is not constant. Designers naturally (and rightfully) focus on worst-case conditions, peak performance requirements and peak utilization. As a result, systems are often designed to operate under high utilization, but they are actually fully exploited during a relatively small fraction of their lifetime.

Obviously there are critical devices which operate almost always under maximum sustainable load. For this type of devices, power management is not effective (or it has marginal impact). Fortunately, this is not the case for the

majority of devices for which low power consumption is key for commercial success. As a motivating example consider portable equipment for personal computing and communication. For all currently available communication devices, the idle (or partially idle) times are usually much longer than the fully operational times. Nevertheless, the devices are designed to provide high performance levels when in use. Dynamic power management explicitly targets these non-stationary-load systems.

In the perspective presented so far, dynamic power management is a remedy whereby the designer (or the CAD tool) may be able to eliminate a sizable fraction of the useless switching activity that consumes power without producing useful results. The key issues to be addressed for the implementation of dynamic power management schemes for elimination of useless switching activity are: i) identification of idle conditions, ii) selection of the most power-consuming idle conditions, iii) optimal design of the power management circuitry. A successful power management strategy must provide satisfying solution to all three issues. Clearly many complex tradeoffs are involved: there is always a compromise between effectiveness in identifying idle conditions and implementation cost of power management circuitry.

Elimination of wasted power has been the primary focus of power management schemes. We take a more aggressive standpoint. Power management can be exploited as a design tool for actively reducing power consumption even when idleness is not apparent. Several examples of this concept will be provided in the next chapters. A system should be designed taking power management into account: the functionality can be partitioned in tasks which are mutually exclusive and idle for a large fraction of the operation time, and communication protocols which allow the exploitation of idleness should be envisioned. These are simple examples of how power management can directly influence the active design cycle, and not only be relegated to a role of post-optimization and tuning.

An even more radical view point could be taken. We have always assumed that power management should never impose a significant performance degradation when the system is fully active. In many practical examples, even this assumption can be relaxed. Consider for example a portable device such as a laptop computer. It is possible to envision a *graceful degradation* operation mode as batteries discharge. The key intuition here is that what really matters in many cases (more than sheer performance) is a balance of performance and availability. In extreme conditions we could tolerate (and even not perceive) a controlled performance degradation if functionality is still provided for a longer period.

There are even more serious reasons for investigating power management policies that reduce performance. High power dissipation is associated with

heat that must be adequately dissipated, otherwise temperature may raise and abruptly reduce reliability or even directly cause functional failure. If cooling devices are insufficient to keep temperature under control, power dissipation must be reduced even at the price of a sizable performance penalty.

In summary, we have outlined three flavor of dynamic power management. The first and most conservative targets the reduction of wasted power at no (or little) performance cost. The second drives system partitioning and decomposition to produce an implementation where locality of computation is high and idleness can be easily detected and exploited. The third and most aggressive allows sizable performance decreases, if either availability or functional correctness may be compromised. The detailed description of how to implement power management schemes at various level of abstraction and how to exploit power management at different levels of aggressiveness is the core contribution of the following chapters.

2 BACKGROUND

Et chaque vérité que je trouvais étant une règle qui me servait après à en trouver d'autres.

And each truth that I found became a rule which served afterwards for finding others.

—R. Descartes

41

2.1 INTRODUCTION

The main purpose of this chapter is to provide the necessary background for the ideas and algorithms presented in the following chapters. In the second section we define the basic concepts of Boolean algebra and of finite-state machines with binary inputs and outputs. This material is prerequisite for the rest of the chapter.

In the third section, we delve into the algorithms and data structures for the manipulation of large Boolean functions and finite-state machines. We will then extend our treatment to similar structures and algorithms for dealing with a generalization of Boolean functions known as discrete functions.

In the fourth section we will describe the analysis techniques that allow the accurate estimation of the power-related cost functions that drive power optimization algorithms. The theory of finite Markov chains will be briefly reviewed and algorithms for dealing with small and large-size Markov chains will be described.

2.2 BOOLEAN ALGEBRA AND FINITE-STATE MACHINES

The following treatment is by no means complete. The reader is referred to the many books on Boolean algebra and sequential machines published in the last few decades (see, for example [koha70, davi78, brow90]). We adopt the formalism and terminology used by De Micheli [dmc94].

2.2.1 Boolean algebra

A *binary Boolean algebra* is defined by the set $B = \{0, 1\}$ and the operations *disjunction* and *conjunction* which are represented by the symbols $+$, $\cdot$, respectively. Disjunction is often called *sum* or OR, while conjunction is called *product* or AND. We will use these terms interchangeably.

The multi-dimensional space spanned by n binary-valued variables is denoted by B^n. A point in B^n is a binary-valued vector of dimension n. Every variable is associated with a dimension of the Boolean space and is called a *Boolean variable*. In addition, any Boolean variable a with values in B has a *complement*, denoted by a', such that $a + a' = 1$, $a \cdot a' = 0$. A *literal* is an instance of a variable or its complement. For convenience, we define the *exclusive or* (XOR) operation, denoted by the symbol $\oplus$, as $a \oplus b = ab' + a'b$.

A *completely specified* Boolean function is a mapping between Boolean spaces: $f : B^n \rightarrow B^m$. An *incompletely specified* Boolean function is defined over a subset of B^n. The points where the function is not defined are called *don't care* conditions. For the sake of compactness, incompletely specified functions can be represented as $f : B^n \rightarrow \{0, 1, -\}^m$, where the symbol "$-$" denotes a *don't*

care condition. For each output, the subset of the domain for which the function assumes values 1, 0 and – are respectively called *on set*, *off set* and *don't care set*. *Don't cares* can be divided into two classes. *Observability don't cares* (ODCs) are points in B^n for which the value of function f is not specified because it is not observed (hence, it can either be one or zero). *Controllability don't cares* (CDCs) are points in B^n for which the value of the function is not specified because its inputs are guaranteed never to assume the value of the points.

In order to simplify the following treatment, we restrict ourselves to *single-output Boolean functions*, i.e., $f : B^n \rightarrow B$. Generic (*multiple output*) Boolean functions can be represented as arrays of single-output Boolean functions $f = [f_1, f_2, ..., f_m]$. We also define two special Boolean functions: i) $\mathbf{1} : B^n \rightarrow \{1\}$ (also known as *tautology*) and ii) $\mathbf{0} : B^n \rightarrow \{0\}$ (the *null function*). Tautology maps all points in B^n to the point 1 in B. The null function maps all points in B^n to the point 0.

Let $f(x_1, x_2, ..., x_n)$ be a Boolean function of n variables. We call the set $\{x_1, x_2, ..., x_n\}$ the *support* of the function. We now give some definitions that will be extensively used.

Definition 2.1 *The positive cofactor of* $f(x_1, x_2, ..., x_n)$ *with respect to variable* x_i *is*

$$f_{x_i} = f(x_1, x_2, ..., x_{i-1}, 1, x_{i+1}, ..., x_n)$$

The negative cofactor is

$$f_{x_i'} = f(x_1, x_2, ..., x_{i-1}, 0, x_{i+1}, ..., x_n)$$

In words, a cofactor of a Boolean function w.r.t x_i is the restriction of the function to the semi-space where variable x_i has a fixed value. Hence, the support of the cofactor does not contain x_i, because f_{x_i}, $f_{x_j'}$ do not depend on x_i. The definition of cofactor allow us to define three important operators of Boolean algebra, namely Boolean difference (Boolean derivative), consensus (universal quantifier) and smoothing (existential quantifier).

Definition 2.2 *The Boolean difference of* $f(x_1, x_2, ..., x_n)$ *with respect to variable* x_i *is* $\partial f / \partial x_i = f_{x_i} \oplus f_{x_i'}$.

The Boolean difference represents the sensitivity of the value of f to the value of x_i, in other words, $\partial f / \partial x_i = 1$ for all values of input variables (different from x_i) for which a value change on x_i causes a value change of f.

Definition 2.3 *The consensus of* $f(x_1, x_2, ..., x_n)$ *with respect to variable* x_i *is* $\forall_{x_i} f = f_{x_i} \cdot f_{x_i'}$.

The consensus is a fundamental operator in Boolean algebra and will be often used in this book. It represents the component of f whose value is independent of x_i. In other words, $\forall_{x_i} f$ is 1 for all values of input variables (different from x_i) for which $f = 1$ no matter what the value of x_i is. We will often call consensus the *universal quantifier* with respect to x_i, because it represents conditions that make $f = 1$ for all values of x_i

Definition 2.4 *The smoothing of* $f(x_1, x_2, ..., x_n)$ *with respect to variable* x_i *is* $\exists_{x_i} f = f_{x_i} + f_{x_i'}$.

The smoothing is the dual of the consensus: $\exists_{x_i} f$ is 0 for all values of input variables (different from x_i) for which $f = 0$ no matter what the value of x_i is. The smoothing is often called *existential quantifier.*

The *greater or equal* relation between two Boolean functions $f \geq g$ holds if and only if $f \cdot g' = \mathbf{1}$. An equivalent notation is $g \Rightarrow f$ (g implies f). The definitions given so far will be clarified through an example.

Example 2.1 *Consider the Boolean function* $f(a, b, c) = ab + b'c + a'b'c'$. *The positive cofactor w.r.t.* c *is* $f_c = f(a, b, 1) = ab + b'1 + a'b'0 = ab + b'$. *The negative cofactor is* $f_{c'} = f(a, b, 0) = ab + b'0 + a'b'1 = ab + a'b'$. *The partial derivative is* $\partial f/\partial c = f_c \oplus f_c' = (ab + b') \oplus (ab + a'b') = ab'$. *The universal quantifier is* $\forall_c f = f_c f_c' = (ab+b') \cdot (ab+a'b') = ab+a'b'$. *Finally, the existential quantifier is* $\exists_c f = f_c + f_c' = (ab + b') + (ab + a'b') = ab + b'$. *Notice that all operations involving cofactors reduce the support of the function by eliminating the variable with respect to which the cofactors are taken. As an example of* $\geq$ *relation, observe that* $\exists_c f \geq \forall_c f$ *(this is actually true for any Boolean function* f*).*

Boolean functions can be represented as sets because there is an isomorphism between set algebra and binary Boolean algebra. We will sometimes adopt the set representation, because it helps clarifying some concepts. A *minterm* of a function is a point in the n-dimensional Boolean domain of a Boolean function where the function has value 1. Thus, a completely-specified Boolean function can be represented by the set of its minterms (i.e., *on set*). An incompletely specified Boolean function is defined by two sets (either *on set* and *off set*, or *on set* and *don't care set*, or *off set* and *don't care set*).

Any Boolean function can be represented as a *sum of products* (SOP) of literals, or as a *product of sums* of literals. A product of literals is often called a *cube.* One particular *sum of product* representation is the *minterm canonical form*, where each product has n literals. Each product in the minterm canonical form is a minterm of the function. This representation is unique for any given function.

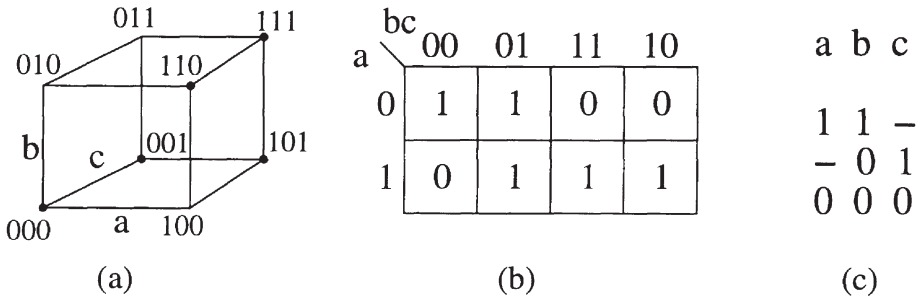


Figure 2.1. Pictorial representations of a Boolean Function

Example 2.2 *Refer to function f of Example 2.1. A sum of product representation of f is $f = ab + b'c + a'b'c'$, but so is $f = abc + abc' + b'c + a'b'c'$. Hence, there are many SOP representations for a single Boolean function. The minterm canonical form of f is: $f = abc + abc' + ab'c + a'b'c + a'b'c'$. The function has five minterms. Figure 2.1 shows three equivalent graphical representations of f. In Figure 2.1 (a), the three-dimensional cube represents B^3 and the darkened vertices are the on set of f. In Figure 2.1 (b), the Karnaugh map of the function is shown [mccl86]. Finally, Figure 2.1 (c) shows the representation of the function as a list of cubes: the "1" represent positive literals, the "0" negative literals and the "–" literals that do not appear in the product. For instance, $ab' \rightarrow 10-$ and $ab'c' \rightarrow 100$.*

Several important definitions and concepts are related to the SOP representation of Boolean functions.

Definition 2.5 *An implicant of a Boolean function f is a product p of literals such that $f \geq p$.*

Any product in a SOP representation is obviously an implicant of f. Particular relevance has the concept of *prime implicant.*

Definition 2.6 *A prime implicant of f is an implicant of f that is not contained in any other implicant.*

To better understand the definition, notice that a product in a SOP represents a set of minterms. If we drop a literal from a product we *expand* the product (i.e., we increase the number of minterms it contains by a factor of 2). A prime implicant is a product that cannot be expanded. If we expand a prime implicant, some of its minterm will not be contained in the *on set* of f. Thus, the product is not an implicant any more.

Definition 2.7 *A cover of a Boolean function is a set (list) of implicants that covers all its minterms*

Any SOP representation of a Boolean function is a cover. A cover is called *prime* if it contains only prime implicants. A *minimum* cover is a cover of minimum cardinality. The concept of minimum cover is important because there is correlation between the number of implicants in a cover of a function (and the number of literals in a cover) and the area of the hardware implementation of the function [dmc94, mccl86]. Numerous algorithms for the minimization of covers have been developed in the recent past (the book by De Micheli [dmc94] contains a good overview). Such algorithms are based on the following theorem:

Theorem 2.1 [*Quine theorem*]. *There is a minimum cover that is prime.*

The most important implication of this theorem is that it defines the search space where a minimum cover can be found. If we restrict our search to covers formed by prime implicants only, we are guaranteed to find a minimum cover. Although the number of prime implicants can be exponential in the number of inputs of a Boolean function, several exact and heuristic algorithms for efficient SOP minimization based on Quine theorem have been developed and successfully applied in industrial-strength CAD tools.

Boolean algebra provides a convenient framework for reasoning about properties of digital circuits. In particular, the steady-state input-output behavior of a feedback-free digital circuits (a.k.a., combinational circuits) can be modeled by a multi-output Boolean function. Large combinational circuits are often better described in a *structural* fashion by *logic networks* (also called *combinational networks*). A logic network is an interconnection of *modules*, representing input/output ports, logic gates or single-output logic functions. The logic network can be represented as a directed acyclic graph (DAG), with vertices corresponding to the modules and edges corresponding to two-terminal nets to which the original module interconnects are reduced.

Notice that a circuit behavior (i.e., a multiple output Boolean function) can be mapped to many equivalent structures. The lack of uniqueness is the main limitation of the structural description. Given two circuit descriptions, checking their equivalence is not an easy task. On the other hand, there are Boolean functions for which all known abstract descriptions have unmanageable size. For such functions, a structural representation is the only viable alternative. Moreover, structural descriptions are conceptually closer to the physical implementation, and are widely used for simulation and performance analysis.

2.2.2 Discrete functions

We have introduced the concept of Boolean function, a mapping from B^n to B. We now present a simple extension of Boolean functions, namely *discrete functions.*

Definition 2.8 *A discrete function $f : B^n \rightarrow V$ is a mapping from a Boolean space to a finite set V.*

Discrete functions encompass Boolean functions as a particular case. When $V = B = \{1, 0\}$, a discrete function is Boolean. Notice however that discrete functions are not more expressive than Boolean functions: since set V is finite, we can "encode" it with $N_e = \lceil log_2|V| \rceil$ binary variable. Thus, a generic discrete function f can be represented by a multi-output Boolean function and an encoding function. Despite this fact, discrete functions are a useful abstraction that will be employed in the development of algorithms for power minimization.

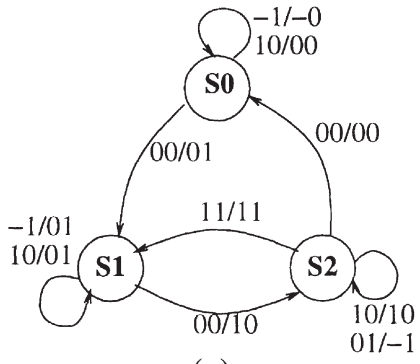
2.2.3 Finite-state machines

We formally define a finite-state machine with binary inputs and outputs as a 6-tuple $(X, Z, S, s^0, \delta, \lambda)$, where $X = B^n$ is the input space, $Z = B^m$ the output space, S is a finite set of states, s^0 is the initial (reset) state, $\delta(x, s) : X \times S \rightarrow S$ is the next state function and $\lambda(x, s) : X \times S \rightarrow Z$ is the output function. FSMs are categorized in two classes [hart66, henn68, koha70]:

Definition 2.9 *A Moore machine is a FSM where $\lambda(x, s) = \lambda(s)$, i.e., the outputs do not depend directly on the input value, but they depend only on the state. A Mealy machine is a FSM for which this property does not hold.*

Finite-state machines can be *incompletely specified.* An incompletely specified FSM is one where $\delta(x, s)$ and/or $\lambda(x, s)$ are incompletely specified Boolean functions. Notice that in the definition of FSM, the state set S is not defined as a Boolean space. In other words, the states are elements of a generic set, and they are uniquely identified by a symbol (a string or any other unique identifier). A FSM can be represented by a graph or, equivalently, by a table. The two representations are called *state transition graph* (STG) and *state transition table* (or state table, for brevity), respectively. The states of the STG are labeled with the unique symbolic state name, the edges are labeled with the inputs and output values. The state table is simply the list of edges of the STG. An example will better illustrate these definitions.

Example 2.3 *Consider the FSM in Figure 2.2 (a), (b). Part (a) shows the STG of the FSM, while part (b) shows the state table. The notation used in the*



PS	In	NS	O
S0	-1	S0	-0
S0	10	S0	00
S0	00	S1	01
S1	-1	S1	01
S1	10	S1	01
S1	00	S2	10
S2	10	S2	10
S2	01	S2	-1
S2	11	S1	11
S2	00	S0	00

(b)

Figure 2.2. State transition graph and state table of a FSM

STG for edge labeling is that the inputs are followed by the outputs. Each edge in the STG corresponds to an entry in the state table. The first two fields in the state table are present state and input. The last two fields are next state and output. The FSM of Figure 2.2 is incompletely specified, because for present state S2 and input 01 the first output is not specified (i.e., it can be either one or zero). The first output is incompletely specified for the first transition in the table as well.

Notice that both STG and state table completely define the input-output behavior of a FSM, but they do not provide any information about the circuit implementation. Hence, STG and state table are *behavioral* representations of the FSM. In order to obtain a representation which is closer to the circuit implementation, we need to introduce the concept of *state encoding.*

Definition 2.10 *A state encoding is a one-to-one mapping from S to B^{N_s}, i.e., a function $E : S \rightarrow B^{N_s}$. The number of state variables is indicated by N_s.*

By specifying the state encoding function E, we associate each symbolic state with a particular binary vector of N_s elements, i.e., with a vertex in the N_s-dimensional Boolean space. The vertex is called *state code.* Notice that we have numerous degrees of freedom in the choice if E. The only important constraint for E is that B^{N_s} has enough vertices to assign a different one to each symbolic state, thus $N_s \geq \lceil log_2|S| \rceil$. Once we have specified N_s and E, the state of a FSM is completely expressed by N_s binary variables called *state variables.*

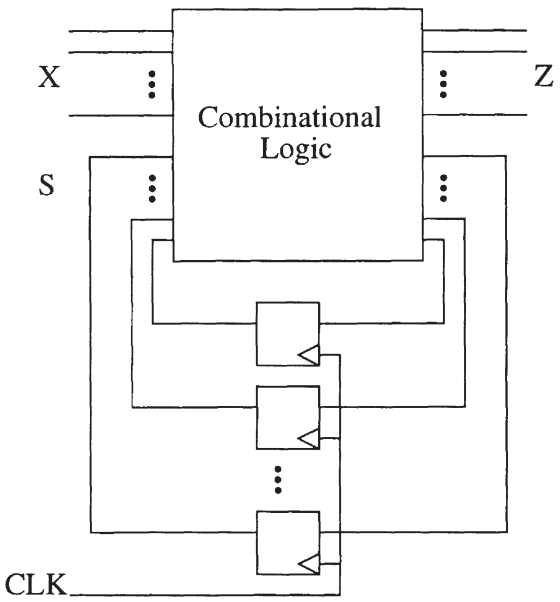


Figure 2.3. Structural representation of a FSM

When we specify E, the output and next state functions become Boolean functions: $\delta(x,v) : B^n \times B^{N_s} \rightarrow B^{N_s}$, $\lambda(x,v) : B^n \times B^{N_s} \rightarrow B^m$, where $v = E(s)$. We consider Mealy FSMs. If the number of elements in B^{N_s} is larger than $|S|$, both $\delta(x,v)$ and $\lambda(x,v)$ become incompletely specified, since their value is not relevant for values of v that do not correspond to any valid state. For the sake of brevity, we will often use the notation $\delta(x,s)$ and $\lambda(x,s)$ even for FSMs where E has been specified, with the convention that s is the state code of the symbolic state.

Once the state encoding has been specified, the structural model of FSM shown in Figure 2.3 can be used to represent the FSM. A (combinational) logic network implements δ and λ, while state elements (flip-flops) store the value of one state variable. The structural model is called *sequential* or *synchronous* network. We assume that the flip-flops are triggered on the raising edge of the clock CLK (the triangular shape within each flip-flop symbol identifies the flip-flops as positive-edge-triggered). Numerous types of sequential elements and clocking styles can be used to implement real-life circuits, but it is useful to reason about a single implementation model.

The representation of Figure 2.3 is *structural*, because it refers to a particular circuit structure implementing the FSM. Notice that there are many structural

representations for a single STG (they can be generated by changing N_s and E). The main reason why structural representations are useful even when developing algorithm or studying properties of FSMs is that they may be much more compact than the state-based representations.

Example 2.4 *Consider a FSM with ten millions of states. If we represent each symbolic state with a string of characters from an alphabet with 32 symbols, we need at least 4 characters (minimum length identifiers) for each state. We want to estimate the memory needed to store the behavioral representation of the FSM. Only storing the names of the states requires* $10^6 * 4 * 5/8 \approx 25Mb$. *In contrast, the minimum number of state variables needed to uniquely represent all states is* $N_s = \lceil log_2 10^7 \rceil = 21$. *Just 21 flip-flops are sufficient for encoding the* 10^7 *states. In other words, the STG of a relatively small sequential circuit with 21 flip-flops may already be unmanageably large.*

Although it is often convenient to reason about small-size sequential system using behavioral representations, structural representations are the only viable alternative for large sequential systems. Similarly to the case of combinational network, the main limitation of the sequential network representation is its lack of uniqueness. Additionally, checking the equivalence of two sequential networks is at least as hard as checking the equivalence of their combinational part.

2.3 IMPLICIT REPRESENTATION OF DISCRETE FUNCTIONS

Digital circuits may have unmanageably large abstract representations. The reason is that both Boolean functions and FSMs can be represented in tabular format, but the size of the tables is worst-case exponential in the number of inputs (and states, in the case of FSMs). One approach that mitigates this problem is to resort to structural representations. The main limit of structural representations is that they are not unique. A Boolean function can be represented by an infinite number of combinational networks, and a FSM can be represented by infinite sequential networks. Hence, it is hard to develop algorithms based on the properties of a Boolean function or a FSM when starting from a structural representation.

In this section we will describe a powerful data structure for the representation of Boolean functions and state machines in a canonical fashion (i.e., with a unique representation) which has in many cases manageable size, even when the tabular representation is excessively large.

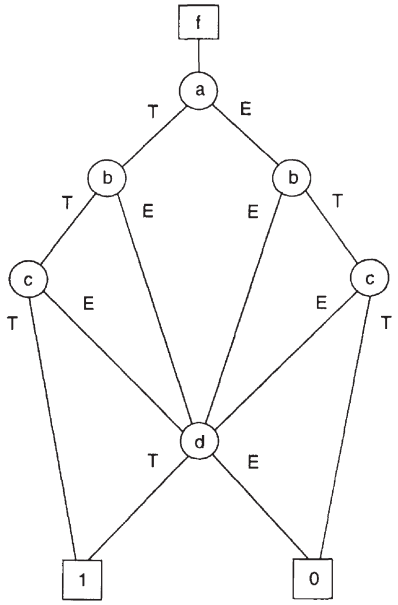


Figure 2.4. A Binary Decision Diagram.

2.3.1 Binary decision diagrams

Binary decision diagrams (BDDs) are a data structure developed for the compact representation of large Boolean functions. Several variants of BDDs have been developed by different groups of researchers (see, for instance, [brya86, brac90, madr88, mina90]). The differences are generally small. In our description we will follow [brya86] and [brac90], though with slightly different notation. We introduce the BDD representation by means of a simple example

Example 2.5 *Let us consider the following Boolean function:*

$$f = abc + b'd + c'd.$$

A BDD for this function is given in Figure 2.4. If we want to know the value of f for a particular assignment to the variables a, b, c, and d, we just follow the corresponding path from the square box labeled f (this node is the root *of the BDD) to one of the square boxes labeled* 1 *and* 0 *(these nodes are the* leaves *of the BDD). Suppose we want to determine the value of function f for the assignment $a = 1$, $b = 0$, $c = 1$, and $d = 0$. The first variable encountered from the root is a, whose value is 1. We then follow the edge labeled T (which stands for* then*). We then come across a node labeled b. Since the value of b is 0, we*

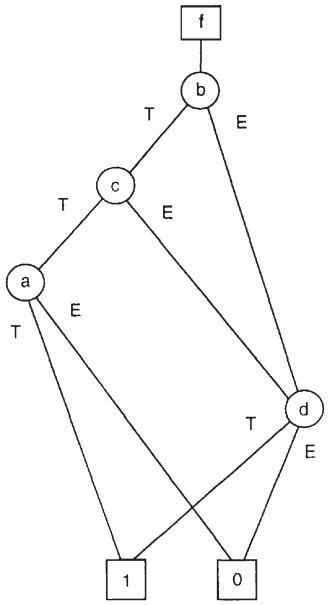


Figure 2.5. An Optimal BDD.

*follow the edge labeled E (*else*). The next node is labeled d, which implies that for $a = 1$ and $b = 0$, the value of f does not depend on c. Following the E edge we finally reach the* leaf *labeled 0. This tells us that the value of the function is 0, as can be easily verified from the sum of products expansion.*

The BDD of Figure 2.4 is an ordered *binary decision diagram, because the variables appear in the same order along all paths from the root to the leaves. The ordering in this case is:*

$$a \leq b \leq c \leq d.$$

The appearance and the size of the BDD depend on the variable ordering. This is illustrated in Figure 2.5, where a different BDD for f is given according to the following variable ordering:

$$b \leq c \leq a \leq d,$$

This is an optimal ordering, since there is exactly one node for each variable. We will assume that BDDs are ordered.

The BDD of f constructed as shown in the example may not be reduced, i.e., it may contain duplicated and superfluous nodes. However, a non-reduced BDD can be systematically transformed into a reduced one by iteratively applying:

- Identification of isomorphic sub-graphs;
- Removal of redundant nodes.

Given an ordering, the reduced graph for a function is unique. Hence, the Reduced Ordered BDD (ROBDD) is a canonical form, i.e., two functions f_1 and f_2 are equivalent (i.e., $f_1 = f_2$) if and only if they have the same BDD. This is the first important characteristics of binary decision diagrams. Other interesting properties of BDDs are:

- The size of the BDD (the number of nodes) is exponential in the number of variables in the worst case. However, BDDs are well-behaved for many functions that are not amenable to tabular representations (e.g., EXCLUSIVE-OR).
- The logical AND and OR of BDDs have polynomial complexity in the size of the operands. Complementation is inexpensive (constant time). All Boolean operators can be implemented using only one basic operator, called ITE (shorthand for *if-then-else*).
- Both satisfiability (i.e., proving that a Boolean function has at least one satisfying assignment) and tautology (i.e., proving that a Boolean function is always **1**) can be solved in constant time. Indeed, f is satisfiable if and only if its BDD is not the terminal node 0. Function f is a tautology if and only if its BDD consists of the terminal node 1.

On the other side, BDDs have some drawbacks:

- BDD sizes depend on the variable ordering. Finding a good ordering is not always simple.
- There are functions for which the sum of products or product of sums representations are more compact than the BDDs.
- In some cases sum of products (and product of sums) forms are closer to the final implementation of a circuit. For instance, if we want to implement a programmable logic array, we need to generate at some point a sum of products or product of sums form.

BDDs allow compact storage and manipulation of multiple Boolean functions, since it is possible to share nodes in the BDD representations of two or more Boolean functions. As a limiting case, two equivalent functions fully share all nodes, i.e., they are represented by the same BDD (not just two identical BDDs).

Given that we are interested in reduced BDDs, instead of generating non-reduced BDDs and then reducing them, we guarantee that, at any time, there are no isomorphic sub-graphs and no redundant nodes in the multi-rooted DAG. This can be achieved by checking for the existence of a node representing the function we want to add, prior to the creation of a new node. A straightforward approach would consist of searching the whole DAG every time we want to insert a new node. However, that would be far too inefficient. Instead, a hash table called *unique table* is created. The unique table contains all functions represented in the DAG. In other words, the table has one unique entry for each node.

BDDs are manipulated with recursive algorithms based on the *cofactor expansion* of Boolean functions f with respect to a variable:

$$f = x_i \cdot f_{x_i} + x_i' f_{x_i}' \tag{2.1}$$

The usefulness of Equation 2.1 lies in the fact that after cofactor expansions, the two cofactors f_{x_i} and $f_{x_i'}$ are simpler functions. All Boolean operations defined in Section 2.2 can be efficiently carried out on the BDDs representing the functions on which they apply. The complexity of the computations is polynomial in the number of BDD nodes of the functions (more precisely, linear in the product of the number of BDD nodes of each operand).

BDD manipulation packages keep a cache of recently computed functions, called *computed table*, for efficiency reasons. The purpose of this table is different from that of the unique table. With the unique table we answer questions like: "Does there exist a node labeled v with children g and h?" On the other hand, the computed table answers questions like: "Did we recently compute the AND of f_1 and f_2?" We can ask this question before we actually know that the AND of f_1 and f_2 is a function whose top node is labeled v and whose children are g and h. Hence we can avoid recomputing the result.

2.3.2 Algebraic decision diagrams

Since discrete functions are a straightforward extension of Boolean functions, the intuition suggests that data structures similar to BDDs could be used to efficiently manipulate them. Indeed, several "BDD-like" data structures have been developed for this purpose [clar93, lai94, baha95]. The one we adopt is the algebraic decision diagram (ADD) [baha95]. We do not describe the definition and the properties of ADDs since they are analogous to the ones we illustrated for BDDs: reduced and ordered ADDs are a canonical representation of discrete functions and all important operations with ADDs can be performed in polynomial time in the ADD size.

The main difference between BDDs and ADDs is in the terminal nodes. While all BDDs terminate in either the "0" or the "1" terminal nodes, ADDs have multiple terminal nodes. The terminal nodes (leaves) of an ADD are associated with the values of the discrete function it represents. Since operators on discrete functions are generalizations of Boolean operators, the ADD algorithms that implement such operators are very similar to the procedures used for BDDs. We review three important operators on ADDs that will be used in later chapters: ITE, APPLY, and ABSTRACT.

ITE takes three arguments: f, an ADD restricted to have only 0 or 1 as terminal values (called "0/1-ADD", and fully equivalent to a BDD), and g and h, generic ADDs. It is defined by:

$$\mathrm{ITE}(f, g, h) = f \cdot g + f' \cdot h \tag{2.2}$$

Intuitively, the resulting ADD has the same leaves as g when f is 1 and the leaves of h when f is zero.

APPLY takes one operator, *op* (e.g., $+$, $-$, $*$), and two operand ADDs as arguments; it applies *op* to all corresponding elements of the two operands and returns the resulting ADD. Notice that APPLY generalizes the Boolean operators. For example, the APPLY of $*$ (multiplication) to two 0/1-ADDs degenerates to the Boolean AND operator.

ABSTRACT reduces the dimensionality of its argument function through existential arithmetic abstraction of some variables. Let x, y be the support of a pseudo-Boolean function $f(x, y)$, where x and y be two disjoint sub-sets of Boolean variables. The arithmetic existential abstraction of x from $f(x, y)$ with respect to the arithmetic sum is defined as:

$$\backslash_x^+ f(x, y) = \sum_x f(x, y) \tag{2.3}$$

This definition tells that, instead of taking the Boolean sum of all the the cofactors associated with the minterms of the x-variables, as in Boolean existential abstraction, the ABSTRACT operator computes precisely the arithmetic sum. Notice that the ABSTRACT operator eliminates x from the support (i.e., $\backslash_x^+ f(x, y) = g(y)$). Similarly, the arithmetic existential abstraction of x with respect to the MAX operator is defined as:

$$\backslash_x^{MAX} f(u) = \max_x f(u) \tag{2.4}$$

Example 2.6 *A discrete function f is given in tabular form in Figure 2.6 (a). The ADD of the function is shown in Figure 2.6 (b), with the variable order $a \leq b \leq c$. The arithmetic existential abstraction of f with respect to variable c is shown in Figure 2.6 (c). As expected, $\backslash_c^+ f$ does not depend on c any more.*

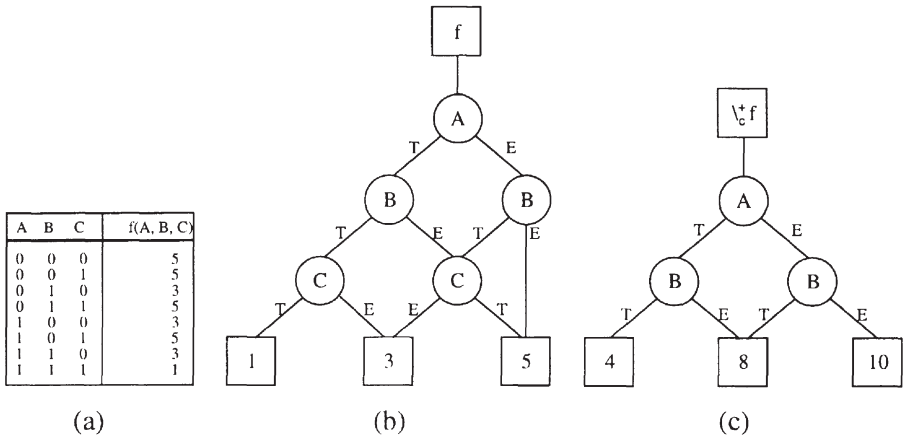


Figure 2.6. (a)-(b) A discrete function f and its ADD. (c) The ABSTRACT of f w.r.t c, $\setminus_c^+ f$.

2.4 MARKOV ANALYSIS OF FINITE-STATE MACHINES

Power is a strongly pattern-dependent cost function, since it depends on the switching activity of a circuit, which in turn depends on the input patterns applied to the circuit. Hence, we need to specify some information about the typical input patterns applied to a circuit if we want to optimize or even only estimate its power dissipation. The most straightforward way to provide information about input patterns is to actually provide a long input stream representing a typical usage pattern together with the specification of the circuit.

This approach gives the most complete information, but it suffers from two serious drawbacks i) the input traces can be very large and cumbersome to manage, ii) in many cases only incomplete information about the environment may be available. A more synthetic description of the input streams provides information about *input probabilities*. Given a circuit, we may simply provide one real number for each input i: the probability for the input to be one, $p_i = Prob(i = 1)$. This statistical parameter is called *signal probability*.

Obviously, signal probabilities express much less information than a complete input trace, for example they do not give any indication about correlation between successive input values (called *temporal correlations*) or between different inputs (called *spatial correlations*). Nevertheless, input probabilities are often used to synthetically describe the statistics of the input patterns.

Even if we assume that the input patterns can be described by signal probabilities with sufficient accuracy (i.e., spatio-temporal correlations are a second-

order effect), it is not obvious how to estimate, in a FSM, the probability of a state transition, since it depends not only on the inputs, but also on the state information. For example, if a FSM has a transition from state s_i to state s_j for all possible input configurations, we may think that this transition will happen with very high probability during the operation of the machine. This may not be the case: if state s_i is unreachable, the machine will *never* perform the transition, because it will never be in state s_i. Similarly, if the probability of being in state s_i is very low, a transition from state s_i to state s_j is very unlikely.

We model the stochastic behavior of a FSM by a Markov chain. A Markov chain is a representation of a finite-state *Markov process* [triv82], a stochastic model where the transition probability distributions at any time depend only on the present state and not on how the process arrived in that state. The Markov chain model for the STG is a directed graph isomorphic to the STG and with weighted edges. For a transition from state s_i to state s_j, the weight $p_{i,j}$ on the corresponding edge represents the *conditional probability* of the transition (i.e., the probability of a transition to state s_j *given* that the machine was in state s_i). Symbolically this can be expressed as:

$$p_{i,j} = Prob(Next = s_j | Present = s_i) \quad i,j = 1,2,...,|S| \tag{2.5}$$

Note that edges with zero conditional probability are never drawn in the graph representation of the Markov chain.

Definition 2.11 *The conditional probability distribution is the set of $p_{i,j}$ values.*

The conditional probability distribution is external input information. It does not depend on the structure of the Markov chain and we assume that it is known. Although conditional transition probabilities can be used as a rough approximation to the transition probabilities [roy93], we need to know the probability of a transition independently of the present state. These probabilities are called *total transition probabilities*, $P_{i,j}$, and can be calculated [triv82] from the *state probabilities*, where the state probability, P_i, represents the probability that the machine is in a given state i.

$$P_{i,j} = p_{i,j} P_i \quad i,j = 1,2,...,|S| \tag{2.6}$$

Equation 2.6 implies that, in order to have high total transition probability *both* the state probability *and* the conditional transition probability must be high. Using only the conditional transition probability can lead to incorrect estimates.

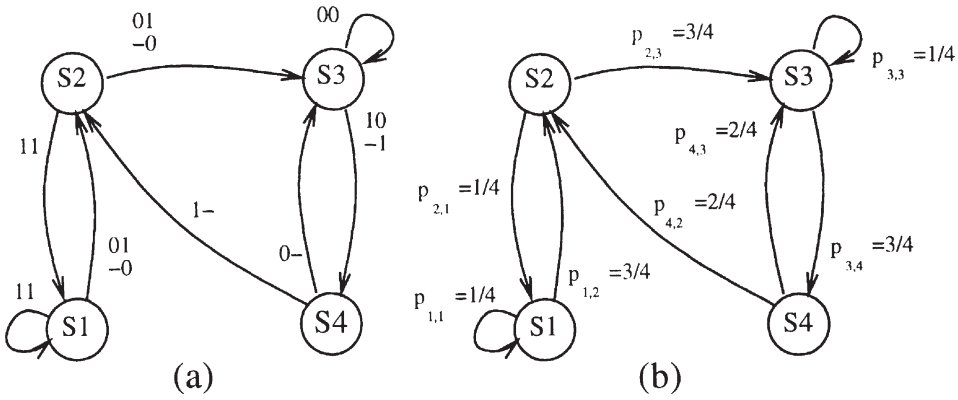


Figure 2.7. (a) A finite-state machine (b) its Markov chain model

Example 2.7 *Consider the FSM shown in Figure 2.7 (a) with two inputs,* in_1 *and* in_2*, and one output. Assume that the input probabilities are* $Prob(in_1 = 1) = .5$ *and* $Prob(in_2 = 1) = .5$*. The Markov chain model of the FSM is shown in Figure 2.7 (b). The edges of the Markov chain are labeled with the conditional transition probabilities of the patterns on the edges of the FSM. For instance, consider the transition between S2 and S1. Its conditional probability is* $p_{2,1} = Prob(in_1 = 1) \cdot Prob(in_2 = 1) = .25 = 1/4$ *(we assumed independent inputs). Observe that the conditional transition probabilities are computed using only the input probability information, and they are generally different from the total transition probabilities.*

The next step is to show that it is possible to compute the state probabilities and, more importantly, to show that these values are not time-dependent. Intuitively, this implies that as the observation time increases, the probability that the machine is in each of its states converges to a constant (stationary) set of real numbers. In other words, we must show that it is possible to compute a *steady state (or stationary) probability vector* whose elements are the stationary state probabilities.

It is quite easy to find STGs for which the stationary state probabilities do not exist, because, for example, their value is oscillatory. The general theory explaining the asymptotic behavior of the state probabilities is too involved to be described here, but it can be found in reference [triv82]. Here we just state an important theorem that allows us to define a large class of STGs whose corresponding Markov chains have a steady state probability vector.

Theorem 2.2 *For an irreducible, aperiodic Markov chain, with all states recurrent non-null, the steady state probability vector exists and it is unique.*

An *irreducible Markov chain with all the states recurrent non null* is a chain where every state can be reached from any other state, and the greatest common divisor of the length of the possible closed paths from every state is one. For the sake of simplicity, we assume in the following that we deal only with FSMs for which the corresponding Markov chain satisfies all requirements of Theorem 2.2 for existence and uniqueness of the state probability vector. We will prove in a later chapter that the presence of a unique reset state is sufficient to ensure the existence of the steady state probability vector. Notice however that state probabilities can be computed in much more relaxed assumptions than those of Theorem 2.2 even for FSMs without a unique reset state [hama94].

In the next subsections we describe the basic methods for computation of the state probability vector i) for small-size FSMs that can be described in a behavioral style, ii) for large FSMs described in a structural fashion.

2.4.1 Explicit methods

The methods for computation of the state probability vector presented in this subsection are called *explicit* because they are based on the direct representation of the conditional transition probabilities with a matrix of size $|S| \times |S|$. Let $\mathbf{P}$ be the conditional transition probability matrix whose entries $p_{i,j}$ are the conditional transition probabilities, and $\mathbf{q}$ the steady state probability vector whose components are the state probabilities P_i (i.e $\mathbf{q} = [P_1, P_2, ..., P_{|S|}]^T$). Then we can compute the steady state probabilities by solving the $(|S| + 1)$ system of equations:

$$\mathbf{q}^T \mathbf{P} = \mathbf{q}^T \tag{2.7}$$

$$\sum_{i=1}^{|S|} P_i = 1 \tag{2.8}$$

The problem of finding the steady state probability vector is thus reduced to finding the left eigenvector of the transition matrix corresponding to the unit eigenvalue, and normalizing it in order to make the sum of its elements equal to unity [triv82]. The normalization condition (Equation 2.8) is required because matrix $\mathbf{P}$ is singular (since all its columns sum to one), and the Equation 2.7 has infinite solutions. The only valid solution is found by constraining the sum of all the components of $\mathbf{q}$ to be one, because the elements of $\mathbf{q}$ are probabilities of mutually exclusive events that cover the complete event space (i.e., the Markov chain is always in one and only one of its states).

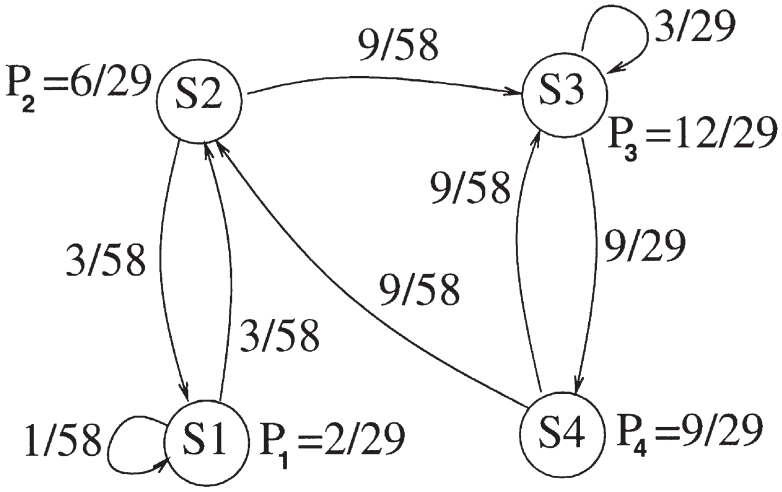


Figure 2.8. Stationary state probabilities and total transition probabilities

Example 2.8 *For the Markov chain of Example 2.7, the stationary state probabilities calculated solving Equation 2.7 are shown in Figure 2.8 besides the nodes in the chain. The state probabilities sum to one, as enforced by the normalization condition. The figure shows also the total transition probabilities (the products $p_{i,j} \cdot P_j$) on the edges. Notice that the total transition probabilities are completely different from the conditional transition probabilities: while all conditional transition probabilities leaving a state sum to one, the sum of all total transition probabilities is one.*

For sparse and large conditional transition probability matrices, the solution of System 2.7 can be carried out using iterative methods that do not require matrix inversion. One iterative method that is very simple and well-suited for state probability computations is the *power method.* With this approach, the state probability vector $\mathbf{q}$ can be computed using the iteration:

$$\mathbf{q}_{n+1}^T = \mathbf{q}_n^T \, \mathbf{P} \tag{2.9}$$

with the normalization condition $\sum_{i=1}^{|S|} P_i = 1$ until convergence is reached. The convergence properties of this method are discussed in [hama94]. The main advantage of the power method is that it can leverage well-developed techniques for the manipulation of large matrices for which the computation of matrix by vector product has complexity proportional to the number of nonzero elements of $\mathbf{P}$.

The main limitation of the methods presented in this section is that they are not applicable to Markov chains derived from large sequential circuits for which the state set is large enough to make even the storage of matrix $\mathbf{P}$ a formidable task.

2.4.2 Implicit methods

When dealing with sequential circuits with thousands or even millions of states, specified by a structural description, it is very hard to extract and manipulate the transition probability matrix. Implicit methods allow the manipulation of large systems by representing the transition probability matrix with an ADD.

The STG of a finite-state machine is implicitly represented by the BDD (or, equivalently, by a 1/0-ADD) of its *transition relation* [coud89]. The transition relation is a Boolean function $T(x, s, ns) : B^N \times B^{N_s} \times B^{N_s} \rightarrow B$. The support of the transition relation consists of: i) the input variables x, ii) the state variables s, iii) the *next state* variables ns. T has value 1 when the STG of the machine has a transition from state s to state ns with input x, zero otherwise.

The transition relation can be extracted directly from the next state function $\delta(x, s)$ with an iterative procedure of fixed point computation. Starting from an initial state (e.g., the reset state of the machine), the set of states that can be reached from it in one clock cycle is computed. The procedure then iterates computing all states reachable from the set of states reached in the previous iterations. The iteration stops as soon as no new states are reached in two successive iterations (i.e., a *fixed point* has been reached).

The fixed point computation can be carried out by operating directly on BDDs. The transition relation is computed without ever enumerating the states. BDD-based algorithms for the computation of $T(x, s, ns)$ have been able to completely explore the state spaces of machines with more than 10^{20} states, for which any enumerative algorithm would be completely impractical. The interested reader can refer to [some96] for a review of BDD-based procedures for transition relation computation.

Similarly, the input probabilities can be represented by an ADD. The ADD $PI(x)$ is extracted from the array of input probabilities, with the simple formula $PI(x) = \prod_{i=1}^{m} PI_i(x_i)$ where each $PI_i(x_i)$ is a single-node ADD, with two leaves with value p_i and $1 - p_i$.

