Precomputation-Based Sequential Logic Optimization for Low Power

by

Mazhar Murtaza Alidina

B.S.E.E., Lehigh University (1992)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

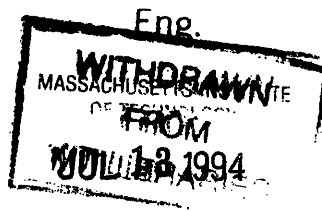
May 1994

© Massachusetts Institute of Technology 1994. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
May 6, 1994

Certified by ....
Srinivas Devadas
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Frederic R. Morganthaler
Chairman, Department Committee on Graduate Students


Eng.
WITHDRAWN FROM
JUL 13 1994

Precomputation-Based Sequential Logic Optimization for Low Power

by

Mazhar Murtaza Alidina

Submitted to the Department of Electrical Engineering and Computer Science
on May 6, 1994, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

In this thesis, we address the problem of optimizing sequential logic circuits for low power. We present a powerful optimization method that selectively *precomputes* the outputs of the circuit one clock cycle before they are required and uses the precomputed values to reduce switching activity in the next clock cycle. We present different precomputation architectures that exploit this observation.

The primary optimization step is the synthesis of the precomputation logic, which computes the output values of the circuit for a *subset* of input conditions. If the output values can be precomputed, the original logic circuit can be "turned off" in the next clock cycle and, thus, has substantially reduced switching activity. The size of the precomputation logic determines the power dissipation reduction, area increase and delay increase relative to the original circuit.

Given a sequential logic circuit, we present an automatic method of synthesizing precomputation logic so as to achieve maximum reductions in power dissipation. We present experimental results on various sequential circuits. Up to 60 percent reductions in power dissipation are possible with marginal increases in circuit area and delay.

Thesis Supervisor: Srinivas Devadas
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

Although this work bears my name, it is a result of the efforts and contributions of several people. First and foremost, I would like to thank Professor Srinivas Devadas, my thesis supervisor, for his original contributions and help throughout this project. His patience, willingness to discuss ideas, and enthusiasm have made it a unique and enjoyable learning experience for me.

I would also like to thank Dr. Abhijit Ghosh of Mitsubishi Electric Research Laboratories and Professor Marios Papaefthymiou of Yale University for their contributions to this work.

Thanks to José Monteiro for his work in power estimation, his help in implementing this project, and all the interesting discussions on low power design. He has been a good friend and a source of inspiration.

I am also grateful to Stan Liao and Amelia Shen for helping me learn the synthesis tools, and to Luis Miguel Silveira for his assistance in customizing this document. In addition, I thank all members of the 8th floor VLSI group especially Xuejun Cai, Songmin Kim, and Ricardo Telichevesky for their encouragement.

Finally, I would like to thank my parents and my two sisters, Ateka and Noren, for their unconditional love and support.

The work described in this thesis was done in the Research Laboratory of Electronics at the Massachusetts Institute of Technology and was supported in part by the Defense Advanced Research Projects Agency under contract N00014-91-J-1698 and in part by a NSF Young Investigator Award with matching funds from Mitsubishi and IBM Corporation.

Contents

1 Introduction **10**

2 Preliminaries **13**

2.1 Introduction 13

2.2 Definitions 13

2.3 Binary Decision Diagrams 14

3 Power Estimation **17**

3.1 Introduction 17

3.2 A Power Dissipation Model 17

3.3 Power Estimation of Combinational Circuits 18

3.3.1 Estimating Switching Activity 19

3.3.2 Symbolic Simulation 19

3.4 Power Estimation of Sequential Circuits 20

3.4.1 Modeling Correlation 21

3.4.2 State Probability Computation 22

3.4.3 Power Estimation Given Exact State Probabilities 24

4 Precomputation Architectures **26**

4.1 Introduction 26

4.2 The Predictor Functions 26

4.3 First Precomputation Architecture 27

4.4 Second Precomputation Architecture 28
4.5 Multiple-Output Functions 30
4.6 Examples 31
4.7 Testability of Precomputation-Based Logic Circuits 33

5 Synthesis of Precomputation Logic **35**
5.1 Introduction 35
5.2 Precomputation and Observability Don't-Cares 36
5.3 Precomputation Logic 36
5.3.1 Selecting a Subset of Inputs 37
5.3.2 Implementing the Logic 39
5.3.3 A Comparator Example 40
5.4 Multiple-Output Functions 41
5.4.1 Selecting a Subset of Outputs 41
5.4.2 Logic Duplication 43

6 Special Cases **46**
6.1 Introduction 46
6.2 Special Circuits 46
6.2.1 An Arithmetic Logic Unit 47
6.2.2 An Array Multiplier 49
6.2.3 A Carry-Select Adder 52
6.2.4 A Maximum Function 53
6.3 Special Architectures 55
6.3.1 Multiplexor-Based Precomputation 55
6.3.2 Combinational Logic Precomputation 56

7 Multiple-Cycle Precomputation **59**
7.1 Introduction 59
7.2 Basic Strategy 59

7.3 Examples 60

8 Experimental Results **64**

8.1 Introduction 64

8.2 Results 64

9 Conclusion and Future Work **67**

List of Figures

2-1 Examples of Ordered Binary Decision Diagrams 15

3-1 A Synchronous Sequential Circuit 21

3-2 Modeling Correlation in a Sequential Circuit 22

3-3 A State Transition Graph 23

4-1 The Original Circuit 27

4-2 First Precomputation Architecture 28

4-3 Second Precomputation Architecture 29

4-4 Precomputation of a Multiple-Output Function 30

4-5 Precomputation of a Comparator Function 31

4-6 Precomputation of a Priority Function 33

4-7 Redundancy in a Precomputation-Based Logic Circuit 34

5-1 Procedure to Determine the Optimal Set of Inputs 38

5-2 A Comparator Example 40

5-3 Procedure to Determine the Optimal Set of Outputs 42

5-4 Logic Duplication in a Multiple-Output Function 44

6-1 Precomputation of an Arithmetic Logic Unit 48

6-2 Partial Products of a Multiplier 49

6-3 The First 3 Stages of an Array Multiplier 50

6-4 Precomputing the ith Stage of an Array Multiplier 51

6-5 A Carry-Select Adder 52
6-6 Precomputation of a Maximum Function 54
6-7 Precomputation Using the Shannon Expansion 56
6-8 Combinational Logic Precomputation 57

7-1 Multiple-Cycle Precomputation 60
7-2 An Add-Compare Function 61
7-3 An Add-Maximum Function 62

List of Tables

4.1 Truth-Table of a Priority Function 32

6.1 Specification of an Arithmetic Logic Unit 47

8.1 Power Reductions for Datapath Circuits 65

8.2 Power Reductions for Random Logic Circuits 66

Chapter 1

Introduction

Average power dissipation has recently emerged as an important parameter in the design of general-purpose as well as application-specific integrated circuits. Portable systems that operate on a battery, such as cellular telephones, personal digital assistants, and laptop computers, are the driving force behind low power electronics. Since these systems are portable, strict requirements are placed on their size, weight, and power. Furthermore, battery lifetime becomes a critical issue as it determines the usefulness of the system, and ultimately its acceptance and success in the mass market. The integrated circuits in battery operated systems, therefore, must efficiently consume power.

In the case of general-purpose circuits, such as microprocessors for personal computers and workstations, power is also becoming a critical factor. With technology dimensions decreasing, more circuits and functionality are being added onto a single chip. In addition, clock frequencies are being increased at staggering rates. These technology trends have a cumulative effect on power dissipation, and in order to continue them, power consumption must be taken into account during design. Furthermore, integrated circuits require more sophisticated packaging if they dissipate large amounts of power. The circuits may also require heat sinks in order to operate efficiently and reliably. All of these combine to increase the most important param-

eter of the product, its cost. The electronics that integrate complex functions and require high throughputs must be carefully designed and optimized for low power in order to be economical and reliable.

Optimization for low power can be applied at many different levels of the design hierarchy. For instance, algorithmic and architectural transformations can trade-off throughput, circuit area, and power dissipation [5]. Logic optimization methods have been shown to have a significant impact on the power dissipation of combinational logic circuits [17]. At the circuit and layout levels, transistors can be sized to improve the power-delay product [18]. In addition, wire and driver sizing can reduce the power consumed by interconnect while maintaining delay constraints [6]. Furthermore, scaling technology parameters such as supply and threshold voltages can substantially reduce power dissipation [5]. To effectively optimize designs for low power, however, accurate power estimation methods must be developed and used.