Given T and PI, the implicit representation of matrix $\mathbf{P}$ can be obtained by the following symbolic formula:

$$\mathbf{P}(x, s) = PI(x) \cdot \backslash_{ns}^{+} T(x, s, ns) \tag{2.10}$$

Equation 2.10 produces an ADD with the conditional input probabilities of the transitions of the STG as leaves. The value of x and s are sufficient to identify

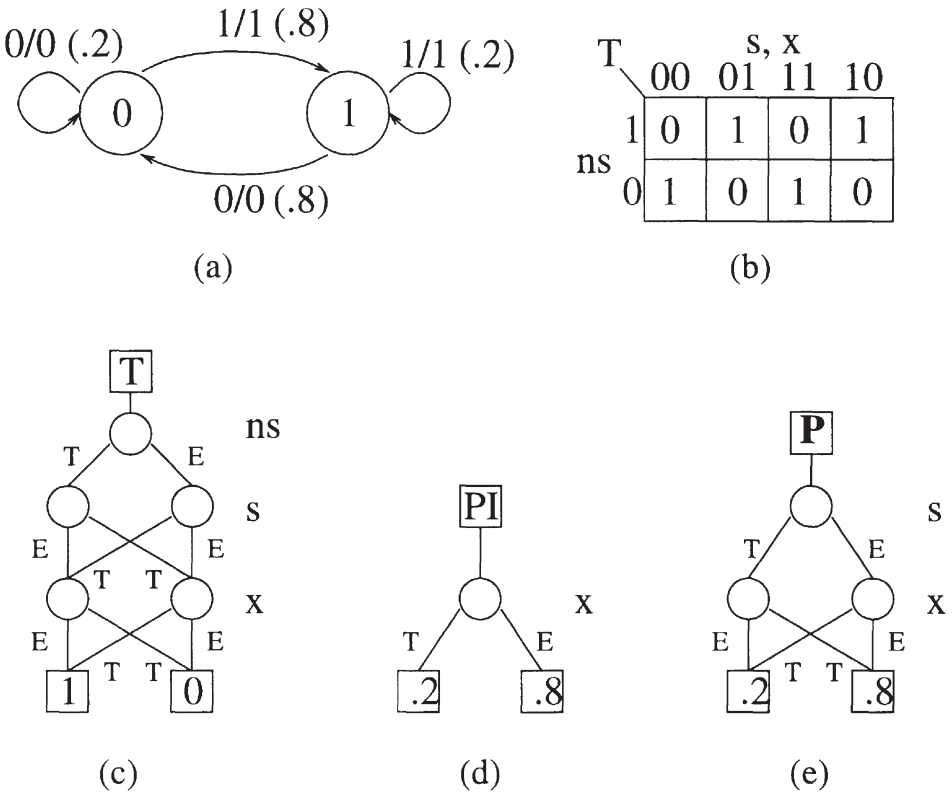


Figure 2.9. Symbolic computation of the conditional transition probability matrix

every transition in the STG. Notice that the ADD of **P** can be exponentially smaller than the traditional matrix representation.

Example 2.9 *The STG of a finite-state machine is given in Figure 2.9 (a). Notice that each edge is labeled with input, output and conditional transition probability. The transition relation $T(x, s, ns)$ is shown in Figure 2.9 (b) in the tabular format. The BDD (0/1 ADD) of T is shown in Figure 2.9 (c). The ADD of the conditional input probability is shown in Figure 2.9 (d). If we compute the ADD of matrix* **P** *using Equation 2.10, we obtain the result shown in Figure 2.9 (e). Although we used a small example for the sake of explanation, notice that symbolic representation based on ADD becomes useful when the STG and the truth table of T are unmanageably large.*

Once $\mathbf{P}(x, s)$ has been computed, the state probability vectors can be computed using the symbolic version of the power method. The matrix-by-vector multiplication required for the application of the power method (Equation 2.9) is implemented as an ADD operator, and the final result is $\mathbf{q}(s)$, the ADD whose leaves are the state probabilities. Notice that other implicit methods have been presented in the work by Macii et al. [hach94], as well as a detailed study of the convergence properties of symbolic algorithm for the calculation of $\mathbf{q}$ in the case of large FSMs that do not satisfy Theorem 2.2.

2.5 CONCLUSION

In this chapter we covered the background material required for full understanding of the power optimization algorithms presented in the following chapters. The key concepts introduced so far are: Boolean and discrete functions, the quantification operators, BDDs, ADDs and Markov models for finite-state machines. The BDD/ADD-based symbolic representation of Boolean functions, FSMs and Markov chains is a useful tool when dealing with systems with a large number of inputs and state variables.

3 FUNDAMENTAL CONCEPTS

Otia corpus alunt, animus quoque pascitur illis.

Idleness nourishes the body and the soul.

—Ovidius

65

3.1 INTRODUCTION

In this chapter we focus on the application of power management techniques to VLSI logic design. Although only a fraction of the power dissipated by a complex digital system is actually consumed by VLSI circuits, it is often a large (and increasing) one. We further narrow our discussion to the dominant CMOS technology and to clocked circuits (also known as *synchronous* circuits). Many of the basic concepts presented in the following treatment have much wider scope of applicability. We believe however that a focused approach may give the reader a more concrete understanding of the basic principles.

The chapter takes a designer's perspective. The fundamental ideas that allow the identification and exploitation of power management opportunities are informally introduced, and practical examples are provided. The problems and tradeoffs involved in the design of power-managed digital circuits are investigated in detail. At the same time, it will become clear why power management is not a panacea, and a hard tradeoff is involved in the choice of which particular power management scheme can be used. Moreover, we analyze alternative implementations of the circuitry that must be added to baseline functional units in order to power-manage them.

The treatment of the power management concept is purposedly informal. We do not want to obscure the key ideas with unnecessary formalism. Moreover, we do not discuss the automation of the described techniques. Computer-aided tools for automatic synthesis of power managed circuits will be carefully analyzed in the next chapters. The following sections are a prerequisite for the complete understanding of the basic and advanced computer-aided techniques.

3.2 FUNDAMENTALS

Consider a sequential logic circuit (a computational *unit*) U embedded in a larger system, as shown in Figure 3.1. Our task is to find when the computation performed by U is not required. In other words, unit U is *idle* when the system does not need its functionality to operate correctly. Observe that idleness is very common in real-life systems. For instance, high-performance microprocessors contain a massive amount of hardware resources which are not always used. Such resources are indeed required to achieve high peak performance, but peak performance is not always achievable or needed. More in general, although "silicon real estate" is still a precious resource, the high level of integration of state-of-the-art devices enables the instantiation of hardware resources which are not always utilized during normal system operation. As a result, digital systems grow in size and are seldom fully utilized at any given time.

The degree of system utilization varies widely during operation. When the environmental conditions require peak performance, utilization is very high,

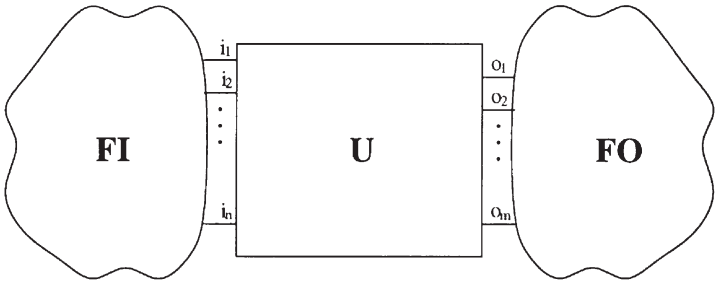


Figure 3.1. A computational unit embedded in the environment

and all computational units are exploited. During low utilization intervals, the idle units are computationally redundant. This observation is the key for the basic power management strategy. Computationally-redundant units can be shut down and their power dissipation minimized. The decision of shutting down a resource is taken *during operation* in a dynamic fashion. Referring to Figure 3.1, unit U is obviously not always functionally redundant (if that is the case, it can be eliminated), but it may be so for a large fraction of the operation time. Ideally, we would like to shut down the resource every time it is idle. This is not always possible. Detecting idleness can be expensive, and there is a tradeoff between the cost of idleness detection and the power savings provided by shutdown.

It is important to notice that power optimization based on dynamic power management has no analogy when considering area optimization under performance constraints. When unit U is instantiated, there is a direct tradeoff between the performance improvement provided by U and its area cost. On the contrary, the tradeoff between performance and power is not as direct. If a unit is instantiated to provide peak performance improvement, and it has very high idleness, power management can minimize its power cost. Hence, dynamic power management introduces completely new degrees of freedom that can be exploited to reduce power dissipation with minimal or no impact on performance.

The computation performed by U becomes redundant only when the environmental conditions are such that its computation is not observed at the system outputs, or when its own outputs do not change even if the input have changed. We call the first type of idleness *external*, as opposed to *internal* idleness, in the second case. These are two distinct types of idleness that can be exploited for devising power management strategies. We analyze them separately in the following subsections.

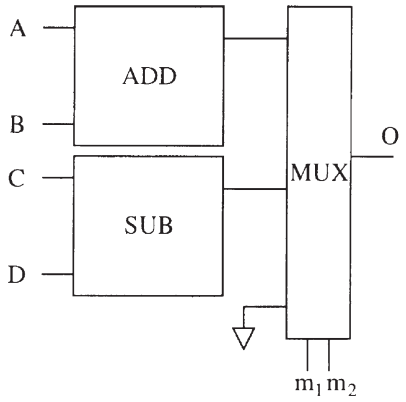


Figure 3.2. An example of circuit with idleness

3.2.1 External idleness

External idleness is not a property of a computational unit in isolation. It is induced by the environment. Consider the circuit shown in Figure 3.2. We focus on the adder unit ADD. The output of the adder is connected to a three-way multiplexer MUX. When the control inputs of the multiplexer have the value $m_1 = 0$, $m_2 = 0$, the output of ADD is driven onto the primary output. Whenever $m_1 \neq 0$ or $m_2 \neq 0$, the value computed by ADD is not observed at the primary output. This is a typical example of external idleness caused by a unit's environment.

The power dissipation of the circuit can be reduced if we prevent the execution of the addition whenever the output of ADD is not selected by the multiplexer. There are several alternative ways to prevent the execution of the addition: for example, we could simply turn off the power supply of all gates in the adder. This and other methods will be described in greater detail in the next section. Preventing the execution of the addition causes some internal nodes of the circuits to have an incorrect logic value (for instance, the outputs of ADD may be fixed at zero when the adder is shut down). The presence of the multiplexer ensures however that the incorrect logic value do not propagate to the primary outputs.

Whenever the multiplexer selects the output of ADD, the adder must be re-activated. Re-activation must be efficient: in the example of Figure 3.2, the adder must be re-activated and complete its computation within the timing constraint imposed by the propagation of the sum to the primary outputs. Hence, the shutdown mechanism must not only be effective in eliminating the

power dissipation, but it must also impose a minimal performance overhead when normal operation resumes.

External idleness is closely related to the concept of *observability*. In intuitive terms, a logic signal is not observable when a change in its value does not cause any change at the primary outputs of the circuit. In our example, multiplexer control values $m_1 \neq 0$ or $m_2 \neq 0$ cause all internal signal and outputs of ADD to become unobservable. Hence, whenever *all outputs* of a unit are unobservable, the unit is externally idle, and can be shut down to reduce power consumption.

External idleness is very common in practical systems. Often several computations are performed in parallel, but only one of them is actually useful. All instances of such behavior are good candidates for investigating the opportunity of power optimizations based on dynamic power management. Unfortunately, in many cases, redundant computation is a price that is willingly paid in order to improve performance, and it may not be possible to power manage a system because of performance constraints.

As an example, consider again Figure 3.2, and assume that m_1 and m_2 are late arriving signals. In this case, ADD and SUB, although mutually exclusive, are performed concurrently for performance reason. Since the multiplexer selection is performed very late, all possible results are computed in advance (before the selector value is known) and multiplexed to the output as soon a the selector value is available. This is a simple example of *speculative execution*, a well-known performance-enhancement practice. Adder shutdown would not be possible in this case, because the decision of shutting down the resource is based on the values of m_1 and m_2. Assume that the adder is shut down in cycle n, and it should be active in cycle $n+1$. The values of m_1 and m_2 are computed too late in cycle $n+1$ to turn the adder on and to compute the sum before the end of the clock cycle.

Several designers have implemented power management schemes based on external idleness. The IBM PowerPC microprocessor [slat95] implements some power-reduction techniques, including power-down of execution units based on instruction activity. For instance, if the processor is running integer code, the floating-point units can be powered down. This is a typical example of exploitation of external idleness. The environment around the floating point unit does not need its computation to perform correctly, hence the unit can be powered down.

We summarize the conditions which enable the exploitation of external idleness.

- Existence of environmental conditions for which unit U is unobservable at the primary outputs.
- Early availability of the signal used to detect the lack of observability.

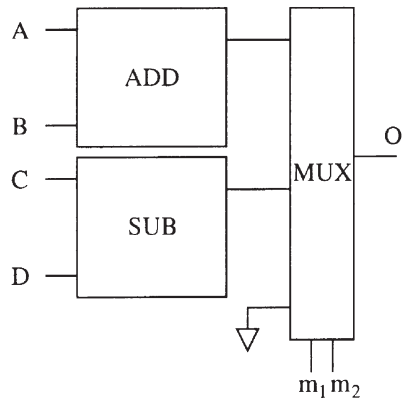


Figure 3.3. Circuit with idleness.

- Efficient shutdown mechanism (low overhead in speed, area and power).

One important point that must be considered is that the existance alone of external idleness conditions is not enough to establish the usefulness of power management. The typical usage patterns for the system must also ensure that the unit is idle for a sufficiently high fraction F of the operation time. The minimum value F_{min} of F for which power management becomes advantageous clearly depends on the cost of the power management circuitry needed to detect idleness and to shut down the unit.

3.2.2 Internal idleness

While external idleness is caused by the environment, internal idleness depends exclusively on the functionality of a unit U. Consider again the example of Figure 3.2 (repeated in Figure 3.3 for convenience). In contrast with the previous discussion, we consider the entire circuit as a single unit U. No information is available on the environment of U, hence we assume that the primary outputs of U are fully observable (this implies the absence of external idleness).

When the control inputs of the multiplexer in Figure 3.2 have value $m_1 = 1$ and $m_2 = 1$, the bottom data input is selected. Observe that the bottom data input is connected to the constant 0 logic value: the output of U is always zero when $m_1 = 1$ and $m_2 = 1$. Suppose that both the control inputs are held at 1 for two or more clock cycles. After the first clock cycle, the outputs of U are stuck at zero until a change of control input values occurs. It is easy to observe that, as long as $m_1 = 1$ and $m_2 = 1$, all power consumed by the circuit is wasted, because the outputs do not change.

Notice that the power consumed in this condition may be significant. Although we assumed that the control inputs of the multiplexer are held constant, we did not make any assumption on the switching activity of the data inputs of the adder and subtracter. Thus, the two arithmetic operators may continue to compute new results, but with no outside visibility because the multiplexer always selects the constant zero value. We call this behavior *internal* idleness because a logic condition *within* the unit itself is causing useless power consumption.

The main difference between internal idleness and external idleness is that the first one is inextricably tied to the observation of the circuit behavior in at least two successive clock cycles. In the example above, it is not possible to take the decision of shutting down U if we do not know that m_1 and m_2 have been equal to zero for at least two clock cycles. Moreover, the behavior of U when it is shut down is more constrained than in the case of external idleness. Some type of state-holding mechanism is required in order to exploit internal idleness.

To clarify this statement, observe that when a unit is externally idle, its outputs are unobservable, hence, their value is irrelevant. On the contrary, the outputs of an internally-idle unit may be observable, although they do not change. Therefore, a shutdown mechanism must guarantee that the values of the outputs are held constant when the unit is shut down (i.e., at least the outputs of the idle circuit must hold their state). Static CMOS logic is inherently suitable for the exploitation of external idleness because it keeps indefinitely correct output values and dissipates little power if its inputs are stationary. Hence, simply freezing the inputs of a combinational logic unit in static CMOS technology guarantees both low power dissipation and state holding. Notice however that some of the shutdown methods presented in the following section are not suitable for exploiting internal idleness. For instance, supply shutdown is not acceptable, because it forces all outputs to logic zero.

Again, several tradeoffs are involved in the exploitation of internal idleness. First, detecting internal idleness requires the presence of some circuitry whose function is to reveal conditions under which outputs do not change. In our contrived example, this is an easy task (it is sufficient to detect when $m_1 = 1$ and $m_2 = 1$), but it is not generally true. Power management based on internal idleness can be successful only if the detection of internal idleness is much less power consuming than the useless computation it avoids. It will become clear in the following chapters that detection of *all* internal idle conditions is as complex as computing the output values. Hence, internal idleness can be fruitfully exploited only if a limited set of idle conditions is detected, namely those that have high probability to be verified.

Fortunately, there are large classes of practical systems for which power management based on internal idleness is very advantageous. Such systems operate in a *reactive* fashion: they sample the inputs and take actions (i.e., update their outputs) only when some (rare) input conditions are observed. Typical examples of reactive systems are pagers. For a very large fraction of the operation time, the pager circuitry is idle and it needs to perform power consuming computations only when the pager number is sampled on the input channel. Indeed pagers achieve very low power consumption thanks to a clever exploitation of idleness [mang95]. The paging functionality is intrinsically a reactive one, and pager systems have a large (and easily detectable) amount of internal idleness.

We will analyze the system-level implications and opportunities in Chapter 6. Even at the logic level, several devices have reactive behavior. For instance, interface circuitry and control logic is often reactive. In Chapter 4 we will introduce techniques for the automatic power optimization of control circuits based on internal idleness.

3.2.3 Taxonomy

Having introduced internal and external idleness and their properties, we can now propose a taxonomy for the classification of power management techniques. The taxonomy is pictorially represented in Figure 3.4. Internal idleness is a property of the unit itself, and does not critically depend on information about its environment. The exploitation of internal idleness requires *state holding* to be available, because the outputs of the unit may be observable when the unit is idle, hence we must guarantee that their value is held constant throughout the idleness interval. There are no requirements on where and how the state is held, as long as the stationarity of the outputs is guaranteed.

External idleness is essentially a property of the environment of a unit. Because of that, exact computation of external idleness can be difficult, if not impossible. In many cases, only partial information about the environment is available and external idleness must be computed conservatively. On the other hand, exploitation of external idleness may not require state holding for the idle unit, because the outputs of the unit are not observable and can assume any value. The only requirement is that correct operation must be resumed as soon as the outputs become observable. For this reason, in some cases state holding is required even for externally idle units. The diagram in Figure 3.4 shows the relation among the types of idleness and the requirements on state holding.

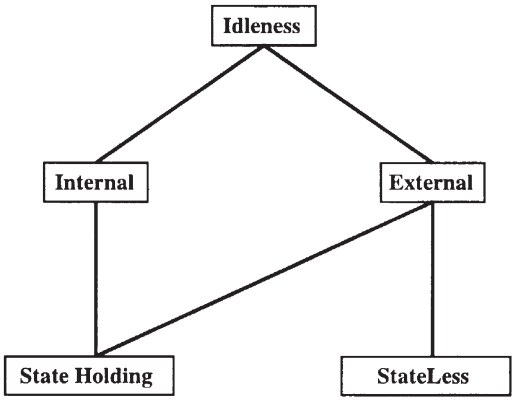


Figure 3.4. Idleness classification and properties

3.2.4 Granularity

Power mangement does not come for free. Idleness must be detected by some means in order to be able to stop the computation when it is not needed. The only way to detect idleness is to instantiate additional circuitry. Moreover, the original unit must be modified to make shutdown possible. As a result, power-managed units require some functionally-redundant circuitry which has a non-negligible cost i.e., an overhead with respect to the baseline implementation without power management. Such overhead must be carefully taken into account when deciding on the opportunity of building a power-managed unit.

A critical issue is the identification of the level of granularity at which it is reasonable and convenient to investigate the opportunity of power management. Consider a generic logic circuit as shown in Figure 3.5. The atomic logic functional blocks are either memoryless combinational gates or sequential memory elements (flip-flops). Applying power management on a gate-by-gate basis (or on a single flip-flop at a time) is obviously counterproductive in the vast majority of the cases, because detection of idleness is more power consuming than the simple computation performed by a gate, not to mention the area and delay overhead.

Power management should be investigated at a level of granularity large enough to tolerate the overhead of power-management circuitry. In practice a good rule of thumb is that the power management logic should be at least ten times smaller than the unit it manages. We shall see in the following section that a very low-overhead scheme requires at least three logic gates. For that particular scheme, the minimum granularity for applicability is a few tens of logic gates. Notice that the 1/10 rule of thumb is only a very rough estimator.

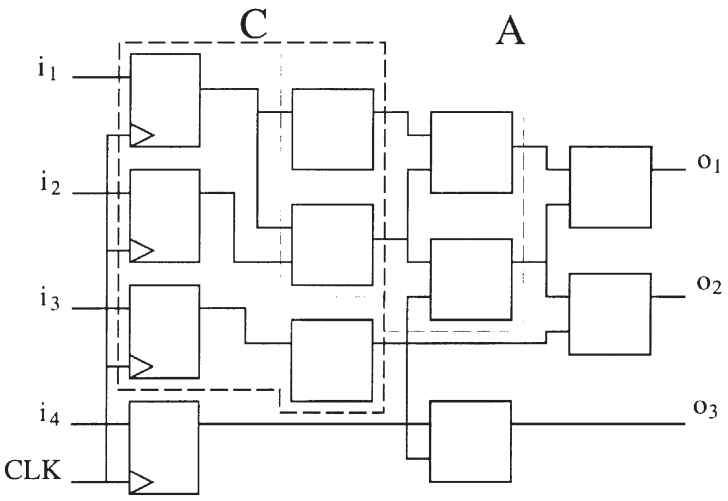


Figure 3.5. A generic digital logic circuit

Area or speed constraints could greatly restrict the sections of the designs where power management is possible. This type of considerations is strongly instance-dependent.

Another important decision is the selection of the boundary of the power-managed unit. In Figure 3.5 two boundary selections are shown. Selection A has at least one combinational gate at its input boundary. Whenever a selection has only conbinational gates at its input boundary, we say that it has *combinational boundary.* On the other hand, the input boundary for selection C consists of only sequential primitives (flip-flops, in the example). In this case we say that C has *sequential boundary.*

Although it is possible to formulate power management techniques which are applicable to regions like A [tiwa95], the implementation of such techniques poses several practical problems. The main concern when considering units with combinational boundary is that combinational gates are *always* sensitive to changes on their inputs. If we want to prevent transitions at the output of a combinational gate, we need to ensure that the power management control is asserted at the gate before the arrival of any of the inputs, otherwise it would be difficult, if not impossible to avoid the propagation of some transition activity into the unit.

When only sequential elements are at the input boundary of the unit, we can exploit a useful property. Generic sequential elements propagate transitions to their outputs only when some condition is verified on their clock inputs. In

Figure 3.5, edge-triggered flip-flops are instantiated. Their outputs propagate the input value only when the clock signal has a raising transition. Hence, when power managing the unit, we just need to ensure that power management controls are asserted before the raising edge of each clock cycle. It is generally much easier to design power management circuitry for units with sequential boundary. Regions like C are the preferred choice for the vast majority of power management schemes.

3.2.5 Input statistics

In the previous subsection we focused on the logic-level specification of the system to define idle conditions and address granularity issues. However, the logic-level specification is only one part of the information needed to perform an optimal choice of power management strategy. The additional information required is data on typical usage patterns of the target unit.

Typical usage patterns can be obtained by simulating the unit within the complete system and collecting the stream of input values. Alternatively, more compact (and less accurate) representations can be used. A widespread compact representation style makes use of *input probabilities* (cfr. Chapter 2). For each input i of the unit, a single parameter is provided, namely the *signal probability* p_i, $0 \leq p_i \leq 1$ (cfr. Chapter 2).

Obviously, given the input pattern stream obtained from simulation, it is possible to compute an estimate of the signal proabibility for each input:

$$p_i = \frac{1}{N_{patt}} \sum_{c=1}^{N_{patt}} i^c \tag{3.1}$$

where N_{patt} is the number of patterns in the stream and i^c is the value of input i in pattern c. Input probabilities are a very compact way to describe input pattern distributions. Compactness does not come for free. Indeed, input proabilities do not completely characterize input streams, and are a lossy description style.

Consider for example two stream of values of input i. In the first stream, i has value 0 for the first $N_{patt}/2$ patterns, and value 1 for the remaining ones. In the second stream, the value of i changes on every clock cycle. According to Equation 3.1, p_i is the same for the two streams, but obviously the streams are very dissimilar. The loss of information caused by using input probabilities to describe typical usage patterns for a unit should always be acknowledged and taken into account when evaluating the opportunity of implementing a power management policy.

The knowledge of typical usage patterns is critical information when selecting which idle conditions to exploit for implementing a power management strategy.

It would be useless to implement circuitry for the detection of idle condition which are never (or seldom) verified during normal operation of the unit. On the contrary, detection effort should be focused on highly-probable idle conditions because they ensure a high proability of unit shutdown.

As an example, consider the circuit of Figure 3.1. Assume that the probability of having $m_1 = 1$, $m_2 = 1$ (i.e., internal idleness) is on the order of 10^{-5} for a typical input stream, while the probability of having $m_1 \neq 0$ or $m_2 \neq 0$ (i.e., external idleness for ADD) is much larger (0.2). Inserting power management circuitry for detecting internal idleness would almost certainly be counterproductive, because the probability of actually stopping unit U is extremely small. On the contrary, the adder is idle (externally) for 20% of the unit operation time, hence it may be very advantageous to power manage it.

3.3 POWER MANAGEMENT CIRCUITRY

The key idea in power management is that unused parts of a complex design can be shut down during system operation. In this section we analyze several alternative implementations of shutdown schemes for static CMOS logic circuits. The alternatives span the tradeoff between implementation complexity and shutdown effectiveness. On one side there are aggressive techniques that virtually eliminate any power dissipation on a shut down component, but they require hard-to-design circuitry and/or impose a significant overhead with respect to the baseline implementation. On the other side we have conservative techniques that are relatively easy to implement but do not completely nullify the power dissipation of a shutdown unit.

3.3.1 Power supply shutdown

The most effective way to eliminate power dissipation for an idle unit is to turn off its power supply. Conceptually, this is a simple operation: it consists of inserting a switch on the power supply of the unit, and turning the switch on or off according to the power management control. This solution is shown in Figure 3.6 (a). When power is turned off, no power is consumed by the unit, hence this solution guarantees the absolute minimum power consumption (zero) during shutdown periods.

The ideal switch depicted in Figure 3.6 (a) is of course a drastic simplification of reality. Any real switch has a resistance and a delay associated with it. For example, the most straightforward implementation of the ideal switch is a MOS transistor. To minimize the ON resistance of the transistor we must increase its width. Unfortunately, transistor width is proportional to gate capacitance. Hence, the power management control input is heavily loaded. Additional buffering circuitry (that consumes power, occupies area and increases

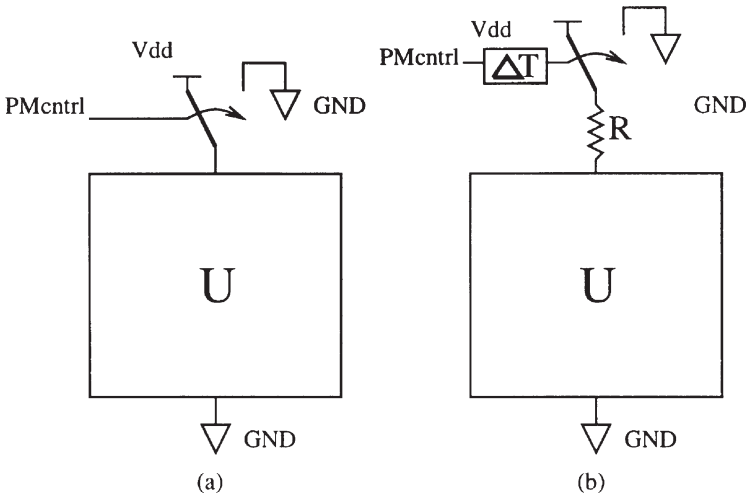


Figure 3.6. Power management by supply turn-off

delay) is needed to drive the control input of the MOS switch with satisfactory performance. Figure 3.6 (b) shows a more realistic model of a circuit where power supply can be turned off. The effect of the switch resistance and of the control delay must be taken into account when evaluating the opportunity of power managing the unit.

Even the model shown in Figure 3.6 (b) is a drastic simplification of reality. In actual circuits, a stabilized voltage supply circuit is required. The analysis of such devices is outside the scope of our treatment. Nevertheless, the simplified model is accurate enough to represent the non-ideal efficiency of the voltage supply and its stabilization delay. These are the key parameters to be considered in tradeoff evaluation. The presence of a stabilization delay ΔT limits the applicability of power-managed solutions to resources that are idle for time intervals much longer than ΔT. Moreover, the loss in efficiency due to the insertion of a controllable voltage supply must be more than compensated by the power saving.

Several other difficulties are involved in the power supply shutdown scheme. The physical placement of the controlled voltage supply (or the power switch) is an important decision. The transient noise generated by turning on and off the power supply must be adequately shielded to avoid cross-coupling with functional signals and consequent functional failure. Moreover, the layout of power the distribution grid is often carefully designed for maximum efficiency,

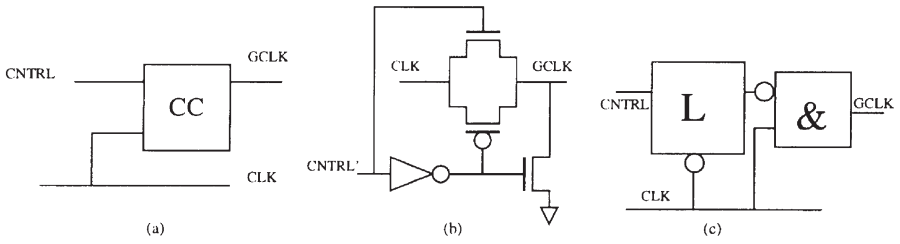


Figure 3.7. Gated clock implementation

and the presence of additional circuitry may perturb this highly tuned and critical setup.

If the unit contains volatile memory (such as registers or RAM), all state information is lost when power supply is shut down. In case some state information is needed when execution resumes, additional overhead is encountered for restoring the state of the unit upon re-assertion of the power supply. First, the state information must be stored elsewhere in the system during the shutdown period. Second, it must be loaded in the internal memory of the unit before normal execution can resume. The overhead for state storage and startup may be substantial in both time and power dissipation.

Because of the aforementioned practical difficulties, the power supply shutdown scheme is employed at a very coarse level of granularity. The power supply of entire chips or boards can be turned off to save power, but we are not informed of any attempt of applying this power management technique on relatively small units inside a chip. System-level power management is studied in a later chapter. At the logic level (and at fine granularity) power supply shutdown is not used and other power management schemes are generally exploited.

3.3.2 Clock gating

If candidate units for power management are selected according to the granularity guidelines specified in the previous section, the input boundary of a unit is a set of sequential elements. In this case, an effective and low-overhead shutdown scheme is based on the concept of *gated clock*. A gated or *qualified* clock is a clock signal that can be stopped (i.e., forced to a constant logic value) by a logic control signal which is computed during normal operation. The abstract gated clock generation scheme is shown in Figure 3.7 (a).

When signal $CNTRL = 1$, the clock-gating logic CC freezes the clock to a constant logic value (either 0 or 1). A simple circuit for implementing CC

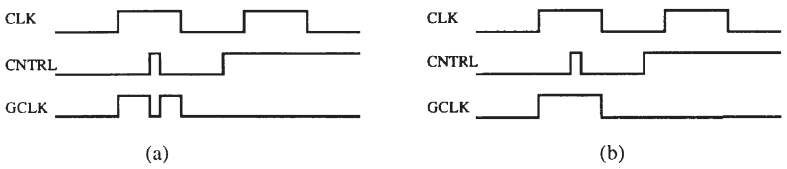


Figure 3.8. Timing diagram of the GCLK signal

(taken from [west92]) is shown in Figure 3.7 (b); the CMOS pass-transistor is controlled by signal $CNTRL$. In this case we use the negative polarity of $CNTRL$. When $CNTRL = 1$, its complement $CNTRL' = 0$. The pass-transistor is OFF and the NMOS transistor is ON, connecting $GCLK$ to ground. In this implementation, the clock is held at the logic zero value when it is gated.

In the gated-clock-based power management scheme, the clock inputs of all flip-flops at the input boundary of the unit are connected to the $GCLK$ signal. When the gated clock is stopped, power consumption is minimized in the power-managed unit because the flip-flops do not receive any raising clock edge, hence their outputs do not change, no matter what happens to their data inputs. Notice that, differently from the supply shutdown case, power dissipation is not nullified in the unit. Leakage power is still dissipated because it does not depend on signal transitions. Moreover, clock gating is specifically targeted towards static CMOS technology. State storage elements in dynamic CMOS logic need to be refreshed periodically in order not to loose the state information. Fortunately, this is not true in static CMOS, where the value stored in the flip-flops is preserved indefinitely (if power supply is maintained).

Although the gated clock implementation of Figure 3.7 (b) has a low overhead, it has one serious limitation. If signal $CNTRL$ has a glitch (i.e., a 0-1-0 transition) when CLK is high, the glitch propagates on the $GLCK$ output, producing one additional raising edge on the flip-flops clock input (as shown in Figure 3.8 (a). The additional raising edge may have disastrous effects on the functionality, because it causes sampling on the inputs before the end of the cycle time. This behavior must be avoided to preserve correct functionality. The simplest way to do that is to require stability of $CNTRL$ when $CLK = 1$. Unfortunately this requirement is a severe one, and it may greatly complicate the logic for computation of $CNTRL$.

An alternative, and better solution is shown in Figure 3.7 (c). The block labeled "L" represents a latch, transparent when the global clock signal CLK is low. The presence of the latch L is required for the correct behavior of the proposed gated clock implementation: it filters any spurious transitions

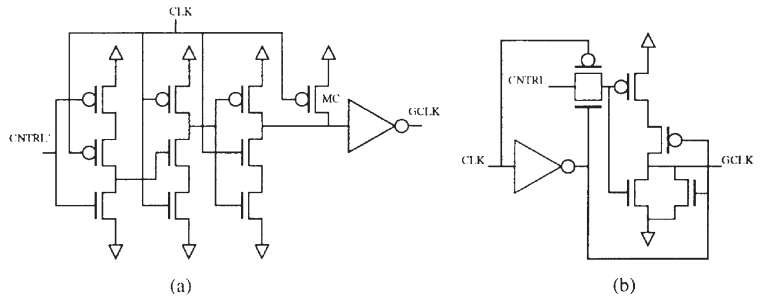


Figure 3.9. Optimized clock-gating circuit

on $CNTRL$ that may occur when CLK is high, as shown in Figure 3.8 (b). $CNTRL$ may also glitch when the clock is low, but in this case the AND gate itself will filter out the spurious transitions, because the global clock signal has the controlling value.

This solution has a larger overhead than the simple implementation based on the pass-transistor gate. In particular, speed overhead must be considered. The simple solution adds only a pass-gate delay on $GCLK$ with respect to CLK, while the latch-based solution adds the latch and the AND gate delay on GLK. In order to verify the correctness of the gated clock implementation, we first need to ascertain that timing constraints are not violated by $CNTRL$. To check this condition, we first perform timing analysis of the logic network that generates the $CNTRL$ signal. We call the critical path delay through this network T_{crit}.

If we collect the delays through the latch and the AND gate (Figure 3.7) and the setup time of the input flip-flops in one worst-case parameter T_{wc} we obtain the following constraint inequality for the activation function:

$$T_{crit} < T - T_{wc} \tag{3.2}$$

Moreover, the presence of a gate on the clock path usually implies increased clock skew. In a completely automated synthesis environment it should be possible for the designer to specify accurate skew control for the gated clock line, thus preventing possible races or timing violation. The insertion of gated clock may also decrease testability. The reader is referred to [fava96] for a detailed treatment of the testability issues involved in clock gating. In [fava96] it is shown that testability can be preserved in gated-clock circuits with a small overhead caused by the insertion of observability (and, optionally, controllability) points.

It is possible to reduce the area, power and speed overhead of the clock-gating circuit shown in Figure 3.7 (c) by optimizing its transistor-level implementation. Chandrakasan [chan95] presented a clock-gating circuit, shown in Figure 3.9 (a), which is based on a small and power-efficient CMOS flip-flop family [yuan89] called *true single phase clock* (TSPC). The circuit without transistor MC is the standard TSPC flip-flop, that behaves as an ordinary raising edge-triggered flip-flop. The presence of transistor MC causes the output to be driven to zero whenever CLK is 1. Hence, when $CNTRL' = 1$, output $GCLK$ is a replica of the clock signal. When $CNTRL' = 0$ the output is stuck at 0. Notice that, similarly to the circuit in Figure 3.7 (c), the only requirement on input $CTRL'$ is to be stable when the raising edge of the clock signal arrives. With this clever implementation, only one modified TSPC flip-flop (12 transistors) is needed to generate the gated clock signal $GCLK$.

An alternative, more compact implementation can be obtained by directly optimizing the logic-level implementation of Figure 3.7. The AND gate is transformed in an equivalent NOR gate with inverted inputs. Then, the double inversion on the top input is canceled. Finally, latch L is implemented by a simple pass transistor gate. The transistor-level schematic of the optimized implementation is shown in Figure 3.9 (b). It only requires 8 transistors, thus it is more compact (and less power consuming) than the TPSC-based implementation. Unfortunately, it introduces larger clock skew. The TPSC-based implementation introduces a PMOS and an inverter delay on CLK (with respect to CLK), while the smaller implementation introduces an inverter and a NOR gate delay.

In summary, area and power of the clock-gating circuit can be much reduced by careful transistor-level optimization. Increased clock skew is probably the main limitation of power management scheme based on clock gating. Although clock skew can be effectively controlled by exploiting recently developed computer-aided algorithms and circuit techniques [cong97], in some conservative design styles (such as those used for FPGAs), the use of gated clocks is strongly discouraged, and less aggressive solutions should be chosen. Nevertheless, the relatively low overhead and the high effectiveness of clock gating are strong points in favour of this scheme.

3.3.3 Enabled flip-flops

When clock gating is not allowed in the logic design methodology of choice, less aggressive (and less effective) solutions can be chosen. A very safe and conservative approach is based on *enabled flip-flops*. Enabled flip-flops are sequential primitives with an additional input control signals EN. When $EN =$

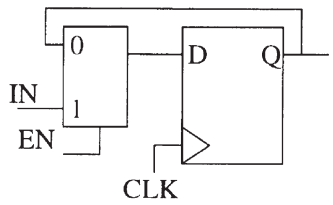


Figure 3.10. Enabled flip-flop

0 the data input value is not loaded into the flip-flop, and the state (and output) value is held constant.

The logic-level schematic of the enabled flip-flop is shown in Figure 3.10. Notice that the clock line is connected directly to the clock input of the flip-flop. To shut down a unit with enabled flip-flops at the input boundary, the powerdown signal $CNTRL$ must be inverted and connected to the EN inputs of all flip-flop. Differently from the gated clock scheme, no logic is inserted on the clock tree, hence the concerns on clock skew are eliminated. This shutdown scheme is suitable for design flows where either clock skew is not easily controllable or design validation tools (timing analysis and testing) cannot deal with gated clocks.

The main advantage of this scheme is that it can be completely analyzed without any special provision by standard logic-level design validation tools (such as static delay analyzers and ATPG programs). Unfortunately, the scheme has a sizable overhead with respect to clock gating, and it is not as efficient in saving power. The main overhead is caused by the replacement of the basic sequential primitives with enabled flip-flops. This must be done for all flip-flops at the inputs of the power-managed units, with a cost proportional to the number of inputs, because enabled flip-flops are larger (and more power-consuming) than standard flip-flops. Clock gating does not require special sequential primitives and the additional clock-gating logic (a latch and an AND gates) istantiated in the gated clock scheme is shared among all input flip-flops. Moreover, an additional, heavily loaded line is added to the circuit (the $CNTRL$ line is loaded with the input capacitance of all EN inputs), and the power dissipation of the clock line is not reduced at all when the unit is shut down.

As a result, power management solutions based on enabled flip-flops are convenient only in the case the unit is idle for a substantial fraction of the operation time and, above all, it has a relatively small number of input flip-flops compared to the number of internal combinational gates. A conservative implementation style have a significant overhead. Although power management based on en-

abled flip-flops can be beneficial, the implementation based on gated-clock is intrinsically superior when power savings are the primary concern.

3.4 CONCLUSION

In this chapter we have introduced the key concepts of power management based on external and internal idleness. External idleness is a condition where the outputs of a unit are not observable at the primary outputs of the system. Internal idleness is a condition where input transitions do not propagate to the outputs of a unit. In both cases, the unit can be shut down while idle, thereby reducing power dissipation. Although we have described internal and external idleness at the logic level, the same concepts are exploited at higher level of abstraction. Hence, a clear understanding of the nature of idleness and the mechanics of idleness exploitation are a prerequisite for the understanding of the more advanced material that will be introduced in the following chapters.

We have also described and analyzed several alternative shutdown mechanisms, namely, power supply shutdown, clock gating and enabled flip-flops. The choice among these alternatives is dictated by the tradeoff between technological challenges and overhead against achievable power reductions. For CMOS logic with small leakage power dissipation, clock gating is probably the most attractive solution, because it conjugates low overhead with high effectiveness in reducing the power dissipation of idle units.

We have addressed the issue of selecting an appropriate level of granularity for the application of logic-level power management techniques. We formulated a simple rule of thumb for estimating the maximum ratio between size of the additional power management circuitry and the size of the target unit. Moreover, we stressed the importance of selecting units with memory elements at the input boundaries. Finally, we emphasized the need of reliable estimates of input pattern distribution. Such information is paramount for discriminating between high-probability idle conditions that should be exploited for power managing the unit and low-probability idle conditions that can be neglected.

4 POWER MANAGEMENT FOR CONTROL UNITS

Non habent isti otium, sed iners negotium.

They are not idle, but they have idle occupations

—Seneca

85

4.1 INTRODUCTION

In this chapter we begin the analysis of computer-aided techniques for logic-level power management. The concepts outlined in the previous chapter are exploited in an algorithmic framework which supports the automatic generation of power-managed circuits with little effort on the designer's side. The input information required by the synthesis tool are a functional specification and a set of typical input patterns (or an input probability distribution). The techniques presented in this chapter specifically target control circuits modeled as finite-state machines.

Two techniques are presented. The first one exploits internal idleness to reduce the power wasted by circuit activity induced by input transitions that do not propagate to the outputs. The second technique is more aggressive and is applicable even to units that do not have any apparent internal idleness. In this case the initial specification is transformed into an equivalent one where idleness is exposed and can be exploited.

Both techniques follow a top-down paradigm. The distinction between top-down and bottom-up approaches is based on the format of the initial specification. For the techniques presented in this chapter, the circuit is specified abstractly as a finite-state machine, and the end product of the optimization tool is a mapped logic network. In the next chapter we will also consider a bottom-up approach where the starting point is a mapped logic network. The top-down paradigm has advantages and limitations. The key advantage is that the high level of abstraction of the specification makes it easy to detect idleness and to apply transformations for better exploiting it. Needless to say, it is hard to accurately estimate the effect of FSM-level transformations on power, speed and area. Because of the high level of abstraction, the optimizations have some uncertainty margin that should always be taken into account. Nevertheless, it is a general rule that applying transformations at a high level of abstraction is more likely to have bigger impact.

Probably the most serious limitation of the top-down optimization techniques based on FSM specification is that they are applicable only to relatively small control units. Although top-down techniques do achieve sizable power reductions on the system components they are applied to, they may not impact the overall power consumption as much, because large data-paths are not specified as FSMs. Bottom-up techniques, introduced in Chapter 5 partially overcome this limitation. Unfortunately, they require a more complex algorithmic machinery to extract and exploit idleness, and may not be as effective in transforming the specification in an equivalent one where idleness is exposed.

4.2 TOP-DOWN SYNTHESIS

The first technique we analyze is based on the exploitation of internal idleness. We move from the observation that during the operation of a FSM there are conditions such that the next state and the output do not change (i.e., the machine is internally idle). Therefore, clocking the FSM only wastes power in the combinational logic and in the registers. If we are able to detect when the machine is idle, we can stop the clock until a useful transition must be performed and the clocking must resume. The presence of a gated clock has a two-fold advantage. First, when the clock is stopped, no power is consumed in the FSM combinational logic, because its inputs remain constant. Second, no power is consumed in the sequential elements (flip-flops) and the gated clock line.

Detecting idle conditions requires some computation to be performed by additional circuitry. This computation dissipates power and requires time. Sometimes it will be too expensive to detect all idle conditions. It is therefore very important to select a subset of all idle conditions that are taken with high probability during the operation of the FSM. As it will be clarified later, idle conditions correspond to self-loops of Moore FSMs, and therefore it is relatively easy to detect them. Idle conditions in Mealy FSMs can also be detected, but with more effort.

An algorithmic framework for the automatic synthesis of power-managed circuits is based on two key procedures. First, idle conditions must be extracted from a given FSM specification. Second, clock-stopping logic must be synthesized. The automatic synthesis of clock-stopping logic must take into account the trade-off between the cost (in area, speed and power) of the additional instantiated circuitry and the power saved by shutting down the FSM during idle periods. More specifically the synthesis algorithm algorithm should choose a subset of all idle conditions such that the clock-stopping circuitry is as small and fast as possible, dissipates minimum power, but stops the clock with high efficiency. Procedures for idleness extraction and synthesis of clock-stopping logic will be described in the following sections.

Experimental results will be presented for a number of benchmark circuits specified by state tables. Since state table specifications are suitable only for small sequential systems, the largest example we present has a few hundreds of states. However, some relatively large designs are specified as interconnection of state machines. If the technique is applied to each component FSM, the overall power savings can be good.

For some circuits more than 50% reduction in average power dissipation has been obtained, but quality of results is strongly instance-dependent. In particular, internal idleness exploitation is well suited for FSMs that behave as

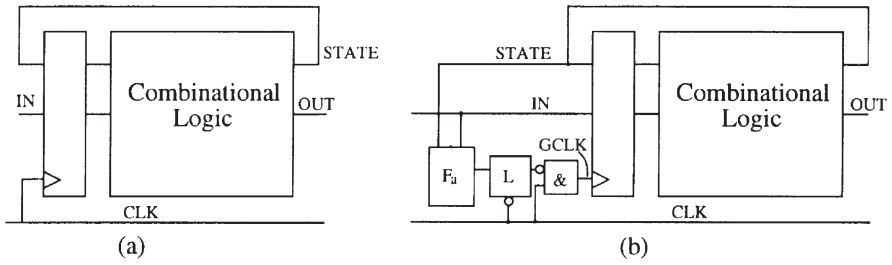


Figure 4.1. (a) Single clock, flip-flop based finite-state machine. (b) Gated clock version.

reactive systems: they wait for some input event to occur and they produce a response, but for a large fraction of the total time they are idle. As previously discussed, examples of such behavior can be found in portable devices such as pagers which are typical *reactive* systems, and for which power minimization in important.

4.2.1 Gated-clock FSMs

Consistently with Chapter 3, we will assume a single clock scheme with edge-triggered flip-flops, shown in Figure 4.1 (a). This is not a limiting assumption. The method can be applied to different clocking schemes as shown in [beni94a] (where transparent latches and multi-phase clocks are used).

The FSM model of Figure 4.1 (a) is different from the generic FSM structure we described in Chapter 2, where the primary inputs are not latched (we define as *latched* an input which is directly connected to a flip-flop). The FSM structure without latched inputs is seldom used in the design practice. In a large system, control logic and data-path are always decomposed in interacting sub-units. The interface between sub-units (interacting FSMs in our case) usually consists of sequential elements [west92] (in our case, D-flip-flops).

A gated clock FSM is obtained by modifying the structure in Figure 4.1 (a). We define a new signal called *activation function* (f_a) whose purpose is to selectively stop the local clock of the FSM, when the machine does not perform state or output transitions (i.e., when the FSM is internally idle). Activation function f_a corresponds to signal $CNTRL$ defined in Chapter 3: when $f_a = 1$ the clock will be stopped. The modified structure is shown in Figure 4.1 (b). Remember that block labeled "L" represents a transparent latch as explained in Chapter 3.

The modified circuit operates as follows. We assume that the activation function f_a becomes valid before the raising edge of the global clock. At this time the clock signal is low and the latch L is transparent. If the f_a signal

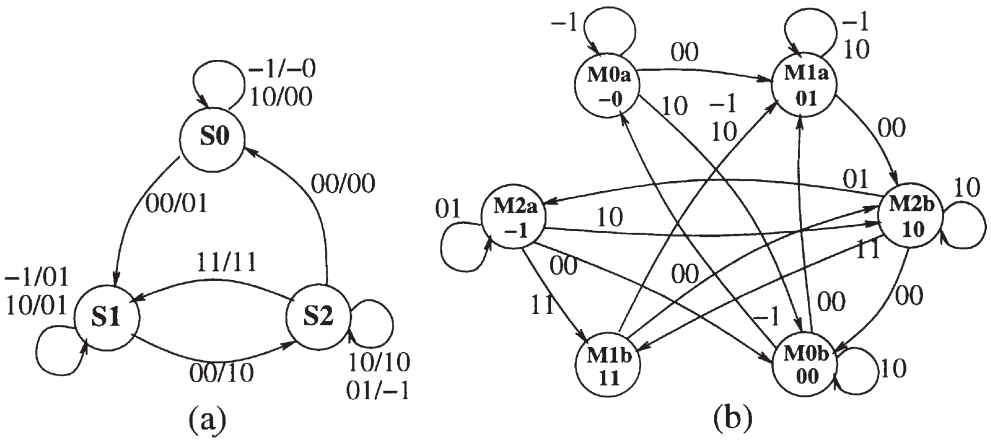


Figure 4.2. (a) STG of a Mealy machine. (b) STG of the equivalent Moore machine.

becomes high, the upcoming edge of the global clock will not filter through the AND gate and therefore the FSM will not be clocked and GCLK will remain low. Note that when the global clock is high, the latch is not transparent and the inverted input of the AND gate cannot change at least up to the next falling edge of the global clock.

The activation function is a combinational logic block with inputs the primary input IN and the state lines STATE of the FSM. No external information is used, the only input data for the algorithm is the behavioral description of the FSM and the probability distribution of the input signals.

4.2.2 Mealy and Moore machines

The definitions of Mealy and Moore FSMs have been given in Chapter 2. Conceptually, Mealy and Moore machines are equivalent, in the sense that it is always possible to specify a Moore machine whose input-output behavior is equal to a given Mealy machine behavior, and vice versa [hart66]. Practically, however, there is an important difference. The Mealy model is usually more compact than the Moore model. Indeed the transformation from Mealy to Moore involves a state splitting procedure that may significantly increase the number of states and state transitions [hart66].

Example 4.1 *In Figure 4.2 (a), a Mealy machine is represented in form of state transition graph (STG). It is transformed into the equivalent Moore machine (using the procedure outlined in [hart66]) and the new STG is shown in Figure 4.2 (b). The STG of the Moore machine has the output associated with*

the states, while in the Mealy model the outputs are associated with the edges. The higher complexity in terms of states and edges of the Moore representation is evident.

4.2.3 Problem formulation

Given the specification of the FSM and its probabilistic model (i.e., the Markov chain associated with it, with the conventions of Chapter 2), we first want to identify the idle conditions when the clock may be stopped. This is a simple task for Moore-type FSMs. For each state s_i we identify all input conditions such that $\delta(x, s_i) = s_i$. We define for each state s_i, $i = 1, 2, ..., |S|$, a *self-loop function* $Self_{s_i} : X \rightarrow \{0, 1\}$ such that $Self_{s_i} = 1 \ \forall x \in X$ *where* $\delta(x, s_i) = s_i$.

We then encode the machine. After the encoding step every state s_i has a unique code e_i and $e_i = [e_{i,1}, e_{i,2}, ..., e_{i,|V|}]$, where V is the set of the state variables used in the encoding .

The *activation function* is defined as $f_a : X \times V \rightarrow \{0, 1\}$:

$$f_a = \sum_{i=1,2,...,|S|} Self_{s_i} \cdot e_i \tag{4.1}$$

Example 4.2 *For the Moore machine in Example 4.1, the self-loop function for state M2a is* $Self_{M2a} = in_0' in_1$ *Similarly, all other self-loop state functions can be obtained. We encode the states using three state variables,* v_1, v_2, v_3. *The encodings are:* $M0a \rightarrow v_1' v_2' v_3'$, $M1a \rightarrow v_1' v_2 v_3'$, $M2b \rightarrow v_1 v_2 v_3'$, $M0b \rightarrow v_1 v_2 v_3$, $M1b \rightarrow v_1' v_2 v_3$, $M2a \rightarrow v_1' v_2' v_3$. *The activation function is therefore:* $f_a = in_2 v_1' v_2' v_3' + in_2 v_1' v_2 v_3' + in_1 in_2' v_1' v_2 v_3' + in_1 in_2' v_1 v_2 v_3' + in_1 in_2' v_1 v_2 v_3 + in_1' in_2 v_1' v_2 v_3$.

If the machine is Mealy-type, the problem is substantially more complex. The knowledge of the state and of the input is not sufficient to individuate the conditions when the clock can be stopped. If only the next state lines and the inputs are available for the computation of the activation function, we do not have a way to determine what was the output at the previous clock cycle. This is a direct consequence of the Mealy model: since the outputs are defined on the edges of the STG, we may have the same next state for many different outputs. The important consequence is that, even if we know that the state is not going to change, we cannot guarantee that the output will remain constant as well, and therefore we cannot safely stop the clock.

Example 4.3 *Consider the Mealy FSM in Example 4.1, and refer to Figure 4.1 for the implementation. If we use only the lines IN and STATE as inputs*

for the calculation of the activation function, we may for example observe state S2 on the next state lines, and input 10. Observing the STG, we know that for this state and input configuration the state will not change and the output will be 10 in the next clock cycle. Unfortunately, we do not have any way to know what is the output value in the current clock cycle (it could be either 10 or -1). The Moore model does not have this problem, since we know the output when we know what the state is.

There are two ways to solve this problem. The simpler way is to use the outputs of the FSM as additional inputs to the activation function. The other approach is to transform the STG in such a way that the FSM will be functionally equivalent to the original one, but it is possible to detect conditions where the clock can be stopped by observing only input and state lines.

In this section, we investigate the second method because the initial specification is in state table format. The state table is a behavioral, implementation-independent specification, therefore we still have the freedom to modify the number of states and the STG structure (this is not the case if we start from a synchronous network that is an implementation of the STG). In Chapter 5 we will assume a structural specification. In that case, the second method is not viable, since the circuit structure is given. We will then explore the first method.

The simplest transformation that enables us to use only input and state signals as inputs of the activation function f_a, is a Mealy to Moore transformation. The algorithm that performs this conversion is well known [hart66] and its implementation is simple, but it may sensibly increase the number of states and edges (correlated with the complexity of the FSM implementation). In the next subsection we propose a transformation that enjoys the advantage of the Mealy-to-Moore transformation (namely, it is sufficient to sample input and state signals to decide when the clock can be stopped), but does not suffer from the drawback (i.e., the excessive increase in number of states).

4.2.4 Locally-Moore machines

We now define and study a transformation that enables us to use a Moore-like activation function without a large penalty in increased complexity of the FSM. We define a *Moore state* as a state such that all incoming transitions have the same output. Formally, the subset of Moore states of a Mealy machine is: $\{s \in S \mid \forall x \in X, \forall r \in S, \delta(x,r) = s \Rightarrow \lambda(x,r) = const\}$. States that are not Moore states will be called Mealy states.

Proposition 4.1 *Consider a Mealy state s with k different values of the output fields on the edges that have s as a destination. The Mealy state can be transformed in k Moore states. No other state splitting is required.*

The FSM could be transformed by simply applying the Mealy to Moore transformations locally to states that have self-loops. The local Moore transformation has the advantage of allowing us to concentrate only on states with self-loops, avoiding the useless state splitting on the states without self-loops. The potential disadvantage is an increase in the number of states related to the number of different outputs on incoming edges of Mealy states (Proposition 4.1). In order to further reduce complexity, it is possible to split Mealy states with self-loops into simple pairs of states, where one state is Moore-type by construction with a self-loop that has maximum probability, and the other may be Mealy or Moore. The activation function f_a for the transformed machine is defined as the union of all input and state values on self-loops leaving Moore states. The details of the splitting procedure are quite involved and will not be described here. The interested reader is referred to [beni96, beni97c] for an exhaustive treatment.

This procedure is advantageous for many reasons. First, the increase in the number of states is tightly controlled. In the worst case, if all states are Mealy-type and have self-loops, we can have a twofold increase in the number of states. Second, the self-loops with maximum probability are selected. Third, if we really want to limit the increase in the number of states, it is possible to define a threshold for splitting. The states are ordered for decreasing probability of their self-loops and only the first k states in the list are duplicated. We call the FSM obtained after the application of this procedure *locally-Moore* FSM, because in general only a subset of the states is Moore-type.

Example 4.4 *The transformation of the Mealy machine of Example 4.1 produces the locally-Moore FSM shown in Figure 4.3. The shaded areas enclose states that have been split. The Moore states with self-loops are drawn with thick lines. The number of states and edges of the locally Moore machine is smaller than those that we obtained with the complete Mealy to Moore transformation (state S0 has not been split).*

The inputs are in_1 *and* in_2. *Assume that we use three state variables for the encoding:* v_1, v_2 *and* v_3. *The state codes are LM0a* $\rightarrow v_1'v_2'v_3'$, *LM1a* $\rightarrow v_1'v_2'v_3$, *LM2a* $\rightarrow v_1'v_2v_3'$, *LM1b* $\rightarrow v_1'v_2v_3$ *and LM2b* $\rightarrow v_1v_2'v_3'$. *The activation function includes all self-loops leaving Moore states:* $f_a = in_2v_1'v_2'v_3' + in_1in_2'v_1'v_2'v_3' + in_2v_1'v_2v_3 + in_1in_2'v_1'v_2v_3 + in_1in_2'v_1v_2'v_3'$.

Once the activation function has been found, we still need to solve the problem of synthesizing the clock-stopping logic in an optimal way. This problem will be addressed in the next subsection.

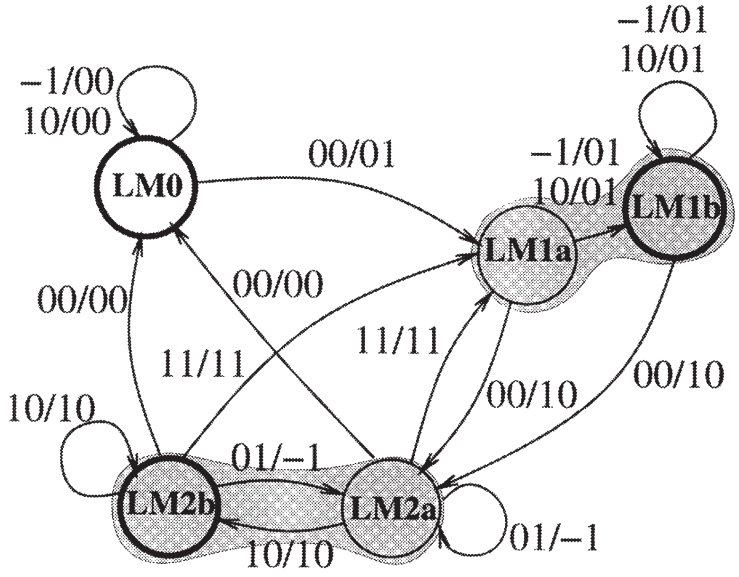


Figure 4.3. STG of the locally-Moore FSM

4.2.5 Optimal activation function

The most obvious approach to the synthesis of the activation function is to use the complete f_a. This is seldom the best solution, because the size of the implementation of f_a can be too large, and the corresponding power dissipation may reduce or nullify the power reduction that we obtain by stopping the clock. Roughly speaking, it is necessary to be able to choose a function contained in f_a whose implementation dissipates minimum power, but whose efficiency in stopping the clock is maximum. We call such function $F_a \subseteq f_a$ a *subfunction* of f_a. Here we exploit the isomorphism between set theory and Boolean algebra. Boolean function are seen as sets of minterms. A *subfunction* of f_a is therefore a function whose *on set* is contained in the *on set* of f_a.

A straightforward solution to this problem is the following. First, a minimum cover of f_a is obtained by a two-level minimizer. Then, the largest cubes in the cover are greedily selected until the number of literals in the partial cover exceeds a user-specified literal threshold. The rationale of this approach is that generally large cubes have high probability and the primes that compose a minimum cover are as large as possible. There are several weak points in this approach.

- There is no guarantee that choosing the largest cubes in a cover will maximize the probability of the cover, because in general the probability of a cube depends on the input and state probability distribution. Even if we assume uniform input probability distribution, the state probability distribution is in general not uniform.

- The function sought may not be found by looking only at a subset of cubes of the minimum cover of the original activation function. The number of possible subfunctions of f_a is much larger than the functions that we can generate using subsets of the cubes of the minimum cover.

- Even if we restrict our attention to the list of cubes in the minimum cover of f_a assuming uniform distribution for input and states, the cubes of the cover are in general overlapping (the minimum cover is not guaranteed to be disjoint). Finding the minimum-literal, maximum-probability subset of cubes becomes a set covering problem that certainly is not solved exactly by a greedy algorithm.

- The relation between the number of literals in a two-level cover of the activation function and the power dissipation of a multilevel implementation is not guaranteed to be monotonic.

We now formulate the problem that we want to solve in a more rigorous way.

Problem 4.1 *Given the activation function f_a, find $F_a \subseteq f_a$ such that its probability $P(F_a)$ is $P(F_a) \geq MinProb = \alpha P(f_a)$, (with $0 \leq \alpha \leq 1$) and the number of literals in a two-level implementation of F_a is minimum.*

We call this problem *constrained-probability minimum literal-count covering* (CPML). Notice that we could as well formulate the dual problem, constrained literal count cover with maximum probability. The two problems can be solved using the same strategy, and are equivalent for our purposes. With the assumption of a good correlation between number of literals and power dissipation, CPML can be solved in an exact fashion by a *branch-and-bound* algorithm [beni96]. The algorithm for the the solution of CPML is too involved to be described here. We will provide the key theoretical results that enables its solution and we will simply sketch the optimization procedure.

Apparently, the first source of difficulty comes from the fact that we are not constrained to completely cover f_a, therefore, the "algorithmic machinery" (i.e., the minimum covering algorithms based on Quine theorem, introduced in Chapter 2) developed in the area of two-level minimization seems not useful. We first show that this is not true. Consider the set of primes of f_a, called $Primes(f_a)$. Consider the set Sub_{f_a} of all possible subfunctions of f_a. The set

of primes of a generic subfunction $F_a \in Sub_{f_a}$ is called $Primes(F_a)$. We state the following theorem.

Theorem 4.1 *For every prime $p \in Primes(F_a)$ only two alternatives are possible.*

- $p \in Primes(f_a)$.
- *p is contained in at least one element q of $Primes(f_a)$ (consequently its literal count is larger than the literal count of q).*

For the proof of the theorem, refer to [beni96]. The important consequence of this theorem is that we do not need to generate all possible subfunctions of f_a in order to find a minimum literal count subfunction F_a that satisfies the probability constraint. We can restrict the search to subfunctions that are formed by subsets of $Primes(f_a)$. Functions that belong to this class have all primes in the first category of Theorem 4.1. The optimal implementation of the activation function can be found by carefully selecting a subset of the primes of f_a. Exact (and computationally expensive) as well as heuristic (and fast) procedures for prime selection are described in [beni96].

4.2.6 The overall procedure

We can now briefly outline the full procedure used for the synthesis of the low-power gated-clock FSMs. The starting point is a FSM specified with a transition table or a compatible format. The synthesis flow is the following.

- The Mealy machine is transformed to an equivalent locally-Moore machine.
- The complete activation function f_a is extracted from the Moore states of the locally-Moore machine.
- The probability of the complete f_a is computed.
- The prime set $Primes(f_a)$ is generated.
- The minimum literal count solution F_a whose probability is a pre-specified fraction α of the probability of f_a is found.
- F_a is used as additional *don't care set* for optimizing the combinational logic of the FSM.

The last step can sensibly improve the quality of the results, in particular if F_a is large. Unfortunately, it is hard to foresee the effects of F_a used as *don't care* set. Sometimes it may be convenient to choose a F_a that is not minimal in

the sense discussed above, if it allows a large simplification in the combinational part of the FSM. A heuristic approach is to try different F_a that range from the complete f_a to a much smaller subfunction, in an attempt to explore the trade-off curve.

This iterative search strategy raises the problems of choosing appropriate values of the parameter $\alpha \leq 1$ (as defined in Problem 4.1) used to scale down the probability of f_a when the reduced activation functions are generated. An effective approach is to generate a set of reduced activation functions $Cand_F$ using different values of α, in such a way that the possible range of solutions is uniformly sampled. We briefly outline a heuristic procedure that generates suitable α values.

Initially, the user specifies the number of candidate solutions N_{cand} that should be generated by the tool. Then, the values of α are generated by uniformly dividing the interval $[0, 1]$:

$$\alpha_i = i/N_{cand} \quad i = 1, 2, ..., N_{cand} \tag{4.2}$$

It may be the case that for two or more consecutive values α_i, the algorithm generates the same solution. This happens for example when eliminating even a single prime from a solution generated for $i + 1$ causes a decrease in the $P(F_a)$ in the solution generated for i larger than $P(f_a)/N_{cand}$. In this case the algorithm adaptively select new values of α such that the new candidates will have a probability between those of solutions generated with two consecutive values of α that have maximum literal cost difference.

Obviously, if large N_{cand} are used, the computational time required to generate $Cand_F$ increases. Notice however that only the last two steps in the overall procedure described before need to be iterated, and usually a small number of different values of α is sufficient to find a satisfying solution.

4.2.7 Experimental results

We implemented the power optimization algorithm described above and we performed several experiments to assess its effectiveness and applicability. The state transition table of the FSM are first read. The first step of the optimization procedure is the transformation of the Mealy machine to a locally-Moore machine and the extraction of the self-loops from the Moore states.

An input probability distribution must be specified by the user. For our experiments we set the input probabilities to 0.5 for all inputs, since we did not assume any information on the environment where the FSMs are embedded. Moreover, we assumed that every input line has a maximum of one transition per clock cycle. This is an optimistic assumption, because multiple input transitions (high transition density) may increase the power dissipated in the activa-

tion function logic (but not in the FSM logic because the input are guarded by flip-flops). If inputs with high transition activity are present, smaller activation function should be allowed.

The *power method* (described in Chapter 2) is applied to compute the exact state probabilities given a conditional input probability distribution. Notice that this step can be modified to use the exact and approximate methods described in [hama94, mont94, marc94] that have been demonstrated to run on very large sequential circuits. Simple sparse matrix techniques are sufficient to efficiently process all MCNC [mcnc91] benchmarks provided in state transition table format ($\approx$ $10sec.$ on a DECstation 5000/240 for the largest example `s298`).

After state probability and transition probability computation, state codes are assigned to both the original machine and the locally-Moore FSM using JEDI [lin89]. Once the state codes have been assigned, the probabilistic-driven procedure for the selection of the activation function can start. CPML is solved and an optimal reduced activation function F_a is computed.

The user specifies the number of activation functions that the procedure should generate, and CPML is solved as many times as it is requested for different probability constraints. Surprisingly, for all MCNC benchmarks this step has never been the bottleneck, the CPU time being in the order of 30 seconds maximum. This is certainly due to the fact that the majority of the FSM MCNC benchmarks do not have a large number of self-loops (in particular the larger ones). Nevertheless, even if difficult cases are found, fast heuristic solution can be computed.

The combinational logic of the locally-Moore FSM is then optimized in SIS [sent92] using the additional *don't care set* given by the activation function. This step is repeated for all activation functions generated in the preceding step, and alternative solutions are generated. The *don't care*-based minimization of the combinational logic using the activation functions is the main bottleneck of the procedure. The user has the possibility to specify a CPU-time limit for each minimization attempt. This of course limits the possible improvements obtainable on large FSMs.

The activation functions are also optimized using SIS, then the alternative solutions are mapped with CERES [mail91], and the gated clock circuitry is generated. Again the same optimization and library binding programs are used for both the original Mealy machine and the locally-Moore gated clock machines. We employed a simple target library which includes two, three and four input gates. The flip-flops have a master-slave structure, and their cost (in terms of area and input load capacitance) is approximatively equivalent to 2 three-input logic gates.

Finally the alternative gated clock implementations and the implementation of the original Mealy FSM are simulated with a large number of test patterns using a switch level simulator (IRSIM [salz89]) modified for power estimation.

The quality of the results strongly depends on two factors. First, how much state splitting has been needed to transform the machine to a locally-Moore one. Second, for what percentage of the total operation time the FSM is in a self-loop condition (this depends on the FSM structure and on the input probability distribution). For machines with a very small number of self-loops or a very low-probability complete activation function, the chance of improvement is limited or null. This is the case for many MCNC benchmarks for which the final improvement is negligible. As for the first problem, it may be worth to investigate if, in case the state duplication is too high, using an activation function with the outputs of the FSM as additional inputs may lead to better results.

Example 4.5 *The Mealy machine of Example 4.1 has been synthesized without any gated clock. The number of states is 3, the mapped implementation has 124 transistor and a total nodal capacitance of 2.32 pF. The average power dissipation is 52 μW.*

The minimum power implementation (obtained with the complete activation function in this case) of the equivalent locally-Moore gated clock machine has 178 transistors and a total nodal capacitance of 3.14 pF. The average power dissipation is 42 μW. Notice that the efficacy of the activation function in stopping the clock allows substantial power savings (24%) even if the total capacitance is larger (35%). This is due to the fact that the locally-Moore machine has 5 states, and its combinational logic is more complex. In contrast, with a complete Moore transformation the minimum power implementation has 196 transistors and total nodal capacitance of 3.39 pF. Its power dissipation is 48 μW.

Table 4.1 reports the performance of power management on a subset of the MCNC benchmarks. The first six columns show the area (number of transistors) and the power dissipation of the normal Mealy FSM, the locally-Moore FSM without gated clock and the locally-Moore machine with gated clock. The last three columns show the percentage power reduction (computed as $100(P_{mealy} - P_{gated})/P_{mealy})$), the size (in transistors) of the activation function and the α factor used in the solution of CPML leading to the best result. Notice that, if there is no power improvement the improvement is set to 0.

The tool is able to process all benchmarks, but in the table we list examples representative of various classes of possible results. The benchmarks `bbara` and `test` are reactive FSMs. The high number and probability of the self-loops allow an impressive reduction of the total power dissipation, even if the

Circuit	Original Size	Original P	Locally-M. Size	Locally-M. P	Gated Size	Gated P	%	F_a Size	α
bbara	330	67	422	72	408	34	49	74	1
bbsse	640	121	742	137	736	119	2	140	1
bbtas	142	56	138	57	164	44	21	34	.93
keyb	721	128	754	132	820	114	10	62	.91
lion9	188	60	226	60	248	52	13	8	.25
s298	7492	899	7496	900	7502	810	10	14	1
s420	544	132	544	132	602	108	18	44	.75
scf	3222	437	3222	437	3169	400	8	26	1
styr	1474	159	2468	230	2534	208	0	560	.75
test	348	73	442	76	374	32	56	64	.88

Table 4.1. Results on MCNC benchmarks. Size is number of transistors. P (power) is in μW.

area penalty is not negligible. For this class of FSMs, power management gives its best results.

In contrast, for `bbsse` and `styr` there is no power reduction or even a power increase. The `bbsse` benchmark is representative of a class of machines where the number and probability of the self-loops is too small to obtain substantial power savings. The `styr` benchmark has many self-loops, but they all have low probability. Moreover, the transformation to locally-Moore machine has a too large area overhead in this case. Therefore, even if there are power savings with respect to the locally-Moore implementation without clock, the smaller Mealy implementation has the lowest power consumption.

For all other examples in the table the power savings vary between 10% and 30%. For some of these machines (`s420` and `scf`), there is no area overhead for the locally-Moore transformation. This happens when all states with self-loops are already Moore states in the original FSM. We included some of the larger examples in the benchmark suite (`s298` and `scf`) to show the applicability of the method to large FSMs.

From the analysis of the results, it is quite clear that several complex trade-offs are involved. First, the transformation to locally-Moore machine can sometimes be very expensive in terms of area overhead. Second, the choice of the best possible activation function is paramount for good results. In fact, for many examples, the complete activation function was too large, and reduced

Circuit	$P^{Comb}_{Gated}/P^{Clk}_{Gated}$	$P^{Comb}_{Loc\_M}/P^{Clk}_{Loc\_M}$	$P^{Comb}_{Gated}/P^{Comb}_{Loc\_M}$	$P^{Clk}_{Gated}/P^{Clk}_{Loc\_M}$
bbara	1.2777	2.7006	0.3624	0.7660
bbsse	1.5317	2.5386	0.7601	1.2597
bbtas	1.2242	1.8715	0.6861	1.0488
keyb	2.4024	3.2242	0.8185	1.0985
lion9	2.0734	2.2749	0.8937	0.9806
s298	17.7560	17.9872	0.9887	1.0016
s420	1.3517	1.5251	0.8126	0.9169
scf	2.8988	2.7368	0.9274	0.8755
styr	4.2330	4.3990	0.8969	0.9320
test	1.1751	2.8957	0.3048	0.7512

Table 4.2. Partition and comparison between power dissipation in clocking logic and FSM logic for locally-Moore and gated-clock FSMs

activation functions gave better results. Notice however that for some examples the efficiency of the activation function in stopping the clock was such that the power was sensibly reduced even with large area overhead.

Having discussed *how much* power is saved, we address now the problem of *where* the power is saved. The FSM clock is stopped only when the next state variables and the outputs are not going to change in the upcoming clock cycle. It may be possible to think that power is saved only in the flip-flops and the clock line. This intuitive observation is deceiving, because power is also saved in the combinational logic, as it is shown in Table 4.2. We have compared in the table the power dissipation of the locally-Moore implementation with and without gated clock. We compare to the locally-Moore FSM because its STG is isomorphic to those of the gated-clock FSM. Hence, all modifications are due only to the insertion of the activation function. The comparison with the Mealy machine is less explicative because the locally-Moore transformation modifies the STG and consequently the next state and output function, making impossible to distinguish how the clock-gating circuitry alone affects the power dissipation.

In the first two columns of the table, for the two implementations, we compare the ratio of the power dissipated in the combinational logic (flip-flops outputs, all nodes in the FSM logic and outputs) and the power dissipated in the clock-related circuitry (activation function, clock lines, NAND gate, latch, inputs and internal nodes of the flip-flops). In the last two columns we show the

power ratio for the combinational logic and the power ratio for the clock-related circuitry for the two implementations.

First, notice that the ratio P^{Comb}/P^{Clk} is almost always smaller for the gated-clock FSMs. This result is quite intuitive, because P^{Clk} in the gated-clock FSM includes the power dissipation of the activation function. The results of columns 3 and 4 are somewhat counterintuitive, because they show that *there is consistently higher power saving in the combinational logic than in the clock-related circuitry.* This result is due to three factors. First, the reduced switching activity on the outputs of the flip-flops, that are generally highly loaded. Second, the absence of propagation to internal nodes in the FSM logic of input transitions when the FSM is in a self-loops. Third, and most importantly, the simplification in the FSM's logic that is obtained using the *on set* of the activation function as additional controllability *don't care set.*

We want to point out that the power management methodology attains consistent power savings not only when the clock line is heavily loaded and large flip-flops are used, but also for very small FSMs with optimized flip-flops. It is however important to remark that the full extent of the possible savings is obtained only if the combinational logic is re-optimized with the increased *don't care* set previously described. Moreover, for FSMs without self-loops, the methodology is ineffective in reducing power consumption.

In summary, the power savings depends on the fraction of the total operation time that the FSM spends in idle condition. *Don't care* optimization is very helpful when the FSM is small and the idle time is a relatively small fraction of the total, because it helps in reducing the overhead of the clock-gating circuitry. For large FSMs the impact of *don't cares* is generally less relevant. Even if the power savings are biased by the initial structure of the FSM, it is important to have an automated synthesis procedure to avoid unnecessary effort from designers in trying to manually design clock-stopping logic.

4.2.8 Summary

In this section we have described a technique for the automatic synthesis of gated clocks for Mealy and Moore FSMs. From the practical point of view, the method is part of a complete procedure, starting from FSMs state-table specification to fully mapped network, and it has been tested with accurate power estimation tools. The quality of results depends on the initial structure of the FSM, but we obtain important power reductions for a large class of finite-state machines, where the probability of being in a self-loop (idle) is high. Even if an automatic synthesis tool cannot fully replace the knowledge of the designer in finding idle conditions at the architectural level, it may enable

design exploration for cases where it is not clear if clock gating may produce sizable power savings.

4.3 TOP-DOWN DECOMPOSITION

In the previous section we demonstrated that clock gating is an effective technique for minimizing power by exploiting internal idleness. Sizable power reduction are obtained for reactive FSMs that are idle for a large fraction of the operation time. We now investigate a more aggressive approach: we target the minimization of "useful" power dissipation. In other words, we focus on reducing power consumption in sub-systems that may never be internally idle [beni97c]. For such systems, the power management techniques presented so far are not effective.

The fundamental intuition behind the optimization technique presented in this section is that a sequential circuit may be decomposed into a set of small interacting blocks. During operation, only one block is active at any given time and controls the input-output behavior. In the remaining blocks, the clock can be stopped and, consequently, the total power consumption is reduced. A decomposition approach allows us to save power even for systems that are never idle. In other words, we expose idleness by properly decomposing a system.

Unfortunately, in many cases the behavior of the system is specified without any consideration for *power-efficient decompositions*. Hence, we need to develop procedures for the detection of such decompositions and the synthesis of a hardware architecture where unused parts of the system can be shut down without compromising the correctness of the global input-output behavior.

We propose a procedure for the automatic synthesis of a network of interacting FSMs starting from a single state-table specification (or an equivalent format). We call *decomposed FSM* the interacting FSM implementation. The straightforward single-machine implementation is called *monolithic FSM*. The monolithic finite-state machine can be decomposed into smaller sub-machines that communicate through a set of interface signals. *Usually, one single sub-machine is clocked* at any given time and it controls the outputs values while all other sub-machines are idle: they do not receive the clock signal and dissipate little power. When a sub-machine terminates its execution, it sends an *activation signal* to another sub-machine which takes control of the computation, then it de-activates itself. This transition is characterized by a single cycle for which both sub-machines are clocked.

There is full cycle-by-cycle equivalence between the input-output behavior of the decomposed and monolithic implementation. If the FSM is embedded in a larger system, the decomposed implementation can replace the monolithic one and the behavior of the system remains unchanged. Given the state transition

graph (STG), or equivalently, the state table of the FSM, the procedure decomposes the STG into a set of smaller STGs. Each component is then synthesized using standard FSM synthesis algorithms. The resulting sequential circuits are then connected and dedicated clock-control circuitry is synthesized, which allows selective clocking of the sub-FSMs.

The total area of the interacting FSM implementation is larger than that of the monolithic counterpart. Notice however that the increase in area is accompanied by more hierarchical structure: the interacting FSMs implementation consists of small and loosely coupled blocks instead of one large highly coupled block. This characteristic can be very useful during synthesis and the placement and routing phase: small blocks are more effectively synthesized, placed and connected. Consequently the improved power and performance of the decomposed implementation will not be damaged by excessive wiring even if it has area overhead. Accurate power and timing estimation shows that the decomposed implementation achieves on average 31% power reduction and 12% speed improvement at the price of a 48% increase in the number of standard cells. Since the method trades off power reduction and speed increase for area usage, it is well-suited for high-performance VLSI systems where speed and power are the primary concerns.

4.3.1 Previous work on FSM decomposition

Finite-state machine decomposition has been extensively studied for several decades. Its theoretical foundations were laid down by Hartmanis and Stearns [hart66] in the sixties. Hartmanis and Stearns defined several flavors of decomposition and formulated numerous decomposition procedures. More recent work [geig91] reports experimental results on the implementation of the decomposition procedures described in [hart66].

A different viewpoint on the problem was proposed in recent years by Ashar, Devadas and Newton [asha91] who presented numerous algorithms for the automatic decomposition of FSMs specified by an monolithic state transition graph (STG). In this work, the authors introduced the concept of *factorization* (an efficient algorithm to detect and exploit decomposition opportunities) as well as exact and heuristic solutions to the problem of two-way FSM decomposition for minimum-area two-level implementation. The work by Hartmanis and Stearns is based on an algebraic model of FSMs rooted in lattice theory. By contrast, Ashar et al. rely on Boolean algebra as the underlying theoretical framework.

More recent work is based on graph theory [royn93, kuo95]. The STG of the FSM is decomposed using graph partitioning algorithms. The approach presented in this section has several points in common with [royn93]. Most notably, they both focus on a partitioning strategy, where the initial FSM is

decomposed in sub-modules that are never performing useful computation at the same time. However, we target the minimization of power consumption and we rely on a different assumption on the hardware implementation (i.e., gated clock versus single clock).

The algorithms mentioned so far target area minimization, or the optimization of area-related cost functions (i.e., number of input and outputs, connectivity). Although sizable area reductions can be achieved, the average effectiveness of decomposition techniques for area is not impressive, mainly for two reasons. First, the majority of real-life FSMs do not have a simple decomposition, and some replication is almost always required. Second, the computational requirements of the exact decomposition algorithms are too high on large examples (where the potential savings are substantial), and heuristic solutions fail to find good decompositions.

We believe that the potential impact of decomposition techniques for low power and high performance is much higher. This conjecture is confirmed in [hasa95], where decomposition techniques have reported very promising delay reductions on multi-level logic implementation. Moreover, Chow et al. [chow96] recently proposed a low-power decomposition approach based on the relationship between state assignment and FSM decomposition that produced very promising results. Compared to the method proposed in the following sections, the method by Chow et al. has a smaller area overhead, a similar performance improvement but lower power savings. Our better power saving are attributed to the fact that Chow et al. do not use gated clocks in the decomposed implementation, while their smaller area overhead is due to the fact that they integrate decomposition and state assignment to reduce the amount of replication of logic caused by decomposition.

4.3.2 Interacting FSM structure

Let $F = (X, Y, S, s_0, \delta, \lambda)$ be a monolithic Mealy-type FSM specification, where X is the input set, Y the output set, S the set of states, $s_0 \in S$ the reset state. The next state function is $\delta : X \times S \rightarrow S$ and the output function is $\lambda : X \times S \rightarrow Y$.

Assume that a *partition* Π on S is given [hart66]. A partition is defined as a collection of n disjoint subsets of S whose set union is S, i.e $\Pi(S) = \{P_1, P_2, ..., P_n\}$, such that $P_i \cap P_j = \emptyset$ for $i \neq j$, and $\bigcup_{i=1}^{n} P_i = S$. We call *partition blocks* the subsets P_i. A monolithic FSM and a two-way partition are shown in Figure 4.4.

Given the monolithic F and $\Pi(S)$ we proceed to the formal definition of the decomposed FSM. The decomposed FSM Φ is a set of n FSMs $\Phi =$

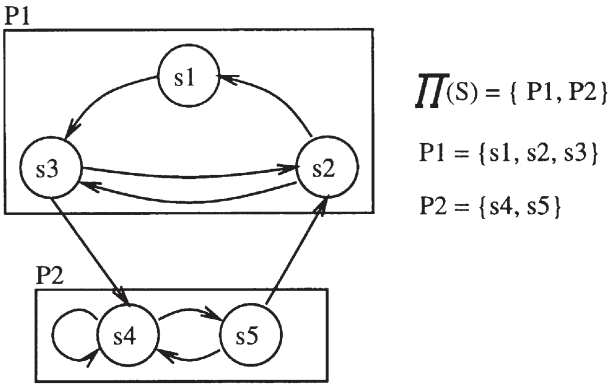


Figure 4.4. A monolithic FSM and a partition of its state set.

$\{F_1, F_2, ..., F_n\}$. We call *sub-machines* the FSMs $F_i \in \Phi$. A generic sub-machine F_i is defined as $F_i = (X_i, Y_i, S_i, s_{0,i}, \delta_i, \gamma_i)$, where:

- $s_{0,i}$ is the reset state of F_i.
- S_i, the state set of F_i is $S_i = P_i \cup \{s_{0,i}\}$.
- The input set X_i is $X_i = X \cup GO_{-,i}$. We define $GO_{-,i}$ as follows. For each transition from a state $t \in P_j$ to a state $s \in P_i$ in the monolithic FSM F, a new signal $go_{t,s}$ is created which is an input for F_i and an output for F_j. The set $GO_{-,i}$ is the set of all *go* signals which are inputs for F_i. In symbols: $GO_{-,i} = \{go_{t,s} \mid \delta(\cdot, t) = s,\ s \in P_i,\ t \in P_j,\ P_i \neq P_j\}$.
- The output set Y_i is $Y_i = Y \cup GO_{i,-}$. The set $GO_{i,-}$ is the set of all *go* signals which are outputs for F_i: $GO_{i,-} = \{go_{t,s} \mid \delta(\cdot, t) = s,\ s \in P_j,\ t \in P_i,\ P_i \neq P_j\}$.
- The next state function $\delta_i(x, go_{q,t}, s)$ is defined as follows:

$$\delta_i(x, go_{q,t}, s) = \begin{cases} \delta(x,s) & \text{if } s \in P_i \text{ and } \delta(x,s) \in P_i \\ s_{0,i} & \text{if } s \in P_i \text{ and } \delta(x,s) \in P_j,\ P_i \neq P_j \\ t & \text{if } s = s_{0,i} \text{ and } go_{q,t} = 1,\ q \in P_j,\ t \in P_i,\ P_i \neq P_j \\ s_{0,i} & \text{if } s = s_{0,i} \text{ and } \forall go_{q,t} \in GO_{-,i}\ go_{q,t} = 0 \end{cases} \tag{4.3}$$

The definition means that every transition between states s and t in the monolithic FSM does not change its source and destination if the two states belong to the same block P_i (this case is shown in Figure 4.5 (a) and (b)). A transition from s to t belonging to different partition blocks, respectively P_i

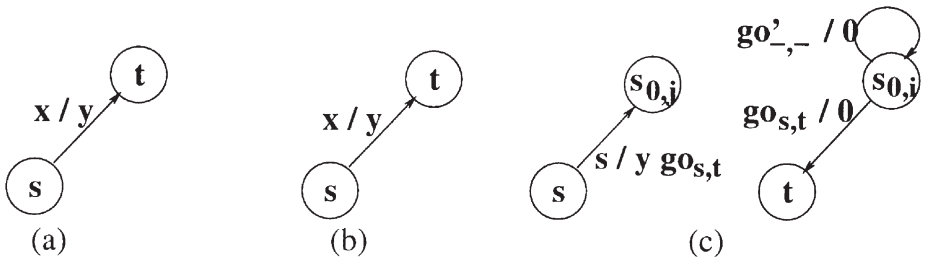


Figure 4.5. Pictorial representation of the definitions of δ_i and λ_i

and P_j, becomes: i) a transition from s to the reset state of machine F_i; ii) a transition from the reset state to t for machine F_j. When a sub-machine F_i is in reset state, it exits from it only when one of the *go* signals is set to one.

- The output function $\lambda_i(x, go_{q,t}, s)$ takes values on the new output set Y_i that includes the original outputs of the monolithic FSM and the *go* outputs. We represent the output value with the notation (x, go).

$$\lambda_i(x, go_{q,t}, s) = \begin{cases} (\lambda(x,s), \mathbf{0}) & \text{if } s \in P_i \text{ and } \delta(x,s) \in P_i \\ (\lambda(x,s), go_{s,t} = 1) & \text{if } s \in P_i \text{ and } \delta(x,s) = t \in P_j,\ P_i \neq P_j \\ (\mathbf{0}, \mathbf{0}) & \text{otherwise} \end{cases} \tag{4.4}$$

We used the shorthand notation $(x, \mathbf{0})$ to indicate that all *go* outputs or are held at value 0. Similarly, the notation $(\mathbf{0}, \mathbf{0})$ is used to indicate that all original outputs in set X and all *go* signals are held at value 0. The notation $go_{s,t} = 1$ is used to indicate that only one *go* output has non-zero value, namely $go_{s,t}$. The definition of λ_i means that whenever there is a transition between two states belonging to the same P_i, the output of sub-machine F_i is the same as in the monolithic FSM. A transition toward the reset state has the output value corresponding to the transition in the monolithic FSM between state s in S_i and state t in S_j, and asserts the $go_{s,t}$ output. All outputs are zero for the self-loop on $s_{0,i}$ and all transitions leaving $s_{0,i}$.

The definition of the sub-machines F_i completely defines the decomposition strategy. To better understand the definitions of δ_i and λ_i refer to Figure 4.5. Part (a) shows a transition in the monolithic FSM. Part (b) shows the transition in the decomposed FSM when its source and destination state both belong to the same partition block P_i (the transition is unchanged). Part (c) shows the case when the source and destination state belong to different partition blocks. For each transition leaving a state sub-set P_i in the monolithic FSM, the sub-

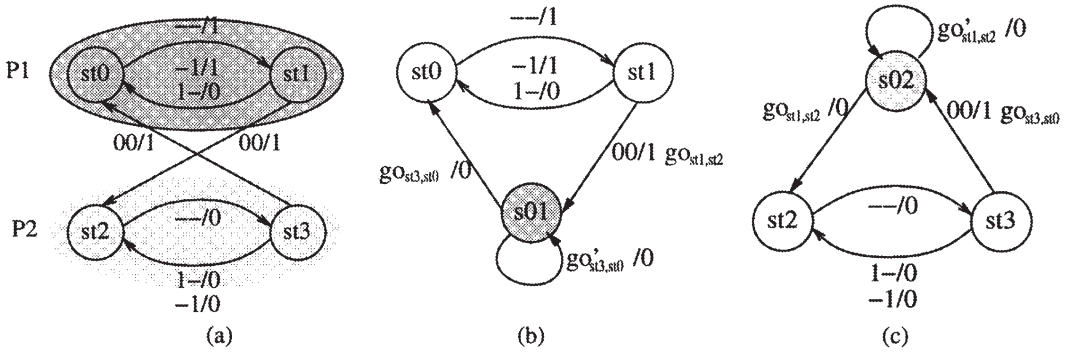


Figure 4.6. Decomposition of the monolithic FSM

FSM F_i associated with P_i performs a transition to its reset state. On the other hand, a transition entering a sub-set P_i from $P_i \neq P_j$ corresponds to a transition exiting the reset state in the sub-FSM F_i. A sub-machine can exit the reset state only upon assertion of a *go* signal by another submachine. At any given clock cycle only two situations are possible: i) one sub-machine is performing state transitions and all other sub-machines are in reset state or ii) one sub-machine is transitioning toward its reset state, while another one is leaving it.

All inputs and outputs of the monolithic FSM are inputs and outputs of the sub-machines. The *go* signals are new, additional inputs and outputs. If an edge $s \rightarrow t$ of the original machine has head and tail state included in sub-machine F_i, the edge is replicated in F_i, with the same input and output fields. Edges in the global FSM connecting states which belong to different partitions are associated with edges representing transitions to and from the reset states of the corresponding sub-FSMs. These transitions are labeled as follows: i) edges toward reset have the same input field as the original edge, assert an additional output $go = 1$ and have the same output field as in the original transition edge of the monolithic FSM; and ii) transitions leaving reset have only one specified input *go* and all outputs set to zero. The outputs of a sub-machine blocked in reset state are zero.

Example 4.6 *Consider the FSM in Figure 4.6 (a). We assume that the state partition is* $\Pi(S) = \{P_1, P_2\} = \{\{st0, st1\}, \{st2, st3\}\}$. *The two sub-machines created by the decomposition procedure are shown in Figure 4.6 (b) and (c). The additional reset states are shaded.* P_1 *originates sub machine (b) and* P_2 *originates sub machine (c). Notice that the "go" signals are shown only on the transitions from and to the reset states. A sub-machine asserts a "go" signal*

only when transitioning to the reset state, in all other cases the signal has value zero. Similarly, a submachine is sensitive to input "go" signals only when it is in reset state. The "go" inputs are not observed for all other transitions.

After describing the decomposition strategy, we now focus on how to reduce the total power dissipation of the interconnection of sub-FSMs.

4.3.3 Clock gating

In the interacting FSM system, most of the machines F_i remain in state $s_{0,i}$ during a significant number of cycles. If we stop their clock while they stay in reset state, we would save power (in the clock line, the flip-flops and in the FSM combinational logic) because only a part of the system is active and has significant switching activity. To be able to stop the clock, we need to observe the following conditions.

- The condition under which F_i is idle. It is true when F_i reaches the state $s_{0,i}$. We use the Boolean function $is\_in\_reset_i$ that is 1 if F_i is in state $s_{0,i}$, 0 otherwise.
- The condition under which we need to resume clocking, even if the sub-FSM is in reset state. This happens when the sub-FSM machine receives a *go* signal and must perform a transition from s_{0i} to any other state.

We can derive F_{ai}, the *activation function* (in negative logic). The clock to F_i is halted when $F_{ai} = 1$. Namely:

$$F_{ai} = is\_in\_reset_i \wedge (\bigvee_{q \in F_i, p \in F_j \neq F_i} go_{p,q})' \tag{4.5}$$

The first term $is\_in\_reset_i(s)$ stops the clock when the machine reaches $s_{0,i}$. The second term ensures that clock is not halted when one of the $go_{p,q}$ is asserted and the sub-FSM must exit the reset state. This activation function allows the newly activated sub-FSM to have its first active cycle during the last cycle of the previously active FSM. The two sub-FSMs make a transition in the same clock cycle: one is transitioning to its idle state, and the other from its idle state. The local clocks of F_i and F_j are both active. We call *transitions of control* the cycles when a sub-FSMs shuts down and another activates.