In *Complementary Metal Oxide Semiconductor* (CMOS) circuits, the probabilistic average switching activity is a good measure of the average power dissipation of the circuit. Several methods to estimate power dissipation for CMOS combinational circuits based on measuring the average switching activity have been developed (e.g. [8, 13]). More recently, efficient and accurate methods of power estimation for sequential circuits have been developed [12].

In this thesis, we are concerned with the problem of optimizing sequential logic circuits for low power. Previous work in the area of sequential logic synthesis for low power has focused on state encoding [15] and retiming [11] algorithms. We present a powerful optimization method that is based on selectively *precomputing* the output values of the circuit one clock cycle before they are required, and using the precomputed values to reduce switching activity in the next clock cycle.

The primary optimization step is the synthesis of the precomputation logic, which computes the outputs for a *subset* of input conditions. If the output values can be precomputed, the original logic circuit can be "turned off" in the next clock cycle

and, hence, will have substantially reduced switching activity. Since the savings in the power dissipation of the original circuit is offset by the power dissipated in the precomputation phase, the selection of the subset of input conditions for which the output is precomputed is critical. The precomputation logic adds to the circuit area and can also result in an increased clock period.

Given a sequential logic circuit, we present an automatic method of synthesizing the precomputation logic so as to achieve a maximum reduction in power dissipation. We present examples and experimental results on various sequential circuits. For some datapath circuits, 60 percent reductions in power dissipation are possible with marginal increases in circuit area and delay.

We begin, in Chapter 2, by introducing some terminology pertaining to Boolean functions. In Chapter 3, we describe how we accurately and efficiently estimate the power dissipated in CMOS combinational and sequential logic circuits. In Chapter 4, we describe precomputation architectures, and we give examples of circuits that use precomputation. An algorithm that synthesizes precomputation logic in order to achieve the maximum reduction in power dissipation is described in Chapter 5. We also present an algorithm that gives the best power reduction in multiple-output functions. In Chapter 6, we give examples of circuits that are precomputable, but for which the precomputation logic cannot be determined using our algorithms. We also describe some additional precomputation architectures. One architecture, for instance, applies precomputation to combinational logic circuits. Multiple-cycle precomputation, which shows how powerful precomputation-based optimization can be, is described in Chapter 7. Experimental results for datapath as well as random logic sequential circuits are given in Chapter 8.

Chapter 2

Preliminaries

2.1 Introduction

We introduce terminology that we need in the power estimation and optimization methods described in subsequent chapters. In Section 2.2, we give some definitions pertaining to Boolean functions. We describe how logic functions can be represented graphically using Binary Decision Diagrams in Section 2.3.

2.2 Definitions

A *Boolean function* f of n input variables, $x_1, \cdots, x_n$, and of m output variables, $f_1, \cdots, f_m$, is a mapping $f : B^n \rightarrow B^m$, where $B^n = \{0,1\}^n$ and $B^m = \{0,1\}^m$. For each output f_i of f, the *ON-set* can be defined to be the set of inputs x such that $f_i(x) = 1$. Similarly, the *OFF-set* is the set of inputs x such that $f_i(x) = 0$. A function in which $m = 1$ is a *single-output function*, and a function with $m > 1$ is a *multiple-output function*.

The *support* of f, denoted as *support*(f), is the set of all variables x_i that occur in f as x_i or $\overline{x_i}$. For example, if $f = x_1 \cdot \overline{x_2} + x_3$, then *support*$(f) = \{x_1, x_2, x_3\}$.

The *cofactor* of a function f with respect to a variable x_i, denoted as f_{x_i}, is defined

as:

$$f_{x_i} \; = \; f(x_1, \cdots, x_{i-1}, 1, x_{i+1}, \cdots, x_n) \tag{2.1}$$

Likewise, the cofactor of a function f with respect to a variable $\overline{x_i}$, denoted as $f_{\overline{x_i}}$, is defined as:

$$f_{\overline{x_i}} \; = \; f(x_1, \cdots, x_{i-1}, 0, x_{i+1}, \cdots, x_n) \tag{2.2}$$

The *Shannon expansion* of function around a variable x_i is given by:

$$f \; = \; x_i \cdot f_{x_i} \; + \; \overline{x_i} \cdot f_{\overline{x_i}} \tag{2.3}$$

2.3 Binary Decision Diagrams

A *Binary Decision Diagram* (BDD) [1, 10] is a rooted, directed graph with vertex set V containing two types of vertices. A *nonterminal* vertex v has as attributes an argument index $index(v) \in \{1, \cdots, n\}$ and two children $low(v)$, $high(v) \; \in V$. A *terminal* vertex v has as an attribute a value $value(v) \in \{0,\ 1\}$.

The correspondence between BDDs and Boolean functions is defined as follows: a BDD G having root vertex v denotes a function f_v defined recursively as:

1. If v is a terminal vertex:
   (a) If $value(v) = 1$, then $f_v = 1$.
   (b) If $value(v) = 0$, then $f_v = 0$.
2. If v is a nonterminal vertex with $index(v) = i$, then f_v is the function:

$$f_v(x_1, \cdots, x_n) \; = \; \overline{x_i} \cdot f_{low(v)}(x_1, \cdots, x_n) \; + \; x_i \cdot f_{high(v)}(x_1, \cdots, x_n) \tag{2.4}$$

   where x_i is the *decision variable* for vertex v.

Ordered BDDs (OBDDs) have a restriction such that for any nonterminal vertex v, if $low(v)$ is also nonterminal, then $index(v) < index(low(v))$. Similarly, if $high(v)$ is also nonterminal, then $index(v) < index(high(v))$. From these conditions, it is easy

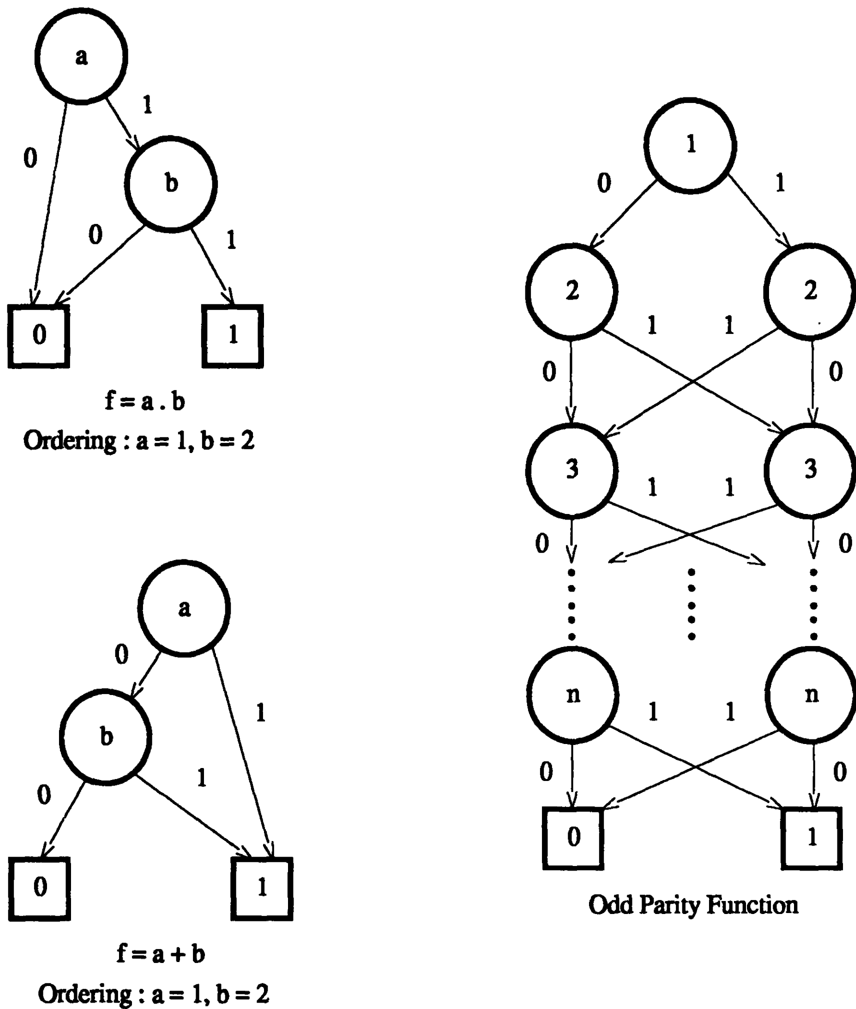


Figure 2-1: Examples of Ordered Binary Decision Diagrams

to see that an OBDD is an acyclic graph. The OBDDs for some simple functions are shown in Figure 2-1. Terminal vertices are represented as squares, while nonterminal vertices are represented as circles. The low child is pointed to by the arrow marked 0, and the high child is pointed to by the arrow marked 1.

Reduced OBDDs (ROBDDs) as proposed in [4] are a minimal OBDD representation for a given function and are defined as follows:

Definition 2.1 *An OBDD G is reduced if it contains no vertex v with* $low(v) = high(v)$ *nor does it contain distinct vertices v and w such that the subgraphs rooted*

by v and w are isomorphic.

In [4], it is also proved that an ROBDD is a canonical representation of a Boolean function. We use ROBDDs to represent logic functions in our power estimation and optimization methods.

Chapter 3

Power Estimation

3.1 Introduction

Before we can optimize a circuit for low power, we need to accurately and efficiently estimate the power that the circuit dissipates. In Section 3.2, we describe a simple model that estimates the dynamic power of a CMOS logic gate. Methods to efficiently determine the power in combinational logic circuits are described in Section 3.3. Finally, in Section 3.4, an accurate method to determine the power in sequential circuits is presented.

3.2 A Power Dissipation Model

In a simple model, the energy dissipated in a CMOS circuit is directly related to the switching activity. In particular, the assumptions are:

- The only capacitance in a CMOS logic gate is at the output node of the gate.
- Current is flowing either from V_{DD} to the output capacitor, or from the output capacitor to ground.

- Any change in the gate's output voltage is a change from V_{DD} to ground, or vice-versa.

All of these are reasonably accurate assumptions for well-designed CMOS gates [9] and they imply that the energy dissipated by a CMOS gate each time its output changes is approximately equal to the change in energy stored in the gate's output capacitance. If the gate is part of a synchronous digital system controlled by a global clock, it follows that the average power dissipated by the gate is given by:

$$P_{avg} = 0.5 \times C_{load} \times V_{dd}^2 \times f_{cyc} \times E(transitions) \quad (3.1)$$

where P_{avg} denotes the average power, C_{load} is the load capacitance, V_{dd} is the supply voltage, f_{cyc} is the global clock frequency, and $E(transitions)$ is the *expected value* of the number of gate output transitions per clock cycle [13], or, equivalently, the average number of gate output transitions per clock cycle. All of the parameters in Equation 3.1 can be determined from the technology or circuit layout information except $E(transitions)$, which depends on the logic function being performed and the statistical properties of the primary inputs. Equation (3.1) is used by the power estimation techniques such as [8, 13] to relate switching activity to power dissipation.

In the optimization method presented in this thesis, we assume that C_{load}, V_{dd}, and f_{cyc} are fixed, and we target the quantity $E(transitions)$, also known as the switching activity, to minimize the average power dissipation. In the following section, we describe how we compute $E(transitions)$.

3.3 Power Estimation of Combinational Circuits

The combinational logic estimation techniques summarized in this section were originally developed in [8].

3.3.1 Estimating Switching Activity

A logical function implemented by a gate g_i in a circuit is denoted as f_i. The probability of the function f_i being a 1 is p_i^{one}, and the probability of f_i being a 0 is $1 - p_i^{one}$.

In the case of static CMOS circuits, the application of a vector pair $\langle I0,\ It\rangle$ causes transitions to occur at gate outputs. Assuming a zero-delay model, each gate in a CMOS circuit can make at most one transition, either from low to high or from high to low, upon the application of a vector pair. In a zero-delay model, all gates switch instantaneously. If the vectors applied are uncorrelated, then the probability that the gate g_i makes a low to high transition is $(1 - p_i^{one})p_i^{one}$. Similarly, the probability of a high to low transition is $p_i^{one}(1 - p_i^{one})$. Hence, the expected number of transitions is:

$$E(transitions) \;=\; 2p_i^{one}(1 - p_i^{one}) \qquad (3.2)$$

In the case of a CMOS circuit with arbitrary gate delays, a gate may make multiple transitions, in other words *glitch*, on the application of a vector pair. In that case, $E(transitions)$ could be greater than 1.

3.3.2 Symbolic Simulation

Given a combinational logic function and the *static probabilities* of the inputs i.e., the probability of the input being a 0 and the probability of the input being a 1, symbolic simulation can be used to calculate the average switching activity at each gate in the circuit. Once the switching activity is known, the average power dissipated can be computed using Equation 3.1. The *total* average power of the circuit is the sum of the average power dissipated by each gate in the circuit.

In symbolic simulation, we construct a Boolean function representing the logical value at a gate output for each time point. For instance, we compute the functions $f_i(t)$ and $f_i(t+1)$ for a particular gate g_i i.e., the value of the function at time t and at time $t+1$. The Boolean condition that corresponds to a $0 \rightarrow 1$ transition on g_i

between times t and $t+1$ is represented by the function $\overline{f_i(t)} \cdot f_i(t+1)$. Therefore, the probability of a $0 \rightarrow 1$ transition occurring between time t and $t+1$ is the probability of the Boolean function $\overline{f_i(t)} \cdot f_i(t+1)$ evaluating to a 1. Similarly, a $1 \rightarrow 0$ transition on g_i can be represented by the function $f_i(t) \cdot \overline{f_i(t+1)}$. Hence, the probability of the node making a transition between times t and $t+1$ is the probability of

$$\overline{f_i(t)} \cdot f_i(t+1) \; + \; f_i(t) \cdot \overline{f_i(t+1)} \; = \; f_i(t) \; \oplus \; f_i(t+1) \tag{3.3}$$

being a 1, where $\oplus$ stands for the exclusive-or operator. These probabilities can be evaluated exactly using BDDs which represent the symbolic simulation equations. For each gate, the probabilities of transitions occurring at each time point are evaluated, and these probabilities are summed over all the time points to obtain the average switching activity.

For a general delay model, symbolic simulation takes into account the correlation due to reconvergence of input signals and accurately measures switching activity.

3.4 Power Estimation of Sequential Circuits

The sequential logic estimation techniques summarized here were originally presented in [12].

Power and switching activity estimation for sequential circuits is significantly more difficult than combinational circuits because the probability of the circuit being in any of its possible states has to be computed. As an example, consider the sequential circuit of Figure 3-1. When a vector pair $\langle v_1, \; v_2 \rangle$ is applied to the combinational logic, it is composed of a primary input part and a present state part, namely $\langle i_1 @ s_1, \; i_2 @ s_2 \rangle$. Given $i_1 @ s_1$, the next state s_2 is uniquely determined by the functionality of the combinational logic. This *correlation* between the vector pairs has to be taken into account in accurate, sequential switching activity estimation.

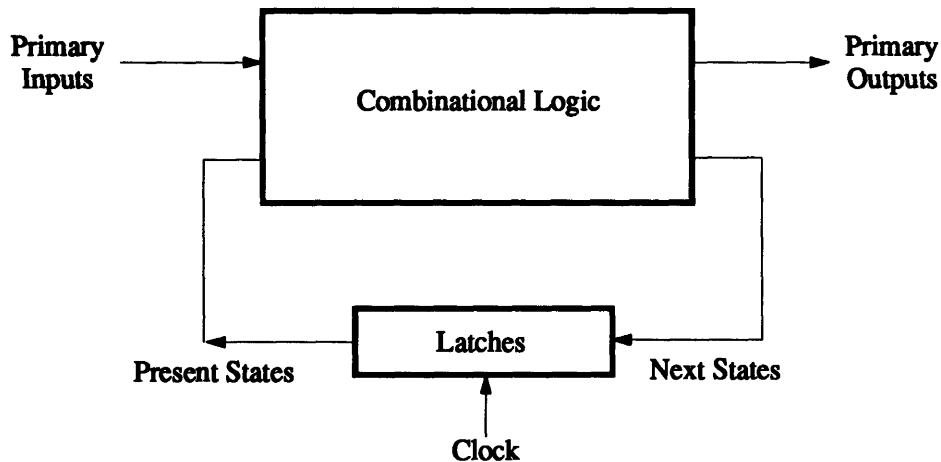


Figure 3-1: A Synchronous Sequential Circuit

3.4.1 Modeling Correlation

To model the correlation between two vectors in a sequential circuit, the combinational estimation method described in Section 3.3 has to be augmented. This is summarized in Figure 3-2.

Figure 3-2 shows the block corresponding to the symbolic simulation equations for the combinational logic part of the general sequential circuit shown in Figure 3-1. The symbolic simulation equations have two sets of inputs, namely $\langle I0,\ It \rangle$ for the primary inputs and $\langle PS,\ NS \rangle$ for the present state lines. However, given $I0$ and PS, NS is determined by the functionality of the combinational logic. This is modeled by prepending the next state logic to the symbolic simulation equations.

The configuration of Figure 3-2 implies that the switching activity can be determined given the vector pair $\langle I0,\ It \rangle$ for the primary inputs and PS for the state lines. Therefore, to compute the switching activity, we require the static probabilities for the primary input and the present state lines.