Each local clock of the FSMs F_i is controlled by a clock-gating circuit. The circuit implements the activation function F_{ai}. We could use F_{ai} directly as *enable* signal on the flip-flops, but this scheme would let the clock lines active and consume power, therefore we choose an aggressive implementation: the clock itself is stopped when a submachine is inactive. The power savings on

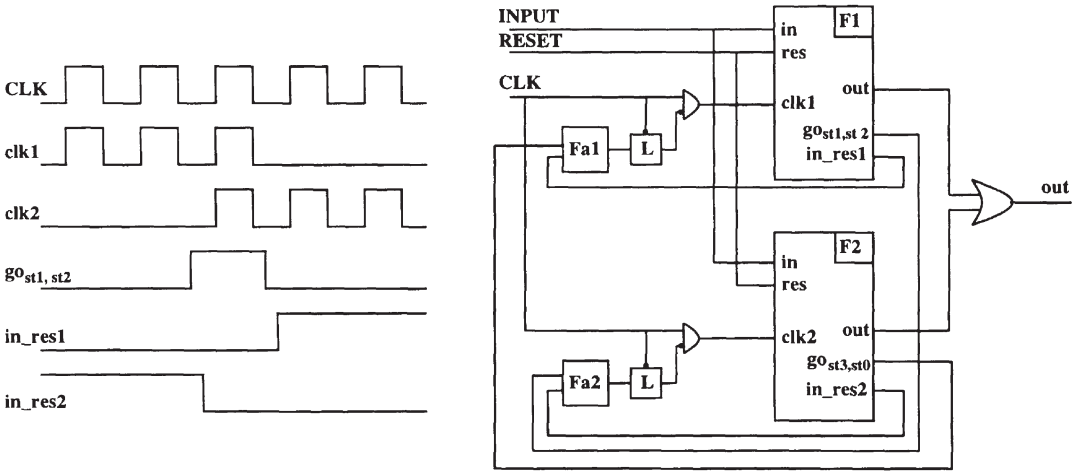


Figure 4.7. Gated-clock implementation of the interacting FSMs

the clock are sizable because the clock line is heavily loaded and switches with high frequency. We use a low-level sensitive latch for F_{ai}, in order to avoid spurious transitions on the clock line that would result in incorrect behavior. Refer to Chapter 3 for a detailed description of the clock-gating circuitry.

The clocking strategy discussed so far has two important functions: reducing power dissipation and keeping the sub-FSM in lock-step. We now address the issue of ensuring cycle-by-cycle equivalence between the monolithic FSM and the decomposed implementation. The outputs of the gated-clock sub-machines are connected to n-way OR gates, one for each primary output. The output equivalence between the decomposed machine and the specification is guaranteed by the fact that, at any given clock cycle, only the active sub-machine is controlling the output value, and all inactive sub-FSMs have their output forced to zero. During the transitions of control, when a machine stops and another resumes execution, the value of the output is controlled by the machine terminating execution.

Example 4.7 *The gated-clock implementation of the interacting FSMs of Figure 4.6 is shown in Figure 4.7. The external output is obtained by OR-ing the outputs of the sub-FSMs. Figure 4.7 also shows the clock waveforms, the "in_reset" signals and the "go" signals. Notice that there is a clock cycle for which both local clocks are enabled. The waveforms show how sub-FSM 1 is deactivated and sub-FSM 2 activates thanks to the assertion of the* $go_{st1,st2}$ *signal.*

The startup of the decomposed FSM is obtained by disabling F_a for all sub-FSMs when a synchronous RESET is asserted (this can be done by inserting a AND gate controlled by RESET on the output of F_a). When the RESET signal is high, all sub-machines receive the clock signal. Notice that only the sub-FSM containing the original reset state of the monolithic FSM is actually set to the corresponding state code. All other machines are put in reset state $s_{0,i}$ and they stop and wait for an external *go* signal.

4.3.4 Partitioning

The power savings in the interacting FSM implementation strongly depends on the quality of the partition $\Pi(S)$. A good partition is characterized by loose interaction between sub-FSMs and small communication overhead. We analyze these requirements in detail.

The ideal mode of operation for the interactive FSM circuit is one of minimum transition of control between different sub-FSMs. When a sub-FSMs disables itself and another one takes control, both machines are clocked for one cycle, the *go* signals involved in the control transfer change value, and the clock control circuitry switches as well. As a result, transitions of control are power consuming and should be avoided as much as possible.

Minimizing the number of *go* signals is another important objective. The generation of such signals requires additional hardware, that increases power dissipation. Moreover, the *go* signals increase the coupling between sub-machines, complicating the placement and routing of the circuit. On the other hand, if we reduce the number of *go* signals to zero, i.e., we do not decompose the FSM, no power savings are achieved.

In summary, we should look for a partition $\Pi(S)$ which maximizes the locality of the computation and minimizes the hardware overhead for communications between sub-FSMs. We formally describe this problem in the next subsection.

4.3.4.1 Partitioning as integer programming. In the following discussion we assume that the probability of occupancy of every state in the original FSM has been computed. This task can be performed by simulating the behavioral description of the FSM or by an analysis based on a Markov chain model, as seen in Chapter 2.

The problem of finding an optimal partition $\Pi(S)$ can be formalized as an Integer Programming (IP) problem [nehm88]:

Given:
a_i, the probability to be in state i, $i = 1, 2, ..., |S|$
b_{ij}, the probability to transition from state i to j, $i, j = 1, 2, ..., |S|$
n_{max}, the maximum number of blocks in the state partition.
A set of binary decision variables $\{x_{ip}, i = 1, 2, ..., |S|; p = 1, 2, ..., n_{max}\}$ such that $x_{ip} = 1$ if state i in partition block p and 0 otherwise.

Minimize:

$$\sum_{p=1}^{n_{max}} K_p \times Prob(F_p) \tag{4.6}$$

Such that:

$$\sum_{p=1}^{n_{max}} x_{ip} = 1, \quad i = 1, 2, ..., |S| \tag{4.7}$$

i.e., every state has to be assigned to exactly 1 partition.

The formulation of the cost function requires further discussion. The cost of the interacting FSM implementation can be expressed as the summation over all sub-FSMs of the cost of each sub-machine (K_p) weighted by its probability to be active ($P(F_p)$). The cost of a sub-machine is expressed by a linear combination of the number of states of the machine and the number of possible transitions from and to the sub-machine (meaning extra I/O). This is can be expressed by the following equation:

$$K_p = \left(1 + \sum_i x_{ip}\right) + \alpha \left(\sum_{p_1 \neq p} \sum_i \sum_j \lceil b_{ij} \rceil x_{ip_1} x_{jp} + \sum_{p_2 \neq p} \sum_i \sum_j \lceil b_{ij} \rceil x_{ip} x_{jp_2}\right) \tag{4.8}$$

The first part of the formula is simply the number of states in partition block p. The second part of the formula accounts for the transitions connecting states in p with states in other partition blocks. The term $\lceil b_{ij} \rceil x_{ip_1} x_{jp}$ is 1 if the transition probability from state i to state j is $b_{ij} > 0$, with i in partition block p_1 and j in p. It is zero otherwise. By summing over all edges (a sum over i and j) and all partition blocks p_1 different from the one under consideration, we obtain the total number of transitions into p. Similarly, the term $\lceil b_{ij} \rceil x_{ip} x_{jp_2}$ holds value 1 only for transitions with non-null probability from a state in partition block p to a state in another partition block.

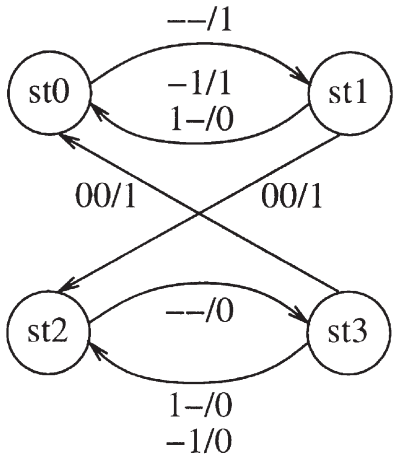


Figure 4.8. STG of the example FSM

The rationale behind the cost function is that we look for a decomposed implementation with minimum interface cost. The first part of the formula penalizes implementation with excessively coarse granularity, while the second part penalizes implementations where the interface overhead is high. The coefficient α expresses the relative weight that should be given to minimizing the additional gates and wires needed to implement the interface between sub-FSMs (i.e., the *go* signals and the logic generating them) with respect to the number of states of each sub-FSM (i.e., the granularity of the decomposition).

The probability that a particular machine F_p is powered, $Prob(F_p)$ is equal to the total state occupancy probability for states in F_p plus the total probability of transition to F_p. From other partition blocks. In symbols:

$$Prob(F_p) = \sum_i a_i x_{ip} + \sum_{p_1 \neq p} \sum_i \sum_j b_{ij} x_{ip_1} x_{jp}. \tag{4.9}$$

Notice that it is important to consider the probability of the incoming edges for each partition because they mark transitions of controls. During transitions of controls *two sub-machines are powered on at the same time.* Therefore, the summation of $Prob(F_p)$ over all sub-machines F_p is larger than one, accounting for the cycles when two sub-machines are enabled. The formalism used for the IP formulation will be clarified through an example.

Example 4.8 *Consider the FSM specified by the STG in Figure 4.6 (a), reproduced in Figure 4.8. We assume equiprobable and independent inputs for*

the sake of simplicity. From the input probability distribution and the STG, the state probabilities are computed: $[1/4, 1/4, 1/4, 1/4]^T$. *The transition probabilities are computed from state probabilities and input probabilities. They are collected in the transition probability matrix:*

$$\begin{bmatrix} 0 & \frac{1}{4} & 0 & 0 \\ \frac{3}{16} & 0 & \frac{1}{16} & 0 \\ 0 & 0 & 0 & \frac{1}{4} \\ \frac{1}{16} & 0 & \frac{3}{16} & 0 \end{bmatrix}$$

We assume a 2-way partition of the states, where `st0` *and* `st1` *are assigned to block 0 and* `st2` *and* `st3` *to block 1 . The partition is expressed by the matrix of* x_{ip}*:*

$$\begin{bmatrix} 10 \\ 10 \\ 01 \\ 01 \end{bmatrix}$$

Notice that this matrix is constrained by the fact that every row sums up to 1, since every state is assigned to exactly one partition. We calculate the cost for partition block P_1. *The first term is* $1 + \sum_i x_{ip} = 3$, *which counts the 2 states of block* P_1, `st0, st1` *and the reset state the sub-machine goes into when it is powered down. The second term in the cost function consists of two contributions: i) incoming edges, ii) outgoing edges. There is only one incoming edge, namely for* $i = 3, j = 0, p_1 = 1$, *and one outgoing edge, for* $i = 1, j = 2, p_2 = 1$. *The cost for* P_1 *is therefore* $3 + \alpha(1 + 1)$.

The probability to be in P_1 *is* $1/2$. *We know from the cost calculation that there is 1 possible transition into this partition, for* $i = 3, j = 0, p_1 = 1$. *The probability of this transition is* $b_{30} = \frac{1}{16}$. *Note that the probability of transitioning out of the partition equals the probability of transitioning into it. We only have to count one of both. Hence, the total probability is for* P_1 *to be powered is* $\frac{9}{16}$.

The total cost function for this example is given by the scalar product of the matrix of costs for each partition and the matrix of the probabilities for each partition to be powered. Considering the symmetry of the example and choosing $\alpha = 1$, *we get* $5\frac{9}{16} + 5\frac{9}{16} = \frac{45}{8}$.

Several techniques have been proposed solve the IP problem exactly and heuristically [nehm88]. The exact solution of IP requires algorithms with above-polynomial worst-case complexity since IP is NP-complete [nehm88]).

4.3.5 Partitioning algorithm

Exact minimization of the cost function is computationally intractable. We describe a heuristic, computationally tractable solution based on a *genetic algorithm* [gold89] (GA) which still leverages the IP formulation of the problem. Some properties of the problem made it well suited for this approach. First, the the solution space is easily representable as a set of bit-strings: a chromosome is encoded as a set of $|S|$ blocks of $\lceil log_2 n_{max} \rceil$ bits. Each block is associated with a state and represents the number identifying the partition block to which the state belongs. The length of the chromosome in bytes is $|S| \cdot \lceil log_2 n_{max} \rceil / 8$. This is a very compact encoding and if n_{max} is chosen as a power of two, every chromosome represents a valid solution.

Second, and more importantly, the cost function can be efficiently evaluated, and its computation is $O(|E|)$ where $|E|$ is the number of edges in the STG of the original machine. The compact encoding (with no invalid solutions) and the inexpensive computation of the cost function allow us to take very large populations (in the order of 10^6 individuals).

To efficiently evaluate the cost function, the state probability vector and the transition probability matrix of the monolithic FSM are stored once and for all at the beginning of each GA run. Other input parameters are the relative cost of the I/O overhead (parameter α in Equation 4.8) and the maximum number of partitions to consider n_{max}. The simplified pseudo-code for the evaluation of the cost function in the genetic algorithm is shown in Figure 4.9.

There are three loops in the algorithm. The first iterates over all the states to compute the first part of the cost function ($\sum_i x_{ip}$) and the cumulative probability of the states in each partition. The second loop computes the second part of the cost function (i.e., the cost of the interface signals) and the probability of the transitions of control. Finally, the third loop iterates on the partitions and computes the final value of the cost function (i.e., the weighted sum of Equation 4.8). The worst case complexity is $O(|E|)$ because, for non-sparse STGs, the number of edges is of the order of $|S|^2$. The second loop usually dominates the execution time.

It is important to notice that the optimization strategy not only tries to find an optimal partition, but also automatically searches for an optimal *number of blocks* in the partition. In particular, if the circuit is not decomposable in a favorable way, the GA run will produce a degenerate partition consisting of a single block containing all states of the original FSM. The only constraint on the partition is the maximum number of allowed blocks, n_{max}.

Concluding this subsection, we point out that the optimization strategy is based on a cost function which is not directly proportional to the actual power dissipation. However, the experimental results confirm that the cost measure is

```
foreach (state i) {
  Cost[partition(i)] ++;
  probability[partition(i)] += StateProbability(i);
}
foreach (edge i → j) {
  if (partition(i) ≠ partition(j)) {
    probability[partition(i)] += TransitionProb(i→j)
    Cost[partition(i)] += α
    Cost[partition(j)] += α
  }
}
TotalCost = 0;
foreach (partition p) {
  Cost[p] ++;
  TotalCost += Cost[p] * probability[p]
}
```

Figure 4.9. Algorithm for the computation of the cost function

very informative in the *relative* sense (i.e.,, it can be used to compare the power dissipation of two alternative decomposed implementations). If one partition has much better locality and lower interface cost than another one, the power dissipation of the first will be usually much smaller than the second one. This is because the gated-clock implementation guarantees that only a small part of the system is active at any given cycle, therefore a system with few transitions of control and few *go* signals dissipates less power than a system with poor locality and high interface overhead.

Notice also that the technique can be biased by increasing the relative cost of the interface overhead in the cost function. Changing the value of coefficient α the user can control the likelihood of the generation of a decomposed implementation. Moreover, by controlling n_{max} the user can set a lower limit on the granularity of the partition.

4.3.5.1 Refined model for partitioning. Until now we have assumed that every transition between states of two different partition blocks corresponds to an additional interface signal in the interacting FSM implementation. This is a safe but conservative assumption. In many cases the number of interface *go* signals can be highly reduced.

The reduction in the number of *go* signals is based on the following observation. When the control is transferred from one sub-FSM to another, the only

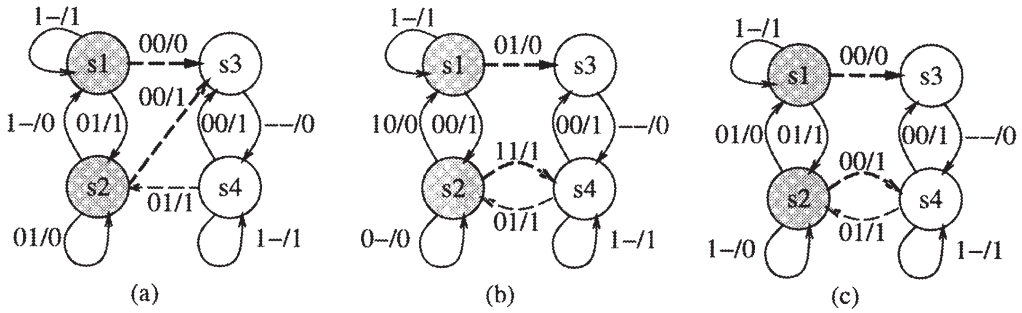


Figure 4.10. Decomposition of the monolithic FSM

information that the newly activated machine needs is the destination state to which it has to transition. Assume that there are two incoming edges from states in block P_i to states in block P_j of the state partition. If the two edges are directed to the same state in P_j, there is no need to have two different *go* signals, because sub-machine F_j should transition to the same state when it is activated by submachine F_i. The output values during the transfer of control are not a problem, because they are set by F_i, which can distinguish between the two transitions, since they are originating from two different states.

Even if the two edges go to different states in P_j, we might be able to use the input signals that caused the transitions to discern between the destination states. If the input value labeling the transitions are different, we can use them in sub-machine F_j to direct the transition from its reset state towards the right destination. The only case when we need to implement more than one *go* signal between two sub-FSMs is if there are transitions to multiple states in F_j which are not distinguishable by their input fields.

Example 4.9 *Consider the STGs of three monolithic machines shown in Figure 4.10. In each FSM, states s1 and s2 belong to the same partition block, states s3 and s4 belong to another. The edges drawn with thick dashed lines represent transitions of control in the decomposed implementation from the first to the second sub-machine, while the edges drawn with thin dashed lines are transitions of control from the second to the first sub-machine. We focus on the transitions of control from the first to the second sub-machine. Figure 4.10 (a) shows a case where two edges between partition blocks do not require two "go" signals in the interacting FSM implementation, because they are both directed towards the same state. Figure 4.10 (b) shows another case that requires only one "go" signal, since the input field in the two edges allows to distinguish the destination state. The situation shown in Figure 4.10 (c) requires two "go"*

```
if( list of edges from partition(i) to partition(j) == NULL ||
    ( list of edges from partition(i) to partition(j)
      contains an edge going to another state &&
      being triggered by the same input &&
      doesn't contain an edge to this state
      for which a signal was added)) {
  Cost[partition(i)] += α
  Cost[partition(j)] += α
}
add (i,j) to list of edges from partition(i) to partition(j);
```

Figure 4.11. Improved computation of the cost function

signals. The input on the edges cannot be used to distinguish the destination state when, in the decomposed implementation, the sub-machine containing $s3$ *and* $s4$ *is activated by the sub-machine containing* $s1$ *and* $s2$.

Reducing the number of interface signals gives us much more freedom in the choice of the partition. Moreover, many controllers have STG that cannot be partitioned without cutting numerous edges. For these controllers the basic technique proposed in the previous subsection would not find any acceptable partition but the trivial one i.e., the unpartitioned state set), while the improved partitioning technique leads to effective decompositions.

Since the improved decomposition technique can greatly reduce the number of *go* signals required in the interface of the sub-FSMs, the cost function used during the GA to evaluate the quality of a partition is modified accordingly. In the cost function, the number of outgoing and incoming edges is replaced by the actual number of output and input signals to be added to the machine. This is calculated using the pseudo-code of Figure 4.11, which replaces lines 8 and 9 in the original algorithm (Figure 4.9).

4.3.6 Experimental results

We implemented the algorithms presented in the previous section in a FSM decomposition tool that consists of two programs: the *partitioner* and the *netlister*. The partitioner reads in the STG of the monolithic FSM and finds an optimal partition $\Pi(S)$. The frame for the genetic algorithm implemented in the partitioner is provided by the Genesis package [gref90]. The netlister reads in the partition $\Pi(S)$, the STG of the specification and produces the decomposed FSM. One important task of the netlister is the reduction of the

number of *go* signal, which is performed following the approach outlined in subsection 4.3.5.1.

The input of the tool is a simple state table description (in Berkeley `kiss` format or in the similar Synopsys `state table` format) and a file containing the state and input signal probabilities. The output is a set of state tables, one for each partition of the decomposition and a synthesizable Verilog description of the clocking circuitry, containing empty modules corresponding to the sub-machines. Thanks to this simple interface, the designer can read the Verilog code and the state tables in the logic synthesis tool of choice and proceed to the logic-level optimization of the full hierarchical design. The state probabilities are computed with the methods described in Chapter 2.

The time spent in decomposition strongly depends on the effort that the user want to dedicate to the search of an optimal solution. This is controlled by a parameter file that is specified once for all by the designer and is used to set-up the GA run and control the parameters in the cost function. The file contains: i) parameters for the GA run (population size, number of cost function evaluations, probability of cross-over and mutation, and other secondary parameters); ii) the values of n_{max} and α for the control of the cost function. In our experiments we wanted to explore the maximum optimization achievable, therefore we specified large population sizes (from 10^5 to 10^6) and large numbers of cost function evaluations (from 10^6 to $5 * 10^7$). The GA runs were scheduled for overnight runs on SGI Indy machines with 64Mb of memory.

The parameters for the control of the cost function have been the subject of careful study. With technology library used for the experiments, $\alpha \approx 1$ (as defined in Problem 4.1) gave the best results. If pre-layout estimates of the interface cost are considered too optimistic, the value of α can be set to a constant larger than one. Also, α is technology dependent and could change with the technology library used for mapping. The maximum number of partition blocks n_{max} was always set $n_{max} = 8$, since initial exploratory analysis with larger n_{max} showed that solutions with more than 8 sub-machines were never included among the best individuals of the GA runs.

Table 4.3 shows the results on a number of benchmarks. The first three examples are controllers of data-path for small full-custom chips implemented in a class project. The remaining FSMs are standard MCNC benchmarks [mcnc91], with the exception of the last one which is a modified version of MCNC benchmark `s298` (we reduced the number of states because the commercial tool we used for FSM synthesis could not optimize the monolithic implementation with the memory resources available on our machines).

The decomposed and monolithic implementation were both optimized with Synopsys `Design Compiler` running on a Sun SPARC10 workstation, using the same optimization script targeting minimum delay. The circuit power was

Name			Original			Partitioned		
	# of states	# of partitions	# of std cells	power	crit. path	# of std cells	power	crit. path
test1	24	4	67	804	3.81	118 (+76%)	679 (−16%)	3.01 (−21%)
test2	18	3	58	930	2.83	89 (+53%)	642 (−31%)	2.98 (−5%)
test6	80	4	252	2115	7.09	336 (+33%)	1209 (−43%)	5.92 (−17%)
bbsse	13	4	112	1146	4.21	145 (+29%)	847 (−26%)	3.60 (−14%)
dk512	14	2	61	1138	3.29	88 (+44%)	853 (−25%)	2.79 (−15%)
keyb	18	3	157	1688	4.15	262 (+67%)	1387 (−18%)	4.69 (+13%)
planet	48	4	360	4967	6.76	503 (+40%)	3241 (−35%)	5.85 (−13%)
s1488	48	4	433	2743	6.68	642 (+48%)	1717 (−37%)	5.82 (−13%)
s820	25	3	191	1717	5.01	238 (+25%)	1171 (−32%)	3.83 (−24%)
s832	25	3	211	1889	4.61	274 (+30%)	1244 (−34%)	3.72 (−19%)
sand	32	4	429	3395	7.86	471 (+10%)	2554 (−25%)	6.78 (−14%)
scf	112	8	672	3719	7.07	988 (+47%)	2280 (−39%)	5.36 (−24%)
test13	166	8	681	7006	7.38	1610 (+137%)	4124 (−41%)	7.57 (+3%)

Table 4.3. Power, area and speed of the decomposed implementation versus the monolithic one

estimated by PPP[bogl96], an accurate full-delay gate-level power estimator. The critical path timing was estimated after technology optimization by the static timing analysis tool within `Design Compiler`. The technology library is a subset of the Berkeley *Low-Power CMOS library* [burd94]. Interestingly, for all larger examples in the table, the run time of the synthesis tool was much decreased for the decomposed implementation.

The differences in area, power and speed between the partitioned machine and the original unpartitioned design is given in Table 4.3 between parentheses in the last three columns. The average power reduction is 31%. There is also an increase in speed of 12%. The number of standard cells increases on average by 48%. The results listed are only for machines that actually are successfully partitioned. Some designs do not have an effective partitioning, and they are left monolithic by the tool. For example, in the MCNC benchmark suite, the FSMs `bbara`, `bbtas`, `dk16` and `donfile` are not decomposed. Notice that, with the values of P_{max} and α we chose, the tool never produced partitioned machines with power consumption larger than the original one. This indicates that: i) the technique is conservative; ii) more aggressive settings (for example, $\alpha < 1$) could lead to the decomposition of more machines, but the uncertainty on the quality of the results would increase.

Notice also that the increase in area is marked on all the examples. The main reason for this phenomenon is the overhead due to additional flip-flops. We specified minimum-length state encoding in all our experiments. This encoding style implies that the number of flip-flops in the monolithic machine increases only logarithmically with the number of states. When the machine is decomposed, the number of states in each sub-machine is decreased by a factor of two if the partition is balanced. In this case each sub-machine has just one flip-flop less than the original machine and the total number of flip-flops is almost doubled. If the partition is unbalanced, the number of flip-flop is generally increased by the flip-flops required in the smaller machine. Obviously the sequential overhead is larger for N-way partitions, with $N > 2$. We could have performed our tests specifying *one-hot* encoding. In this case, the sequential overhead would have been null. However, we feel that the comparison with minimum-length encoding is fairer towards the monolithic implementation.

It can be observed that the increase in area may translate after layout in increase in power and delay. However the decomposed implementations are more modular, because they consist of small and loosely connected blocks. This characteristic may actually improve the quality of the layout. Moreover, the power savings are quite large and unlikely be completely swamped during layout.

4.3.7 Summary

We have described an algorithm for finite-state machine decomposition for low power consumption. We leverage clock-gating techniques to produce an interacting FSM implementation in which only one or two sub-machines are clocked at any clock cycle, while the others are inactive and dissipate small power. The decomposition procedure integrates easily in synthesis-based design methodologies and can be seen as a pre-processing step on the state table specification of the original FSM. Standard synthesis tools (or the techniques illustrated in previous chapters) can then be used for optimizing the sub-FSMs and additional power reductions can be obtained.

The partitioning algorithm takes into account the overhead imposed by the interface signals required for the interaction of the sub-machines and automatically chooses not only how to partition the state set of the original specification, but also how many partition blocks will be generated. The algorithm is based on conservative assumptions and avoids the generation of decomposed FSMs if the expected power savings are not high. An important byproduct of the technique is the increase in speed of the interacting FSM.

4.4 CONCLUSION

In this chapter we have presented two basic CAD techniques for implementing power management at the logic level. We took a top-down approach, starting from an abstract FSM-based specification of the behavior of a sequential circuit. The first technique reduces power by exploiting internal idleness, and it can be seen as a "power recovery" strategy. Part of the power wasted by input transitions that do not propagate to the outputs can be saved by gating the clock.

The second technique, is more aggressive and aims at reducing power dissipation even for system that are never internally, nor externally idle. We apply a decomposition strategy where a monolithic specification is sub-divided into components that operate in a mutually exclusive fashion. Mutual exclusiveness implies idleness of all but one component during normal operation. Hence, all idle components can be shut down by gating their clocks.

Both techniques do require careful consideration of the trade-off involved in instantiating power-management circuitry. Area increases (sometimes significantly) and speed may decrease. Such undesired side effects must be taken into account when deciding on the applicability of the power minimization techniques.

The main limitation of both approaches is that they target circuits specified as finite-state machines. Such specification style is practical only for control-dominated circuits of moderate size. In the next chapter we shall focus on

techniques that, although based on the same principles, have a much wider applicability, because they either extract idleness from circuits described in a structural fashion or apply to specifications at a higher level of abstraction.

5 POWER MANAGEMENT FOR FUNCTIONAL UNITS

Otio tibi commendo, in quo maiora agas et pulchriora quam quae reliquisti.

I recommend you to be idle so that you can dedicate yourself to more noble and better occupations than the ones you left.

—Seneca

5.1 INTRODUCTION

Although the computer-aided design techniques presented in the previous chapter successfully exploit idleness to reduce power, they have limited applicability. Designers are interested in minimizing power consumption on entire chips, not only on controllers or interface circuitry. Thus more powerful techniques are needed for optimizing parts of a design which are not specified as finite-state machines. To increase the size of the circuits that can be automatically optimized by power management, we can resort to two alternative approaches. We can formulate techniques which operate at a higher level of abstraction (i.e., register-transfer or behavioral level). Some accuracy in idleness detection is lost, but efficiency is much improved. Alternatively, we can work at a level of abstraction even lower than that of FSM specifications. In this case, detection accuracy is high, but specialized algorithms are required to cope with the computational burden. We first outline the salient features of specification styles at diffrent abstraction levels, then we briefly discuss advantages and limitations of the two alternative approaches.

Hardware description languages (HDLs) are almost universally used to specify digital designs in an abstract fashion. HDLs are very expressive and they can describe functionality at many different levels of abstraction. The most common levels are: *behavioral level*, *register-transfer level*, *logic level*. At the behavioral level, the functionality of the design is completely expressed, but the actual scheduling in time of the basic operations required to implement the functionality is not specified. The operations themselves are abstracted by their functionality and the choice of the number or type of hardware resources needed to implement them (i.e., the resource binding) is left to the synthesis tool. Power management techniques that operate at the behavioral of abstraction are potentially very effective, because they can exploit the wide degrees of freedom in the specification.

At the register-transfer level (RTL), the scheduling of operations is specified by the designer. A RTL specification matches the behavior of the final implementation at clock cycle boundaries. Nevetheless, the designer specifies only *when* an operation is carried out, while he/she has still the freedom to let a synthesis tool decide *how* (i.e., by which functional unit) the operation is executed. Clearly, the degrees of freedom available to a RTL synthesis tool are much smaller than those available at the behavioral level. At the RT level the designer has much tighter control on the final architecture (and cost) of the hardware implementation. Data-path circuitry is often described at the RT level because this specification style is detailed enough to accurately model the computation and the flow of data words in a synthetic fashion. Thanks to its generality and its conceptual closeness to actual hardware implementation, the

RTL specification style is probably the most commonly used in large-scale designs. At a lower level of abstraction we have logic-level description, where data and control flow are both represented by the same structure: a logic network. The distinction between control and data-path is lost, and transformations are applied with very fine granularity. In this chapter we shall explore two classes of techniques. The first class detects and exploits power management opportunities on RT or behavioral level descriptions. The second class, in contrast, targets low-level specifications, namely, logic-level netlists.

The main advantage of the first class of methods is that they require a relatively low computational effort for detecting idleness, and operate at a coarse granularity, where power saving are substantial. On the other hand, idleness detection is not very accurate, and good opportunities may be missed. Moreover, although in principle high-level descriptions are very general, in practice there are severe restrictions on the classes of designs that can be synthesized from abstract descriptions. Depending on the synthesis tools employed in the design flow, different and incompatible flavors of high-level specifications are used. Hence, high level techniques are intrinsically limited in scope by the lack of standardization of high-level description styles.

Logic level techniques, on the contrary, have very high idleness detection accuracy, because they analyze the design with fine granularity. Moreover, they have extremely wide applicability. Although several different logic-level specification formats are available, the translation from one format to another is a relatively straightforward task, because all of them have the same descriptive power. On the other hand, the lack of high-level information poses serious challenges to the algorithmic engines of gate-level power management tools. Idleness detection, which is computationally efficient at high level of abstraction, is a very demanding task that requires highly efficient and robust Boolean manipulation data structures and algorithms. Another limitation of logic-level techniques is their limited restructuring power. In other words, it is very hard to transform a logic network with little apparent idleness into an equivalent one where idleness is exposed. This is much easier for techniques operating at higher level of abstraction.

In summary, RTL and behavioral techniques are computationally efficient and potentially very powerful, but they lack accuracy and have limited applicability. Logic-level techniques are computationally demanding, but they are very accurate and have wide applicability. Notice that the approaches are not mutually exclusive. High-level techniques can be applied in a top-down fashion during design refinements. Logic-level techniques operate in a bottom-up fashion to incrementally improve the power dissipation of a netlist by inserting power management circuitry.

In this chapter we review several recently-proposed advanced power management techniques that are applicable to *functional units*, i.e., designs (or design components) larger than those targeted by the basic techniques presented in the previous chapter. We first analyze the important issue of clock distributions for power-managed systems based on gated clocks and we study two optimization techniques for minimizing the power wasted on clock lines. Then we describe two logic-level techniques, namely an extensions of the internal idleness exploitation approach of the previous chapter, and a power management technique known as precomputation that exploits external idleness. We consider next RTL techniques that reduce the computational burden of idleness detection by exploiting information available at the register-transfer level. Finally, we analyze a few behavioral level techniques, that leverage the numerous degrees of freedom available in behavioral specifications to better exploit idleness.

5.2 CLOCK DISTRIBUTION DESIGN

In Chapter 3 we reviewed several alternative methods for shutting down idle units. Among these methods, clock gating is probably the one achieving the best trade-off between effectiveness and applicability. Clocking large designs is a challenge even when the clock is unconditionally distributed. Gated clocks further complicate the distribution problem, because of increased clock skew, but they also provide some opportunity for reducing the power dissipation on the clock network itself, not only on the logic which is disabled by clock gating.

An approach to power optimization of clock distribution networks based on clock gating has been presented by Tellez and co-authors [telle95]. In this work, the authors move from the assumption that a power management methodology based on clock gating is implemented. Moreover, the authors assume that the system is operated in a periodic fashion, where the period is an integral number of clock cycles N_{cycles}. All state-holding elements in the design (single flip-flops at the logic level, n-byte-wide registers and the RT or behavioral level) are associated with *activity patterns*. The activity pattern of register is a vector of Boolean variables, one for each clock cycle in the period of the circuit. In symbols, the activity pattern for register i is: $\mathbf{A}_i = [a_{i,1}, a_{i,2}, ..., a_{i,N_{cycles}}]^T$.

Variable $a_{i,j} = 0$ if the clock signal of register i is gated during clock cycle j, $a_{i,j} = 1$ otherwise. Hence, each register is characterized by a unique activity pattern $\mathbf{A}_i$. In [telle95] the authors consider the problem of synthesizing a power-optimal clock distribution structure by optimally distributing clock-gating blocks. The authors assume that the clock signal is distributed by a *clock tree*. More specifically, a H-tree [cong97] topology is assumed, as represented schematically in Figure 5.1 (a). The leaves of the tree are the clock

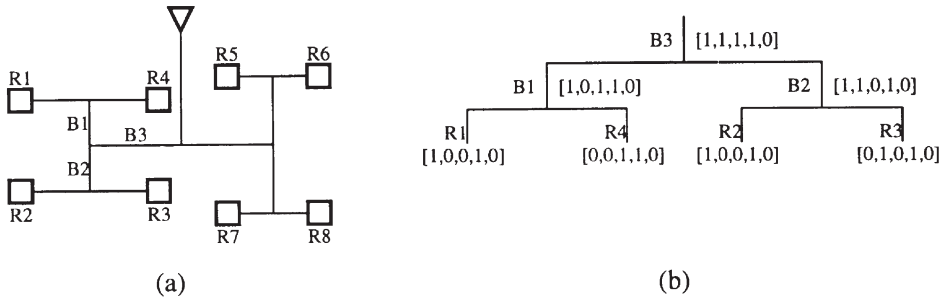


Figure 5.1. (a) A clock distribution network (b) Clock sub-tree and activity patterns

wires which are directly connected to the registers. Thus, an activity pattern is associated with each leaf. Moreover, we can associate an activity pattern with each internal wire of the clock distribution network. The activity pattern of an internal wire is simply bit-wise OR of the activity patterns of all leaves that can be reached from it.

Example 5.1 *Consider internal wire $B3$ in Figure 5.1 (a). The computation of its activity pattern can be carried out in a recursive fashion as shown in Figure 5.1 (b). The activity patterns of the registers are shown at the bottom of Figure 5.1 (b). First, the activity pattern from the internal wires $B1$ and $B2$ are computed by OR-ing the patterns of the registers connected to them. Finally, the activity pattern of $B3$ is computed by OR-ing the activity patterns of $B1$ and $B2$. The disjunction is performed in a bit-wise fashion.*

Notice that in this instance, the periodicity of operation for the circuit whose clock distribution is represented here is $N_{cycles} = 5$. In other words, the activity patterns repeat every 5 clock cycles. For example, for register $R1$ the activity pattern is $\mathbf{A}_i = [1, 0, 0, 1, 0]$. This means that a new value is loaded in $R1$ only in the first and fourth clock cycle of the period. In the other cycles, the clock signal can be gated.

In [telle95], the *activity-driven clock tree construction problem* is formulated and solved. In simple terms, the problem consists of constructing the topology of the clock tree that minimizes the switching activity in the distribution network. It is assumed that activity on a clock subtree can be eliminated for the cycles where the activity pattern of the subtree's root is 0. To effectively solve the problems, registers with similar activity patterns should receive the clock signal by wires belonging to the same sub-tree. A recursive weighted matching algorithm is proposed in [telle95] for the heuristic solution of the activity-driven clock tree construction problem.

Given the optimal tree topology, power can be reduced only by inserting clock-gating logic the internal branches of the clock tree. The straightforward approach of inserting a clock-gating block on each internal branch (and on each leaf) of the tree may be sub-optimal, because the clock-gating blocks dissipate additional power, that may swamp the power saved by gating the clock. Hence, a second optimization problem is formulated in [telle95], namely the *activity-driven minimum power gate insertion problem.* Here the objective is to select the optimal subset of internal branches (and leaves) where the clock-gating blocks should be placed in order to minimize the total power dissipation of the clock tree. The problem is solved with a dynamic programming algorithm that moves in a bottom-up fashion from the leaves of the tree and explores the impact of alternative choices. An alternative algorithm is proposed that takes clock skew into account. The idea is to insert the same number of clock-gating blocks or clock buffers on all root-to-leave paths.

Example 5.2 *Consider again the subtree of Figure 5.1 (b). A better topology swaps $R4$ with $R2$. In the resulting tree, internal branch $B1$ has activity pattern $[1,0,0,1,0]$ which is less active than $[1,0,1,1,0]$. At the same time, internal branch $B2$ has activity pattern $[0,1,1,1,0]$ which has the same activity (i.e., number of ones) as the one shown in Figure 5.1 (b). Internal activity of the new tree is reduced because of the reduction in internal branch $B1$.*

Notice that the reduction of internal activity becomes actual switching activity reduction only if a clock-gating block is inserted on a clock tree branch. The clock-gating logic consumes power, thus there is a tradeoff between its power consumption and the power saved by preventing useless transitions of the clock wires. For instance, it may be more convenient to insert a clock-gating block on internal branch $B1$ of the optimized clock topology because its eliminates useless clock transitions in three clock cycles (over 5), while it may not be as convenient to insert clock-gating on branch $B2$, because the activity pattern has only two zeros (i.e., the clock can be stopped only in three of the five clock cycles of the period).

The main limitation of the approach to clock power management presented by Tellez et al. is the lack of consideration of floor-planning issues. The optimization algorithm does not take the location of the registers into account. If two registers have similar activation patterns, the clock-tree construction algorithm will assign them to the same clock sub-tree. Unfortunately, if the two registers are placed far from each other on the chip, very long wires will be needed to build the subtree, resulting in larger clock skew, higher clock loads and larger area. Moreover, in many practical applications, usage is not periodic, and it is not possible to define activity patterns in a crisp fashion. Nevertheless, the approach presented by Tellez et al. is the first attempt to

solve the important clock distribution issues that arise from the application of clock-gating techniques on large-scale designs.

Another clock power optimization algorithm that is related to clock-gating techniques has been presented by Chung and co-authors [chun95]. The authors assume a tree distribution network, where buffers can be placed on internal branches. The key observation that motivates the work is that there is no need to assert the clock signal on all internal branches with the same frequency used to clock the sequential primitives connected at the leaves of the tree. Clock can be distributed with a reduced frequency by inserting frequency dividers instead of simple clock repeaters/buffers in the inner branches of the clock tree. Frequency multipliers must then be inserted on the leaves of the tree to re-establish the correct clock frequency.

In [chun95] the author solve an optimization problem that entails: i) the choice of the optimal insertion points for frequency dividers and multipliers, ii) wire width optimization for the tree branches, iii) sizing of clock buffers, dividers and multipliers. The objective is to reduce power under skew constraints. Although the optimization algorithm is heuristic, the results are promising. Almost 50% power reductions are obtained by reducing the internal frequency in the clock distribution network. It is however important to mention that more traditional clock optimization techniques for low power (see [cong97] for an excellent review) that do not involve frequency division and multiplication do obtain similar power savings. Thus, it is not clear how much of the power reduction is due to the frequency reduction with respect to the effect of wire and buffer sizing.

Interestingly enough the approaches by Chung and Tellez can be merged, and both can leverage the general power clock optimization techniques presented in [cong97]. It is apparent that clock gating and clock frequency control can be very effective techniques not only for power reduction in functional logic but also to save power on the clock distribution network, which is probably them single largest contributor to the overall on-chip power budget. Moreover, the skew control techniques presented by both authors demonstrate that clock skew can be kept under control even when a significant amount of logic circuitry is inserted on the clock lines. Hence, power management based on clock gating appears to be viable even for large designs.

5.3 LOGIC-LEVEL TECHNIQUES

Logic-level techniques follow a bottom-up paradigm. The initial data consists of a synchronous network (i.e., combinational gates and flip-flops) and input probability distributions of typical input patterns. The output of the optimization tool is a modified network where power management circuitry has been

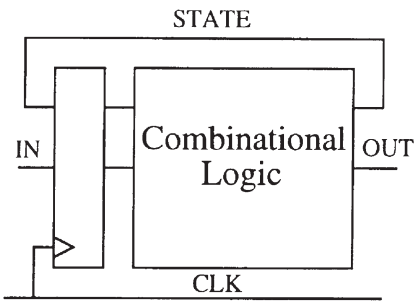


Figure 5.2. Synchronous network architecture

inserted. We assume that clock gating is used. The techniques presented in this section are applicable with minor changes to power management schemes based on enabled flip-flops.

We describe two logic-level techniques. The first is based on the exploitation of internal idleness, while the second leverages external idleness. The first technique is the adaptation of the clock-gating approach developed in Chapter 4 to sequential systems described in a structural fashion, while the second is known as *precomputation* and it is has been proposed by Alidina and coworkers [alid94]. Both approaches target flat synchronous networks with the structure shown in Figure 5.2. All input signals are latched (i.e., they are stored in flip-flops before being fed to the combinational logic). Some of the output signals may be fed back to the inputs of the combinational logic after being stored in flip-flops. No hierarchy information is available.

The signals which are fed-back to the inputs are called *state variables.* The architecture of Figure 5.2 is fully general. Any FSM specification can be implemented in this style. If there are no state variables (*feedback-free* architecture), the architecture degenerates into a typical pipeline stage. Notice that even the feedback-free network is sequential because the value of the outputs at any given clock cycle n depends on the value of the inputs in cycle $n-1$.

5.3.1 Exploiting internal idleness

We consider first the approach based on internal idleness, which is based on the concept of self-loops as idle conditions developed in Chapter 4. To extend such an idea to designs described in a structural fashion, the most straightforward approach is to extract the abstract behavior of the circuit from the synchronous network and re-synthesize the circuit using the tool presented in Chapter 4. The extraction procedure can be automated. Its input is the synchronous network

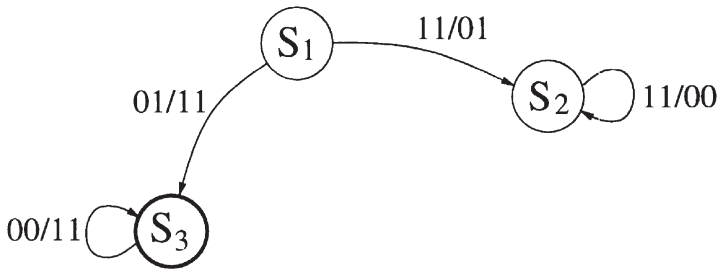


Figure 5.3. Fragment of a Mealy FSM. S_2 is Mealy-State while S_3 is a Moore-State.

and the output is a state transition graph (STG) which fully describes the FSM implemented by the synchronous network.

Unfortunately, it is often computationally infeasible to extract the STG representation for large sequential circuits. As a consequence, the transformation from Mealy-states to Moore-states and the activation function synthesis procedure presented in the previous chapter is not practically applicable and we must restrict ourselves to the Moore-states of the Mealy FSMs. In other words, while in the previous chapter we could *increase* the number of idle conditions by transforming the STG of the FSM, we will for now assume that only the self-loops leaving Moore-states of the original FSM can be selected as clock-gating conditions. An extension to deal with arbitrary internal idle conditions will be introduced later.

Given an FSM implemented by a synchronous network, we want to find the self-loops of Moore-states. Such self-loops are uniquely identified by the present-state and input values and represent the set of idle conditions that may be exploited to stop the clock. For example, for the FSM fragment in Figure 5.3, the only useful idle condition is the self-loop on S_3 (identified by input value 00 and state value S_3).

The *complete activation function* $f_a(x, s)$ (i.e., the set of input-state conditions for which the clock of the unit can be stopped because of internal idleness) is defined by the union of all self-loop conditions for Moore-states (x and s are respectively the input and state variables). The set of all self-loops in the FSM includes f_a, because it contains also the self-loops of Mealy-states.

The identification of the Moore-states can be performed implicitly (i.e., without extracting the STG) by a procedure that requires a single *unrolling* of the sequential circuit, i.e., duplicating the combinational logic to represent two consecutive time frames, as shown in Figure 5.4. There are two cascaded logic blocks: the inputs of the first combinational block are x and s, representing respectively primary and state inputs. The outputs are z and t. The next state

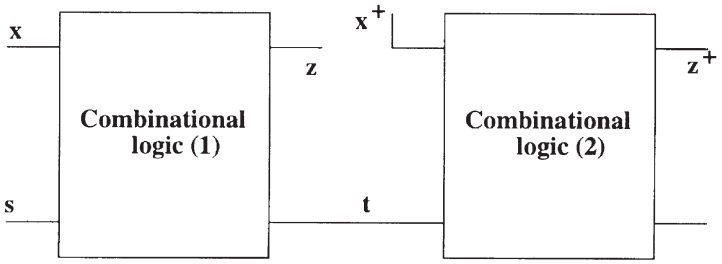


Figure 5.4. Unrolling of a FSM.

outputs t of the first block are fed into the state inputs of the second block. The primary input values in the second block are represented by x^+, while the output of the second block are z^+ and the next state outputs.

With this model, finding the Moore-states is quite simple. Remember that a generic sequential circuit is fully described by its output function $\lambda(x,s) = [\lambda_1(x,s), \lambda_2(x,s), ..., \lambda_m(x,s)]^T$ and its next-state function $\delta(x,s) = [\delta_1(x,s), \delta_2(x,s), ..., \delta_k(x,s)]^T$, where m and k are the number of outputs and state variabes, respectively. For a Moore state t, the following property holds: if in the second combinational logic block the state transition is a self-loop (i.e., $\delta(x^+,t) = t$), for each state transition $s \rightarrow t$ in the first block, the output $z = \lambda(x,s)$ and $z^+ = \lambda(x^+,t)$ are the same. Intuitively, this property expresses the requirement that every incoming edge for state t has the same output value, but we are interested only in states with self-loops, because otherwise no idle conditions are available. Finding all states for which the condition holds is equivalent to finding all Moore-states with self-loops, but no STG extraction is required. This procedure lends itself to an elegant symbolic formulation.

The expression giving the activation function f_a (i.e., the set of all idle conditions) in symbolic form is the following:

$$f_a(x^+,t) = \prod_{i=1}^{k} \left(\delta_i(x^+,t) \equiv t_i\right) \cdot \forall_{x,s} \left(\prod_{i=1}^{m} \left(\lambda_i(x,s) \equiv \lambda_i(x^+,t)\right) + \left(\prod_{i=1}^{k} (\delta_i(x,s) \equiv t_i)\right)' \right) \quad (5.1)$$

We analyze each term of the seemingly complex Equation 5.1 in greater detail.

- The term $\prod_{i=1}^{m} (\delta_i(x^+,t) \equiv t_i)$ imposes the condition that, in the second frame of the unrolled circuit, the machine has a self-loop. This is expressed by having each present-state variable t_i identical to the next state function $\delta_i(x^+,t)$.