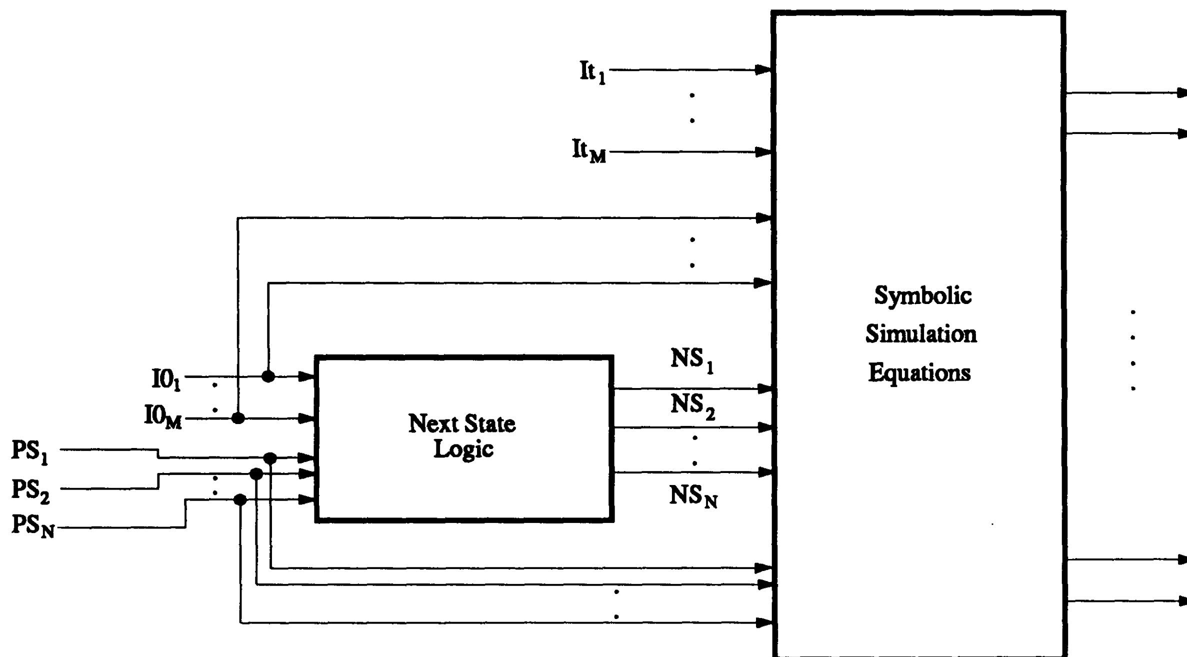


Figure 3-2: Modeling Correlation in a Sequential Circuit

3.4.2 State Probability Computation

The static probabilities for the present state lines marked PS in Figure 3-2 are also correlated. Knowledge of *present state probabilities* as opposed to present state line (PS) probabilities is required. The state probabilities depend on the connectivity of the State Transition Graph (STG) of the circuit and can be computed using the Chapman-Kolmogorov equations for discrete-time Markov Chains [14]. This method is described below.

For each state s_i, $1 \leq i \leq K$ in the STG, a variable $prob(s_i)$ corresponds to the steady-state probability of the machine being in state s_i at $t = \infty$. For each edge e in the STG, $e.Current$ signifies the state that the edge fans out from, $e.Next$ signifies the state that the edge fans out to, and $e.Input$ signifies the input combination corresponding to the edge. Given static probabilities for the primary inputs to the machine, $prob(Input)$, the probability of the combination $Input$ occurring, can be

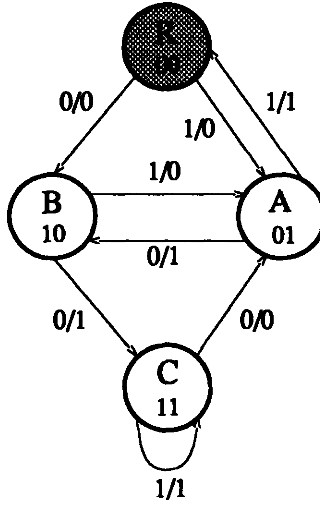


Figure 3-3: A State Transition Graph

computed as follows:

$$prob(e.Input) = prob(e.Current) \times prob(Input) \tag{3.4}$$

For each state s_i, an equation can be written as:

$$prob(s_i) = \sum_{\forall\ e\ such\ that\ e.Next\ =\ s_i} prob(e.Input) \tag{3.5}$$

Given K states, $K - 1$ equations are obtained. One final equation that is needed is:

$$\sum_{i=1}^{K} prob(s_i) = 1 \tag{3.6}$$

This linear set of K equations can be solved to obtain the different $prob(s_i)$'s.

For example, consider the STG of Figure 3-3. The following equations are obtained assuming a probability of 0.5 for the primary input being a 1:

$$prob(\mathbf{R}) = 0.5 \times prob(\mathbf{A}) \tag{3.7}$$

$$prob(\mathbf{A}) = 0.5 \times prob(\mathbf{R}) + 0.5 \times prob(\mathbf{B}) + 0.5 \times prob(\mathbf{C}) \tag{3.8}$$

$$prob(\mathbf{B}) = 0.5 \times prob(\mathbf{R}) + 0.5 \times prob(\mathbf{A}) \tag{3.9}$$

The final equation is:

$$prob(\mathbf{R}) + prob(\mathbf{A}) + prob(\mathbf{B}) + prob(\mathbf{C}) = 1 \tag{3.10}$$

Solving this linear system of equations results in the state probabilities, $prob(\mathbf{R}) = \frac{1}{6}$, $prob(\mathbf{A}) = \frac{1}{3}$, $prob(\mathbf{B}) = \frac{1}{4}$ and $prob(\mathbf{C}) = \frac{1}{4}$.

3.4.3 Power Estimation Given Exact State Probabilities

A power estimation method that uses the exact state probabilities obtained from the Chapman-Kolmogorov method is described below. As was shown in Section 3.3, the symbolic simulation equations express the exact switching conditions for each gate in the circuit. Prepending the next state logic block, as illustrated in Figure 3-2, accounts for the correlation between the present and next states. Finally, computing the exact state probabilities models the steady-state behavior of the circuit.

As described in Section 3.3, power estimation of a combinational circuit can be carried out by creating a set of symbolic functions, such that summing the signal probabilities of the functions corresponds to the average switching activity in the circuit. In the case of sequential circuits, some of the inputs to the symbolic functions are the present state lines of the circuit and the others are the primary input lines.

The signal probability calculation procedure has to appropriately weigh these combinations. As an example, consider the function

$$f = i_1 \wedge ps_1 \vee i_1 \wedge \overline{ps_1} \wedge ps_2 \quad (3.11)$$

whose signal probability is to be computed. Assume that the probability of i_1 being a 1 is 0.5, and the state probabilities are $prob(00) = \frac{1}{6}$, $prob(01) = \frac{1}{3}$, $prob(10) = \frac{1}{4}$ and $prob(11) = \frac{1}{4}$ (the first bit corresponds to ps_1 and the second to ps_2). The probability of f can be calculated by summing the probabilities of the two product terms in f since the two terms have a null intersection. The probability of the first term is

$$\begin{aligned} prob(i_1 \wedge ps_1) &= prob(i_1) \times (prob(10) + prob(11)) \\ &= 0.5 \times (\tfrac{1}{4} + \tfrac{1}{4}) \\ &= \tfrac{1}{4} \end{aligned}$$

Similarly, the probability of the second term is

$$\begin{aligned} prob(i_1 \wedge \overline{ps_1} \wedge ps_2) &= prob(i_1) \times prob(01) \\ &= 0.5 \times \frac{1}{3} \\ &= \frac{1}{6} \end{aligned}$$

Finally we have

$$\begin{aligned} prob(f) &= \frac{1}{4} + \frac{1}{6} \\ &= \frac{5}{12} \end{aligned}$$

The major disadvantage of this estimation method is its *average-case* exponential complexity – the probability of each state is computed and the number of states grows exponentially with the number of flip-flops in the circuit. In [12], however, approximate methods are described that are computationally efficient. We use these methods to obtain accurate estimates of power dissipation in our optimization experiments.

Chapter 4

Precomputation Architectures

4.1 Introduction

We present a sequential logic optimization method that is based on selectively *precomputing* the outputs of the circuit one clock cycle before they are required and using the precomputed values to reduce switching activity in the next clock cycle.

We begin by defining the predictor functions, which are the basis for precomputation, in Section 4.2. In Sections 4.3 and 4.4, we describe two different precomputation architectures and discuss their characteristics in terms of power dissipation, circuit area, and circuit delay. In Section 4.5, we generalize precomputation to multiple-output functions. We illustrate examples in Section 4.6. Finally, in Section 4.7, we briefly discuss the testability of precomputation-based logic circuits.