- The term $\prod_{i=1}^{m}(\lambda_i(x,s) \equiv \lambda_i(x^+,t))$ describes the constraint on the output values. Since we are detecting Moore-states, we require that the output values of the incoming edge and the self-loop are the same. Notice that the unrolling implies the use of different variables for the two frames of the unrolled circuit.

- The term $(\prod_{i=1}^{m}(\delta_i(x,s) \equiv t_i))'$ is OR-ed with the second term to express the fact that the equality of the outputs in two frames does not need to be enforced for transitions not in the next state functions of the FSM.

The universal quantification on the inputs x and the state variables s enforces the condition for all states and input values. Is is important to notice that in Equation 5.1 all states are considered as reachable, even if this is not generally true. Fortunately, this assumption does not lead to incorrect implementation.

To justify this claim, consider the following situation. Assume that state s_0 is unreachable, and for input i_0, $f_a(s_0,i_0) = 1$. In other words, the clock would be stopped when the state is s_0 and the input is i_0. However, since s_0 is unreachable, the state lines will never hold that state value, therefore the point s_0, i_0 in the *controllability don't care set* of f_a.

The activation function f_a produced by Equation 5.1 is expressed in terms of the auxiliary variables (x^+,t) for convenience, and can be easily re-expressed as a function of the inputs x and present states s by variable renaming.

Example 5.3 *Consider the FSM of Figure 5.5 (a). The next state function $\delta(x^+,t)$ and output function $\lambda(x^+,t)$ of the FSM are shown in Figure 5.5 (b) and (c), respectively. For the sake of explanation, we represent Boolean functions using truth tables. In practice all Boolean functions are computed and manipulated in BDD form (as explained in Chapter 2). The functions $\delta(x,s)$ and $\lambda(x,s)$ can be obtained by simply rotating the truth tables of $\delta(x^+,t)$ and $\lambda(x^+,t)$ by 90 degrees (clockwise), and are not shown.*

The truth table of the first term in Equation 5.1 ($\prod_{i=1}^{m}(\delta_i(x^+,t) \equiv t_i)$) is shown in Figure 5.5 (d). The truth table of the second term ($\prod_{i=1}^{m}(\lambda_i(x,s) \equiv \lambda_i(x^+,t))$) is shown in Figure 5.5 (e), while the third term (i.e., $(\prod_{i=1}^{m}(\delta_i(x,s) \equiv t_i))'$) is shown in Figure 5.5 (f).

The truth table of the activation function $f_a(x^+,t)$ is shown in Figure 5.5 (g). The reader can verify the correctness of the result by observing that the f_a is one in both self-loops of the STG. This is correct, since both states of the FSM are Moore-states.

A direct application of Equation 5.1 yields, in the general case, activation logic whose power dissipation may partially mask off the potential power savings. Therefore, it is mandatory to develop a systematic method to reduce the

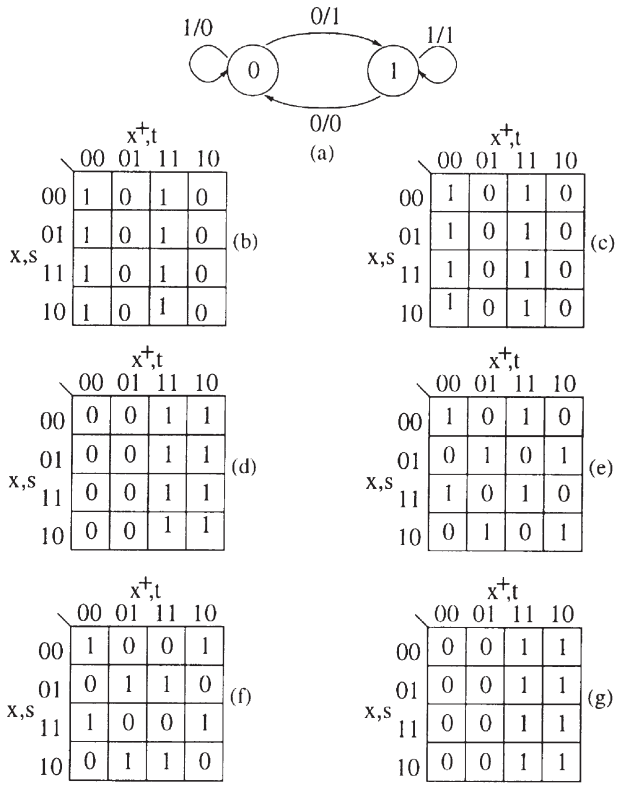


Figure 5.5. Example of symbolic computation of f_a.

power consumption of the activation logic implementing f_a, while keeping as high as possible the probability of its ON-set.

The reader should notice that this is exactly the same problem discussed in the previous chapter where an exact procedure was presented for the minimization of a sum-of-product implementation of f_a. The two main limitations of such procedure were i) its high computational cost, ii) the fact that we are interested in minimizing power dissipation of f_a, thus minimizing the number of literals is only a rough approximation. Both limitations are addressed here. the algorithm outlined in this section is efficient because it operates directly on the BDD representation of f_a and it targets explicitly the power minimization of a multi-level implementation of the reduced activation function F_a.

First, the ADD of a pseudo-Boolean function, P_{f_a} is constructed. It implicitly represents the probability of the minterms in the ON-set of f_a. Then, some of the ON-set minterms of f_a are removed until a given cost criterion breaks the

```
procedure Reduce_Fa(f_a, P_fa) {
    F_a^Best = f_a; P_Best = P_fa;
    F_a^Current = f_a; P_Current = P_fa;
    Best_Cost = Compute_Cost (F_a^Current);
    while (not Stop_Test(F_a^Best, P_Best)) {
        F_a^Current = Prune_Fa (F_a^Current);
        Curr_Cost = Compute_Cost (F_a^Current);
        if (Curr_Cost ≤ Best_Cost) {
            F_a^Best = F_a^Current;
            P_Best = P_Current;
            Best_Cost = Curr_Cost;
        }
    }
    return (F_a^Best);
}
```

Figure 5.6. The Reduce_Fa Algorithm.

loop. The pruning procedure returns the reduced activation function $F_a \leq f_a$. Clearly, both the minterm removal and the stopping condition must be guided by a combination of the size improvement in the implementation of F_a and the probability decrease of the ON-set of F_a.

5.3.1.1 Iterative reduction of f_a. Given the complete activation function, f_a, and its probability function P_{f_a}, the reduction algorithm iteratively prunes some of its minterms until an acceptable solution F_a is found. The pseudo-code of the procedure is shown in Figure 5.6.

The objective of procedure `Reduce_Fa` is to determine a new activation function, F_a^{Best}, which is contained into the original f_a, has a high global probability, and is less costly (in terms of both power and area) if compared to f_a. Three main routines are called inside `Reduce_Fa`:

- `Prune_Fa` eliminates some of the minterms of f_a producing a function whose ON-set is strictly contained into that of the original f_a.
- `Compute_Cost` evaluates the power cost of the implementation of the current F_a. It can be designed to take into account different cost metrics such as area and timing.
- `Stop_Test` is the exit condition. It returns 1 when it estimates that further reduction of the ON-set of F_a cannot improve the circuit's power dissipation.

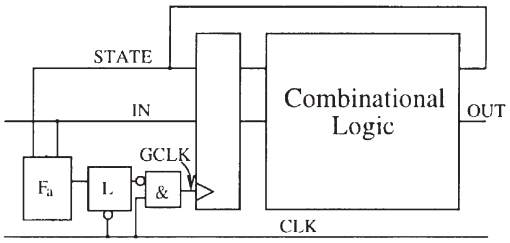


Figure 5.7. Gated clock architecture

The algorithm in Figure 5.6 is a simple greedy procedure that decreases the size of the activation function until the point of diminishing returns. The quality of the optimization depends on the implementation of the three routines `Prune_Fa`, `Compute_Cost` and `Stop_Test`. The detailed description of their implementation is beyond the scope of this section. The interested reader is referred to [beni97a] for a complete treatment.

5.3.1.2 Global circuit optimization. The result produced by procedure `Reduce_Fa` is a gate-level specification of the reduced activation function, $F_a \leq f_a$, which is expected to reduce power dissipation when appropriately connected to the original sequential design and to introduce only a small overhead.

After the logic is included in the circuit as shown in Figure 5.7, some global optimization can be performed. Notice that the activation function is functionally redundant. Since we employed redundancy removal procedures targeting area minimization rather than power minimization, the optimizer may remove the clock-gating logic in its entirety, thus producing a circuit which is very similar to the original one. This is most likely to happen when F_a is used as external *don't care set* for each primary and state output and redundancy removal methods are used for the optimization. Clearly, this is something we must avoid.

To overcome this problem it is possible to add an extra pin to the circuits, to make function F_a directly observable. With this artifact, redundancy removal procedures can be applied to the circuit. This type of optimization has highly beneficial effects on the gated-clock circuits: it may not only reduce the power dissipation, but also increase the testability of the system, because it eliminates the untestable faults in the combinational logic generated by the insertion of the redundant clock-activation logic [fava96].

5.3.1.3 Covering additional self-loops. If a sequential circuit is an implementation of a Mealy FSM with no Moore-states, or a feedback-free stage,

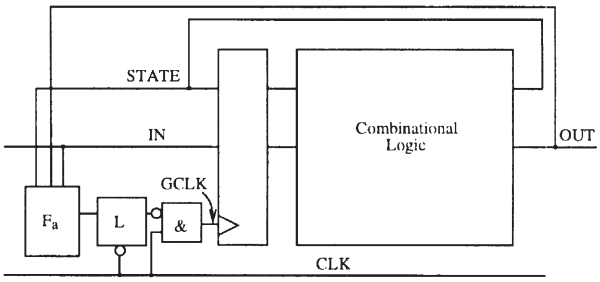


Figure 5.8. Modified Gated-Clock Architecture to Take into Account Circuit Outputs.

the activation function obtained by Equation 5.1 is the null function **0**. We describe a variation of the procedure used to find the initial f_a that allows us to exploit different kinds of idle conditions.

We target self-loops on Mealy states and, more generally the detection of input-output conditions for which clocking the input flip-flops would only result internal spurious transition activity with no changes on the primary output. Such conditions can be detected if the outputs of the sequential circuit are taken as inputs of the activation function as well as the state and primary inputs.

The gated-clock architecture can be modified as shown in Figure 5.8. If all outputs are taken as inputs of the activation function, *all self-loops can be exploited to stop the clock.* As an example, consider again Figure 5.3: if we are allowed to observe the output values, then a state value of S_2, an input value 00, and an output value 11 uniquely identifies the self-loop in S_2. Observing these values we can stop the clock because: i) the FSM is in a self-loop, ii) the output is not going to change in the next clock cycle.

The expression of the activation function including output values is very similar to Equation 5.1:

$$f_a(x^+,t,z^+) = \prod_{i=1}^{k}(\delta_i(x^+,t) \equiv t_i) \cdot \forall_{x,s}\left(\prod_{i=1}^{m}\lambda_i(x^+,t) \equiv z_i^+) + (\prod_{i=1}^{k}(\delta_i(x,s) \equiv t_i))'\right) \quad (5.2)$$

Notice that the support of f_a has been extended to include the output variables z^+. The term $\prod_{i=1}^{m}(\lambda_i(x^+,t) \equiv z_i^+)$ expresses the condition that the observed output value must be equal to the output that would be computed if we clocked the machine when traversing a self-loop. If this is true, we do not need to clock the circuit, hence, $f_a = 1$. This term is the main difference between Equation 5.1 and Equation 5.2: since we have increased the input support

of f_a, we now possess additional information for stopping the circuit more frequently.

We observed in Chapter 4 that the number of outputs in a sequential circuit is often very large, thus, the size of the activation logic may increase too much if we include all outputs in its support. However, it is often the case that we do not need to use *all outputs* as inputs of f_a. For example, when referring to Figure 5.3, it is sufficient to sample the second output to exploit the self-loop on S_2, because the first output does not change on all transition reaching S_2. Formula 5.2 can be modified so that only a subset of the outputs becomes part of the support of f_a. We have:

$$\begin{aligned} f_a(x^+, t, z^+) \quad = \quad & \prod_{i=1}^{k} (\delta_i(x^+, t) \equiv t_i) \ \cdot \\ & \forall_{x,s} \left(\prod_{i=1}^{w} (\lambda_i(x^+, t) \equiv z_i^+) \cdot \prod_{i=w+1}^{m} (\lambda_i(x, s) \equiv \lambda_i(x^+, t)) + \right. \\ & \left. (\prod_{i=1}^{k} (\delta_i(x, s) \equiv t_i))' \right) \end{aligned} \tag{5.3}$$

where w is the number of circuit outputs we want to include in the support of the activation function. Equation 5.3 can be seen as a compromise between Equation 5.1 and Equation 5.2. The smaller w is, the closest the activation function is to the one computed by Equation 5.1.

There is clearly a trade-off between the additional self-loops that can be included in the activation function by adding one or more outputs to its support and its size (and power dissipation). Moreover, the activation function must be drastically simplified by pruning its ON-set for obtaining activation logic of reasonable size. Refer to [beni97a] for a detailed treatment of these issues. The logic-level techniques presented so far allow the designer to exploit the internal idleness of the system. In the next subsection we briefly describe a technique for exploiting external idleness.

5.3.2 Exploiting external idleness

External idleness is caused by lack of observability. Whenever a logic unit is unobservable, its output values are not relevant for correct operation of the entire system. Hence we can freely decide to force the unit in an operation mode that is functionally incorrect, but has minimum power dissipation. Alidina and co-authors [alid94] exploit this intuition to formulate a power management technique known as *precomputation*. In their paper, the authors propose several

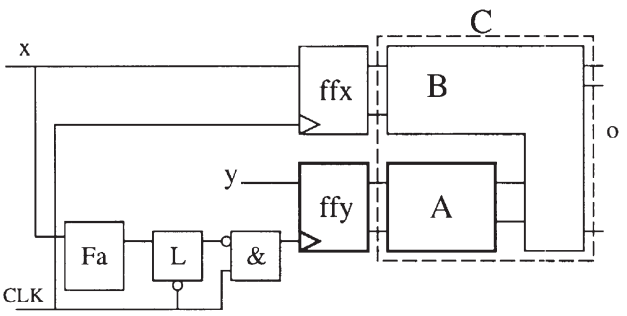


Figure 5.9. Precomputation architecture

architectures for external idleness exploitation. Here, we focus on the simplest one for the sake of explanation.

The basic precomputation architecture is shown in Figure 5.9. Given a monolithic synchronous network C, the purpose of the optimization technique is to find a decomposition in two sub-networks, A and B. Network A is unconditionally clocked, while network B can be shut down by clock gating, controlled by activation function F_a. The conditions for which A can be shut down are all values of inputs x such that the computation performed by A is unobservable at the primary outputs o. As a practical remark on the precomputation architecture, notice that Alidina and co-authors propose a shut-down scheme based on conditionally enabled flip-flops which is safer (but less power efficient) than clock-gating.

The unobservability conditions for block A can be formally computed, given the functionality of block C, described by a multi-output Boolean function $\mathbf{o}(x, y)$. The symbolic equation for the computation of the complete set of unobservability conditions f_a is:

$$f_a(x) = \prod_{i=1}^{N_{out}} (\forall_y o_i(x, y) + \forall_y o_i'(x, y)) \tag{5.4}$$

where o_i are the components of $\mathbf{o}$ and N_{out} is the number of outputs for unit C. Equation 5.4 can be interpreted as follows. The conditions for which block A is externally idle are all values of inputs x such that the outputs of C has constant value 0 or constant value 1 independently from the value taken by inputs y. The ON-set of the activation function can be computed by applying Equation 5.4 to the BDD representation of function $\mathbf{o}$.

The main algorithmic issues in the application of precomputation technology are:

- The choice of the variables x (or equivalently, variables y). A good partition of the input variables is paramount for achieving acceptable power savings. Ideally, the number of x variables should be kept as small as possible in order to minimize the support of F_a and to maximize the number of gated flip-flops and the size of A.

- The implementation of the activation function F_a. Similarly to the technique based on internal idleness, F_a is not constrained to be exactly equal to f_a. Any function $F_a \leq f_a$ would produce a correct implementation.

In their paper [alid94], the authors address only the first problem, claiming that the complexity of F_a can be controlled simply by reducing its support size, i.e., by reducing the number of x variables. The second problem can be solved by the same algorithms proposed in the previous sub-section.

The computation of the optimal support of F_a is based on a recursive algorithm based on signal probabilities. The purpose of the algorithm is to select the subset of cardinality k of input variables for which f_a has maximum probability to be 1. Such subset becomes input set x. The algorithm can be repetitively applied for different values of k to explore the tradeoff between complexity (and power consumption) of f_a and power saving in block C. The algorithm has worst-case exponential complexity (in the number of inputs), thus the authors propose a polynomial-complexity heuristic algorithm that is applicable to circuits with a large number of inputs. Notice again that no distinction is made in [alid94] between the complete f_a and the reduced activation function F_a, hence some degrees of freedom for the optimization of the activation function are lost.

Interestingly, the precomputation architecture based on external idleness and the architecture that exploit internals idleness can be merged together in an integrated solution, as shown in Figure 5.10. The clock-gating block is obtained by combining two activation functions. The first one F_a^i signals internal idleness, while the second F_a^e signals external idleness for C. When $F_a^i = 1$ the entire unit can be shut down. When $F_a^e = 1$ only block C is idle, hence only the flip-flops on y can be stopped. The actual implementation of the activation logic is obtained by merging F_a^i and F_a^o in a single two-outputs network, in order to exploit sharing opportunities and decrease area and power overhead.

5.3.3 Experimental results

We present the results we have obtained by applying the three power management strategies (i.e., internal and external idleness-based shutdown and integrated architecture) discussed in this section on a set of standard benchmarks.

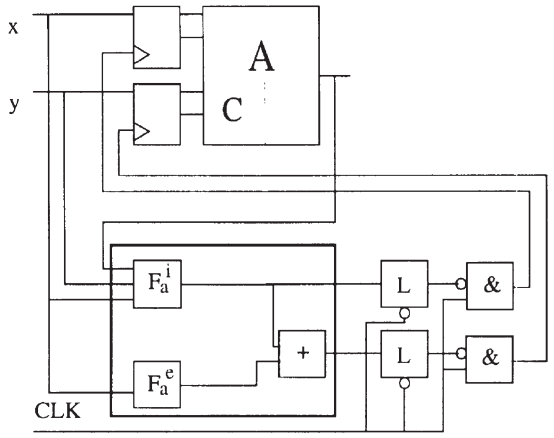


Figure 5.10. Integrated architecture

From the results, we can draw two conclusions. First, pre-computation and the internal idleness-based technique are both efficient for feedback-free (pipelined) circuits, but the former fails for most of the sequential examples. Second, integrated architecture can be advantageous when the target of the optimization are pipelined designs, but it is not very helpful for sequential circuits with feedback.

Tables 5.1 and 5.2 report the experimental data. Pipelined circuits have been constructed by adding input and output latches to some combinational circuits taken from the `MCNC'91` [mcnc91] suite. In particular, we have chosen the examples for which the best power savings have been obtained in [alid94]. Synchronous designs, on the other hand, are taken from the `ISCAS'89` set [brgl89], and are the same as the ones we have used for the experiments in [beni97a].

The selected circuits have been initially optimized using the standard SIS `script.rugged` [sent92], and mapped for delay with the SIS command `map -n 1 -AFG`. The optimized netlists have been used as the starting point for the experiments. The technology library used for mapping includes buffers and inverters with three different strengths, and NAND/NOR gates with up to four inputs. The power estimates, measured in μW, have been calculated using the IRSIM-CAP simulator [salz89].

Circuit	*PI*	*PO*	*Orig.*	*Int.*	*Idle*	*Precomp.*		*Both*	
			P	*P*	*Sav.*	*P*	*Sav.*	*P*	*Sav.*
9sym	9	1	123	62	49%	134	−9%	67	45%
apex2	39	3	286	286	0%	114	60%	114	60%
cm138	6	8	35	13	63%	29	17%	25	28%
cm150	21	1	94	60	36%	74	21%	48	49%
cmb	16	4	66	15	77%	45	32%	14	79%
comp	32	3	144	116	19%	68	53%	75	48%
cordic	23	2	111	38	66%	81	27%	44	60%
mux	21	1	99	61	38%	70	29%	60	39%
sao2	10	4	93	58	37%	48	48%	37	60%

Table 5.1. Results for feedback-free circuits.

Circuit	*PI*	*PO*	*FF*	*Orig.*	*Int.*	*Idle*	*Precomp*		*Both*	
				P	*P*	*Sav.*	*P*	*Sav.*	*P*	*Sav.*
s208.1	10	1	8	75	49	34%	71	5%	52	31%
s298	3	6	14	89	72	19%	120	−35%	112	−26%
s386	7	7	6	63	58	8%	70	−11%	65	−3%
s400	3	6	21	90	63	30%	77	14%	71	21%
s420.1	10	1	16	106	66	36%	97	8%	72	32%
s444	3	6	21	101	76	25%	86	15%	77	24%
s510	19	7	6	95	81	15%	99	−4%	85	11%
s526	3	6	21	119	114	4%	147	−23%	136	−14%
minmax4	3	5	15	85	40	52%	66	22%	34	60%

Table 5.2. Results for sequential circuits with feedback.

The use of internal idleness power management on the pipelined circuits has shown to be more effective than pre-computation in six of the nine examples we have considered; in the remaining cases, pre-computation has worked better, with a peak performance on benchmark `apex2`, where external idleness exploitation has not given any advantage. From an absolute point of view, the integrated architecture has produced the best savings in four cases.

Concerning sequential designs, the application of pre-computation has always given worse results than those obtained with the external idleness exploitation alone; in some cases, it has even resulted in a higher power consumption than the original circuit. The reason for this poor behavior lies in that the pre-computation function never attempts to stop the propagation of present-state inputs, which represent the majority of the inputs to the combinational logic for sequential circuits with a realistic number of memory elements. As a consequence, the results obtained with the integrated solution are not satisfactory, except for the case of benchmark `minmax4`. This was somehow expected, since `minmax4` contains quite many comparators.

5.3.4 Summary

We conclude the section with a few general observations on bottom-up techniques. First, experimental evidence shows good power savings on benchmark circuits, confirming the validity of the approach. Unfortunately, extensive experimentation has shown that good power savings can be obtained only on a small subset of the available benchmarks. Moreover, the computational burden of the calculation of idle conditions (i.e the computation of f_a) can be quite high even for the relatively small (compared to industrial designs) circuits included in the benchmark suite.

As a result, the scope of applicability of bottom-up techniques is currently quite limited. For many large logic-level circuits either the computation of f_a fails because of excessive time and memory requirements, or the activation function is empty (no idleness is detected). Industrial-strength synthesis tools must address these limitations before bottom-up techniques become practically viable in a commercial environment.

Widening the scope of bottom-up techniques is a matter not only of perfecting and engineering the current approaches, but also a challenging research area. In particular, more research is needed to formulate robust algorithms for the computation of f_a leveraging the fact that its exact computation is not required as long as we can guarantee that a safe approximation is obtained (i.e., a function $F_a \leq f_a$). An even more promising direction of research is the study of partitioning algorithms that, starting from a flat logic netlist, partition the specification into a set of sub-units. The techniques described in this section

could then be applied to each sub-unit. Partitioning seems to be the only viable approach to deal with extremely large designs like the ones that are customary in the industrial practice.

5.4 RTL TECHNIQUES

At the register-transfer level, the design description is usually partitioned in two main blocks: controller and data-path. While the data-path performs the actual computation as required by the functional specification, the controller enables such computation by assigning in each cycle the correct inputs to functional units, deciding where the computed outputs should be stored and making decisions on future computations based on results of data-path operations. Controller and data-path communicate through two types of signals. *Control signals* are outputs of the controller and inputs of the data-path. *Conditional signals* are outputs of the data-path and inputs of the controller.

The data-path usually consists of two types of resources: *computational units* such as adders, comparators, multipliers and *steering logic*, which consists usually of multiplexers and three-state buffers. Another important component of the data-path is the interconnection structure (busses and global signals). State storage in the data-path is based on *registers*, which are banks of flip-flops. *Register width* is the number of single-bit flip-flops in a register. The state of the controller is stored in a register as well. The control signals drive the selection inputs of the multiplexers and three-state buffers of the data-path. The conditional signals are sampled by the controller for computing its next state and for setting the control signals to the correct value. The structure of a generic RTL description is shown in Figure 5.11. Hardware description languages such as Verilog HDL or VHDL are the preferred choice for specifying RTL descriptions, although graphical specification are quite common as well.

In this section we review a few techniques for automatic synthesis of power-managed circuits starting from RTL specifications. The common characteristic of all RTL techniques is that they leverage the coarser granularity of the description and the knowledge of the type of signals and resources to detect and exploit idleness in a computationally efficient fashion. As a result, RTL techniques are applicable to designs of much larger size than those targeted by the bottom-up approaches described in the previous section. On the other hand, RTL techniques are not applicable if the information on types of signals and resources is missing or incomplete.

5.4.1 Detection of hold conditions

Register-transfer level descriptions of large circuits are characterized by the presence several registers that hold their data for a large fraction of the op-

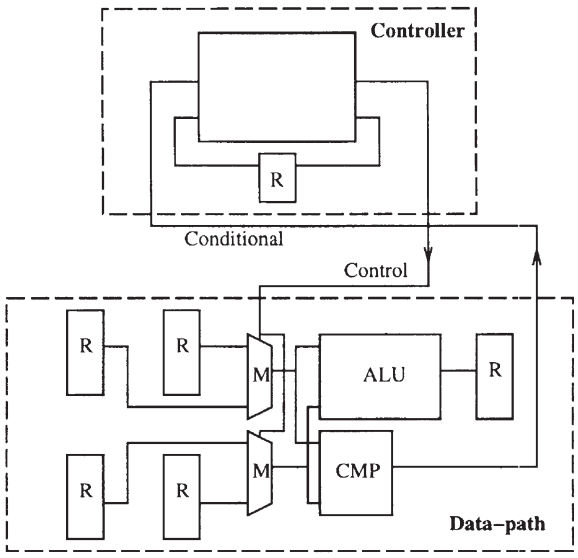


Figure 5.11. Structure of a generic RTL specification.

eration time. State holding functionality is often implemented by providing a conditional feedback loop from the output of the register to its input. If the feedback loop is active, the register is in *hold mode* and does need to be clocked because it does not store any new value. Hence, power can be saved by gating the clock of registers when they are in hold mode.

This observation is the basis of the work by Theeuven and Seelen [thee96]. The author propose a technique to identify flip-flops that are in hold mode for a large fraction of the operation time. For these flip-flops, a *hold expression* is computed. The hold expression is a logic function h that has value 1 when the flip-flop is in hold mode, value 0 otherwise. Power management circuitry controlled by h is then inserted, to stop the clock of flip-flops in hold mode.

The approach of Theeuven and Seelen can be categorized as a technique for exploitation of internal idleness. A flip-flop is internally idle when its input value is equal to its output value. Notice that even if re-loading an input which is identical to the current state does not dissipate power on the output of the flip-flop, internal power dissipation may be non-null and power is wasted to toggle the clock line.

It is important to observe that the minimum amount of circuitry required to stop the clock for a single flip-flop dissipates more power than what is saved in the flip-flop itself. Therefore, the technique in [thee96] is applicable only to clusters of flip-flops that have the same hold condition (or very similar ones). The

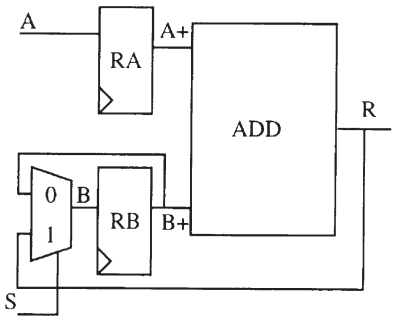


Figure 5.12. Example circuit for the computation of hold function

hold condition for a set S of flip-flops is computed by taking the intersection of their hold conditions $h_S = \prod_{i=1}^{|S|} h_i$. If the ON-sets of the hold conditions h_i do not overlap, their intersection becomes the null function and the clock is never stopped. Another important requisite is that the logic for the computation of the hold function is as small as possible.

The authors of [thee96] propose a simple, yet effective method to compute the hold functions of individual flip-flops in the data-path of large designs such as micro-controllers and digital signal processors. They exploit the information available at the RT level to identify the control signals, and compute only the hold conditions that are directly controlled by such signals, which are a subset of the complete hold conditions.

Consider generic flip-flop k with input d_k and output q_k. Let us call $\mathbf{c} = [c_1, c_2, ..., c_{N_c}]^T$ the control variables, $\mathbf{x} = [q_1, q_2, ..., q_k, ..., q_{N_s}, i_1, i_2, ..., i_{N_i}]^T$ the variables in the data path (i.e., the primary data inputs and the outputs of data-path registers), where N_c, N_s, N_i are the number of control variables, of state variables and of input variables, respectively. Variable q_k is the flip-flop output variable (notice that q_k belongs to $\mathbf{x}$), The input of flip-flop k is a function $d_k(\mathbf{c}, \mathbf{x})$ of control and data variables. The hold condition for flip-flop k is formally defined as:

$$h_k(\mathbf{c}, \mathbf{c}) = d_k(\mathbf{c}, \mathbf{x}) \overline{\oplus} q_k \tag{5.5}$$

The hold condition that depends only on control variables can be expressed as $h_k^c(\mathbf{c}) = \forall_{\mathbf{x}} h_k$. From the definition of the universal quantifier (see Chapter 2) it is clear that $h_k^c \leq h_k$. The hold conditions that depend only on control signals are a safe approximation of the complete hold conditions.

Example 5.4 *Consider the circuit shown in Figure 5.12. It represents a simple accumulator where accumulation can be disabled by a control signal S. We*

focus on the flip-flops in register RB. We want to compute their hold function. By definition the hold function consists of all conditions for which we do not need to store a new value in a flip-flop. For the flip-flop RB_n storing the most significant bit of RB, the hold function has the following expression:

$$h_{RB_n} = S' + S \cdot (q_{RB_n} \overline{\oplus} SUM_n(A+, B+))$$

where the notation $SUM_n(A+, B+)$ represents the most significant output bit of the binary addition function.

If we universally quantify out h_{RB_n} with respect to A+ and B+ (the data inputs), we obtain $h^c_{RB_n} = S'$. This hold condition is much simpler than h_{RB_n} and, most importantly, it is the same for all flip-flops in register RB, while the complete hold conditions change for the various bits in RB.

As demonstrated by the example, hold conditions h^c_k are in general much more suitable than the complete hold conditions for implementing clock-gating logic. They are usually much simpler to compute, and they are shared among large clusters of flip-flops, i.e., registers or sets of registers. Although some internal idle conditions are not exploited, this loss is more than compensated by the compactness of the clock-stopping logic and the large number of flip-flops that can be gated by it.

Theeuwen and Seelen propose a heuristic algorithm for the automatic construction of *hold domains*, i.e., clusters of flip-flops whose clocks are connected to the same clock-stopping block. The algorithm starts by computing hold conditions h^c_k for all flip-flops in the design, then it selects as cluster *seed* the flip-flop whose h^c_k has maximum probability to be 1. The cluster is grown by selecting more flip-flops and including them in the cluster if the product between the probability of the cluster's hold condition and the size of the cluster increases. The hold condition for a cluster is simply the intersection of all h^c_k that belong to it. In order to efficiently implement the hold condition for a domain, the algorithm explores the network trying to locate pre-existing signals that can be used to implement the hold condition logic. Clock gating is actually added to the network only if a user-defined complexity threshold is not exceeded.

Power management based on hold conditions has produced sizable power reductions in a few relatively large circuits. In [thee96], the authors report results for a 8-bit micro-controller and a 16-bit digital signal processor. Power reductions of 19% and 29%, respectively, are reported. Area increases by 7.7% and 0.7%, respectively. No data is reported regarding speed reduction in the power managed circuits.

As a final note, we observe that, although this method achieves good results, the power reductions are somewhat inflated by the design style adopted in the

original designs. When enabled flip-flops are used, there is no need to feed back the data to the inputs of the registers when new values are not loaded. Disabling the flip-flops will be sufficient to guarantee that the output value is held constant. We conjecture that the power savings with respect to designs that make effective use of enabled flip-flop would be much smaller than the ones reported in [thee96]. Nevertheless, the technique represents a significant step forward in extending the basic idleness exploitation techniques to large designs by exploiting information available at the register-transfer level such as the distinction between control and data signals.

5.4.2 Unobservability detection

The technique presented in the previous subsection exploits a form of internal idleness for registers in a RTL description. We now analyze a technique for exploiting external idleness in data-path modules. Data-paths have long and wide busses, which are generally heavily loaded. The large parasitic and load capacitance of such busses causes large power dissipation whenever bus lines are switched. As device technology shrinks, bus capacitances become increasingly larger than internal capacitances of data-path components.

Reducing the switching activity on data-path busses is a very effective power reduction strategy. Power management techniques based on external idleness can be exploited to this purpose. The rationale is that the values driven on bus lines which are not observed by the environment should be frozen in a quiescent state by stopping their drivers. In other words, bus transitions are to be avoided when the "bus resource" is externally idle.

External idleness on data-path busses can be efficiently detected in a RTL description. Remember that we distinguished data-path logic modules in two categories: computational units and steering modules. Computational units are arithmetic or logic circuits such as adders or multipliers. Steering modules are multiplexers and three-state drivers. Steering modules do not perform computation, they are used for communicating data from a source to a destination unit. We can consider registers as a special type of steering resource that communicates data values over time, while multiplexers and three-state drivers communicate over space.

The distinction between steering modules and computational units is exploited to efficiently compute a useful approximation of external idle conditions. The approximation is based on the assumption that computational units are fully observable and that they do not reduce the observability of their inputs. Only steering modules reduce the observability of their inputs, and they are the ones that need to be analyzed for extracting idle conditions.

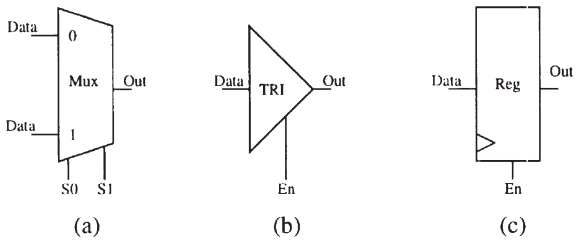


Figure 5.13. Steering modules in a RTL description

Example 5.5 *Consider the steering modules shown in Figure 5.13 (a), (b) and (c). All but one data inputs of the multiplexer are unobservable at any given time. The three-state driver makes its input unobservable if its enable signal is de-asserted. The same holds for the enabled register. In symbols, the unobservability conditions for the inputs (i.e., the observability don't cares, cfr. Chapter 2) are:* $ODC_{Data_0} = S_0'$ *and* $ODC_{Data_1} = S_1'$ *for the multiplexer;* $ODC_{Data} = En'$ *for the three-state driver and the register.*

It easy to realize that the assumption of complete observability of all computational units inputs is a conservative approximation of reality. Consider for instance a multiplier. If one of the operands is zero, the result is zero independently from the value of the other operand. Hence, zero on one operand masks (i.e., makes unobservable) the second operand. The observability loss caused by typical computational units is usually difficult to compute. On the contrary, it is easy to compute observability loss caused by steering modules and, more importantly, the logic for idleness detection is compact (simple functions of control inputs of the steering modules). From these observations, we can formulate an algorithm for efficiently detectingt and exploiting external idleness. The algorithm has the following flow.

- The steering modules in the data-path are levelized. Higher level is assigned to modules that are topologically closer to the primary outputs.
- The steering modules are examined moving from high levels towards low levels.
- For each data input of a steering module, its *observability don't care* set (ODC) is computed.
- Moving from low level to high level, the clock gating logic for the drivers of the data busses is generated.

- Timing constraints are checked in order to ascertain that the clock gating logic can stop the clock in time.

Two steps of this high-level algorithm require further explanation, namely the computation of the ODCs and the insertion of clock gating logic.

Given a steering module, the ODC of its output is the intersection of the ODCs of all its fanout stems. The ODC of one of its inputs is the logic sum of the ODC of its output and the additional ODC condition produced by the steering module itself. If no information on external *don't cares* is available, the ODC of the primary outputs is assumed to be empty. The presence of a computational units is completely transparent. The ODCs of all its inputs is assumed to be the same as the ODC of its output. These simple rules are applied for each steering level moving backwards in the network. The computation is very efficient because the ODCs for steering modules are extremely compact and there are generally not many levels.

Once the ODC for a bus connected to the input of a steering module has been computed, its driver can be gated by a function that freezes it whenever the bus is unobservable. Notice that the driver can be a register, a multiplexer, a three-state driver or a computational unit. If the driver is a register or a three state driver, freezing the bus is relatively straightforward. It is sufficient to qualify the enable signals for the register (or the three-state) with the ODC conditions. If the driver is a multiplexer, the flip-flops driving the control inputs can be clock-gated whenever the multiplexer output is unobservable. Finally, if the driver is a computational unit, the bus cannot be frozen but the inputs of the unit should be frozen instead.

Example 5.6 *In Figure 5.14 we show an example of the power management technique based on external idleness. The data-path has two steering levels. Three-state driver Tri has level 2. Multiplexer Mux has level 1. The input registers have level 0. We assume that the ODC for the output bus is empty. The ODCs for all internal busses B1, B2, ..., B6, are shown in the figure (in boldface). For instance, consider bus B2. It has two fanouts to two computational units. The ODC of the input of a computational unit is the ODC of the output (remember that we neglect the ODCs introduced by computational units). Hence, the ODC of B2 is the intersection of the ODCs of the outputs of the computational units:* $ODC_{B2} = (En' + S_1')(En' + S_0') = En' + S_1'S_0' = En'$, *because* S_1 *and* S_0 *cannot be zero at the same time for correct multiplexing.*

In order to prevent useless switching activity on bus B6 the flip-flops of the control inputs of the multiplexer are disabled by En'. Switching is reduced on B1, B2, ..., B5 by disabling the registers on level 0. Notice that here we used enabled registers, for the sake of schematic readability. A more efficient implementation exploits clock gating to stop the registers. From the schematic,

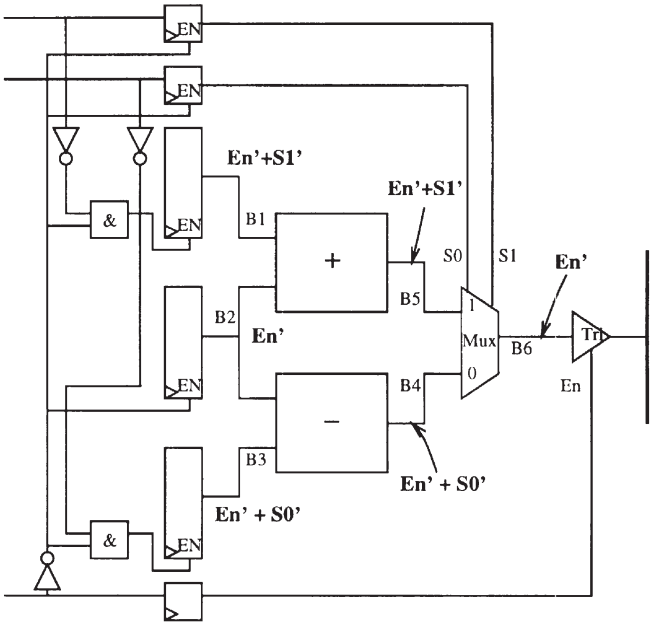


Figure 5.14. Data-path power management based on external idleness.

it is evident that the insertion of power management circuitry has minimal area impact (only three inverters and two AND gates are added). Hoverer, timing analysis should be applied to the modified design to check if the timing of the enable signals does not violate cycle time constraints. This may be the case if the input control signal arrives very late at the flip-flops from the previous stage of the data-path.

A preliminary implementation of the technique presented in this subsection has been applied to the integer data-path of a high performance microprocessor. Power savings around 10% have been measured [kapa97]. More extensive experimentation is required to assess the applicability and the impact of this approach. To summarize, the main features of the techniques are: i) exploitation of external idleness, ii) computational efficiency obtained by exploiting information available at the RT level, iii) applicability to large data-paths such as those found in high-performance micro-processors, iv) low overhead.

5.4.3 Glitch minimization

All RTL techniques described so far do not consider the effect on power of propagation delays in control and data-path logic. Static delay analysis is only performed to ensure that the power-managed implementations do not violate timing constraints. Raghunathan and co-authors [ragh96b] propose a set of techniques aimed at eliminating the power consumption of spurious transitions caused by unequal propagation time of signals within combinational logic circuits. Such transitions, often called *glitches*, are completely irrelevant from a functional point of view (in a synchronous implementation), because the signals stabilize to their final value in time to be correctly sampled and stored in registers. Unfortunately, glitches do dissipate a consistent amount of power.

The authors of [ragh96b] move from two pieces of experimental evidence. First, they claim that steering data-path modules are major contributors to power dissipation of application-specific integrated circuits (ASICs) designed using a synthesis-based methodology. They report that functional units are responsible for only 9% of the total power consumption, random logic (controller and glue logic) consumes 4% of the total, registers 39.5% and steering modules (multiplexers) consume 46.7% of the total.

The second observation is that the estimated power consumption of a design simulated with accurate timing models can be as much as 20% larger than the estimate obtained by a zero-delay simulation, where all gates propagate signals instantaneously and glitches are not accounted for. The 20% difference is entirely due to glitches in the combinational logic. Hence, spurious transition activity is an important contribution to power consumption.

Although it is very expensive (if possible at all) to design completely glitch-free circuits, the authors claim that some architectures are inherently less prone to glitch power consumption than other functionally equivalent ones. In [ragh96b] several RT level transformations are proposed, targeting the reduction of glitches (and the power consumption thereof). Such transformations target specifically control logic and steering modules and have a limited area overhead.

Glitch minimization techniques at the RT level are somewhat marginal to the topic of this book, but they have some relevance because they can be thought as a generalized form of idleness elimination. To better understand this claim, observe that in a synchronous environment any signal can perform at most one functional transition per clock cycle. In case of multiple transition, only the last one is functionally relevant, while all others are spurious. In other words, a combinational logic block is externally idle during a clock cycle until the arrival time of the last output transition, because all previous transitions are not observed by the registers the block fans out to. Glitch reduction tech-

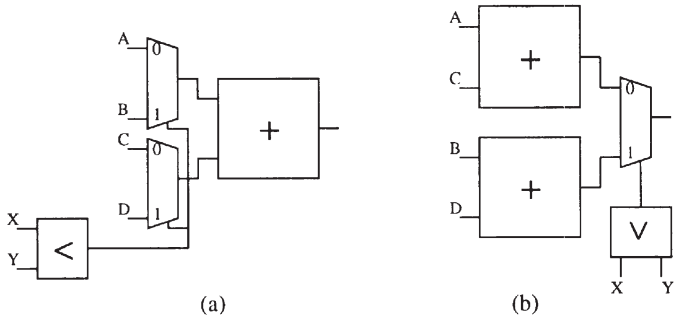


Figure 5.15. Equivalent RTL architectures with different glitching.

niques exploit this form of external idleness, and attempt to reduce the number of unobserved transitions to reduce power dissipation. Before describing the transformations proposed by Raghunatan et al., we give an example (taken from [ragh96b]) of the impact of glitch power consumption on RTL designs.

Example 5.7 *Consider the RTL circuit shown in Figure 5.15 (a). Assume that all its inputs are glitch free. Comparators such as the one on the bottom left side of the figure often generate glitches on their output even if the inputs are glitch free (this is due to their internal architecture that often has paths of unequal length). Assume that a 0-1-0 transition appears on the output of the comparator. The adder and the multiplexers will dissipate a consistent amount of power in spurious transitions, because the adder will first start executing addition $A + C$, then $B + D$, finally $A + C$ again. Only the final $A + C$ result will be sampled on the output.*

In contrast, the equivalent architecture shown in Figure 5.15 (c) has smaller glitch power dissipation, because each of the two adders will perform just one addition per clock cycle. Notice that glitching is not completely eliminated, because still the adder may internally generate glitches, and the glitch on the comparator output is not eliminated, thus the circuit's output is still glitchy.

If power estimation is performed with zero-delay simulation that does not take glitches into account, architecture (a) will certainly appear to be more power efficient than architecture (b). Indeed, if propagation delays are not taken into account, architecture (a) performs just one addition per clock cycle, while architecture (b) performs two in parallel. However, if a full-delay simulation is performed, the power consumption of architecture (a) is larger than that of architecture (b) by more than 17% because of glitch power dissipation.

To better attack glitch power consumption, the authors of [ragh96b] distinguish two types of glitching behavior. *Glitch generation* occurs within RTL

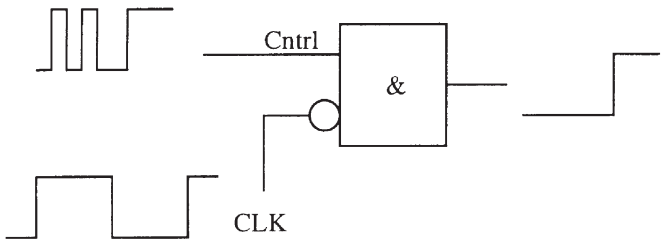


Figure 5.16. Glitch filtering through clock gating.

components, because of uneven delays in the internal gate-level structure of the components. *Glitch propagation* is caused by glitchy inputs which cause multiple spurious transitions within RTL components and on their outputs. The techniques presented in [ragh96b] address glitch propagation. Glitch generation can be reduced by careful logic-level design of the RTL primitives.

5.4.3.1 Glitches in control logic. The control logic by itself accounts only for a small portion of the total power consumption. However, glitches generated in control signals may cause a significant amount of power dissipation when they propagate into the data-path. Glitches propagate from control to data-path mainly through multiplexer select lines. As seen in the previous example, a glitch on a select line of a data-path multiplexer may have disastrous effects on power consumption. Glitch propagation through multiplexer select lines can be reduced by choosing a multiplexer implementation which is slightly more complex, but is less sensitive to glitches on the select line than the simplest implementation [ragh96b]. Alternatively, if several multiplexers are cascaded in a multiplexer tree to select among more than one data signal, the multiplexer tree can be restructured trying to reduce the glitches on the output of the last multiplexer of the tree. Finally, glitches can be filtered by gating the control signal with the clock, as shown in Figure 5.16.

Notice however, that gating the control signal with the clock may not be helpful if the signal has glitches after the falling edge of the clock. Moreover, the technique is not applicable if the required arrival time for the multiplexer control signal occurs is before the falling edge of the clock (i.e., if the multiplexer selection is needed quite early in the clock cycle). The authors in [ragh96b] propose also glitch reduction techniques based on delay insertions on selected signals. Unfortunately, these techniques are difficult to apply at the RT level, because at high level of abstraction accurate delay information is not available, and it is practically impossible to finely tune delay elements.

5.4.3.2 Glitches in data-path. The techniques devised for eliminating glitches on control signals can be adapted to the data-path. Unfortunately, data-path lines are much more numerous than control signals. Therefore any technique that adds gates to data-path is likely to be very expensive in terms of area and power, and not convenient. The authors propose a *selective delay insertion* technique for delaying raising or falling transition on some critical data-path signals [ragh96b].

Although the authors claim that selective delay insertion can be very beneficial, concerns may be raised by the limited accuracy of timing analysis at the RT level, not only for control signals but also for data signals. What is worse, the uncertainty on delay estimations tend to become increasingly large for deep sub-micron technology where wiring delay is significant. We conjecture that techniques involving delay insertion are not likely to have significant impact in the future, but nevertheless they may still be useful in some cases. Probably the best way to reduce glitching in the data path is to synthesize combinational logic with few stages between register. In this way, glitches are not allowed to propagate much.