4.2 The Predictor Functions

Consider the circuit of Figure 4-1. We have a combinational logic block **A** that is separated by registers R_1 and R_2. While R_1 and R_2 are shown as distinct registers in Figure 4-1, they could, in fact, be the same register. We will first assume that the logic block **A** has a single output and that it implements the function f.

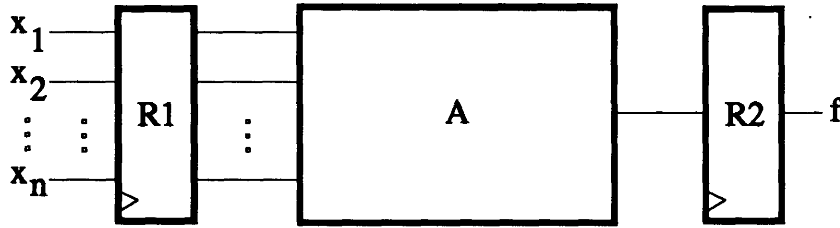


Figure 4-1: The Original Circuit

We define two Boolean functions g_1 and g_2, called the *predictor* functions, that depend on the same subset of inputs to the logic block **A** as follows:

$$g_1 = 1 \Rightarrow f = 1 \quad (4.1)$$

$$g_2 = 1 \Rightarrow f = 0 \quad (4.2)$$

When g_1 and g_2 are both 0, we do not know anything about the function. Also, g_1 and g_2 cannot both be 1 during the same clock cycle as that would imply that the the function f is both 0 and 1.

4.3 First Precomputation Architecture

In Figure 4-2, the first precomputation architecture is shown. During clock cycle t, if either g_1 or g_2 evaluates to a 1, we set the load-enable signal of register R_1 to a 0. This means that in clock cycle $t+1$ the inputs to the combinational logic block **A** do not change. If g_1 evaluates to a 1 in clock cycle t, the input to register R_2 is set to a 1 in clock cycle $t+1$, and if g_2 evaluates to a 1, then the input to register R_2 is set to a 0.

A power reduction in block **A** is obtained because for a subset of input conditions corresponding to $g_1 + g_2$, the inputs to **A** do not change implying zero switching activity. However, the area of the circuit has increased due to the additional logic of g_1 and g_2, the two additional gates shown in the figure, and the two flip-flops marked **FF**. The delay between R_1 and R_2 has increased due to the addition of the OR-AND

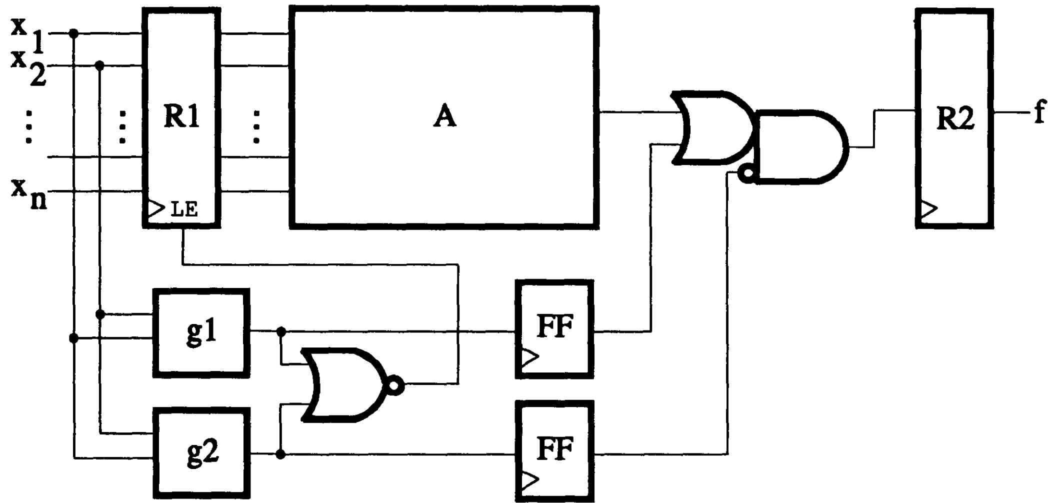


Figure 4-2: First Precomputation Architecture

gate. Note also that g_1 and g_2 add to the delay of paths that originally ended at R_1, but now pass through g_1 or g_2 and the NOR gate before ending at the load-enable signal.

The choice of g_1 and g_2 is critical. We wish to include as many input conditions in g_1 and g_2. In other words, we wish to maximize the probability of g_1 or g_2 evaluating to a 1 as that corresponds to turning off the logic block the largest percent of the time. In the extreme case, this probability is unity if $g_1 = f$ and $g_2 = \overline{f}$. However, this implies a duplication of the logic block **A** and no reduction in power with a twofold increase in area! To obtain a reduction in power with marginal increases in circuit area and delay, g_1 and g_2 have to be significantly less complex than f. One way of ensuring this is to make g_1 and g_2 depend on significantly fewer inputs than f.

4.4 Second Precomputation Architecture

In the second precomputation architecture shown in Figure 4-3, the inputs to logic block **A** have been partitioned into two sets, corresponding to the registers R_1 and R_2. If g_1 or g_2 evaluates to a 1 during clock cycle t, the load-enable signal of register

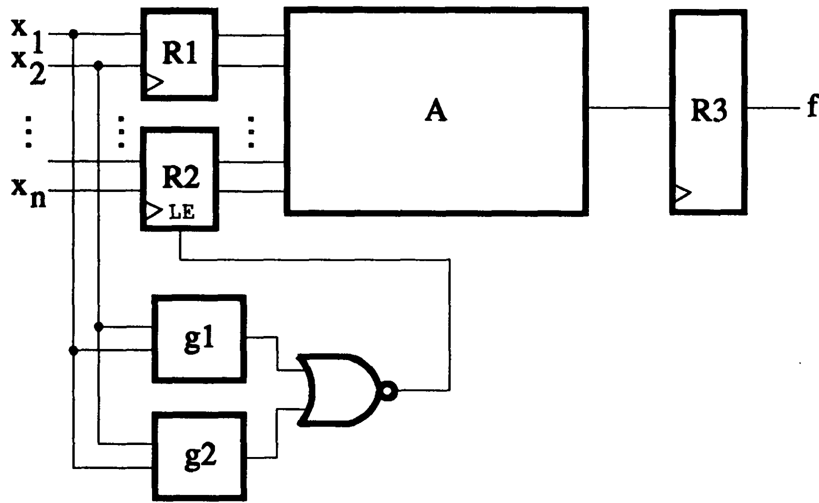


Figure 4-3: Second Precomputation Architecture

R_2 is set to a 0. This means that the outputs of R_2 do not change during clock cycle $t+1$. However, since register R_1 is updated in clock cycle t, the function f will evaluate to the correct logical value.

As in the case of the first precomputation architecture, a power reduction is achieved because only a subset of the inputs to block **A** change, implying reduced switching activity. The area of the circuit has increased, but not as much as in the first architecture. The delay of the paths that ended at R_1 have increased, but the delay from R_1/R_2 to R_3 has remained the same. Once again, g_1 and g_2 have to be significantly less complex than f, and the probability of $g_1 + g_2$ being a 1 should be high in order to substantially reduce power dissipation.

In Chapter 5, we describe an algorithm to select inputs to g_1 and g_2 so that the probability of $g_1 + g_2 = 1$ is maximized. We also discuss how the g_1 and g_2 functions that satisfy Equations 4.1 and 4.2 can be obtained.

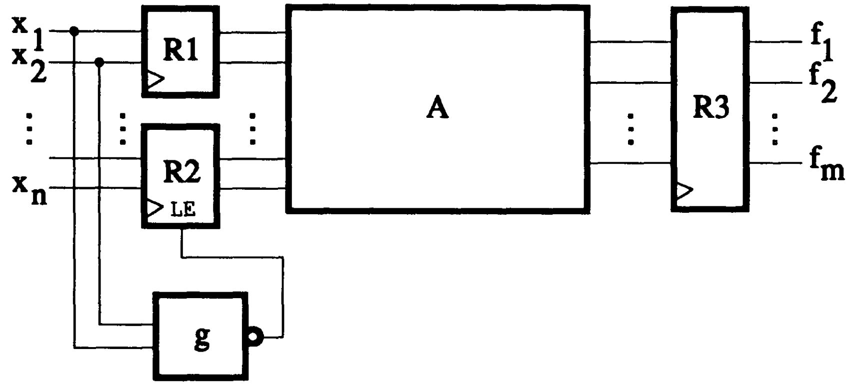


Figure 4-4: Precomputation of a Multiple-Output Function

4.5 Multiple-Output Functions

In general, the logic block **A** shown in Figures 4-2 and 4-3 can be a multiple-output function with outputs $F = \{f_1, \cdots, f_m\}$ as shown in Figure 4-4.

The *predictor* functions are then defined as:

$$g_{1i} = 1 \quad \Rightarrow \quad f_i = 1 \tag{4.3}$$

$$g_{2i} = 1 \quad \Rightarrow \quad f_i = 0 \tag{4.4}$$

for each output f_i. The function g whose complement drives the load enable signal is given as:

$$g \; = \; \prod_{i=1}^{m} (g_{1i} \; + \; g_{2i}) \tag{4.5}$$

Once again, we want to maximize the probability of g being a 1 in order to get the most power reduction. In the case of multiple-output functions, we want to select a subset of the outputs to precompute because the probability of g being a 1 rapidly decreases as more outputs are precomputed. Typically, we want to select the most complex functions (in terms of area) to precompute. In Chapter 5, we describe an algorithm to select the best outputs to precompute. We also discuss, in detail, how we construct the final, precomputed multiple-output function since logic corresponding to the outputs not selected for precomputation may need to be duplicated.

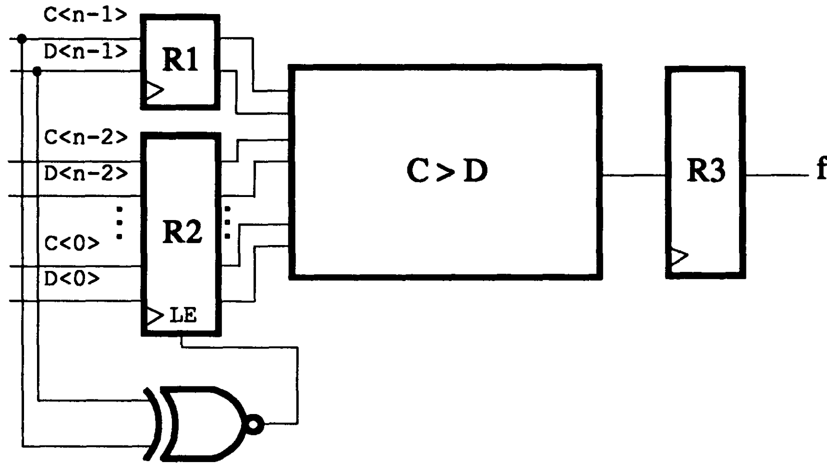


Figure 4-5: Precomputation of a Comparator Function

4.6 Examples

We give examples that illustrate that substantial power gains can be achieved with marginal increases in circuit area and delay.

Consider an n-bit comparator, a function that takes two n-bit inputs C and D and computes $C > D$. The circuit with its precomputation logic is shown in Figure 4-5. The precomputation logic is as follows:

$$g_1 = C\langle n-1\rangle \cdot \overline{D\langle n-1\rangle} \tag{4.6}$$

$$g_2 = \overline{C\langle n-1\rangle} \cdot D\langle n-1\rangle \tag{4.7}$$

Clearly, when $g_1 = 1$, C is greater than D, and when $g_2 = 1$, C is less than D. We have to implement

$$\overline{g_1 + g_2} = C\langle n-1\rangle \otimes D\langle n-1\rangle \tag{4.8}$$

where $\otimes$ stands for the exclusive-nor operator.

Assuming a uniform probability for the inputs, [1] the probability that the XNOR gate evaluates to a 1 is 0.5 regardless of n. For large n, we can neglect the power

[1]The assumption here is that each $C\langle i\rangle$ and $D\langle i\rangle$ has a 0.5 static probability of being a 0 or a 1.

x_1	x_2	x_3	x_4	f_1	f_2	f_3	f_4
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	–	0	0	1	0
0	1	–	–	0	1	0	0
1	–	–	–	1	0	0	0

Table 4.1: Truth-Table of a Priority Function

dissipated by the XNOR gate, and, therefore, achieve a power reduction close to 50%. The reduction, however, will depend on the relative power dissipated by the vector pairs with $C\langle n-1\rangle \otimes D\langle n-1\rangle = 1$ and the vector pairs with $C\langle n-1\rangle \otimes D\langle n-1\rangle = 0$. If we add the inputs $C\langle n-2\rangle$ and $D\langle n-2\rangle$ to g_1 and g_2, it is possible to achieve a power reduction close to 75%.

We can continue adding more inputs to the g_1 and g_2 functions, thereby, increasing the percent of the time that the function can be precomputed. However, as more inputs are added, the precomputation logic becomes more complicated (and we need to account for the power that it dissipates), and we are disabling fewer inputs. Hence, we expect to reach an optimal point where the power dissipation is at a minimum. If more inputs are included in the precomputation logic beyond this point, the power gains will begin to diminish.

Another example is an n-bit priority function with inputs $X = (x_1, \cdots, x_n)$ and outputs $F = (f_1, \cdots, f_n)$. This function selects the highest priority input. In other words, an output f_i is a 1 if x_i is a 1 and x_j is a 0 for all $j > i$. The truth-table for a 4-input, 4-output priority function is shown in Table 4.1.

Figure 4-6 shows the priority function with its precomputation logic. Since x_1 is the highest priority input, we can disable all the low priority inputs whenever x_1 is a 1. Again, assuming a uniform probability for the inputs, the low priority inputs can be disabled 50% of the time. If x_2 is added to the precomputation logic, the inputs can be disabled 75% of the time.

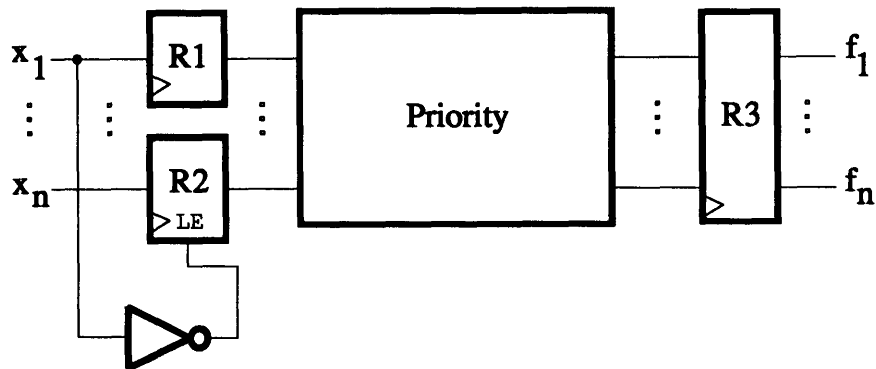


Figure 4-6: Precomputation of a Priority Function

4.7 Testability of Precomputation-Based Logic Circuits

Ensuring that logic circuits are testable is often a critical part of the design process. Considerable attention has been devoted to improving the testability of combinational as well as sequential logic circuits [7]. The focus of this thesis, however, is not to completely evaluate the testability of precomputation-based logic circuits, but to just briefly discuss the issue.

If a scan design methodology is used to test the original circuit, then the precomputed circuit is also testable. For instance, in the second precomputation architecture, during testing, the input registers can be set to specific vectors and the value of the function can be observed at the output register. However, precomputation could affect the testability of the preceeding logic block. Figure 4-7 shows a simple case where we are interested in determining if a particular node is stuck-at-0. A stuck-at-0 fault, for example, occurs when a particular input or node is inadvertently connected to ground during the manufacturing process.

To test the node c for a stuck-at-0 fault, nodes a and b must be set to 0s and a 1 must be applied to node c. However, since a and b are part of the precomputation logic in the next stage, the output of the OR gate computing $a + b + c$ can never be

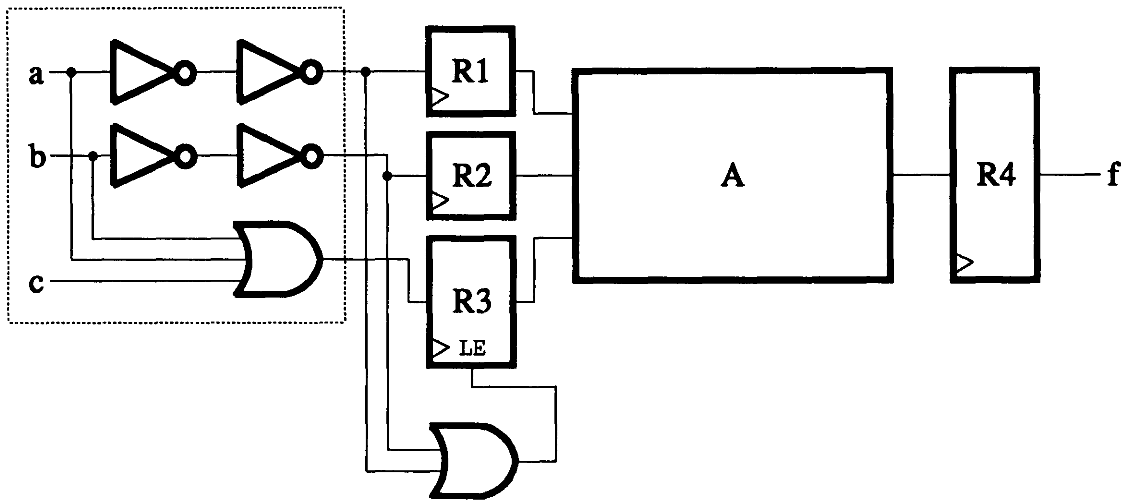


Figure 4-7: Redundancy in a Precomputation-Based Logic Circuit

observed at the register R_3, because whenever $a + b = 0$, the load-enable signal is a 0 and the register is not updated with a new value. In essence, the precomputation logic that has been added is *redundant* as it is already contained in the logic of the previous stage.