5.4.4 Summary

In this section we described several RTL power management techniques. One common characteristic to all of them is the exploitation of high-level information such as the distinction between control signals and data signals or the distinction between functional units and steering modules. This information is lost when the RTL specification is mapped into a logic-level network. The exploitation of RTL-specific information is key for computational efficiency. All techniques described in this section are highly efficient compared to logic-level approaches. On the other hand, all RTL techniques resort to some kind of approximation, thereby neglecting a fraction of the idle conditions. In summary, accuracy is traded off for computational efficiency.

5.5 BEHAVIORAL-LEVEL TECHNIQUES

Register-transfer level descriptions are very general and flexible. Nevertheless, the increasing size and complexity of digital systems motivates using a higher level of abstraction which has the benefit of increasing the designer's productivity. However such a paradigm shift must be supported by adequate computer-aided tools.

Behavioral synthesis (also known as *high-level* synthesis) targets the automatic generation of digital designs starting from abstract specifications. The main differentiating factor with respect to RTL specifications is that in a behavioral description only partial information is provided on when operations must

be executed. More specifically, the designer provides a functional specification of the target system (usually expressed as a HDL program) and a few *constraints* on how the system interacts with the environment. The synthesis tool has complete freedom to decide when to perform the individual operations in the functional specification, as long as the constraints on the external interface are satisfied.

The core task performed during behavioral synthesis is called *scheduling*. During scheduling, the synthesis tool decides when to execute individual operations in the specification, and assigns them to specific clock cycles of the execution. Scheduling information is the key data exploited by behavioral-level power management techniques. Such information is not explicitly expressed in RTL or logic descriptions. Consider a functional unit which has been instantiated during behavioral synthesis. If the scheduling information is available, it is possible to establish when the unit is idle in a very straightforward way, by just observing the clock cycles when the unit does not produce results used by subsequent computations. The two techniques briefly described in the following subsection directly exploit this information to reduce power dissipation.

5.5.1 Operand isolation and controller re-specification

During scheduling, a *utilization table* [knapp95] is produced that summarizes usage information for each computational resource instantiated in the design. If the execution of the entire computation requires N_{cycle} clock cycles, and N_{res} resources, the utilization table has N_{res} rows and N_{cycle} columns (or vice versa). An element $u_{i,j}$ of the table has value 1 if resource i is used during clock cycle j, zero otherwise.

It might seem logical to think that good schedules are characterized by very high resource utilization (i.e., a large number of ones in the table), because designs with many, highly underutilized units tend to have large area. In many cases, the utilization tables produced by scheduling algorithms have a significant number of zero elements. Two are the reasons for this fact. First, timing constraints usually require the instantiation of additional (not fully utilized) resources, to avoid serialization of concurrent operations and the consequent latency increase. Second, resource sharing increases the complexity of the steering logic required to drive the correct operands on the resource's inputs and to direct the results to the correct storage elements. Hence, resource sharing may be limited by routability constraints.

When $u_{i,j} = 0$, unit i is not used during clock cycle j. In the hardware implementation, the unit receives input signals and computes a result, but the result is not observed and not stored. Hence, the unit is externally idle in clock cycle j. Notice that the availability of the utilization table makes the

detection of this form of idleness completely straightforward. This is not the case for RTL or gate-level specifications, where extracting external idleness information requires some computation.

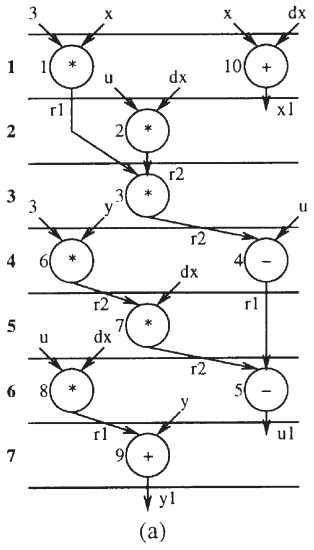
Operand isolation is a *power recovery* technique that exploits external idleness. It can be succintly described as follows. For each unit i, the corresponding row in the utilization table is explored. If the number of zeros divided by the total number of columns N_{cycle} in the table is larger than a user-defined fraction R_i, the unit is modified as follows.

- Latches are inserted on each input line of the unit. Such latches must be transparent in clock cycles when the units is utilized, they are opaque otherwise.
- An additional block of control logic is synthesized, which, depending on the utilization, sets the latches in transparent or opaque state.

Operand isolation is often used in the industrial practice [slat95], and its exploitation during behavioral synthesis has been considered in [muss95]. The applicability and the impact of the technique critically depends on two factors. First, operand isolation relies on transparent latches to prevent useless switching activity in idle resources. Hence, fast and low-power latches are needed. Transparent latches are relatively small cells, that do not consume much power, but they introduce an additional level of logic in the data-path, and they may cause unacceptable slowdown in units that are tightly timing-constrained.

Second, latch control requires the instantiation of additional control logic that dissipate power and may increase delay. The overhead of increasing the number of control signals and the complexity of the controller should be taken into account when evaluating the opportunity for operand isolation. Depending on the estimated control and latch overhead, larger or smaller values of R_i should be specified. A large value leads to conservative implementations, where operand isolation is implemented for unit i only when the units is idle for a large fraction of the execution time. Moreover, tightly timing constrained units should not be considered as candidates for operand isolation.

Example 5.8 *An abstract behavioral-level representation of the functionality of a design, known as data-flow graph [dmc94], is shown in Figure 5.17 (a). Three types of operation are performed, namely addition, subtraction and multiplication, represented by nodes in the graph. Edges represent functional dependencies among operations. Scheduling of the computations has been performed assuming that only one resource of each type is available. Seven clock cycles are needed to complete the computation. The graph is levelized according to the schedule. Each node is put on the level corresponding to the clock cycle of its execution.*



	1	2	3	4	5	6	7
Mul	1	1	1	1	1	1	0
Add	1	0	0	0	0	0	1
Sub	0	0	0	1	0	1	0

(b)

Figure 5.17. A scheduled data-flow graph and its utilization table

The usage table for the units is shown in Figure 5.17 (b). Notice that the multiplier has very high utilization, and it is likely to be the slowest resource (i.e., the one that defines the critical path). Operand isolation is not convenient for the multiplier, since it is idle only for $1/7$ *of the operation time, and inserting latches on its inputs would further slow down the cycle time of the circuit.*

On the contrary, both the adder and the subtracter have poor utilization, or, equivalently, high idleness $(5/7)$*. Moreover, they are not time-critical, because the clock cycle is set by the multiplier. Hence, operand isolation is very convenient for the two units, and would produce substantial power savings with limited area overhead (the input latches and the additional control logic) and no speed overhead. The data-path implementing the data-flow graph (with operand isolation) is shown in Figure 5.18. Blocks labeled* $M1$ *to* $M5$ *and* M *are multiplexers. Blocks labeled* $L1$ *to* $L4$ *are the operand isolation latches. The control signals for multiplexers and latches are not shown for the sake of readability.*

In some designs, operand isolation with input latches may not be allowed because of tight area constraints, speed constraints or testability issues (the insertion of latches complicates testing). Raghunathan and co-authors [ragh96a, ragh97] propose an alternative, low overhead approach for the exploitation of external idleness at the behavioral level. Their approach is bases on two trans-

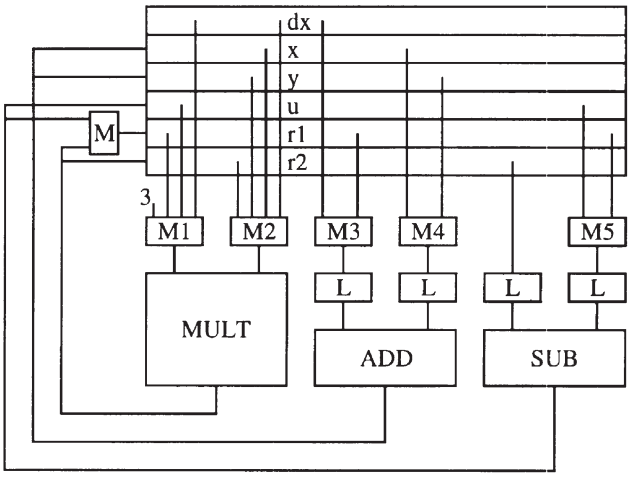


Figure 5.18. Data-path with operand isolation

formations, namely *controller re-specification* and *steering logic re-structuring.* The intuition is that spurious activity on the inputs of idle units can be eliminated at small cost by appropriately choosing the value of the control signals that drive the steering logic at the inputs of the unit itself.

The control signals for the steering logic are *don't care* conditions for the control state machine in states where the unit is not used. Such *don't cares* can be specified to increase the correlation between input values fed to the unit in active cycles and idle cycles. For instance, assume that unit i is active in cycle j, and performs computation $A + B$. In cycle $j + 1$ the unit is idle. The power-reduction technique based on control re-specification attempts to specify the values of the control outputs that drive the steering logic at the input of unit i so as to maintain values A and B. In this way, the unit will not dissipate any power because its input do not change during idle periods.

If control signals have spurious transitions during idle cycles, the benefits of controller re-specification can be much reduced. In order to reduce the amount of glitching on control signals, the techniques presented in the previous subsection can be exploited. Steering logic restructuring is the glitch-reduction technique that, when combined with control-respecification, leads to implementations with little spurious transitions.

As a final comment, notice that although the authors of [ragh96a, ragh97] claim that controller re-specification is a low-overhead technique, it is not clear if it always compares favorably with optimized implementations of operand isolation. Transparent latches are relatively small, low-power and high-speed

circuit primitives, and they can actually be merged with combinational logic blocks to further decrease the overhead. Moreover, operand isolation has been used in industrial practice and can be implemented even in design flows that do not rely on behavioral synthesis.

5.5.2 Memory segmentation

The remaining advanced power management techniques considered in this chapter apply transformations on the behavioral-level representation of a design to expose idleness which is not readily exploitable as in the case of operand isolation. These approaches are more aggressive in exploiting the information available during the behavioral synthesis process, and they actually modify the basic behavioral synthesis algorithms for memory allocation and scheduling.

Farrahi and co-authors [farr95] propose a memory partitioning (also called *segmentation*) scheme that reduces power by exposing idleness in memory accesses. At the behavioral level of abstraction, memory is a resource whose functionality is to store data when it is written and return it when read. Farrahi proposes a decomposition approach. Memory is not seen as a monolithic resource but as a collection of independent *memory segments*, with independent clock and refresh signals. Whenever a memory segment is idle, it can be put in *sleep* mode, i.e., its clock can be stopped, or its refresh signal can be shut down, thereby minimizing its power dissipation. Memory is idle when no useful information is stored into it. Notice that memory is *not* idle when it is not accessed, because it may store useful state information. In this case, we cannot put it in sleep mode to avoid the risk of losing data.

Memory accesses (writes and reads) performed during a computation can be performed. For each variable (called *memory element* in [farr95]), a *lifetime* can be defined as a time interval that starts when the variable is written and ends when the variable is last read. The lifetime of a variable v is completely defined by an ordered pair $L_v = (t_{init_v}, t_{fin_v})$. Given N_{seg} memory segments which can be independently put in sleep mode, the partitioning technique attempts to store in the same segment variables which have overlapping lifetimes. A segment is idle when no variable stored in it is alive. Hence clustering variables with overlapping lifetimes in the same module increases the idle time of the memory segment that stores them.

Finding the optimum assignment of variables to memory segments that maximize the total idle time of all segments is a hard problem. In [farr95] the authors propose a worst-case exponential time algorithm for solving the optimization problem. Fortunately, a practical extension of the optimum assignment problem has an efficient solution. The extensions try not only to maximize the total idle time, but also to bound the total number of transitions between sleep

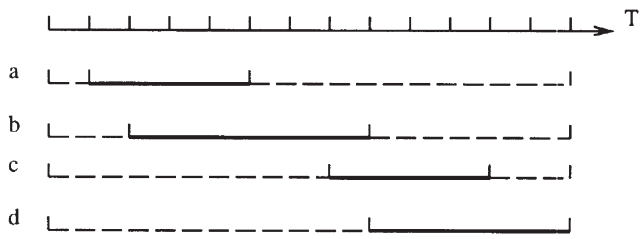


Figure 5.19. Lifetimes of four variables

and active mode of operation for all segments. The total number of transitions between sleep and active and vice versa is correlated to the power overhead in switching the control circuitry needed for putting the unit in sleep mode or waking it up.

Example 5.9 *Consider variables a, b, c, d. Their lifetimes are shown in Figure 5.19 in bold. If all variables are placed into the same memory segment (i.e., monolithic memory), the segment is idle only in the first period. Assume that two memory segments,* S_1 *and* S_2 *are available. If we store variables a and b in* S_1*, and variables c and d in* S_2*, the total number of idle periods is* $I = I(S_1) + I(S_2) = 6 + 7 = 13$*, and the total number of transitions between idle and sleep or vice versa is* 4*. If we store variable a and c in* S_1*, and variable b and d in* S_2*, the total number of idle periods is* $I = 5 + 2$*, while the total number of transitions is* 6*. Clearly the first partition would produce better power savings because it exposes more idleness and reduces the number of transitions between active and sleep.*

Memory segmentation is a binding problem that assumes the knowledge of scheduling information (i.e., to apply the technique, we need to know exactly when each variable is accessed). Partitioning exposes memory idleness that is hidden in the monolithic implementation. The main limitation of the technique is its high level of abstraction. In [farr95] many important practical issues are not discussed in detail. Instantiating memory segments increases the number of memory components, the area and the wiring overhead. The control circuitry that manages memory access increases in complexity and power dissipation. Power savings can be achieved only if the power saved by putting memory in sleep mode compensates for the overhead due to increased component count and control complexity. The authors do not provide any experimental evidence that this is the case in practical situations. Nevertheless the technique is an interesting aggressive power optimization modifies the hardware architecture to

enhance power management opportunities instead of simply exploiting idleness in a given architecture.

5.5.3 Scheduling

All techniques described so far share the assumption that scheduling information is fully known when power optimization is performed. However, it is possible to be even more aggressive and formulate scheduling algorithms that take power management into account. A power-conscious scheduling algorithm should assign operations to time steps trying to maximize the amount of *exposed idleness*. In other words, if there are conditions that imply idleness for a resource, they should be computed as soon as possible (i.e the computation of such conditions should be scheduled very early) so that the resource can be shut down. If this is not done, it might not be possible to shut down an idle resource because of timing constraints. *Exposing* idleness means modifing a design to enable the shut-down of idle resources without violationg timing constraints.

This useful observation is exploited by Monteiro and Devadas [mont96] who propose scheduling techniques which create power management opportunities. Consider a circuit that implements the conditional computation of $A + B$ or $C * D$ depending on the value of condition $X > Y$. If the condition is true, the sum is computed. Otherwise the product is computed. If we schedule the computation of the conditional in the same clock cycle as the two operations, the selection among the two results will be performed while they are being computed. Although one of the two computations is always unneeded (i.e., either the multiplier or the adder are externally idle in any clock cycle), it is possible to exploit idleness to reduce power. The conditional value becomes available too late to prevent the execution of one of the two mutually exclusive operations.

At the behavioral level, the information on mutually exclusive computations is readily available, and it is associated with conditionals in the specification. The power-conscious scheduling algorithms proposed by Monteiro and Devadas attempts to move the evaluation of conditionals as soon as possible in the schedule. Referring to the example, above, if $X > Y$ is computed at least one clock cycle before the execution of both multiplication and addition, it will be possible to expose idleness. Power can be reduced then by applying operand isolation on the adder and the multiplier resource, and controlling the latches with the result of the comparison. The power-efficient circuit with operand isolation is shown in Figure 5.20. This optimization is enabled by scheduling, and it may be completely lost if the circuit is optimized for power after scheduling has been computed.

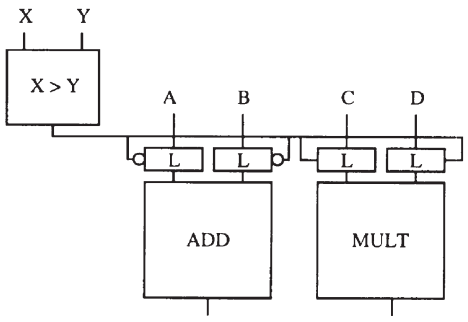


Figure 5.20. Low power implementation enabled by power-conscious scheduling

In [mont96], the authors show that power-conscious scheduling can reduce power by more than 30% for a few benchmark circuits. Unfortunately, the applicability and the impact of the technique can be impaired if the designs are tightly timing constrained. Referring to the above example, timing constraints could force the execution of the conditional on the same cycle as the two operations. The power-conscious schedule requires at least two clock cycles for computing the final results, while an aggressive schedule completes execution in just one clock cycle. Indeed, selecting the result of several alternative (mutually exclusive) computations after they have been computed is a well-known performance-enhancement technique known as *speculative execution.*

Even if the power-conscious scheduling technique proposed by Monteiro and Devadas may have limited impact on the critical computation, it may still very useful for non-critical computations. The slack of non-critical computations can be exploited by pre-computing conditionals whenever possible without increasing the latency of the circuit (which is set by the number of cycles required by the critical computation). The most important advantage of power-conscious scheduling is that it exposes idleness and makes it easy to exploit it to reduce power. Thus, it not a simple "power recovery" technique, but it can be seen as a high-level transformation for low power.

5.5.4 Summary

Behavioral-level power management techniques have just begun to emerge. Several promising directions of development are under active investigation. Behavioral specifications have two basic properties that should be exploited, namely locality and mutual exclusiveness of computation. Locality of computation is a well-known property of programs. A high fraction of the execution time is spent in a relatively small fraction of the program. Behavioral HDL hardware

descriptions are indeed programs which enjoy similar properties. From the point of view of power management, locality is a very useful property, since it implies that only a relatively small subset of the complete functionality is actually used for a large fraction of the operation time. Thus, the little-utilized features can be shut-down and activated only when needed.

Mutual exclusiveness of sections of a computation is caused by the presence of conditionals in the behavioral description. The scheduling technique described above is an example of how mutual exclusiveness can be exploited for power management. However, other approaches are possible. For instance, in [beni96b] a techique is presented to reduce the power dissipation of controllers of sytems specified at the behavioral level. Such technique leverages the mutual exclusiveness of computations caused by conditionals.

More work is needed to reach a complete understanding of the opportunities available at the behavioral level. However we firmly believe that behavioral-level power management techniques can have a sizable impact on reducing the power dissipation of digital designs.

5.6 CONCLUSION

In this chapter we have reviewed several power management techniques targeting large digital circuits described at the logic, register-transfer and behavioral level of abstraction. All techniques are based on the fundamental ideas described in Chapter 3, namely exploitation of internal and external idleness. While some of the approaches focus on detecting and exploiting idleness in a given specification, others transform a specification to expose idleness and obtain further power savings.

Some of the advanced techniques presented in this chapter are not yet mature enough to be integrated in standard industrial design methodologies, although several commercial EDA tool vendors and system design companies are developing industrial-strength tools for power management. The main obstacle to overcome for logic-level techniques is the lack of an effective partitioning strategy that can direct the computational effort on promising sections of large logic-level netlist, while avoiding blocks where idleness computation is prohibitively expensive. On the other hand, the main challenge for approaches at the RT and behavioral level is to increase their generality and prove themselves on large industrial designs.

Although the path to maturity can be long, we believe that the advanced techniques described in this chapter hold good promise of becoming effective power optimization approaches in the domain of VLSI digital design. Unfortunately, all approaches discussed so far are not powerful and general enough to be applied at an even higher level of abstraction, the *system level*, where atomic

components are digital VLSI chips and electro-mechanical devices, disks, displays, etc. We address system-level power management in the next chapter.

6 POWER MANAGEMENT FOR SYSTEMS

To do nothing at all is the most difficult thing in the world, the most difficult and the most intellectual.

—Oscar Wilde

6.1 INTRODUCTION

In all previous chapters we focused on power management schemes for VLSI digital circuits. We also assumed mainstream static CMOS technology and synchronous (i.e., clocked) implementation style. The assumption on technology and on clocking style are not fundamental, and many techniques previously presented can still be applied, with minor modifications, to circuits implemented with different technologies and clocking styles. Nevertheless, all power management techniques presented so far focus on the reduction of chip-level power consumption.

The vast majority of power-critical electronic products are far more complex than a single chip. Portable devices such as cellular telephones and laptop computers contain tens or even hundreds of components. In most electronic products, the digital components are responsible for only a fraction of the power consumed. Analog, electro-mechanical and optical components are a significant fraction of the total, and are often responsible for the largest contributions to the power budget. For example, the power breakdown for a well-known laptop computer [uda96] shows that, on average 36% of the total power is consumed by the display, 18% by the hard drive, 18% by the wireless LAN interface, 7% by non-critical components (keyboard, mouse etc.), and only 21% by digital VLSI circuitry (mainly memory and CPU). Reducing the power in the digital logic portion of this laptop by *10X* would reduce the overall power consumption by less than 19%.

Laptop computers are not an isolated case. Almost all electronic appliances are complex and heterogeneous systems containing a wide variety of devices that do not fall within the scope of the computer-aided power optimization techniques presented in the previous chapters. Nevertheless, designers have reacted promptly to the new challenges posed by low-cost, low-power portable appliances. Battery lifetime (or time between recharges) is steadily increasing and the physical dimensions of portable devices are progressively shrinking. These surprising results are achieved thanks to a well-balanced mix of technological innovation, architectural design and optimization.

One of the most successful techniques employed by designers at the system level is dynamic power management. Building a complex system that supports dynamic power management is a difficult and error-prone process. Long trial-and-error iterations cannot be tolerated when fast time to market is the main factor deciding the success of a product. Unfortunately, system-level computer-aided design environments are still in their infancy, and EDA vendors are lagging far behind the needs of this segment of the electronic industry.

To compensate for this lack of support, several system developers and vendors [onno97a, ACPI96] are aggressively pursuing a long-term, wide-scope strat-

egy to greatly simplify the task of designing large and complex power-managed systems. The strategy is based on a standardization initiative known as the *advanced configuration and power interface* (ACPI). The structure and characteristics of ACPI will be described in Section 6.3. Although the initiative targets personal computers, it contains useful guidelines for a more general class of systems. The characterizing feature of ACPI is that it recognizes dynamic power management as the key to reducing overall system power consumption, and it focuses on making the implementation of dynamic power management schemes in personal computers as straightforward as possible.

The ACPI specification forms the foundation of the *OnNow initiative* launched by the Microsoft Corporation. The *OnNow* initiative targets the migration of power management algorithms and policies into the computer's operating system (OS). The *OnNow* initiative is described in Section 6.2. Although *OnNow* requires ACPI as the interface between OS-directed power management procedures and the hardware platform, ACPI is an OS-independent open specification. Any OS developer who is interested in supporting power-manageable hardware may develop an ACPI back-end to its operating system.

ACPI and *OnNow* only provide a framework for designers to implement power management strategies. The choice of power management policy is left to the engineer. Moreover, the system architecture assumed in ACPI (and *OnNow*) is processor-centric and tightly bound to the abstract model of a personal computer. In this chapter we go beyond ACPI and *OnNow* to formulate a model for a generic power-managed system which is more general (and less detailed) than the model proposed by ACPI. We define the key problem of *policy optimization*, which is the central issue in any power management scheme. Roughly speaking a *policy* is an algorithm that decides how and when to activate and shut down system resources to minimize power, depending on usage patterns and performance constraints. Finding the power management policy that minimizes power without compromising performance beyond acceptable levels is a complex optimization problem.

The chapter is organized as follows. In Section 6.2 we describe the *OnNow* initiative for OS-directed power management. In Section 6.3 we give an overview of the ACPI specification. In Section 6.4 we propose a system-level modeling approach which is more general and less detailed than ACPI. The model is suitable for describing very large and complex systems at a high level, where functionality is almost completely abstracted away and power consumption is modeled in a simplified, yet effective fashion. Finally, in Section 6.4 we describe some empirical results on policy optimization and outline directions for future research in this area.

6.2 OPERATING SYSTEM-BASED POWER MANAGEMENT

Personal computers (PCs) are becoming an integral part of daily life, as portable appliances such as wristwatches and cellular phones have become over the last few years. The *flexibility* of the PC platform is pushing the entire electronics market toward the new era of *convergence*. Communication, data processing and entertainment will be supported by the same platform, enhanced by the world-wide connectivity provided by the Internet. Although industry leaders have contrasting visions on the architecture of the *integrated electronic appliance* of the future, there is general agreement on a few aspects: it will be portable, it will be multi-functional, and it will be connected to the Internet.

Personal computer manufacturers are strategically positioned to exploit the huge business opportunities opened by the convergence era. PCs are flexible, general-purpose machines that can deliver an impressive amount of computational power at low cost. Two main obstacles to the success of the PC platform as the basis for the new generation of integrated electronic appliances are the relatively long delay between the user switch-on signal and the "fully-operational" state and the high power dissipation of current portable PCs, which still cannot guarantee battery duration longer than the average "awake time" of the human user. To address both problems, the software industry leader, Microsoft, has launched a comprehensive initiative, called *OnNow* [onno97a], to provide efficient system-wide power control on PC platforms.

As with other electronic appliances (portable audio and video equipment, cameras, pocket calculators, etc.), the personal computer should be instantly available to respond to the user's requests, it should appear off when not in use, and dissipate a minimal amount of power. In order to achieve this goal, the PC hardware and software must be able to move very rapidly (from the user's point of view) from an *apparently off* state to a *fully-functional* ON state.

The *OnNow* initiative has the goal of bringing the PC platform (i.e., hardware and software) to new levels of usability, robustness and power efficiency. This goal can be achieved by promoting tight integration and cooperation between hardware, the operating system and the application. An *OnNow*-compliant PC platform must conform to the following basic requirements [onno97a]:

- The PC is ready for use as soon as the user turns it on. Turn-on delays should be imperceptible to the user.
- The PC appears as off when not in use, but it must be capable of responding to wake-up events that may be triggered not only by the user, but also by input devices such as a telephone or by software-scheduled wakeup calls.
- Software tracks hardware status changes and adjusts accordingly. Operating system and applications cooperate to effectively perform power management

in accordance to the user's expectations and requirements. Applications should not be designed assuming that the hardware is always active and fully functional. Whenever an application is not performing useful work, it should inform the operating system that it is idle and release control of unused/unneeded resources.

- All hardware devices participate in the power management scheme, whether originally installed in the baseline PC configuration or added later according to the user needs. Any device connected to the PC should be capable of responding to power management commands in a consistent and effective fashion.

Current PC platforms do not meet the requirements of *OnNow*. PCs must boot the operating system when turned on, which is a lengthy process that can take several minutes. Once fully operational, the PC must be left ON to respond to asynchronous events coming from its environment. Unfortunately, when a personal computer is ON, it is often noisy and power-consuming. Environmental regulations and user discomfort often impose a "default off" policy, where the PC is kept electrically off (i.e., disconnected from the power supply) whenever not in use. The failure of current PCs to serve as standard appliances is caused by several shortcomings in hardware and software architecture. First, there is lack of cooperation among system components. In particular, the *basic operating system* (BIOS), the operating system and the applications attempt to autonomously control the hardware. This miscoordination causes inexplicable erratic behavior such as displays blanking or turning on unexpectedly and disks spinning down and spinning up when not needed. Second, although current PCs are expandable, newly added components are not guaranteed to participate in the system-level power management effort. Third, even for devices where power management is in place, opportunities for saving power are lost because devices are controlled at too low a level, ignoring high-level information on what the user actually needs during system operation. Fourth, applications assume that the computer is always fully on. This assumption often causes excessive power consumption. For example, polling cycles, when an application is waiting for a busy resource, but it continuously tests its availability, are very power inefficient. Polling cycles prevent hardware shutdown by creating useless activity in the system.

OnNow addresses these shortcomings by taking advantage of the observation that in a PC platform the *operating system is the most suitable component for the control of the power management strategy of the entire system.* All modern computers are multi-tasking machines. Any single application does not have sufficient knowledge of the status of the entire system to make autonomous decisions on when to enter a power-down mode. On the other hand, resources are

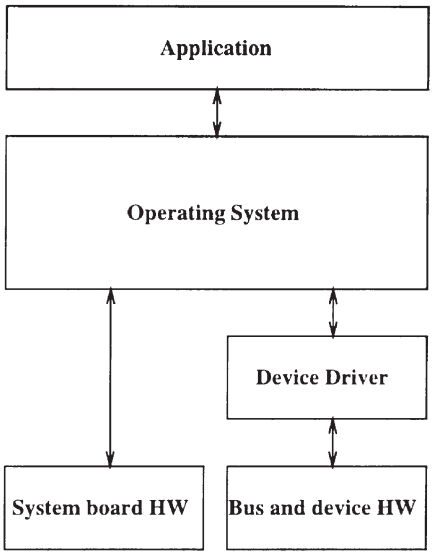


Figure 6.1. High-level architecture of a generic PC platform

shared among applications, and can be exploited according to widely varying usage patterns, depending on the operating conditions of the entire systems. Hence, power management decisions made on a resource-by-resource basis are often highly sub-optimal as well.

In a multi-tasking, resource-shared environment, the complete knowledge of global system status resides in the OS, the system component that takes care of dispatching concurrent applications and managing shared resources. For this reason, in the *OnNow* initiative the operating system is chosen as the global power manager for the PC platform. The high-level architecture of a PC platform is shown in Figure 6.1. It illustrates the central role of the operating system in bridging the gap between hardware resources and software applications.

Applications communicate with the operating system through a standardized *application programming interface* (API). The interface of the operating system to hardware resources is through *device drivers*. A device driver is a software component residing in the OS that hides the details of the hardware implementation of any particular resource, and offers a standard interface between hardware services and system requests. Devices on the core system board are managed directly by the OS, while modularity is supported by external drivers that can be registered with the OS kernel whenever a new external resource is added to the system. The schematic representation of Figure 6.1

State	Description
Working	The system is ON and fully usable.
Sleeping	The system appears to be off. Power consumption is reduced, the system returns to the working state in an amount of time inversely proportional to the power consumption.
Soft off	The system appears to be off. Power consumption is very low. A full OS reboot is needed to restore the working state. During reboot the working context is restored from non-volatile memory.
Mechanical off	System off, with no power consumption. It needs to be reconnected to the power supply to resume working after a full reboot.

Table 6.1. Global system power states and their meaning

shows the distinction between devices on the system board and external devices connected on a communication bus.

6.2.1 Operating system support

To provide full support to the *OnNow* initiative, and enable OS-directed dynamic power management, all system components must be modified. The OS must do the following [onno97a].

- Support *system power states* which represent states of operation with radically different level of functionality and power dissipation. There are four main system power states, as shown in Table 6.1.
- Define and manage the *global power policy.* The power policy is the procedure that decides when the machine should transition from one power state to another. Decisions are made based on user input, application activity and hardware signals from device drivers. Moreover, the policy is influenced by performance constraints that may be set either by the user or by applications.
- Integrate power-awareness into the OS. Not only should the OS decide when to transition into low-power states, but also it should manage system ac-

tivities during working periods in a power-efficient fashion. Even when the system is in the working state, power can be saved by turning off unused resources. The OS should schedule system activities so as to satisfy the performance constraints while maximizing the opportunities for shutting down idle resources.

- Maintain robustness. Power state transitions should never compromise functionality. Particular care must be taken in preserving the context of operation when transitioning from working to any low-power state. Context preservation should be safe and efficient to minimize both the risk of state loss and the time required to return to the fully working state.

6.2.2 Device driver support

OnNow assumes a specific device driver structure which is tightly coupled with Microsoft's OS architecture [onno97b]. Device drivers are low-level components of the OS that encapsulate the hardware resource in the system.

As explained in the previous section, the OS is responsible for managing the global power policy. Additionally, the OS contains separate *policy owners* for each class of devices. A policy owner is a software module in the OS which is responsible for managing the usage of a group of resources with similar characteristics, called a *class*. The required power management features for several device classes are defined by the *OnNow* initiative in a set of specifications. The following classes are defined:

- Audio device. This specification describes the power management features of the PC audio hardware components, including audio input, output, duplex (i.e., simultaneous playing and recording) and advanced devices such as speech recognition boards.

- Communication device class. The specification is addressed primarily to modems and similar devices such as ISDN terminal adapters and computer-connected telephone devices.

- Display device. The specification defines the power management features of video controllers and drivers. It applies to CRT monitors and LCD panels.

- Input device. The specification describes the power management features of keyboard controllers, mice and other similar input devices.

- PC Card device. The specification defines the behavior of PC Card controller devices as it relates to power management.

- Storage device. The specification focuses on the required power management features of hard disk drives, CD-ROMs, removable storage devices, floppy disk drives.

- Network device. This specification applies specifically to Ethernet and Token Ring adapters and describes required power management features.

- Default device class. The specification describes the minimum set of power management features to be included in any device class. New types of devices that do not fall into any of the previously described classes can be considered as belonging to the default device class.

- Bus device classes (PCI, IEEE1394, USB). Busses are special devices that link several hardware resources (belonging to any of the classes described above) to the PC board. Thus, bus drivers know how to perform power management on devices residing on their bus. The power state of the bus resource itself depends on the state of all the connected devices. In particular, the bus resource can be shut down only if all connected resources have already been shut down. Power management specifications of bus devices are substantially more complex than those of normal devices, because they not only specify the power management features of the bus circuitry, but they also describe how to communicate power-related information from the OS to the resources on the bus and vice versa.

Each device class has a policy owner. The owner communicates with its resources through messages, called *input/output request packets* (IRPs). The communication is bidirectional and goes through the OS. A policy owner can dispatch four basic types of IRPs to its devices:

- Request for power-related capability report. This information is needed upon system boot to tune the policy to the features of the resource.

- Query, set or get the power state of a device. During system operation, the policy owner controls the activity of the devices and its mode of operation.

- Enable a device to be a source of wakeup events when the system is in sleep state.

- Get the battery status of the device if device is self-powered.

Device policy owners contain specific knowledge about the usage patterns and the specific requirements for a resource, and have some autonomy in deciding how to manage its state of operation to minimize its power consumption without compromising performance. All device policy owners are controlled by the

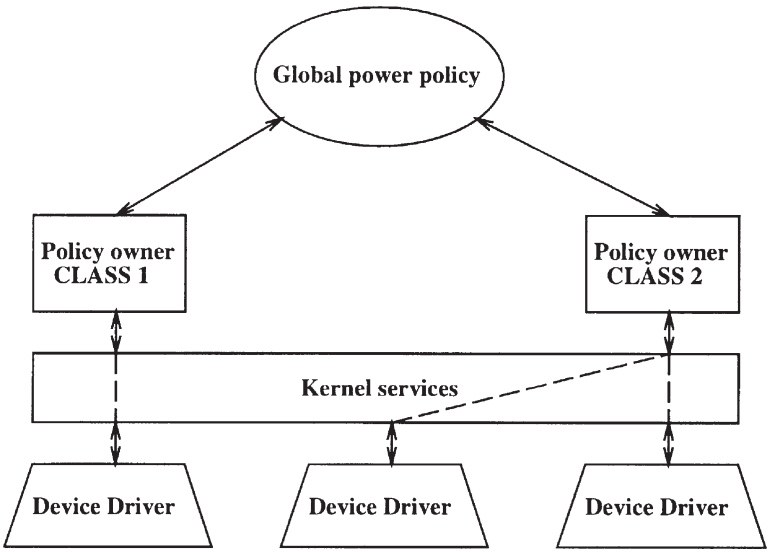


Figure 6.2. Interaction between global power policy and class policy owners

global power policy that makes high-priority decisions on the state of the entire system. The interaction between device policy owners and the global power policy is depicted in Figure 6.2.

In the figure, two device policy owners are shown. The *policy owner CLASS 2* controls two devices (through their device drivers), while *policy owner CLASS 1* controls one device. Both owners can make autonomous decisions on the power management policy of the resources they control, but they are ultimately controlled by the global power policy that can override their decisions. Notice that the flow is bidirectional. Policy owners send commands to the drivers and receive acknowledgments and status information. The same holds for the communication between policy owners and the global power policy.

Concluding our description of device drivers and device policy owners, notice that we did not discuss how devices are power managed, or which power states should be supported. The ACPI standard, that will be described in Section 6.3, specifies that level of detail.

6.2.3 Application support

Applications should be designed in a power-efficient fashion [appl97]. First of all, applications should be cognizant of the fact that modern operating systems are multi-threaded and preemptive. In simple terms this means that several

```
do {
  if (TestMessageReady()) {
    done = ProcessMessage();
  }
  else if (BckgProcReq()) {
    DoBckgProc();
  }
} while (!done);
```

Figure 6.3. Power-inefficient application

programs are run in time-shared fashion, each taking a fraction of the CPU time. In addition, even a single application may have multiple threads of execution.

Let us consider a single-threaded application as shown in Figure 6.3. The pseudo-code represents a correct but power-inefficient implementation of a single-threaded application. First, function `TestMessageReady` tests for the arrival of a message. If the message has arrived (for instance, if a network packet has been received), it is processed. Based on the content of the message, the flag `done` is a set/reset to decide if the computations should proceed or terminate. If no message has been received, the application checks if there is some background work waiting (function `BckgProcReq`). If this is the case, the background task is carried out (function `DoBckgProc`).

Although the code of Figure 6.3 is correct, it is highly power inefficient. From the OS point of view, the application running the code is always active, and it prevents the shutdown of the system even if no background work is performed and no messages arrive. A more power-efficient implementation of the same functionality is shown in Figure 6.4.

The only difference between the two implementations is that the second blocks the process in a wait state (with function `WaitMessage`) if no new message is received and no background processing is required. Function `WaitMessage` makes the process inactive until the arrival of a new message. When a new message arrives, execution resumes and the message is detected by `TestMessageReady` and processed. From the OS viewpoint, if the process is waiting on `WaitMessage`, it is marked as idle. Thus, if no other processes are active, the power management policy may decide to move to a sleep state. Notice that this simple application does not interact directly with the power management features of the OS, but it may have a big impact on the power consumption of the entire system if it prevents transitions to a sleep state.

```
do {
  if (TestMessageReady()) {
    done = ProcessMessage();
  }
  else if (BckgProcReq()) {
    DoBckgProc();
  }
  else {
    WaitMessage();
  }
} while (!done);
```

Figure 6.4. Power-efficient application

The *OnNow* initiative encourages software developers to write applications that directly interact with the power management policy of the OS. If the hardware is power managed at a too low level (i.e., by device drivers), the application-unaware shutdown of system resources may worsen the quality of the user interaction with the PC. Examples of this problem are when the screen blanks during a presentation, or when the disk unexpectedly spins up and spins down (causing annoying delays).

Needless to say, application interaction with power management should be implemented with particular care. The main pitfall for application developers is to take an application-centric view on power management, and think that application-level idleness implies system-level idleness. This is almost never true because of the multi-threading features of modern operating systems. A safer interaction paradigm between OS and applications is based on a simple exchange of information. The application should inform the OS that it does not need resources or that it is entering an idle state, and let the power management policy of the OS decide when to enter the system sleep state. On the other hand, applications may need to access information stored in the OS on power management features and policies.

The system architecture proposed by *OnNow* provides an application programming interface that helps application developers ensure that their applications actively contribute to power efficiency and maintain high-quality user interfaces [onno97c]. The *OnNow* API provides two key features:

- An interface that allows applications to request information to the OS about power events and status. When the OS is preparing to transition to a sleep state, a message is broadcast to all applications which can react accordingly.

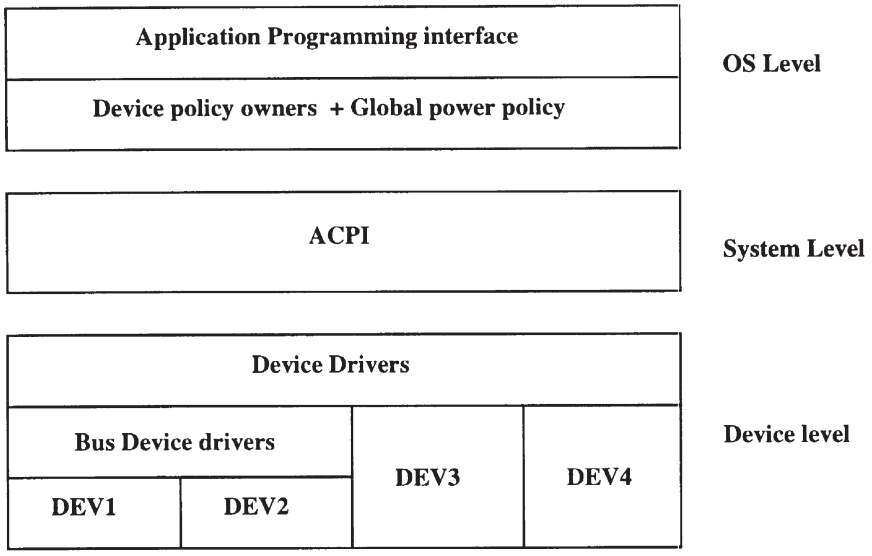


Figure 6.5. Summary of the OnNow architecutre

Several actions may be taken: for example the application may decide to save some context information, or may inform the OS that the system should not transition to the sleep state.

- A mechanism for applications to inform the OS about specific requirements that should influence the power management policy. The basic functionalities required for this type of interaction are: i) a command that can be issued to prevent the OS from putting the system or some resource to sleep when they appear idle, and ii) a command for forcing the state of the system in low-power sleep mode.

Thanks to the communication between OS and applications, the global power policy can be based on both user and application activity in the system. Moreover, applications should be allowed to control the wakeup of the system from a sleep state. The *OnNow* API provides system wakeup functions based on timers. Before the system enters the sleep mode, the application may schedule a wakeup time in the future, or inform the OS that it should be sensitive to events generated by some hardware device.

6.2.4 The *OnNow* architecture

OnNow is a complex and multifaceted initiative. In the previous subsections we outlined its main features. We now summarize the architecture of an *OnNow* personal computer. A pictorial summary of the *OnNow* components is shown in Figure 6.5. At the boundary between the application world and the PC platform, *OnNow* provides a standard API for power-conscious applications. Within the OS, device policy owners are instantiated as well as the global system power policy. ACPI provides the standard OS-independent layer between power management strategies and system hardware. Finally, low-level device drivers manage the most intimate details of hardware control. Three types of device drivers are defined. *Motherboard device drivers* (DEV3 and DEV4 in Figure 6.5) encapsulate hardware components that interface directly with the ACPI layer. Such components are typically located on the PC's motherboard and provide the core system functionality. *Bus drivers* control standard bus resources where additional devices can be connected (and disconnected) to configure the PC according to the user's needs. Finally, *External device drivers* encapsulate devices that are connected to the busses.

The scope of the *OnNow* architecture is very wide. It provides guidelines for the design of power-managed computers ranging from a high-performance desktop to a notebook. In this section we focused on the description of the architecture and its components, without entering into the details of how power can be reduced and controlled for the complex devices that build a PC platform. In the next section we describe the ACPI standard, and we provide more details on the hardware support that is a prerequisite for the implementation of OS-directed power management.

6.3 ADVANCED CONFIGURATION AND POWER MANAGEMENT

The advanced configuration and power management interface is the basic building block for supporting *OS-directed power management* (OSPM). ACPI is an OS-independent, general specification that applies to desktop, mobile and home computers as well as high-performance servers. ACPI is jointly developed and supported by Intel, Microsoft and Toshiba [ACPI96]. The specification has emerged as an evolution of previous initiatives that attempted to integrate power management features in the low-level routines that directly interact with hardware devices (firmware and BIOS). It also provides some form of *backward compatibility* since it allows ACPI-compliant hardware resources to coexist with legacy non-ACPI-compliant hardware.

ACPI is the key element for implementing OSPM strategies such as *OnNow*. It is an open standard that is made available for adoption by hardware vendors and operating system developers. The main goals of ACPI are to:

- Enable all PCs to implement motherboard dynamic configuration and power management. PCs include servers, mobile, home, desktop and workstation machines. Hardware developers targeting completely different market segment can rely on full OS support for power management. The diffusion of ACPI-compliant hardware with OS support will push application developers to exploit the power management features offered by the API of the OS.

- Enhance power management features and the robustness of power managed systems. Policies that are too complex to be implemented at the BIOS level can easily be supported at the OS level. Inexpensive hardware support (i.e., ACPI-compliant devices) will be sufficient to implement elaborate power management policies because the complexity of the power management policies has migrated to the OS software layer. Debugging and customization is made much easier and centralized management improves reliability and coordination. OS support is essential for gathering power management information from application, users and hardware. This information can be exploited to formulate more effective policies.

- Accelerate implementation of power-managed computers, reduce costs and time to market. The availability of an open standard specification reduces the amount of non-recurring costs of developing multiple interfaces.

The ACPI specification defines the interfaces between OS software and hardware. The software and hardware components relevant to ACPI are shown in Figure 6.6. Applications interact with the OS kernel through APIs. A module of the OS implements the power management policies, as discussed in the previous section. The power management module interacts with the hardware through kernel services (system calls). The kernel interacts with the hardware using device drivers. The front-end of the ACPI interface is the *ACPI driver*. The driver is OS-specific, it maps kernel requests to ACPI commands, and ACPI responses/messages to kernel signals/interrupts. Notice that the kernel may also interact with non-ACPI-compliant hardware through other device drivers.

At the bottom of Figure 6.6 the hardware platform is shown. Although it is represented as a monolithic block, it is useful to distinguish three types of hardware components. First, hardware resources (or *devices*) are the system components that provide some kind of specialized functionality (e.g., video controllers, modems, bus controllers). Second, the *CPU* can be seen as a specialized resource that need to be active for the OS (and th ACPI interface layer) to run. Finally, the *chipset* (also called core logic) is the motherboard logic that controls the most basic hardware functionalities (such as real-time clocks, interrupt signals, processor busses) and interfaces the CPU with all other devices.

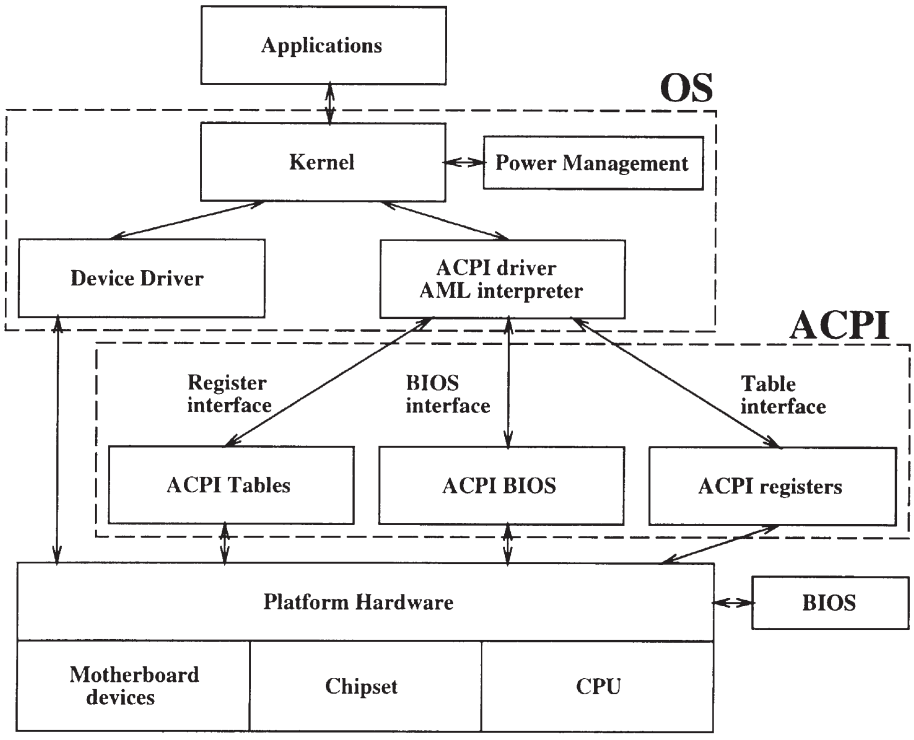


Figure 6.6. ACPI interface and PC platform

Although the CPU runs the OS, no system activity could be performed without the chipset. From the power management standpoint, the chipset, or a critical part of it, should always be active, because the system relies on it to exit from sleep states. Hence, ACPI does not define power management strategies for the chipset itself. The power consumption of the chipset should be managed at the firmware level, transparently to the OS.

It is important to notice that ACPI does not specify *how* to implement hardware devices or the power management support in the operating system. No constraints are imposed on implementation styles for hardware and on power management policies. Implementation of ACPI-compliant hardware can leverage any technology or architectural optimizations as long as the power-managed device is controllable by the standard interface specified by ACPI.

The dashed box represents the organization of ACPI and consists of three main components. *ACPI tables* describe the interface to the hardware. *ACPI BIOS* refers to the portion of the firmware that is compatible with ACPI. This is typically the code that boots the machine and implements interfaces for global power management commands such as sleep, wakeup and some restart operations. Part of the BIOS may not be compatible with ACPI. For this reason, the BIOS is represented as two separate blocks in Figure 6.6, one within the scope of the ACPI specification and the other outside it, close to the hardware. Finally, *ACPI registers* are the constrained part of the hardware interface. ACPI-compliant resources must provide ACPI registers. The ACPI driver controls and monitors the state of resources by writing and reading the ACPI registers.

The ACPI tables contain procedures encoded in a language called *ACPI machine language* (AML). They describe arbitrary sequences of low-level commands for controlling the operation of hardware devices. Every high-level power control (or power monitoring) command is associated with an AML procedure. Whenever a high-level power management command is issued to a resource by the OS, the ACPI drivers calls its AML interpreter module that executes the procedure corresponding to the command. Execution of the procedure involves direct interaction with the hardware. The AML language defines the instruction set of a virtual machine. All ACPI-compliant hardware resources are instances of the AML virtual machine. When AML code is executed, ACPI registers are written and read and resources are thereby controlled and monitored.

6.3.1 ACPI state definitions

As discussed in Section 6.2, the computer viewed as a monolithic system can be in several *global power states*. In the ACPI specification, five global system states are defined. State $G3$ is *mechanical off*, $G2$, also called $S5$ is *soft off*, $G1$ is *sleeping*, $G0$ is *working* and *legacy* is *the state entered when the system*

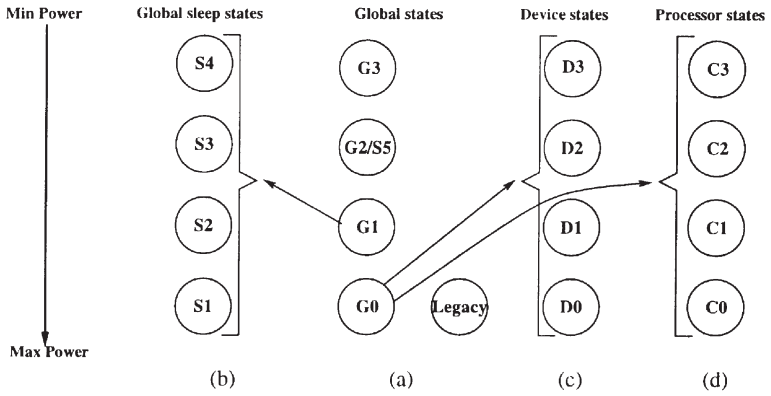


Figure 6.7. State definitions for ACPI

does not comply to ACPI (refer to Table 6.1 for a description of the distinctive characteristics of the first four global power states). The global states are shown in Figure 6.7 (a). They are ordered by increasing power dissipation. The state with lowest power dissipation is at the top.

The ACPI specification refines the classification of global system states by defining four sleeping states within state $G1$, as shown in Figure 6.7 (b):

- State $S1$. It is a sleeping state with low wake-up latency. No system context is lost in the CPU or the chipset.
- State $S2$. It is a low wake-up latency sleeping state. This state is similar to the S1 sleeping state with the exception that the CPU and system cache context is lost. The OS is responsible for saving the content of the caches and the CPU context prior to entering S2. Control starts with the CPU in reset state after the wake-up event.
- Sleeping state $S3$ is a low wake-up latency state where all system context is lost except system memory. CPU, cache, and chipset context are lost in this state. Hardware maintains memory context. Control starts from the processor's reset state after wake-up.
- State S4 is the sleeping state with the lowest power and longest wake-up latency. To reduce power to a minimum, it is assumed that the hardware platform has powered off all devices. Platform context is maintained.

Additionally, the ACPI specification defines states for system components. There are two types of system components, *devices* and *processor*, for which power states are specified. Devices are abstract representations of the hardware

resources in the system. The processor is the central processing unit that controls the entire PC platform. Special devices are *embedded controllers*, that function as resources for the main CPU.

ACPI defines a specialized interface for embedded controllers. Although from a power management point of view embedded controllers are treated as normal resources, they have specialized drivers because they may be used to monitor power-related system characteristics, perform low-level complex calculations, and they may provide data that is required to implement power management policies. For example, an embedded controller can be used to control board temperature sensors and provide valuable data for thermal management.

Four states are defined for devices, as shown in Figure 6.7 (d). In contrast with the global power states, device power states are not visible to the user. For instance, some devices can be in an inactive state, but the system appears to be in working state. Furthermore, state transitions for different devices can be controlled by completely different power management policies. The device states are:

- D0. This state has the highest level of power consumption. The device is fully active and it is expected to maintain all relevant context continuously. State D0 is obviously required for ACPI compliance.

- D1. Again the meaning of this device state is defined by each class of device. Its definition is not required for ACPI compliance. A device in D1 is expected to dissipate more power than a device in D2, but it preserves more context (hence, wake-up is faster).

- D2. The detailed characteristics of state $D2$ are defined by each class of device. D2 may not be defined (i.e., it is not required for ACPI compliance). In general, D2 is expected to save more power and preserve less device context than D1 or D0. If bus resources are in state D2, some of the devices connected to the bus may lose some context.

- D3 (*Off*). Power has been fully removed from the device. The device context is lost when this state is entered. Thus, the OS will re-initialize the device when powering it back on. Since device context and power supply are lost, devices in this state cannot receive any command. Devices in state D3 have the longest restore times. The D3 state is defined for all classes of devices.

Example 6.1 *An example of a device with multiple power states is a disk driver. We can define a minimum set of two states: off (D3) and fully on (D0). In state D3, the disk is spun down and the disk controller interface is turned off. In state D0 the disk is spinning and data can be written or read to it. ACPI supports the definition of a more complex device model, where two*

additional states (D2 and D1) are defined. In D2 the disk is still spinning, but the disk controller is powered down. In D1 the disk is spinning, the disk controller is powered up, but the clock signal of the controller is gated. States D1 and D2 provide additional freedom in the choice of power management policies, because they have lower power than D0 and faster transition time than D3.

Finally, processor power states (Figure 6.7) are processor power consumption states within the global working state G0. Notice the intrinsic asymmetry of the ACPI model. The central role of the CPU is recognized, and the processor is not treated as a simple resource. Hence, processor power states, shown in Figure 6.7 (d), are distinct from device power states.

- State C0. In this state the processor is fully operational and executes instructions.
- State C1. The processor is not executing instructions. The latency for exiting from C1 and entering C0 is required to be low enough that the OS software does not consider the effect of latency when deciding whether to use it.
- State C2. It offers improved power savings with respect to C1. The worst-case hardware latency for resuming execution is defined and it is provided to the OS. The OS can decide to put the CPU in state C2 when the maximum latency for resuming execution is tolerable. From the application or the OS point of view no context is lost.
- State C3. Offers improved power savings with respect to C2. The worst-case hardware latency for resuming execution is larger than that of C2, and it is provided to the OS as well. While in C3, the CPU caches maintain their content but ignore external snoop activity. The OS is responsible for ensuring that the caches maintain coherency.

Figure 6.8 summarizes the power states and state transitions for an ACPI platform. Sub-states within a global power state are shown beside their global state. In general use, the computer alternates between working (G0) and sleeping states (G1 and sub-states). In state G0 the computer actually performs some work. User application threads are active and running. Individual devices that are not used can be in a low power (D1, D2, D3) state. Any device in a low power state can be turned on without leaving the G0 state. From the user's viewpoint, the entire machine is functional. Various working sub-states differ in speed of computation, power consumed and heat produced. The choice of a particular working sub-state is dictated by the need of minimizing power (heat, noise) under performance constraints.

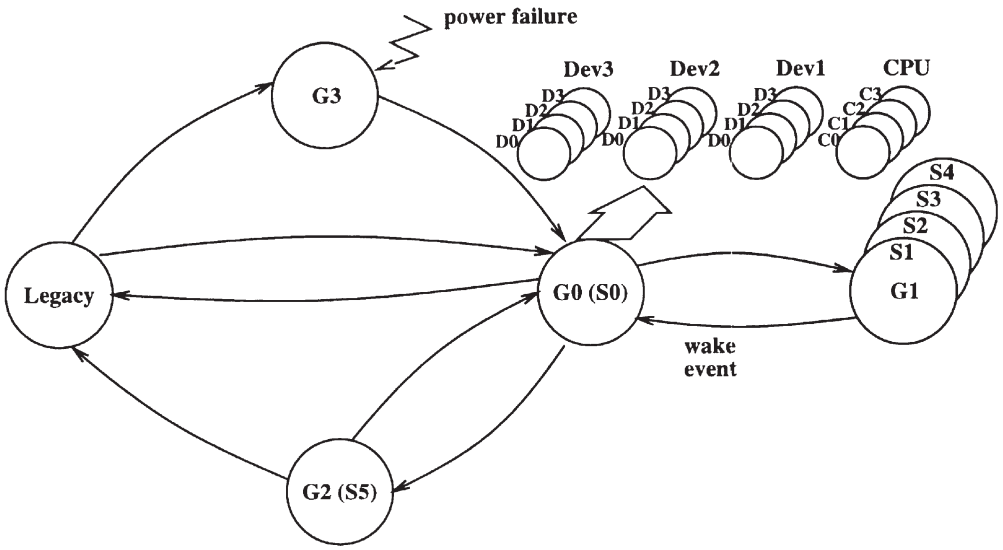


Figure 6.8. Global and power states and substates

When the *entire system* is idle or the user has pressed the power-off button, the OS will drive the computer into one of the sleeping states (left side of Figure 6.8). From the user's viewpoint, no computation occurs. The sleeping sub-states differ in which *wake* events can force a transition into a working state, and how long the transition should take. If the only wake-up event of interest is the activation of the user turn-on button and a latency of a few minutes can be tolerated, the OS could save the entire system context into non-volatile storage and transition the hardware into a soft-off state (G2, at the bottom of Figure 6.8). In this state, power dissipation is almost null and context is retained (in non-volatile memory) for an arbitrary period of time.

Mechanical off is entered in the case of power failure or mechanical disconnection of power supply. Complete OS boot is required to exit the mechanical off state. Finally, the *legacy* state is entered in case the hardware does not support OSPM.

6.3.2 Device power management

In the previous subsection we described the device power states (D0-D3) that model the various modes of operation for power managed devices spanning the trade-off between power consumption and latency required to resume normal operation. We now summarize the mechanisms provided by the ACPI specifica-

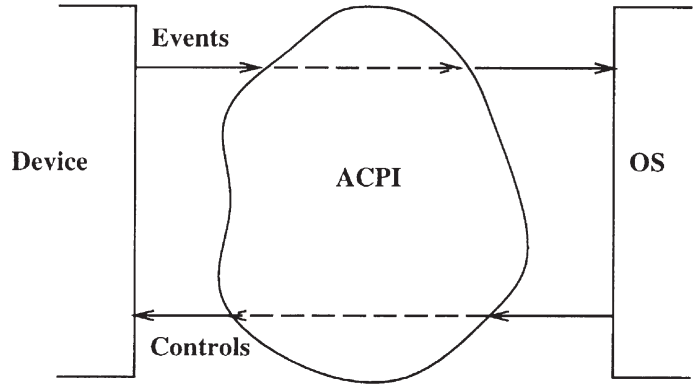


Figure 6.9. Communication between ACPI and devices

tion to support the communication between devices and OS (refer to [ACPI96] for a complete description). To manage power for all devices in the system, the OS needs standard methods for sending commands to a device. Device power management specifications (introduced in Section 6.2) define the operations required to manage power of devices belonging to a particular class (e.g audio devices, storage devices etc.). ACPI defines two flavors of device power management specifications. First, *bus devices* have special specifications that provide commands for controlling the bus circuitry itself, but also all devices connected to the bus. Second, devices on the system board are directly controlled using the basic ACPI specification.

Devices that are directly connected to the motherboard and bus devices are managed directly using standard ACPI controls (to be described later). Devices connected the bus are managed by sending ACPI messages to the bus device driver that translate them to power management messages that comply with the power management specification of the bus. Then the bus device driver forwards the translated command to the destination resource. We focus here on standard ACPI controls which can be divided in four basic types.

- *Identification of device power capabilities.* When the OS enumerates all devices in the system (at boot time, or when requested by the user), it gets information about the power management features supported by the devices. Every device in the system responds to this control by returning a *differentiated definition block* to the OS. The differentiated definition block contains the following information. i) A description of the power resources (power supplies and clock drivers) the device needs in each supported power state. For example, a device might need clock signal in D0 state but no clock signal

in the D2 state. ii) A description of the power resources a device needs in order to wake the system (or indication that the device does not support wakeup). This information can be used by the OS to know when to expect wakeup signals from a device. iii) The control method the OS can use to set the power state of the device and to get and set resources.

- *Setting device power states.* The set-power-state operation is used by the OS to put a device into one of four power states. Whatever power state a device is put into, the OS must be able to issue a new set-power-state command that can resume normal activity. In some cases the OS must turn on the power supply of the device before it can sent the set-power-state command. The time needed to restore the power supply is part of the latency for exiting from the low-power state. The set-power-state can also be used to enable features like wakeup (i.e., the device is authorized to wake up the system and force an exit from global sleep state). When a device is about to transition to a new power state, the OS first decides which power resources will be used and which can be turned off. The OS also tracks all devices on a power supply resource. If all devices on a source have been turned off, the OS can decide to turn off that power supply.

- *Getting device power status.* This operation is used by the OS to determine the current power configuration, namely state and features, as well as the status of any batteries supported by the device. The device can trigger an interrupt to inform the OS of changes in power status. In response to the interrupt, the OS uses this operation to fetch detailed information on the events that caused the status transition. Such information is needed to decide on the actions for dealing with the new operating conditions of the device. For instance, a device can signal with an interrupt that its battery is almost exhausted. In response to the interrupt the OS may need to get the device power status in order to actuate a graceful shutdown.

- *Enabling device-controlled wakeup.* The wakeup functionality allows devices to wake the system from a global sleeping state. Clearly, this functionality cannot rely on the CPU because the CPU is not active in sleep states. When transitioning into a sleep state, the OS enables wakeup for the devices that are defined by the user's application as sources of wakeup events. The OS must also guarantee that any bridges between the wakeup devices and the core chipset logic are in the lowest power states in which the can still transmit the wakeup call. When the computer is in a sleep state and a wakeup device decides to issue a wakeup, it signals to the ACPI-compliant core logic on the chipset. The chip set will wakeup the machine. When the OS is up and running, it will handle the wakeup event.

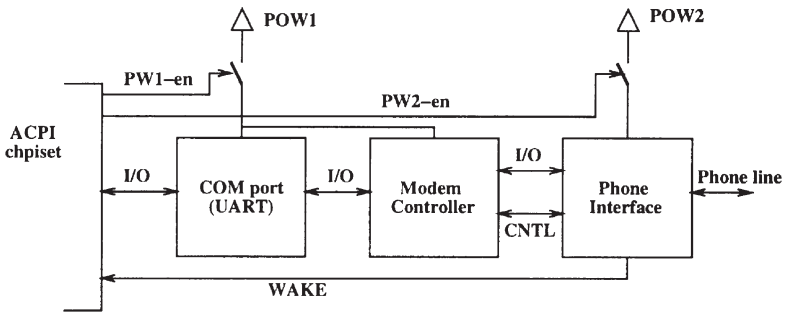


Figure 6.10. Block diagram of an ACPI-compliant modem

The CPU is a special component with specialized power states because it is the hardware that runs the OS. Hence, special control methods are used to manage the CPU state transitions. In particular, transitions to low-latency CPU sleep states are based on observations of the percentage of time in which the OS runs the *idle process* (i.e., the lowest priority process in the OS, which does nothing useful). In the previous section we discussed the impact of application coding style on the amount of idle time.

The two-way communication between ACPI and hardware devices, depicted in Figure 6.9, is based on *controls* (for communication from ACPI to devices) and on *events* (for communication from devices to ACPI). Events are asynchronously generated by devices. They are processed following a standard protocol based on two registers: the *event status register* and the *event enable register*. When an event occurs, the core logic sets a bit in the status register to indicate the event. If the corresponding bit in the enable register is set, the core logic will assert an interrupt to signal the OS. The OS will then run the interrupt handlers corresponding to the bits set in the event status registers. These handlers are called *control methods*. Details on event handling can be found in [ACPI96].

Example 6.2 *We illustrate the main features of device power management support in ACPI considering an integrated modem (this example is adapted from [ACPI96]). The block diagram of the modem is shown in Figure 6.10. The modem consists of three functional blocks: communication port for data and control communication with the core PC logic, the modem controller and the external receiver/transmitter connected to the telephone line. The power states of the modem are:*

- *D0. Modem controller is on, as well as phone interface and communication port. The device is fully functional, can be on or off hook and can be waiting for answer.*

- *D1. The modem controller is in low power mode (context is retained). The phone interface is powered by the phone line or in low power mode. The speaker is off. The modem must be on hook.*

- *D2. This state (optional in ACPI) is equivalent to D1.*

- *D3. The modem controller is off (all context is lost). The phone interface is powered by phone line or off. Speaker is off. The modem is on hook.*

The power policy for the modem is defined as follows. In the transition from D3 to D0, the communication port is opened, while in the opposite transition it is closed. When transitioning from D1 to D3 the `COMM` *port is opened as well. The transition from D0 to D1 puts the modem in answer mode, while the opposite transition is performed when a user application requests dial or the phone rings (an wakeup is enabled). The wakeup policy consist of sending a* `WAKE` *signal when the phone rings and wakeup is enabled. The signal will wake the PC from a sleep state.*

Although ACPI merely specifies the power states, we briefly discuss how they could be implemented in the device to save power. In state D0, both power source `POW1` *and* `POW2` *are connected. D1 requires only* `POW1` *(*`POW2` *is disconnected by signal* `PW2-en`*. D0 does not require any power source. To wake the machine, the modem does not need any power source (i.e it can wake the machine from either D0, D1, D3).*

6.3.3 Battery management

In the ACPI specification, batteries are seen as another hardware resource. An ACPI-compatible battery needs to conform either to the *Smart Battery* subsystem interface [smba97] or a *control method battery* (CMBatt) interface. Smart batteries are controlled by the OS directly through an embedded controller. Control method batteries have an interface which is completely specified by methods written in AML. For the sake of illustration we describe the features of the CMBatt interface.

The CMBatt is accessed by an AML interface that gives the hardware designer the choice of any communication interface at the hardware level. The AML code that handles the battery interface should generate events to notify the OS of battery status changes. Events are generated when the battery device is inserted and removed from the system. Moreover, CMBatt reports

the designed capacity, the latest full-charged capacity and the current remaining capacity. Remaining battery capacity decreases during usage and changes depending on environmental conditions. CMBatt issues three special status messages as the stored charge deceases. The messages are *warning*, *low* and *critical*.

Warning and low messages are defined somewhat arbitrarily for the purpose of giving the OS advanced notice when the battery is approaching discharge. In contrast, *critical* is issued when the battery is fully discharged and cannot supply any more power. When a *critical* message is issued the OS must react quickly, because not enough charge is left to perform a normal shutdown procedure. Hence the OS performs emergency shutdown. The system cannot be turned back on until the battery is replaced or recharged.

In principle, the information provided by the battery to the OS should be sufficient to implement a *battery gauge* (i.e an algorithm that computes the amount of charge left in the battery). Unfortunately, battery gauges are not accurate and designers protect themselves against the risk of unwanted power failure by *down-rating* the batteries. In other words the *critical* message is issued when the battery is not actually fully discharged in order to guarantee with a good degree of confidence that emergency OS shutdown can be accomplished. OS-controlled battery management has the big advantage of being flexible and easily re-programmable for different classes of batteries. ACPI provides full support for OS-controlled battery management.

6.3.4 Thermal management

Another important feature of ACPI is the support for OS-controlled thermal management [ACPI96]. ACPI moves the hardware cooling policies from the firmware to the OS. This is a very powerful concept. Effective cooling decision can be made based on application load on the CPU and instance dependent, flexible thermal heuristics. Moreover, the OS has the capability to gracefully shutdown the computer in case of high-temperature emergencies or to temporarily decrease performance to allow cooling of hot spots.

The ACPI thermal management support is based on the concept of *thermal zones*. A thermal zone is a portion of a physical board (or an entire board) for which temperature is measured and reported to the OS (by dedicated *temperature events* in the ACPI interface). In simple implementations, the entire computer hardware belongs to the same thermal zone, but multiple thermal zones may be defined in complex, high-performance platforms. Thermal management is based on cooling. ACPI supports *active cooling* and *passive cooling*. Passive cooling is obtained by reducing the power consumption of hardware

components in the thermal zone. Active cooling exploits cooling devices such as fans.

The OS can define thermal policies based on a mix of active and passive cooling. Active cooling increases power to reduce the heat within the system. Passive cooling reduces power to decrease temperature. As a result, active cooling allows maximum performance, but increases overall power dissipation and equipment noise. On the contrary, passive cooling reduces performance but also reduces power and does not create additional noise. In battery-powered computers where battery lifetime is the primary concern, passive cooling is generally preferred, while active cooling is usually a good choice for desktop systems.

There are cases in which these general guidelines can be invalid. For example, when a desktop computer must adhere to strict environmental noise rules, active cooling may not be acceptable. ACPI offers a considerable flexibility in implementing thermal management policy. The user may select the best mix of active and passive cooling according to his/her needs. Alternatively the OS may automatically select the best cooling method based on workload conditions and information provided by active applications.

While active cooling requires additional devices, passive cooling is based on the basic power management capability offered by ACPI. Devices can be shut down more frequently and aggressively (at the price of latency increase) when temperature is too high. Alternatively, the system may be slowed down by decreasing clock speed. This technique is often applied to CPUs which typically have extremely high nominal clock speed and tend to become hot spots. The effectiveness of thermal management can be improved by defining multiple thermal zones, and by implementing localized policies.

Example 6.3 *Figure 6.11 shows an example of a PC board with two thermal zones. Each zone has independent temperature sensors (S1 and S2), and different thermal management policies. The zones are enclosed by the dashed boxes. Two zones have been defined because the topmost thermal zone is clocked at a very high frequency and tends to generate much more heat than the lower thermal zone. Hence, the thermal management policies are different.*

For the topmost thermal zone, a mix of active and passive cooling is used. The policy defines two threshold temperatures, $T_0 = 50^0C$ *and* $T_1 = 70^0C$. *When the measured temperature reaches* T_0, *passive cooling is started: the CPU clock is slowed down by a factor of two. If critical temperature* T_1 *is reached, the fan is activated, and the zone is actively as well as passively cooled. The policy targets low noise requirements and activates the fan only when absolutely necessary for maintaining system integrity.*

The bottom zone is less critical since it is clocked at a much lower clock speed. However, if external environmental conditions are particularly hostile

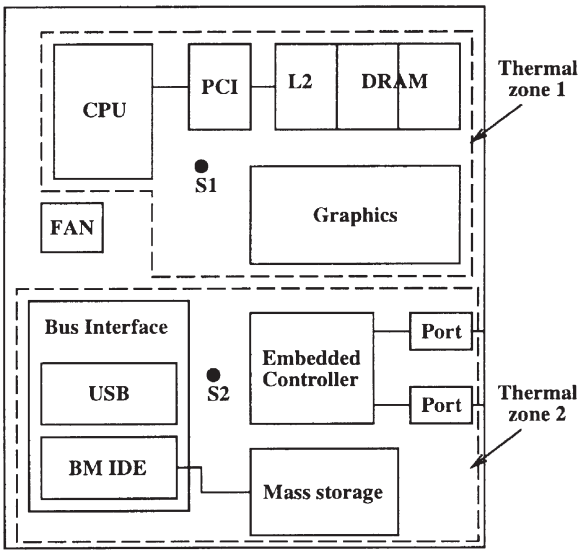


Figure 6.11. Thermal zones in a PC board

and the zone reaches the temperature of 60^0C*, the fan is activated. Notice that clock slowdown is not possible for the components of this bottom zone.*

6.4 MODELING POWER MANAGEMENT

Both ACPI and *OnNow* specifically target general-purpose computers ranging from portable PCs to servers. *OnNow* is an initiative proposed by a single software company that, despite its quasi-monopolistic status in the area of operating systems for personal computers, has only a partial view of the complex market of portable electronic appliances. ACPI is more general in scope than *OnNow* since it does not depend on any specific OS implementation and it is not bound to any proprietary hardware model. Nevertheless, ACPI shares with *OnNow* the assumption of an *OS-centric architecture*, where the activities of the system are managed by a software process (the OS) that runs on a central processing unit.

Moreover, ACPI was developed to support the *implementation* of power-managed computer systems and is too detailed to effectively support *design exploration.* In the early phases of the design of a new architecture, the system designer needs to experiment with alternative configurations and choices. In many cases key system components (such as interface logic) have not yet been designed and even off-the-shelf devices have not been specified (or are not yet

commercially available). Power management adds a new dimension to design complexity. Even at the first stages of architectural definition, it is highly desirable to be able to assess the impact of power management policies.

In the case of personal computers, new products have often been derived in an evolutionary fashion from older products. As faster and more power efficient commodity components (such as microprocessors, memories, communication devices) become available, older designs are incrementally modified to incorporate the new components and achieve better performance (or reduced power dissipation). If a new design is a straightforward evolution of an old product, it might be possible to extrapolate the power dissipation of the new system from the old implementation, or from early prototypes of the new implementation. Unfortunately this approach is not feasible when a radically new architecture is designed.

We introduce a modeling approach that is aimed at providing support for system-level architectural exploration of power-managed systems. The key features of our model are:

- Generality. Any power-managed electronic appliance (not only computer systems) can be modeled. Cellular phones, pagers, digital cameras are few examples of portable appliances for which power is a major concern.

- High level of abstraction. The choice of abstraction level is probably the most important decision in system-level power modeling. In order to manage complexity, irrelevant (or marginally relevant) information should be abstracted. We take an aggressive approach. Only functionality relevant to the power behavior is modeled. Resources are described by abstract power and performance models.

- Non-determinism, The uncertainty on estimation caused by the abstraction process (and the consequent information loss) is mapped to the inherent uncertainty in the estimation of the expected value of random variables. The power and performance estimates provided by our model are indeed random variables. They are not described by simple deterministic values, but by expected values and confidence intervals. Estimation involves some form of statistical sampling of random variables that is required to compute expectations and confidence intervals.

In the next subsections we describe the components of our model and we provide insights on how the model can be used to provide early power and performance estimates for power managed systems.

6.4.1 Model definition

In our model, a system is defined by a set of *components* and a communication pattern between components. Communication is modeled by abstract *events* that are transmitted from a source component to one or more destination components. System components belong to one of the following three types.

- *Central control.* There is only one component of this type. It represents the abstract entity that controls power management. For example, if we are modeling an *OnNow* PC, it represents the OS module that implements the power management policies. Central control receives and sends events to all other components in the systems. From the communication point of view, the central control is the center of a *star* communication pattern.

- *Service provider.* There are generally multiple service providers in a systems. They abstractly represent the hardware system components that implement functionality (and dissipate power), e.g, a modem control unit, a video display controller, a disk driver unit. Service providers belong to *classes* that represent the type of service. Example of classes are: storage device class and video device class.

- *Workload source.* There are generally many instances of workload sources in a system. They are generators of requests to a particular class of service providers. Sources abstractly represent the system workload that is generated by external users.

The details of the structure and behavior of components will be described later. System components communicate through abstract events (also called *messages*. There are four types of events.

- *Work requests (WR).* These events are generated by workload sources and are directed to the central control. The central control will forward them to service providers according to the type of work request and to a work dispatch policy.

- *Work orders (WO).* These events are generated by central control and are directed to service providers. Work orders are always associated with work requests.

- *Power commands (PC).* These events are generated by central control and directed to resource providers. They are used to control changes of power states in the devices.

- *Completion notifications (CN).* These events are generated by service providers and are directed to the central control. They are issued in response to work

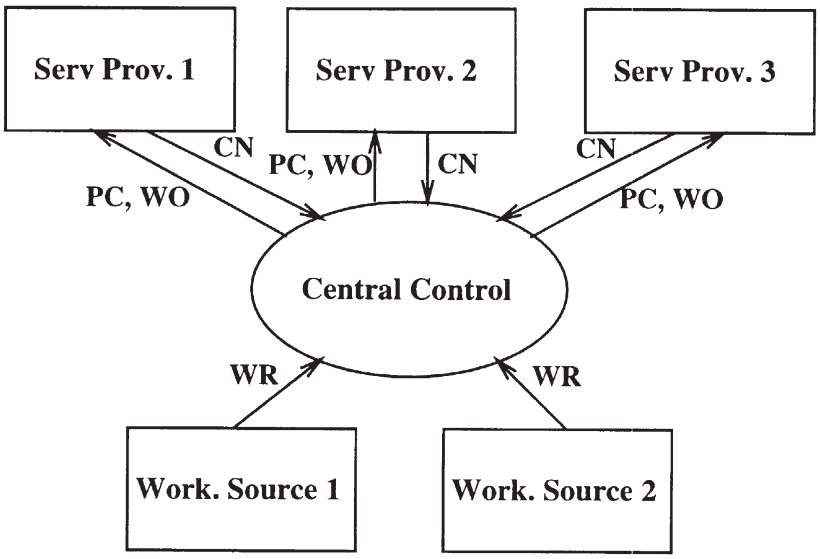


Figure 6.12. The structure of a generic system.

orders and power commands, to inform the central control of the completion of the required task.

Each event instance is marked by a unique identifier. There are causal relationships between events. WOs are generated in response to WRs. CNs are generated in response to PCs or WOs. Hence, each WO contains the identifier of the WR that has generated it and each CN is marked with either the PC or the WO that generated it. Events are the only form of communication between components. Functional information is abstracted away.

The structure of a system with three service providers and two workload sources is shown in Figure 6.12. The communication pattern between system components is a star. The central control is the center of the star. There is no direct communication among service providers and workload sources.

6.4.2 Power states and transitions

Each service provider class is associated with a set of *power states* and state transitions. Transitions are triggered by events (dispatched by the central control). Each state is characterized by a power consumption value. The state structure is similar to that of ACPI. However, the abstraction of the model is much higher, and no details are specified about the functional behavior of service providers in each power state. Moreover, the power consumption in

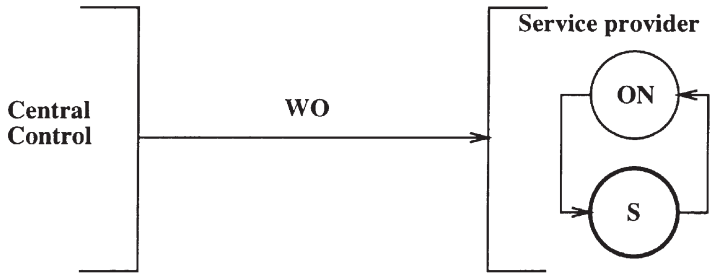


Figure 6.13. A work order issued to a service provider in sleep state

each state can be defined as a random variable of known distribution to model the uncertainty on the estimate of the power consumption due to the high abstraction level and the lack of functional information.

Similarly to the ACPI specification, for each service provider there is only one state, called ON, in which full functionality is provided. There can be an arbitrary number N_s of sleep states $S = \{S_i, i = 1, 2, ..., N_s\}$. In these states service providers are not operational (i.e., they cannot absorb workload). Power dissipation is highest in the ON state and lower in S states. Each $S_i \in S$ represents a sleep mode of the hardware device modeled by the service provider.

When a PC or a WO event is issued by the power manager to a service provider, the provider reacts by initiating a state transition. For example, if a WO is received and the provider is in a sleep state S_i, a wake-up transition towards state ON is initiated. State transitions take a non-deterministic amount of time. In other words, once a transition is initiated, the provider does not change its state instantaneously, but it waits a for some time before moving to the destination state. The wait time is a random variable of known distribution. This condition is called *transient*. Each transient condition is characterized by a power consumption (which is a random variable with known distribution).

Transient conditions model the two-fold overhead incurred in turning on and off hardware devices. First, device performance is decreased (this is modeled by the non-deterministic transition time). Second, transitioning to a new state costs additional power (to save/restore context and perform various shutdown/wakeup activities). The uncertainty on the detailed behavior of the service provider caused by the abstraction and lack of functional information are modeled by non-deterministic transition times and power.

Example 6.4 *Consider a service provider with two power states, as shown in Figure 6.13. Assume that the service provider is in a sleep state S when a work order event is sent to by the central control. Before being able to service the request, the provider must transition to state ON. The transition takes a non-*

deterministic amount of time T_{tran} with known distribution. For instance, the transition time has exponential distribution with average value 2sec. The power dissipation during the transition time is a random variable as well (for instance a uniformly distributed random variable with value between 0.5W and 2W). Once in the ON state, service time is random variable T_{serv} (with exponential distribution). The total time for servicing the WO is $T_{tran} + T_{serv}$.

Notice that transitions between some power states may be forbidden. For example, it may be impossible to transition from S_i to S_j without going through ON. Forbidden transitions model features of the hardware device represented by the service provider.

6.4.3 System behavior

The state of the system changes over time, as workload sources generate WR events. Central control receives the WRs and manages the usage of service providers. Central control has two main tasks.

- Dispatch of WO events according to the arrival distribution of work requests. The decision procedure used by central control to decide when to dispatch WO events is called *dispatch policy.*
- Power management of the service providers by issuing PC events. The decision procedure used by central control to shut down and turn on service providers is called *power policy.*

In many cases power policy and dispatch policy interact, or are formulated as a single procedure that takes care of both dispatch and power management. Such merged procedure is simply called *policy.*

The central control must minimize power through power management and ensure that tasks are executed correctly and within the specified constraints on performance (to be discussed later). The power dissipation of the system is controlled by selectively shutting down resources when they are not servicing requests. Central control has the same role as the operating system in the ACPI specification. Notice that in the case central control is a software process running on some hardware resource (e.g., the OS running on the CPU), the power management policy of central control must activate the service provider that represents the hardware resource whenever a new WR is received. This procedure models the wakeup functionality specified by ACPI.

Service providers respond to WO and PC events sent by central control by initiating transitions to *service the event.* When one service is completed, the service provides sends a CN event to central control. The CN event is labeled with the unique identifier of the WO (or PC) it correspond to. A

PC is considered to be serviced when the service provider has reached the destination power state requested by the PC. A WO is considered as serviced a non-deterministic (with known distribution) amount of time (called *work time*) after the service provider has reached the ON state.

The amount of time it takes to service a WO depends on two factors: the transition time required to reach the ON state and the work time once the provider has reached ON. Because of the delays in servicing WO events, it may be the case that multiple service requests overlap. While the first request is still being serviced, a second WO (or multiple WO events) may arrive. Collision of multiple requests is flagged and monitored, but no requests are lost, i.e., every WO is sooner or later acknowledged by a CN. Given a workload distribution, the power consumption and the performance of the system depend on the policy adopted by central control.

6.4.3.1 Power dissipation model. At any given time, the power dissipation of the system is computed as the sum of the power of components that are stable in either ON or any S_i state and the power for components that are transitioning between power states. Over time, system power dissipation is a *piecewise constant* waveform $P(t)$. The discontinuity points in the waveform correspond to changes of state for the service providers. We may be interested in the *average value* of power $P_{avg}(T)$ over a time interval T, or in the *maximum value* of power $P_{max}(T)$ in the same period.

Another relevant metric is the *cumulative energy consumption* over a time interval T, $E_{tot}(T)$. It is defined as the area below the power dissipation waveform, and can be computed by multiplying the power value of each constant interval by the duration of the interval and by summing over all intervals. We are interested in this quantity, because for battery-operated systems there is a tight constraint on the total energy that can be provided by the battery. Notice that instantaneous power, average power, maximum power and cumulative energy consumption are stochastic variables that depend on the system workload and the policies chosen by central control.

Example 6.5 *The power dissipation $P(t)$ of a system as a function of time is shown in Figure 6.14. The time-domain power waveform for the model is piecewise constant. Changes in power states for service providers in the system correspond to discontinuities in the waveform. It is important to notice that the waveform in Figure 6.14 is merely one sample of a stochastic process. In other words, both the power values P_i and the transition times t_i are random variables. The uncertainty on P_i and t_i is represented in Figure 6.14 by the shaded areas around the expected values.*

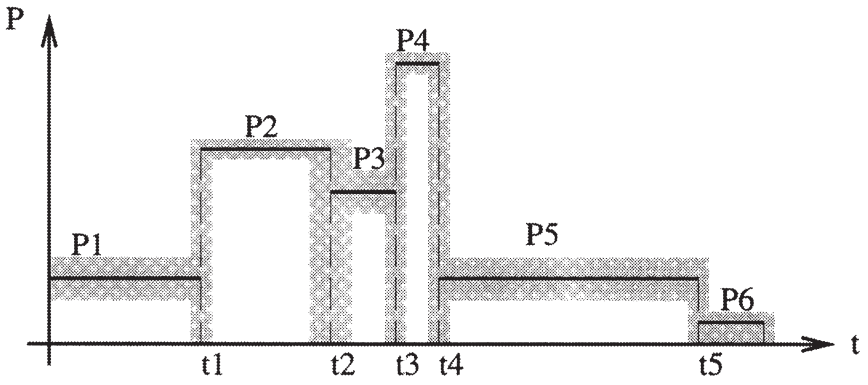


Figure 6.14. A power waveform $P(t)$ for a power-managed system

6.4.3.2 Performance model. The purpose of the performance metric is to model and measure the effects of power management on system performance. In other words, we want to estimate how power management changes the overall performance with respect to the same system with no power management capability. Relevant performance-related metrics include:

- Incremental collision probability. How the wake up delays of power managed service providers change the probability of having two or more unserviced requests waiting on a provider
- Incremental average latency penalty. How much more (on average) an incoming request needs to wait before being serviced because of the slowdown due to power management.
- Incremental throughput penalty. How much the average throughput (i.e., serviced requests per unit time) is changed by the presence of power management
- Average time spent in a given power state transition by a given service provider.

All of these quantities are stochastic variables that depend on the policies chosen by central control. Notice also that we are interested mainly in *incremental* metrics, i.e., differences between the baseline implementation where all providers are always ON and power-managed implementations where central controls tries to reduce power dissipation by selectively shutting down unused resources. This is an important modeling choice. To obtain meaningful

estimates of the performance impact of power management, the baseline implementation without power management should provide a sufficient level of performance to manage the workload.

Example 6.6 *Consider a service provider with two sleep states, S_1 and S_2. The average time to exit from S_1 is $50msec$ and the power in S_1 is $1W$. The average time to exit from S_2 is $500msec$ and the power in S_2 in $500mW$. We use as a performance metric the average latency penalty of incoming request. Assume that the service provider is operating under light load, hence it requests arrive almost always when the provider is idle. A power management policy that does not take performance into account would always put the resource in state S_2 as soon as it becomes idle. If performance is a concern, however, this might not be the best choice, because it causes an increase of $10X$ in average latency with respect to the policy that always puts the idle resource in sleep state S_1.*

6.4.4 Stochastic simulation

The system model described in the previous sections can be simulated within an event-driven simulator. Workload sources generate events that cause other events to be generated in the system. Simulation proceeds over time. The user can specify an observation interval after which the simulation will be stopped and the power and performance metrics will be reported.

The event-driven paradigm and the high level of abstraction are essential features for achieving sufficient simulation speed. In the abstract simulation environment, functionality is modeled by randomization of transition and service times. Only power-relevant events are generated, which are an extremely small fraction of the total number of functional events (data communications, computations, etc.) in a complex system. Moreover, service providers can respond to requests with widely-varying delays. For instance, a disk driver model in sleep mode may need to spin up the disk in order to service a data request. This could take seconds. In contrast, a CPU in sleep mode may require a few milliseconds to re-activate (i.e., the time to activate the PLL). As a result, events are generated with typical delays that may differ by several orders of magnitude. Because the performance of event-driven simulation depends on the *number* of events and not on the time intervals between them, event-driven simulation is highly recommended for systems where delays can vary widely and the number of events is relatively small.

There are two critical issues in the modeling approach. First, power and performance are, without exception, stochastic variables. Performing a single simulation provides one sample of a stochastic process. In order to obtain meaningful estimates, average values and confidence levels should be computed.

This requires multiple simulation runs (under the same workload conditions) until some statistical convergence test on the metrics of interest has been satisfied with a user-specified confidence level. Monte Carlo techniques similar to those introduced in Chapter 1 may be used. We call *stochastic simulation* the iterative simulation process driven by statistical convergence criteria on the cost metrics.

The second key issue is workload modeling. The communication pattern in our model is highly simplified. We assume that requests for service providers come only from the external workload sources. In reality, system components may generate requests for other components with very complex patterns. In our model, these effects must be accounted by modifying workload sources. For instance, consider a disk write request coming from a remote machine to a file server. The request comes first through the network interface, then is processed by the CPU and finally is forwarded to the disk driver interface for storage. In real system, the "propagation" of the request among system components is performed by the component themselves. In our model this effect should be modeled by generating sets of WR events one for each component involved in servicing the request.

Workloads can be modeled by collecting traces from functional simulations (or from monitoring system activity on real hardware). Alternatively, probability distributions of the arrival times of the requests can be provided. Notice that both methods imply some loss of accuracy with respect to an integrated approach where functionality is simulated in lock-step with power management. This is due to the fact that there may be some form of feedback between generation of requests and services. This cannot be modeled by stochastic simulation. To conclude the section we provide an example description of a simple system.

Example 6.7 *Consider the system shown in Figure 6.15, representing a very abstract view of a network file server. The system consists of three service providers:* `NINT`, `CPMEM` *and* `DISK`. *The network interface* `NINT` *has two power states:* ON^{NINT} *and* S_1^{NINT}. *Average power in* ON *state is* $P_{ON}^{NINT} = 1.5W$, *while* $P_{S_1}^{NINT} = 0.1W$ *(both have Gaussian distribution). The time for transition* $S_1 \rightarrow ON$ *has exponential distribution with average value* $15ms$. *The time for* $ON \rightarrow S_1$ *has exponential distribution with average value* $2ms$. *The power costs for transitions have Gaussian distribution with average values* $1.6W$ *and* $1.2W$. *Service time for a request is deterministic, and is equal to* $20ms$.

The second service provider `CPMEM` *models the CPU and memory system. It has two power states* ON^{CPU} *and* S_1^{CPU}. *The power state and transition times of CPMEM are shown in Figure 6.15. The third service provider (*`DISK`*) models the disk and its driver. It has three power states:* ON, S_1 *and* S_2. *An interesting feature of this provider is that the transition time between* S_2 *and*

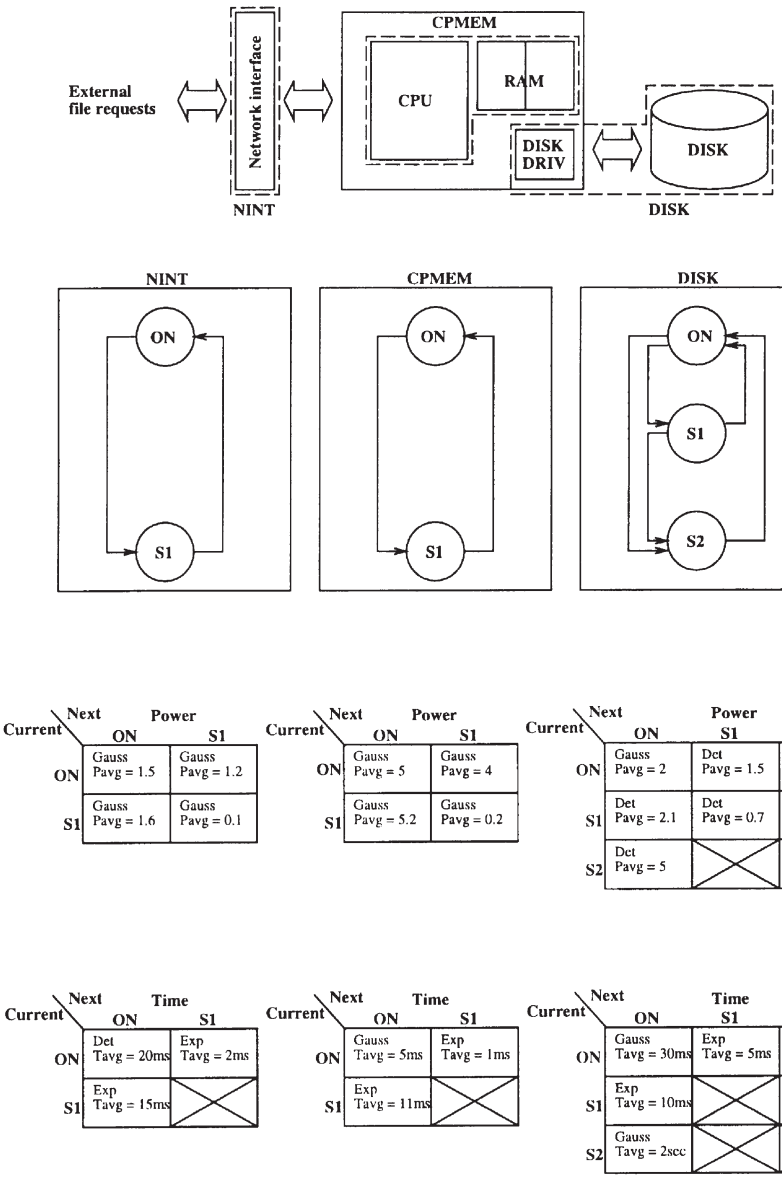


Figure 6.15. Stochastic simulation model

any other state has an average delay 2sec, with narrow Gaussian distribution. The long delay models the time required do spin up the disk.

In this system, whenever a network request comes in, the network interface processes the incoming packets and forwards them to the CPU. The CPU has a disk cache in main memory. If the requested file is found in disk cache, the disk is not accessed, otherwise the disk is accessed and the data is returned to the CPU which then forwards it to the network interface. The cache has an average hit rate of 70%. This activity pattern can be modeled by the following workload. Work requests come in clusters: a WR for NINT *always comes with a WR for* CPMEM*. A WR for the disk is generated only 30% of the times the other request arrive. Moreover there is a constraint on the power management policy.* CPMEM *cannot be put in sleep state if* DISK *is* ON*. This constraint models the fact that CPU must be active to manage the disk controller while servicing a file access request. Performance is modeled by the average number of requests pending on the* DISK *service provider (which is the system bottleneck).*

This model can then be used to evaluate: i) the impact of power management policies on performance and power i) the sensitivity of a power management policy to variations in workload, iii) the sensitivity of performance and power to variations in parameters in service providers (for example, disk spin up time).