Chapter 5

Synthesis of Precomputation Logic

5.1 Introduction

We describe algorithms to determine the best subset of inputs to the precomputation logic and to find the best set of outputs to precompute in the case of multiple-output functions. We focus primarily on the second precomputation architecture illustrated in Figure 4-3. To ensure that the precomputation logic is significantly less complex than the original circuit, we restrict ourselves to identifying g_1 and g_2 such that they depend on a relatively small subset of the inputs.

In Section 5.2, we discuss observability don't-cares and their relationship to precomputation. In Section 5.3, we show how we can determine the functionality of the precomputation logic, and we describe an algorithm to select inputs to the precomputation logic. We also give an example to illustrate how the algorithm works. Finally, in Section 5.4, we present an output-selection algorithm, and we discuss the need for logic duplication when precomputing multiple-output functions.

5.2 Precomputation and Observability Don't-Cares

Assume that we have a logic function $f(X)$ with $X = \{x_1, \cdots, x_n\}$, as in Figure 4-2. The ***observability don't-care set*** for an input x_i is defined as:

$$ODC_i = f_{x_i} \cdot f_{\overline{x_i}} + \overline{f}_{x_i} \cdot \overline{f}_{\overline{x_i}} \tag{5.1}$$

where f_{x_i} and $f_{\overline{x_i}}$ are the *cofactors* of f with respect to x_i, and $\overline{f}_{x_i}$ and $\overline{f}_{\overline{x_i}}$ are the cofactors of $\overline{f}$ with respect to x_i.

Observability don't-cares arise when a logic function's structure limits, under certain input conditions, the observability of a node at an output [7]. If an input x_i is in ODC_i, then we can disable the loading of x_i into the input register. If we wish to disable the loading of registers $x_{k+1}, \cdots, x_n$, we need to implement the function

$$g = \prod_{i=k+1}^{n} ODC_i \tag{5.2}$$

and use $\overline{g}$ as the load-enable signal for the registers corresponding to $x_{k+1}, \cdots, x_n$.

5.3 Precomputation Logic

Let us now consider the architecture of Figure 4-3. Assume that $x_1, \cdots, x_k$, with $k < n$ have been selected as inputs to the predictor functions g_1 and g_2.

We need to find g_1 and g_2 such that they satisfy the constraints of Equations 4.1 and 4.2, and such that $prob(g_1 + g_2 = 1)$ is maximized.

We can determine g_1 and g_2 using universal quantification. The ***universal quantification*** of a function f with respect to a variable x_i is defined as:

$$U_{x_i} f = f_{x_i} \cdot f_{\overline{x_i}} \tag{5.3}$$

Given a subset of inputs $S = \{x_1, \cdots, x_k\}$, we define a set $D = X - S$. The ***universal quantification*** of a function f with respect to a set of variables D is given as:

$$U_D f = U_{x_{k+1}} \ldots U_{x_n} f \tag{5.4}$$

Theorem 5.1 *$g_1 = U_D f$ satisfies Equation 4.1. Furthermore, no function $h(x_1, \cdots, x_k)$ exists such that $prob(h = 1) > prob(g_1 = 1)$ and such that $h = 1 \Rightarrow f = 1$.*

Proof. If some input combination $a_1, \cdots, a_k$ causes $g_1(a_1, \cdots, a_k) = 1$, then for that combination of $x_1, \cdots, x_k$ and all possible combinations of variables in $x_{k+1}, \cdots, x_n$ $f(a_1, \cdots, a_k,\ x_{k+1}, \cdots, x_n) = 1$.

We cannot add any minterm $x_1, \cdots, x_k$ to g_1 because for any minterm that is added, there will be some combination of $x_{k+1}, \cdots, x_n$ for which $f(x_1, \cdots, x_n)$ will evaluate to a 0. Therefore, we cannot find any function h that satisfies Equation 4.1 and such that $prob(h = 1) > prob(g_1 = 1)$. ∎

Similarly, for a subset of inputs S, the function g_2 is given as:

$$g_2 \ = \ U_D \overline{f} \ = \ U_{x_{k+1}} \ldots U_{x_n} \overline{f} \qquad (5.5)$$

When we implement the load-enable signal $g = \overline{(g_1 + g_2)}$, we are implementing the function given by Equation 5.2.

5.3.1 Selecting a Subset of Inputs

Given a function f, we wish to select the "best" subset of inputs S of cardinality k. Given S, we have $D = X - S$ and we compute $g_1 = U_D f$ and $g_2 = U_D \overline{f}$. The best set of inputs are those which result in $prob(g_1 + g_2 = 1)$ being a maximum for a given k. We know that $prob(g_1 + g_2 = 1) = prob(g_1 = 1) + prob(g_2 = 1)$ since g_1 and g_2 cannot both be 1 during the same clock cycle. The above cost function ignores the power dissipated by the precomputation logic, but since the number of inputs to the precomputation logic is significantly smaller than the total number of inputs, this is a good approximation.

We present a branching algorithm that determines the set of inputs that maximize the probability of $g_1 + g_2 = 1$. This algorithm is shown in pseudo-code in Figure 5-1.

```
SELECT_INPUTS( f, k ):
{
    BEST_PROB = 0 ;
    SELECTED_SET = φ ;
    SELECT_RECUR( f, f̄, φ, X, |X| − k ) ;
    return( SELECTED_SET ) ;
}

SELECT_RECUR( g1, g2, D, Q, l ):
{
    if( |D| + |Q| < l )
        return ;
    pr = prob(g1 = 1) + prob(g2 = 1) ;
    if( pr ≤ BEST_PROB )
        return ;
    else if( |D| == l ) {
        BEST_PROB = pr ;
        SELECTED_SET = X − D ;
        return ;
    }
    choose x_i ∈ Q such that i is minimum ;
    SELECT_RECUR( U_{x_i} g1, U_{x_i} g2, D ∪ x_i, Q − x_i, l ) ;
    SELECT_RECUR( g1, g2, D, Q − x_i, l ) ;
    return ;
}
```

Figure 5-1: Procedure to Determine the Optimal Set of Inputs

The procedure **SELECT_INPUTS** has as arguments the function f and the desired number of inputs k to the precomputation logic. **SELECT_INPUTS** calls the recursive procedure **SELECT_RECUR** with five arguments. The first two arguments are the g_1 and g_2 functions, which are initially f and $\overline{f}$. An input is selected within the recursive procedure and the two functions are universally quantified with respect to that input. The third argument D corresponds to the set of inputs not in g_1 and g_2. The fourth argument Q corresponds to the set of "active" inputs, which may be selected or discarded. Finally, the argument l corresponds to the number of inputs that have to be universally quantified in order to obtain g_1 and g_2 with k or fewer inputs.

If $|D| + |Q| < l$, it means that we have selected too many inputs in the earlier recursions and we will not be able to universally quantify enough inputs. The functions g_1 and g_2, hence, will depend on too many inputs ($> k$).

We calculate the probability of $g_1 + g_2 = 1$. If this probability is less than the maximum probability encountered thus far, we can immediately return because of the following invariant:

$$prob(U_{x_i} f) \;=\; prob(f_{x_i} \cdot f_{\overline{x_i}}) \;\leq\; prob(f) \quad \forall x_i,\ f \tag{5.6}$$

Therefore, as we universally quantify inputs from the g_1 and g_2 functions, the *pr* quantity monotonically decreases because f always contains $U_{x_i} f$.

We finally store the selected set of inputs with the best probability.