The simulation model presented in this section targets early estimation of power and performance implication of power management policies at the system level. Very large and complex systems can be modeled and simulated with high efficiency thanks to the event-driven paradigm and the high abstraction level. The uncertainties due to the abstractness of the approach are modeled by randomization. Several stochastic variables are introduced in the model. As a result, the cost metrics of interest are stochastic variables as well. Estimation of average values and confidence intervals for the cost metrics requires repeated simulation runs. The complexity of internal interactions between system components is modeled (with some loss in accuracy) in the formulation of the workload.

6.5 THE POLICY OPTIMIZATION PROBLEM

Although it is obvious that power management policies have great impact on power and performance, assessing the impact in a quantitative fashion is still an open problem. The modeling approach presented in the previous section may shed some light on the problem. Given a system model, a workload and a policy, stochastic simulation can rapidly estimate the power and performance of the system. This process can be iterated for several policies and the results can be used to select among alternatives.

The *policy optimization problem* consists of finding an optimal power management policy for a system given a workload and performance constraints. The critical issue in policy optimization is defining the search space. In its most general formulation, a power policy can be any algorithm. The only practical way to explore such a huge search space is through human guidance. The designer specifies a policy and uses estimation tools to assess the validity of his/her choices. Stochastic simulation, prototyping, back-of-the-envelope calculations may be viable estimation approaches. The advantage of these techniques is that they can provide estimates before the final (or even functional) implementation is available at the expense of accuracy.

An alternative approach is to design systems where the power management strategy is reconfigurable (re-programmable) during system operation, and provide sufficient degree of programmability of the system to enable the specification of optimal policies. Unfortunately, flexibility is expensive in terms of hardware resources and performance. A compromise solution is to design systems with limited, low-overhead programmability in which optimal policies can be implemented for a wide range of workloads.

Several empirical studies have been performed to assess the quality of power management policies [gold95, doug95]. The purpose of these studies is to provide designers with general guidelines for choosing among *families* of policies. This is valuable information because it allows the designer to focus his/her attention on promising regions of the search space, or alternatively to implement reconfigurable solutions with low overhead. In the next subsection we briefly review the results of a detailed study of power management policies for hard disks [gold95]. Although power management of hard disks is only one facet of global system-level power management, this problem has received considerably more attention than any other. We believe that similar analyses should be performed for other sub-systems. The methodology employed in [gold95] provides valuable insights on experimental setup, classification criteria and presentation of results.

6.5.1 Disk power management

Golding and coauthors [gold95] tackle the general problem of finding the best way to exploit idle periods during the operation of a hard disk. When power is the primary concern, the best way to exploit idleness is to shut down the unused device. Shutting down a disk is a relatively easy task, but unfortunately turning it on is much more expensive in time and power. When turning on a disk driver, the motor that spins the disk must overcome the inertia of the disk itself. This is time-consuming task. The authors of [gold95] used in their experiments a disk with the following characteristics: i) power consumption when ON: $1.5-1.7W$,

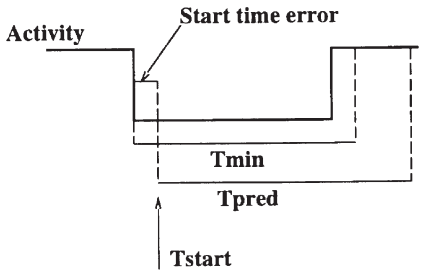


Figure 6.16. Idleness prediction

ii) time to spin down the disk: ≈ 0, iii) power consumption when in sleep (disk spun down): ≈ 0, iv) time to spin up (i.e., transition $ON \rightarrow S$): 1.5sec, v) power consumption when spinning up: 2.2W. Notice that the disk model used by [gold95] is fully compatible with the resource provider model presented in the Section 6.4. The main feature of the model is that it is convenient to spin down the disk as long as it remains in sleep state for a period of time longer than 2.2sec. This value is a threshold T_{min}. The disk should be left on when the idleness period is shorter than T_{min}.

A power management policy that spins down the disk when it is idle can be successful only if it can predict with sufficient accuracy the start time and the duration of the idle interval. Referring to Figure 6.16, the policy predicts an idleness start time T_{start} which follows the actual termination of activity. The difference between T_{start} and the time of termination of activity should be as small as possible. Moreover, if the predicted duration T_{pred} is longer than T_{min} but the actual duration of the idle interval is shorter than T_{min}, the disk will be spun down but it will need to be spun up before the break-even point for power consumption has been reached. We call this situation a *misprediction* and the power cost associated with it *misprediction penalty*. Figure 6.16 shows a case when a misprediction will occur. A succesful power management policy minimizes both start time error and frequency of mispredictions.

The key component of a power management policy for a disk drive is a *predictor* that monitors its environment and issues predictions as couples $[T_{start}, T_{pred}]$, where T_{start} is the expected start time of an idle period and T_{pred} is the expected length of the idle period. The structure of a generic idleness detector that implements a power management policy is shown in Figure 6.17. Golding et al. [gold95] introduce two components besides the predictor, namely the *skeptic* and the *actuator*.

The skeptic observes one or more predictions and generates new ones. Skeptics are mainly used to filter out bad predictions or to combine several predic-

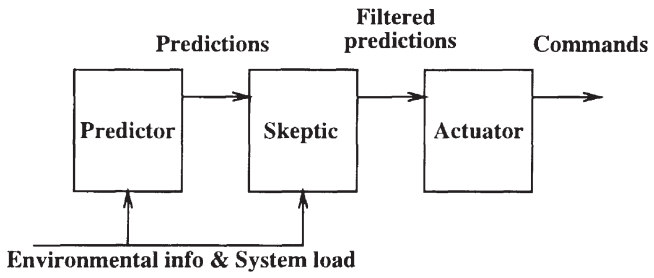


Figure 6.17. Components of an idleness detector

tors and produce a single improved prediction. The main use of skeptics is to compute a single stream of predictions by performing the weighted average of many alternative predictors. In [gold95] only single-stream skeptics were used (i.e skeptics that observe a single predictor, as shown in Figure 6.17). The actuator's only purpose is to translate prediction into commands for the controlled resource.

Golding and coauthors also formulated two cost metrics to evaluate the effect of power management policies. The *power saved* and the *performance penalty* represented by the number of disk operations that had to be delayed because the disk was not ready when the operation was issued and had to be spun up. Comparison between policies was based on power savings and performance penalties on a number of different workloads. Workloads were disk usage traces obtained by monitoring several computer systems.

All idleness detectors compared in [gold95] have a trivial T_{pred} computation procedure. The expected length of the idle period was assumed to be constant $T_{pred} = 10sec$. The value was chosen because it is large enough to guarantee sizable power savings in case of correct prediction (remember that the break-even point for the disk under consideration was $2sec$).

Since no prediction effort was made in estimating T_{pred}, all power savings came from the start time prediction algorithm. Good T_{start} predictors showed an *adaptive behavior*: they adapted the detection policy to produce a T_{start} only if it was likely that the duration of the idle period would be longer than $T_{pred} = 10sec$. Interestingly enough, none of the T_{start} prediction policies tried to anticipate future idleness. In other words, new pairs $[T_{start}, T_{pred}]$ were generated only after an idle period had effectively begun.

Among the eight T_{start} prediction policies tested by the authors of [gold95], two did the best on the available workloads. Interestingly enough, several plausible policies performed badly on several workloads, revealing worrisome pitfalls in heuristic policy design. Quality of results was measured evaluating

the power saving achieved for a given performance degradation. The two best policies were:

- A constant timer-based policy. If the disk has been idle for T_{timer}, it is assumed that a long idle period has started (i.e longer than $T_{pred} = 10sec$), and the shutdown command is issued.
- An adaptive timer-based policy. The algorithm starts with an initial T_{timer} value as for the previous policy. If the start time predictor does not produce a T_{start} for an idle period longer than $T_{pred} = 10sec$ (i.e., the disk is not stopped when it should have been), the timeout T_{timer} is decreased geometrically. If a T_{start} is produced for an idle period shorter than $T_{pred} = 10sec$ (i.e., the disk is stopped when it should not have been), the timeout T_{timer} is increased arithmetically.

Besides giving indications as to the best policies for power minimization under performance constraints, the authors of [gold95] also performed a detailed study of the sensitivity of policies to tuning parameters (for example the value of T_{timer} for the fixed-timer-based policy, or the rate of increase and decrease of the adaptive-timer-based policy). The effect of several classes of skeptics on the quality of the results was also investigated. One important conclusion of Golding et al. is that adaptive policies performed better than fixed policies. The increased flexibility that stems from adaptation improves the average power savings over widely varying workloads.

Although the work by Golding and coauthors presents a solid experimental methodology and a number of interesting conclusions, it does not provide any indication on how to search for optimal policies. Even the large number of alternatives considered by the authors is merely a minuscule sample of the possible policies. Moreover, the policies studied in [gold95] depend on several parameters. No indications are provided on how to choose optimal values of the parameters (if not by trial-and-error).

In summary, experimental studies have shown that a careful choice of power management policy may have a sizable impact on power consumption. Thus the policy optimization problem is of great practical interest. Unfortunately, no directions have been provided on how to select optimal policies or how to tune parameters in a given policy. Hence, the policy optimization problem is still open. We believe that policy optimization is a new and very promising area of research. Industrial standardization efforts such as ACPI have provided the support for the implementation of large and complex power managed system. Simulation models based on the concepts presented in Section 6.4 may be of great help in early estimation of the impact of new policies. What is missing is an optimization strategy for helping the system engineer focus on promising policies and to optimally tune the best candidates.

6.6 CONCLUSION

In this chapter we studied power management from a system perspective. We moved from the observation that all power management techniques presented in the previous chapters fail to address the formidable complexity of large electronic systems. Nevertheless, power management has been recognized by the industrial world as a key technique to reduce system power consumption. Standardization initiatives have been proposed by market-leading companies to help designers rapidly implement effective power management schemes at the system level. In the area of computer systems, the new paradigm of OS-based power management is emerging. The global control function of the OS makes it ideally suited to make decisions on how and when to shut down the system or some of its components, trying to minimize power without compromising performance or functional correctness.

In a more general setting, the abstract concept of a *power management policy* has been introduced. A power management policy is an algorithm that selectively shuts down idle resources based on the observation of present and past workload and operating conditions. We have described a system-level *stochastic simulation* approach for estimating the impact on the power and performance of a policy. Stochastic simulation leverages an event-driven simulation engine and aggressive functional abstraction to achieve high efficiency. The uncertainty introduced by the high abstraction level in the system description is modeled by non-determinism.

Industry-promoted standards have created the implementation framework for power management policies, and stochastic simulation provides rapid evaluation of power management policy choices. Policy optimization is the final challenge. Power management policies are currently based on simple heuristics and designer intuition. Recent investigations have shown that "reasonable" policies may produce widely-varying results. Hence, there is a strong need for a rigorous approach to policy optimization. While we conjecture that algorithmic design of policies will remain based on human intuition and ingenuity, we believe that policy tuning and search of optimal algorithms in a restricted domain can be at least partially automated. The formulation of a rigorous theory and optimization techniques for power management policies is an exciting area for future research.

7 CONCLUSION

The important thing is not to stop questioning.

—Albert Einstein

211

7.1 SUMMARY

From the beginning of the electronic age, designers have been used to think of performance as the primary goal. For more than thirty years, performance optimization has been extensively studied at all abstraction levels. Technology innovation has been driven by the need for faster processors, faster memory, faster communication. The performance-centric perspective that has ruled the world of electronics is slowly giving the way to a dualistic view, where performance and power dissipation have similar relevance and often require antithetical choices.

Power dissipation has become a major concern because of the ever-increasing density of solid-state electronic devices, coupled with a paradigm shift toward portable communication and mobile computing. As this trend continues, power becomes a critical cost metric for increasingly large classes of digital systems. Designers and researchers have responded to the new challenge by developing numerous technological innovations and optimization techniques to reduce power. One of the most successful approaches to power minimization is dynamic power management.

Dynamic power management is a technique that spans multiple level of abstraction (from the logic level to the system level). We first introduced the fundamental idea that motivates all power management techniques, namely *idleness exploitation.* We defined two types of idleness. External idleness is a condition of unobservability. The output of a component is not observed by its environment, or, equivalently the unit is not involved in the computation performed by the system. Internal idleness is a condition for which the input of a component can change and the output are observable, but they do not change. Both internal and external idleness depend on the operating conditions. A unit that is *always* internally or externally idle, is completely redundant and can be eliminated. Clearly, in well-designed circuits no unit is completely redundant, but many are unused for a large fraction of the operation time. A power-managed unit is designed to dissipate minimum power when it is idle.

In Chapter 3 we described several basic techniques for designing units that support power management. From an abstract point of view, power-managed units operate in several different *states.* Transitioning from one state to another should be as fast and as power-efficient as possible. Power management circuitry must provide effective support to power state changes. Ideally, it should be possible to transition to a fully active state to a zero-power sleep state instantaneously and without power penalty, as soon as the unit becomes idle. Unfortunately, implementing a power-managed resources may incur a sizable overhead. First, the power-management circuitry is functionally redundant and dissipates power, occupies area and may limit speed. Moreover, transitioning

from fully-functional to sleep state may require to save the resource's state. This takes time and power. Additional penalty is paid for restoring the saved state when the resource goes back to fully active.

Some of the schemes illustrated in Chapter 3 have relatively low overhead. In particular, clock gating has several desirable features. First, transitions from active (clock enabled) to sleep (clock stopped) state are extremely fast, since they are completed in a single clock cycle. Second, internal state is preserved (assuming a static CMOS implementation). Finally, the interface circuitry needed to control clock gating in very compact. On the other hand, clock gating is particularly suited for digital, static and synchronous CMOS. Units implemented in different technologies may not be as suited for this power management scheme. Moreover, clock gating does not completely eliminate power dissipation, and more aggressive shutdown schemes (such as power supply shutdown) can be exploited for units that are idle for long time intervals.

In general, there is a tradeoff between how much power dissipation is reduced in the sleep state and how long it takes to resume activity when exiting the sleep state. Mainly for this reasons, multiple sleep states can be implemented. Depending on the usage pattern, the power-managed unit can be put in different sleep states, in an attempt to minimize power without degrading too much the responsiveness of the resource to wake-up events.

Chapter 4 and Chapter 5 deal with the problem of designing power-managed units. The techniques presented in these chapters are based on the simple concepts introduced in Chapter 3, but they attempt to automate the task of idleness detection and the construction of power management circuitry. In Chapter 4 medium-scale control circuits are targeted, while the approaches presented in Chapter 5 are applicable to large circuits of general nature.

Although the automatic optimization techniques presented in Chapter 5 can be successfully applied to large sub-units in chip-level designs, the scope of dynamic power management is much wider. Chapter 6 deals with power management at the system level. In this area, only preliminary research results have been reported. Interestingly, the main thrust in system-level power management does not come from academia but from industry: standardization initiatives are being actively supported by industry leaders in both hardware and software for personal computers (e.g., portables, desktops, servers). This effort shows that dynamic power management is being recognized as a value-added technology for current and future personal computers.

With the adoption of the ACPI standard by the major players in the PC arena, an explosive expansion of the market for power-manageable devices has to be expected in the near future. Hence, the design and synthesis techniques presented in Chapters 3-5 will become strategic for rapidly and effectively designing the new generation of low-power chips.

7.2 PROSPECTS

Design for low power consumption is still a young discipline. In the last few years, an impressive amount of research results has been reported and many contrasting claims on the impact and potentials of power optimization techniques have been made. It is difficult for circuit designers and system architects to select the most effective techniques and to adapt them to the severe requirements of industry practice. The industry-promoted effort to support dynamic power management seems to prove that this technique has the features that make it viable and attractive for a wide range of applications. Although extensive research on power management has been carried out in the recent past, none of the available results are applicable to large systems.

System-level power management is the new frontier of research and development. While in the past the main effort has been to *reduce* the power dissipation of hardware, recent developments clearly indicate a new viewpoint. Low-power systems still need hardware support (through power-manageable components), but can be controlled by software. Hardware resources should be able to provide uncompromised peak performance when needed, and reduced power consumption when idle or partially used. New and unexplored degrees of freedom become available when dynamic power management is supported at the system level. Software-controlled power management is a completely new research area that holds great promise.

[alid94] M. Alidina, J. Monteiro et al., "Precomputation-based sequential logic optimization for low power," *IEEE Transactions on VLSI Systems*, vol. 2, no. 4, pp. 426–436, Jan. 19

[asha91] P. Ashar, S. Devadas and A. Newton, *Sequential logic synthesis.* Kluwer, 1991.

[atha94] W. C. Athas, L. J. Svensson et al., "Low-power digital systems based on adiabatic-switching principles," *IEEE Transactions on VLSI Systems*, vol. 2, no. 4, pp. 398–407, Dec. 1994.

[baha93] R. I. Bahar, E. A. Frohm et al., "Algebraic decision diagrams and their applications," in *Proceedings of the International Conference on Computer Aided Design*, pp. 188-191, Nov. 1993.

[baha95] R. I. Bahar and F. Somenzi, "Boolean techniques for low-power driven re-synthesis," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 428–432, Nov. 1995.

[beni97c] L. Benini, "Automatic synthesis of sequential circuits for low power dissipation," *Stanford University Ph.D. Dissertation No. CSL-TR-97-717*, available at *http://elib.stanford.edu.*

[beni96b] L. Benini, P. Vuillod, C. Coelho and G. De Micheli, "Synthesis of low-power selectively-clocked systems from high-level specification," in *IEEE International Symposium on System Synthesis*, pp. 57–62, Oct. 1996.

[beni95] L. Benini and G. De Micheli, 'State assignment for low-power dissipation," *IEEE Journal of Solid State Circuits*, vol. 30, no. 3, pp. 258–268, March 1995.

[beni96] L. Benini and G. De Micheli, "Automatic synthesis of low-power gated-clock finite-state machines," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 6, pp. 630–643, June. 1996.

[beni97a] L. Benini, G. De Micheli, E. Macii, M. Poncino and R. Scarsi, "Symbolic techniques for power optimization of large control-oriented synchronous networks," in *IEEE European Design and Test Conference*, pp. 514–520, March 1997.

[beni97b] L. Benini et al., "Asymptotic zero-transition activity encoding for address busses in low-power microprocessor-based systems," in *IEEE Great Lakes Symposium on VLSI*, pp. 77–82, March 1997.

[beni94a] L. Benini, P. Siegel and G. De Micheli, "Automatic synthesis of gated clocks for power reduction in sequential circuits," *IEEE Design and Test of Computers*, pp. 32–40, Dic. 1994.

[benn88] C. H. Bennet, "Notes on the history of reversible computation," *IBM Journal of Research and Development*, vol. 32, no. 1, pp. 16, 1988.

[berg95] R. Bergamaschi, R. O'Connor et al., "High-level synthesis in an industrial environment," *IBM Journal of Research and Development*, vol. 39, no. 1-2, pp. 131–148, 1995.

[birt95] G. M. Birtwistle and A. Davis (editors), *Asynchronous digital circuit design* Springer-Verlag, 1995.

[bogl97] A. Bogliolo, L. Benini and G. De Micheli, "Adaptive least mean square behavioral power modeling," *IEEE European Design and Test Conference*, pp. 404–410, March 1997.

[bogl96] A. Bogliolo, L. Benini and B. Riccò, "Power estimation of cell-based CMOS circuits," in *Proceedings of the Design Automation Conference*, pp. 433–438, June 1996.

[alpha96] G. Bouchard, "Design objectives of the $0.35\mu m$ Alpha 21164 microprocessor," in *Hot chips symposium*, Aug. 1996.

[brow90] F. M. Brown, *Boolean reasoning.* Kluwer, 1990.

[brac90] K. S. Brace, R. Rudell, R. Bryant, "Efficient implementation of a BDD package," in *Proceedings of the Design Automation Conference*, pp. 40–45, June 1990.

[brgl89] F. Brglez, D. Bryan and K. Koźmiński, "Combinational profiles of sequential benchmark circuits," *ISCAS-89: Intl. Symp. on Circuits and Systems*, pp. 1929-1934, May 1989.

[brya86] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, Vol. C-35, No. 8, pp. 79–85, August 1986.

[bult96] K. Bult, A. Burstein et al., "Low power systems for wireless microsensors," in *International Symposium on Low Power Electronic and Design*, pp. 17–21, Aug. 1996.

[burc93] R. Burch, F. Najm, P. Yang, T. Trick, "A Monte-Carlo approach for power estimation," *IEEE Transactions on VLSI Systems*, Vol. 1, No. 1, pp. 63-71, January 1993.

[burd94] T. Burd, "Low-power CMOS library design methodology," *M.S. Report, University of California, Berkeley, UCB/ERL M94/89*, 1994.

[burd95] T. Burd and R. Brodersen, "Energy efficient CMOS microprocessor design," in *Proceedings of the Hawaii International Conference on System Sciences*, vol. 1, pp. 288-97, Jan. 1995.

[came85] N. Cameron, *Introduction to linear and convex programming.* Cambridge, 1985.

[chan95] A. Chandrakasan and R. Brodersen, *Low power digital CMOS design.* Kluwer, 1995.

[chan95b] A. Chandrakasan, M. Potkonjak et al., "Optimizing power using transformations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 1, pp. 12–31, Jan. 1995.

[chang95] J. Chang and M. Pedram, "Register allocation and binding for low power," in *Proceedings of the Design Automation Conference*, pp. 29–35, June 1995.

[cho93] H. Cho, G. Hachtel and F. Somenzi, "Redundancy identification/removal and test generation for sequential circuits using implicit state enumeration," *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, vol. 12, no. 7, pp. 935–45, July 1993.

[chow96] S. H. Chow, Y. C. Ho et al., "Low power realization of finite state machines - A decomposition approach," in *ACM Transactions on Design Automation of Electronic Systems*, vol. 1, no. 3, July 1996.

[chun95] J. W. Chung, D.-Y. Kao et al., "Optimization of power dissipation and skew sensitivity in clock buffer synthesis," in *International Symposium on Low-Power Design*, pp. 179–184, March 1995.

[clar93] E. M. Clarke, M. Fujita et al., "Multi-terminal binary decision diagrams: an efficient data structure for matrix representation," in *International Workshop on Logic Synthesis*, pp. 6a:1-15, May 1993.

[cong97] J. Cong, L. He and C.-K. Koh, "Layout optimization," in W. Nebel and J. Mermet (Eds.), *Low power design in deep submicron electronics.* Kluwer, 1997.

[coud89] O. Coudert, C. Berthet, J. C. Madre, "Verification of sequential machines using Boolean functional vectors," in *IFIP Intl. Workshop on Applied Formal Methods for Correct VLSI Design*, pp. 111–128, Nov. 1989.

[coud96] O. Coudert, R. Haddad and S. Manne, "New algorithms for gate sizing: a comparative study," in *Proceedings of the Design Automation Conference*, pp. 734–739, June 1996.

[dasg95] A. Dasgupta and R. Karri, "Simultaneous scheduling and binding for power minimization during microarchitecture synthesis," in *International Symposium on Low Power Design*, pp. 69–74, April 1995.

[davi78] M. Davio, J. P. Deschamps and A. Thayse, *Discrete and switching functions.* McGraw-Hill, 1978.

[debn95] G. Debnath, K. Debnath and R. Fernando, "The Pentium processor-90/100, microarchitecture and low power circuit design," in *International conference on VLSI design*, pp. 185–190, Jan. 1995.

[dmc94] G. De Micheli, *Synthesis and optimization of digital circuits.* McGraw-Hill, 1994.

[denk94] J. S. Denker, "A review of adiabatic computing," in *Symposium on Low Power Electronics*, pp. 10–12, Oct. 1994.

[deva95] S. Devadas and S. Malik, "A survey of optimization techniques targeting low power VLSI circuits," in *Proceedings of the Design Automation Conference*, pp. 242-247, June 1995.

[doug95] F. Douglis, P. Krishnan and B. Bershad, "Adaptive disk spin-down policies for mobile computers," in *Proceedings of the Second USENIX Symposium on Mobile and Location-Independent Computing*, pp. 121–137, April 1995.

[elli91] S. C. Ellis, "Power management in notebook computers," in *Proceedings of the Personal Computer Design Conference*, pp. 749--754, July 1991.

[expo96] Exponential Corporation, "Exponential X^{704} microprocessor," *Press release*, Oct. 1996.

[farr95] A. H. Farrahi, G. E. Tellez and M. Sarrafzadeh, "Memory segmentation to exploit sleep mode operation," *Proceedings of the Design Automation Conference*, pp. 36–41, June. 1995.

[fava96] M. Favalli, L. Benini, G. De Micheli, "Design for testability of gated-clock FSMs," in *Proceedings of the European Design and Test Conference*, pp. 589-596, Mar. 1996.

[gars96] J. Garside, "Amulet2e," in *Hot chips symposium*, Aug. 1996.

[geig91] M. Geiger and T. Muller-Wippefurth, "FSM decomposition revisited: algebraic structure theory applied do MCNC benchmark FSMs," in *Proceedings of the Design Automation Conference*, pp. 182–185, June 1992.

[gebo97] C. H. Gebotys, "Low energy memory and register allocation using network flow," in *Proceedings of the Design Automation Conference*, pp. 435–440, June 1997.

[gold89] D. Goldberg, *Genetic Algorithms in search, optimization and machine learning*, Addison-Wesley, 1989.

[gold95] R. Golding, P. Bosh et al, "Idleness is not sloth," in *Proceedings of Winter USENIX Technical Conference*, pp.201–212, Jan. 1995.

[gref90] J. J. Grefenstette, *A user's guide to GENESIS*, 1990.

[guer94] L. Guerra, M. Potkonjak, J. Rabaey, "System-level design guidance using algorithm properties," in *IEEE VLSI Signal Processing VII* pp. 73 - 82, 1994.

[hahe94] G. Hachtel, M. Hermida et al., "Re-encoding sequential circuits to reduce power dissipation," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 70–73, Nov. 1994.

[hama94] G. Hachtel, E. Macii et al., "Symbolic algorithms to calculate steady-state probabilities of a finite state machine," in *Proceedings of IEEE European Design and Test Conference*, pp. 214–218, Feb. 1994.

[hach94] G. Hachtel, E. Macii et al., "Probabilistic analysis of large finite state machines," in *Proceedings of the Design Automation Conference*, pp. 270-275, June 1994.

[harr95] E. P. Harris, S. W. Depp et al., "Technology directions for portable computers," *Proceedings of the IEEE*, vol. 83, no. 4, pp. 636–658, April 1995.

[hart66] J. Hartmanis and H. Stearns, *Algebraic Structure Theory of Sequential Machines.* Prentice-Hall, 1966.

[hasa95] Z. Hasan and M. Ciesielski, "FSM decomposition and functional verification of FSM networks," *VLSI Design*, vol. 3, no. 3–4, pp. 249–65.

[henn68] F. C. Hennie, *Finite-State models for logical Machines.* Wiley, 1968.

[hong96] I. Hong and M. Potkonjak, "Power optimization in disk-based real-time application specific systems," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 634–637, Nov. 1996.

[hsie96] C-T. Hsieh, C-S. Ding et al., "Statistical sampling and regression estimation in power macromodeling," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 583–588, Nov. 1996.

[hsie97] C-T. Hsieh, M. Pedram et al., "Profile-driven program synthesis for evaluation of system power dissipation," in *Proceedings of the Design Automation Conference*, pp. 576–581, June 1997.

[huan95] C. X. Huang et al., "The design and implementation of PowerMill," in *International Symposium on Low Power Design*, pp. 105–110, April 1995.

[kapa97] H. Kapadia, *Private communication*, 1997.

[keut96] K. Keutzer, "What is the state of the art in commercial CAD tools for low power?" in *International Symposium on Low Power Electronics and Design*, pp. 101–107, Aug. 1996.

[kiro97] D. Kirovski and M. Potkonjak, "System-level synthesis of low-power hard real-time systems," in *Proceedings of the Design Automation Conference*, pp. 697–702, June 1997.

[knapp95] D. Knapp, T. Ly et al., "Behavioral synthesis methodology for HDL-based specification and validation," in *Proceedings of the Design Automation Conference*, pp. 286–291, June 1995.

[koha70] Z. Kohavi, *Switching and Finite automata theory.* McGraw-Hill, 1970.

[ku92] D. Ku and G. De Micheli, *High-Level synthesis of ASICs under timing and synchronization constraints.* Kluwer Academic Publishers, 1992.

[kuma95] N. Kumar, S. Katkoori et al., "Profile-driven behavioral sythesis for low-power VLSI systems," *IEEE Design & Test of Computers*, vol. 12, no. 3, pp. 70–84, Fall 1995.

[kuo95] M. Kuo, L. Liu and C Cheng, "Finite-State Machine decomposition for I/O minimization," in *IEEE International Symposium on Circuits and Systems*, pp. 1061–1064, April 1995.

[iman96] S. Iman and M. Pedram, "POSE: Power optimization and synthesis environment," in *Proceedings of the Design Automation Conference*, pp. 21–26, June 1996.

[iman97] S. Iman and M. Pedram, "Combinational circuit optimization," in W. Nebel and J. Mermet (Eds.), *Low power design in deep submicron electronics*. Kluwer, 1997.

[inde94] T. Indermaur and M. Horowitz, "Evaluation of charge recovery circuits and adiabatic switching for low power CMOS design," in *Symposium on Low Power Electronics*, pp. 102-103, Oct. 1994.

[appl97] Intel, "Recommendations to Windows 95 application/driver developers for writing power friendly software," available at *http://www.intel.com/ial/ipm/w95app.htm*, 1997.

[smba97] Intel, Duracell, "Smart Battery System specification," available at *http://www.sbs-forum.org/specs.htm*, 1997.

[ACPI96] Intel, Microsoft and Toshiba,
"Advanced Configuration and Power Interface specification", available at *http://www.intel.com/ial/powermgm/specs.html*, Dec. 1996.

[lai94] Y. T. Lai, M. Pedram, S. B. K. Vrudhula, "EVBDD-Based algorithms for integer linear programming, spectral transformation, and function decomposition," *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, Vol. 13, No. 8, pp. 959-975, Aug. 1994.

[land95] P. Landman and J. Rabaey, "Architectural power analysis, the Dual Bit Type method," *IEEE Transactions on VLSI Systems*, vol. 3, no. 2, pp. 173–187, 1995.

[land96] P. E. Landman, R. Mehra and J. Rabaey, "An integrated CAD environment for low-power design," *IEEE Design & Test of Computers* vol. 13, no. 2, pp. 72–82, Summer 1996.

[lee95] M. T.-C. Lee et al. V. Tiwari et al., " Power analysis and low-power scheduling techniques for embedded DSP software," *International Symposium on System Synthesis*, pp. 110-115, Sept. 1995.

[lids96] D. Lidsky and J. Rabaey, "Early power exploration - A World Wide Web application," in *Proceedings of the Design Automation Conference* pp. 27–32, June 1996.

[lin89] B. Lin and A. R. Newton, "Synthesis of multiple-level logic from symbolic high-level description languag es," in *Proceedings of the International Conference On Computer Design*, pp. 187–196, Aug. 1989.

[madr88] J. C. Madre, J. P. Billon, "Proving circuit correctness using formal comparison between expected and extracted behavior," in *Proceedings of the Design Automation Conference*, pp. 205-210, June 1988.

[mail91] F. Mailhot and G. De Micheli, "Algorithms for technology mapping based on binary decision diagrams and on Boolean operations," *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, pp. 599–620, May 1993.

[mang95] B. Mangione-Smith, "Low power communication protocols: Paging and beyond," in *Symposium on Low Power Electronics*, pp. 8–11, Oct. 1995.

[marc94] R. Marculescu, D. Marculescu and M. Pedram, "Switching activity analysis considering spatiotemporal correlations," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 294–299, Nov. 1994

[mars94] A. Marshall, B. Coates and P. Siegel, "Designing an asynchronous communication chip," *IEEE Design & Test of Computers*, vol. 11, no. 2, pp. 8–21, Summer 1994.

[mart96] T. L. Martin and D. P. Siewiorek, "A power metric for mobile systems," in *International Symposium on Low Power Electronics and Design*, pp. 37–42, Aug. 1996.

[mccl86] E. McCluskey. *Logic design principles.* Prentice-Hall, 1986.

[mehr96] R. Mehra, L. Guerra et al., "Exploiting locality for low-power design," in *Proceedings of the Custom Integrated Circuits Conference*, pp. 401–406, May 1996.

[meht96] H. Mehta, R. M. Owens, and M. J. Irwin, "Energy characterization based on clustering," in *Proceedings of the Design Automation Conf.*, pp. 702–707, 1996.

[meh96a] H. Mehta, R. M. Owens, M. J. Irwin, "Some issues in gray code addressing," in *IEEE Great Lakes Symposium on VLSI*, pp. 178–180, March 1996.

[mein95] J. Meindl, "Low power microelectronics: retrospect and prospect," *Proceedings of the IEEE*, vol. 83, no. 4, pp. 619–634, April 1995.

[meng95] T. H. Meng, B. Gordon et al., "Portable Video-on-Demand in wireless communication," *Proceedings of the IEEE*, vol. 83, no. 4, pp. 659–680, April 1995.

[onno97a] Microsoft, "OnNow: the evolution of the PC platform," available at *http://www.microsoft.com/hwdev/pcfuture/ONNOW.HTM*, Aug. 1997.

[onno97b] Microsoft, "OnNow: device power management," available at *http://www.microsoft.com/hwdev/pcfuture/DEVICEPM.HTM*, Jan. 1997.

[onno97c] Microsoft, "OnNow power management architecture for applications," available at *http://www.microsoft.com/hwdev/pcfuture/onnowapp.HTM*, Aug. 1997.

[mina90] S. I. Minato, N. Ishiura, S. Yajima, "Shared binary decision diagram with attributed edges for efficient Boolean function manipulation," in *Proceedings of the Design Automation Conference*, pp. 52-57, June 1990.

[mont97] J. Monteiro and S. Devadas, *Computer-aided techniques for low power sequential logic circuits*. Kluwer 1997.

[mont96] J. Monteiro, S. Devadas, P. Ashar and A. Mauskar, "Scheduling techniques to enable power management," in *Proceeding of the Design Automation Conference* pp. 349–352, June 1996.

[mont93] J. Monteiro, S. Devadas and A. Ghosh, "Retiming sequential circuits for low power," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 398–402, Nov. 1993.

[mont94] J. Monteiro, S. Devadas and B. Lin, "A methodology for efficient estimation of switching activitiy in sequential logic circuits," in *Proceedings of the Design Automation Conference*, pp. 315–321, June 1994

[moor96] G. Moore, *Computerworld-Smithsonian Monticello Lecture*, Computerworld Leadership Series, May 1996.

[muss95] E. Mussol and J. Cortadella, "High-level synthesis techniques for reducing the activity of functional units," in *International Symposium on Low Power Design*, pp. 99–104, April 1995.

[najm95] F. Najm, "Power estimation techniques for integrated circuits," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 492–499, Nov. 1995.

[nebe97] W. Nebel and J. Mermet (Eds.), *Low power design in deep submicron electronics*. Kluwer, 1997.

[nehm88] G. Nemhauser and L. Wolsey, *Integer and combinatorial optimization*. Wiley, 1988.

[niel94] L. S. Nielsen, C. Niessen et al., "Low-power operation using self-timed circuits and adaptive scaling of the supply voltage," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 4, pp. 391–397, Dec. 1994.

[olso94] E. Olson and S. Kang, "State assignment for low-power synthesis using genetic local search," in *Proceedings of IEEE Custom Integrated Circuits Conference*, pp. 140–143, May 1994.

[pand96] P. R. Panda, N. D. Dutt, "Reducing address bus transitions for low power memory mapping," in *IEEE European Design and test Conference*, pp. 63-67, March 1996.

[raba96] J. M. Rabaey and M. Pedram (editors), *Low power design methodologies*. Kluwer, 1996.

[ragh94] A. Raghunathan and N. K. Jha, "Behavioral synthesis for low power," in *Proceedings of the International Conference on Computer Design*, pp. 318–322, Oct. 1994.

[ragh96a] A. Raghunathan, S. Dey et al., "Controller re-specification to minimize switching activity in controller/data path circuits," in *International Symposium on Low Power Electronics and Design*, pp. 301–304, Aug. 1996.

[ragh96b] A. Raghunathan, S. Dey and N. K. Jha, "Glitch analysis and reduction in register transfer level power optimization," in *Proceedings of the Design Automation Conference*, pp. 331–336, June 1996.

[ragh97] A. Raghunathan, S. Dey et al., "Power management techniques for control-flow intensive designs," in *Proceedings of the Design Automation Conference*, pp. 429–434, June 1997.

[rofl96] R. Roflcich, A. Kobl and B. Wurth, "Reducing power dissipation after technology mapping by structural transformations," in *Proceedings of the Design Automation Conference*, pp. 789–794, June 1996.

[roy93] K. Roy and S. Prasad, "Circuit activity based logic synthesis for low power reliable operations," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 1, no. 4, pp. 503–513, Dec. 1993.

[royn93] S. Roy and H. Narayanan, "Application of the principal partition and principal lattice of partitions of a graph to the problem of decomposition of a finite-state machine," in *IEEE International Symposium on Circuits and Systems*, pp. 2564–2567, May 1993.

[salz89] A. Salz, M. Horowitz, "IRSIM: an incremental MOS switch-level simulator," in *Proceedings of the Design Automation Conference*, pp. 173–178, June 1989.

[saku96] T. Sakurai and T. Kuroda, "Low-power circuit design for multimedia CMOS VLSI," in *Workshop on Synthesis and System Integration of Mixed Technologies*, pp. 3–10, Nov. 1996.

[sanm96] R. San Martin and J. P. Knight, "Optimizing power in ASIC behavioral synthesis," *IEEE Design & Test of Computers*, vol. 13, no. 2, pp. 58–70, Summer 1996.

[sema97] SEMATECH, Inc., "The national technology roadmap for semiconductors," *http:www.sematech.org*, 1997.

[sent92] E. Sentovich et al., "Sequential circuit design using synthesis and optimization," in *Proceedings of the International Conference on Computer Design*, pp. 328–333, Oct. 1992.

[slat95] J. Slaton, S. P. Licht et al., "The PowerPC 603e microprocessor: an enhanced, low-power, superscalar microprocessor," in *Proceedings of the International Conference on Computer Design*, pp. 192–203, Oct. 1995.

[some96] F. Somenzi and G. D. Hachtel, *Logic synthesis and verification algorithms*. Kluwer, 1996.

[stan95] M. R. Stan, W. P. Burleson, "Bus-invert coding for low-power I/O," *IEEE Transactions on VLSI Systems*, Vol. 3, No. 1, pp. 49–58, March 1995.

[stra94] A. J. Stratakos, S. R. Sanders and R. Brodersen, "A low-voltage CMOS DC-DC converter for a portable battery-operated system," in *Proceedings of the Power Electronics Specialists Conference*, vol. 1, pp. 619–26, June 1994.

[sues94] B. Suessmith and G. Paap III, "PowerPC 603 microprocessor power management," *Communications of the ACM*, no. 6, pp. 43–46, June 1994.

[telle95] G. Tellez, A. Farrahi and M. Sarrafzadeh, "Activity-driven clock design for low power circuits," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 62–65, Nov. 1995.

[thee96] F. Theeuven and E. Seelen, "Power reduction through clock gating by symbolic manipulation," in *Symposium on Logic and Architecture Design*, pp. 184–191, Dec. 1996.

[tiwa93] V. Tiwari, P. Ashar and S. Malik, "Technology mapping for low power", in *Proceedings of the Design Automation Conference*, pp. 74–79, June 1993.

[tiwa94] V. Tiwari, S. Malik and A. Wolfe, "Power analysis of embedded software: a first step towards software power minimization," *IEEE Transactions on VLSI Systems*, vol. 2, no. 4, pp. 437–445, Dec. 1994.

[tiwa95] V. Tiwari, S. Malik and P. Ashar, "Guarded evaluation: pushing power management to logic synthesis/design," in *International Symposium on Low Power Design*, pp. 221–226, April 1995.

[triv82] K. Trivedi, *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*, Prentice-Hall, 1982.

[tsui93] C. Tsui, M. Pedram, and A. Despain, "Technology decomposition and mapping targeting low power dissipation," in *Proceedings of the Design Automation Conference*, pp. 68–73, 1993.

[tsui94] C. Tsui, M. Pedram and A. Despain, "Low-Power state assignment targeting Two-and Multi-level logic implementation," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 82–27, Nov. 1994.

[tsui94b] C. Y. Tsui, M. Pedram and A. M. Despain, "Exact and approximate methods for calculating signal and transition probabilities in FSMs," in *Proceedings of the Design Automation Conference*, pp. 18–25, June 1994.

[uda96] S. Udani and J. Smith, "The power broker: intelligent power management for mobile computing," *Technical report MS-CIS-96-12, Dept. of Computer Information Science, University of Pennsylvania*, May 1996.

[usam95] K. Usami and M. Horowitz, "Clustered voltage scaling technique for low-power design," in *International Symposium on Low Power Design*, pp. 3–8, Aprim 1995.

[veen84] H. J. Veendrick, "Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits," *Journal of Solid-State Circuits*, vol. SC-19, no. 4, pp. 468–473, Aug. 1984.

[vitt94] E. Vittoz, "Low-Power design: Ways to approach the limits," in *Proceedings of the IEEE Solid-State Circuits Conference*, pp. 14–18.

[vuill97] P. Vuillod, L. Benini and G. De Micheli, "Re-mapping for low power under tight timing constraints," in *International Symposium on Low Power Electronics and Design*, pp. 287–292, Aug. 1997.

[west92] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design (Second Edition).* Addison-Wesley, 1992.

[wuyt96] "Transforming set data types to power optimal data structures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 6, pp. 619–629, June 1996.

[mcnc91] S. Yang, "Logic synthesis and optimization benchmarks user guide. Version 3.0," *Technical report, Microelectronics Center of North Carolina*, Jan. 1991.

[jeon92] S-W. Yeong and F. Somenzi. "A new algorithm for 0-1 programming based on binary decision diagrams," in *International Workshop on Logic Synthesis*, pp. 177–184, 1992.

[yeun94] N. Yeung et al., "The design of a 55SPECin92 RISC processor under 2W," in *Proceedings of the IEEE Solid-State Circuits Conference*, pp. 206–207, Feb. 1994.

[yuan89] J. Yuan and C. Svensson, "High-speed CMOS circuit technique," *IEEE Journal of Solid-State Circuits*, pp. 62-70, Feb.1989.

Index

ACPI, 169
  battery, 191
  BIOS, 183
  event, 190–191
  machine language, 183
  register, 183
  thermal management, 192
Activation function, 89, 93, 131, 140
Activity matrix, 31
Adiabatic computation, 19
Algebraic decision diagram, 54
Algorithmic transformations, 17
Application programming interface, 172, 178
Asynchronous circuits, 18
Awake time, 170
Battery lifetime, 4, 9, 168
Behavioral level, 124
Behavioral transformations, 32
Binary decision diagrams, 51
BIOS, 171
Boolean algebra, 42
Bus encoding, 36
Chipset, 181
Clock tree optimization, 23
Clock, 8
  distribution, 126
  frequency, 6
  period, 6
  skew, 126
CMOS technology, 2, 4
Communication protocols, 17
Computational unit, 66
Computer-aided design, 3
Computer-aided power optimization
  behavioral-level techniques, 30
  circuit-level techniques, 21
  logic-level techniques, 25
  system-level techniques, 33
Conditional probability, 57
Conditional signal, 144
Consensus, 44
Control signal, 144
Controller re-specification, 159
Cooling, 192
Cover, 46
Data-path, 149
Decomposition, 102
Device class, 174
Device driver, 174
  ACPI-compliant, 181
  bus, 180, 188
  motherboard, 180
Discrete function, 47
Disk power management, 206
Don't cares, 43, 97, 149
Dynamic power management, 37
Embedded controller, 185
Enabled flip-flop, 81
Energy, 6
External idleness, 68, 138, 148
Finite-state machine, 47
  Decomposed, 102
  Mealy, 47, 89
  Monolithic, 102
  Moore, 47, 89
Flip-flop, 49
Floor-planning, 128
Frequency divider, 129

Gate Resizing, 28
Gated clock, 78, 87–88, 108, 128, 130, 147
Glitch, 16
  minimization, 152, 154
Global power policy, 173, 176, 179
Granularity, 73
H-tree, 126
Hold condition, 144
Idleness, 4, 37
Implicant, 45
Input reordering, 24
Input statistics, 75
Input/output request packet, 175
Integer programming, 110
Interacting FSM structure, 104
Internal idleness, 70, 131, 144
Laptop computer, 168
Legacy state, 187
Locally-moore machine, 91
Logic network, 46
Markov chain, 57, 90
Memory
  allocation, 17
  element, 160
  segmentation, 160
Microprocessor, 2
Moore's Law, 2
Multi-tasking, 171
Multiplexer, 68, 70, 148
Observability, 69
OnNow, 169
Operand isolation, 156
Operating system, 169
OSPM, 180
Partition, 104, 110
Performance model, 201
Pipelined circuit, 130, 141
Placement, 24
Policy optimization, 169, 205
Policy owner, 174
Polling cycle, 171
Power consumption
  average, 4
  dynamic, 5
  leakage, 7
  peak, 4
  short circuit, 6
  static, 8
Power estimation
  instruction level, 36
  behavioral level, 32
  circuit level, 21
  logic level, 26
  Monte-Carlo, 22
  register-transfer level, 30
  system level, 35
Power minimization, 8
  capacitance reduction, 14
  frequency reduction, 9
  switching activity reduction, 16
  voltage scaling, 10
Power states
  device, 185
  processor, 186
  system, 173, 184
Power supply, 5, 7, 14
  shutdown, 17, 76
  voltage drops, 4
Precomputation, 139
Predictor, 207
Probabilistic simulation, 26
Pull-down, 5–6
Pull-up, 5–6
Re-mapping, 28
Reactive behavior, 72
Redundancy removal, 136
Register, 144, 148
Register-transfer level, 30, 124
Resource binding, 31
Retiming, 29
Scheduling, 31, 162
Self loop, 87, 132
Signal probability, 56
Skeptic, 207
Sleep, 175, 178, 183–184, 198, 207
Smart battery, 191
Smoothing, 44
Software power minimization, 36
Spatial correlation, 56
Speculative execution, 69
State assignment, 28
State encoding, 48, 90
State probability, 58
State transition graph, 47
Steering module, 148
Stochastic process, 200
Stochastic simulation, 202
Sub-machine, 106
Supply voltage, 2, 5, 8
  and delay, 11
  variable, 35
Synchronous network, 29, 49, 91
System-level model, 196
Task scheduling, 35

Technology mapping, 27
Technology-independent optimization, 27
Temporal correlation, 56
Testability, 80
Thermal management, 192
Three-state driver, 148
Timer-based policy, 209
Timing analysis, 80
Top-down, 87
Transistor reordering, 24
Transistor sizing, 23
Transition activity, 6, 31
  functional, 16
  spurious, 16, 32, 38
Turn-on delay, 170
Wake-up, 170, 175, 179, 187, 189, 198