5.3.2 Implementing the Logic

The Boolean operations of OR and universal quantification required in the input-selection procedure can be carried out efficiently using ROBDDs [4]. In the algorithm, we obtain a ROBDD for the $g_1 + g_2$ function, which then can be converted into a multiplexor-based network (see [2]) or into a sum-of-products cover. The network or cover can be optimized using standard combinational logic optimization methods that reduce area [3] or those that target low power dissipation [17].

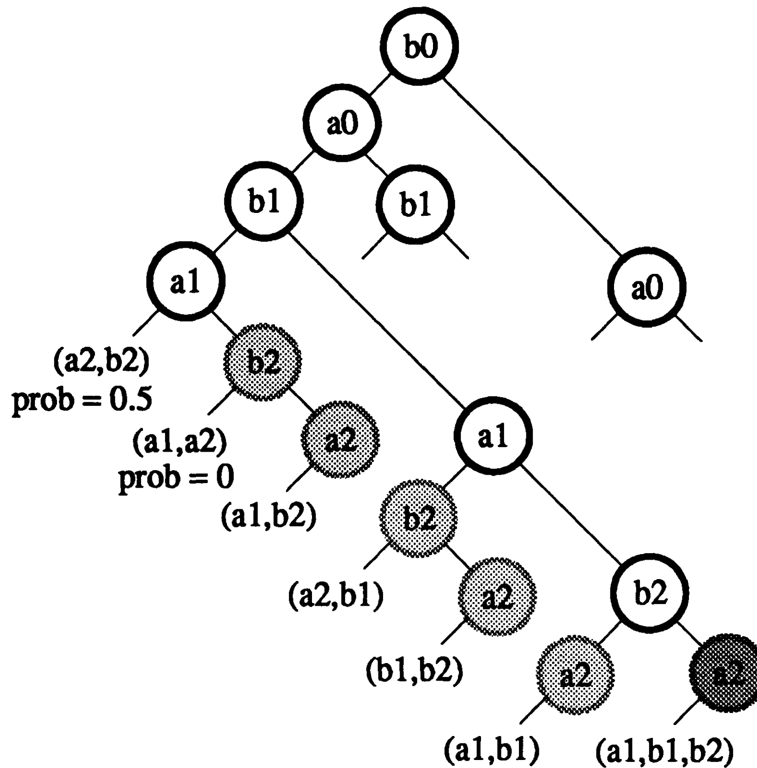


Figure 5-2: A Comparator Example

5.3.3 A Comparator Example

The input-selection algorithm presented in Section 5.3 traverses a binary tree to find the best inputs to the precomputation logic. A partial tree for a $3-bit$ comparator, with inputs $a_0, b_0, \cdots a_2, b_2$ and $k = 2$, is shown in Figure 5-2.

The algorithm begins its search with $g_1 = f$ and $g_2 = \overline{f}$. At a particular node, if the left branch is taken, the input is universally quantified from the precomputation logic, and, hence, is discarded. Whenever the right branch is taken, the input is selected to be in the precomputation logic. The algorithm finds the solution with inputs a_2 and b_2 since it maximizes the probability of $g_1 + g_2 = 1$. The light-shaded nodes in the tree are never reached because of the pruning condition i.e., that the probability of $g_1 + g_2 = 1$ monotonically decreases. The dark-shaded node marked a_2 is also never reached because too many variables have been selected to be in the

precomputation logic. The solution obtained at that point has more than $k = 2$ inputs.

5.4 Multiple-Output Functions

The procedures described so far can be generalized for a multiple-output function with outputs $F = \{f_1, \cdots, f_m\}$ like the one shown in Figure 4-4.

The functions g_{1i} and g_{2i} are given as:

$$g_{1i} = U_D f_i \quad (5.7)$$

$$g_{2i} = U_D \overline{f}_i \quad (5.8)$$

where, again, $D = X - S$.

5.4.1 Selecting a Subset of Outputs

We describe an algorithm, which for a multiple-output function, selects a subset of outputs *and* a subset of inputs in order to maximize a cost function that depends on the probability of the precomputation logic and the number of selected outputs. The pseudo-code for this algorithm is shown in Figure 5-3.

The inputs to procedure **SELECT_OUTPUTS** are the multiple-output function F, and the number k, which corresponds to the number of inputs to the precomputation logic.

The procedure **SELECT_ORECUR** receives as inputs two sets G and H, which are the current set of outputs that have been selected and the set of outputs which can be added to the selected set. Initially, $G = \phi$ and $H = F$. The cost of selecting a set of outputs G is $prG \times$ gates(G)/total_gates, where prG corresponds to the signal probability of the precomputation logic, gates(G) corresponds to the number of gates in the outputs of G, and total_gates corresponds to the total number of gates in the network (across all outputs of F).

```
SELECT_OUTPUTS( F = {f_1, ..., f_m}, k ):
{
    BEST_COST = 0 ;
    SEL_OP_SET = φ ;
    SELECT_ORECUR( φ, F, 1, k ) ;
    return( SEL_OP_SET ) ;
}

SELECT_ORECUR( G, H, proldG, k ):
{
    lf = gates(G ∪ H)/total_gates × proldG ;
    if( lf  ≤ BEST_COST )
        return ;
    BEST_PROB = total_gates/gates(G ∪ H) × BEST_COST ;
    if( G ≠ φ )
        if( SELECT_INPUTS( G, k ) == NULL )
            return ;
    prG = BEST_PROB ;
    cost = prG × gates(G)/total_gates ;
    if( cost > BEST_COST) {
        BEST_COST = cost ;
        SELECTED_SET = G ;
        return
    }
    choose f_i ∈ H such that i is minimum ;
    SELECT_ORECUR( G ∪ f_i, H − f_i, prG, k ) ;
    SELECT_ORECUR( G, H − f_i, prG, k ) ;
    return ;
}
```

Figure 5-3: Procedure to Determine the Optimal Set of Outputs

There are two pruning conditions used in the procedure **SELECT_ORECUR**. The first corresponds to assuming that all the outputs in H can be added to G without decreasing the probability of the precomputation logic. This is a valid condition because the quantity *proldG* in each recursive call can only decrease with the addition of outputs to G, and the only way the cost can improve is if more outputs are selected. We can then set a lower bound on the probability of the precomputation logic prior to calling the input-selection procedure. Assuming that all the outputs in H can be added to G, we are not interested in a precomputation logic probability that results in a cost that is equal to or lower than BEST_COST.

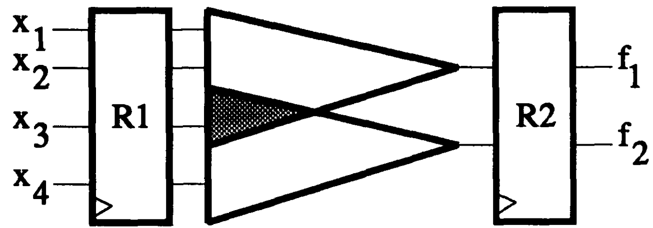
5.4.2 Logic Duplication

Since we are only precomputing a subset of outputs, we may incorrectly evaluate the outputs that we are *not* precomputing as we disable certain inputs during particular clock cycles. If an output that is not being precomputed depends on an input that is being disabled, then the output will be incorrect.

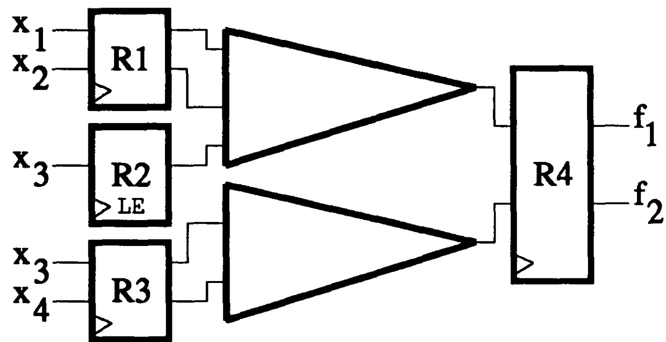
Once a set of outputs $G \subset F$ and a set of precomputation logic inputs $S \subset X$ have been selected, we need to duplicate the registers corresponding to $(support(G) - S) \cap support(F - G)$. The inputs that are being disabled are in $support(G) - S$. Logic in the $F - G$ outputs that depends on the set of duplicated inputs has to be duplicated as well. It is precisely for this reason that we maximize $prG \times$ gates(G)/total_gates rather than prG in the output-selection algorithm as we want to reduce the amount of duplication as much as possible.

An example of a multiple-output function where the registers and logic need to be duplicated is shown in Figure 5-4.

The original network has outputs f_1 and f_2 and inputs $x_1, \cdots, x_4$. The function f_1 depends on inputs $x_1, \cdots, x_3$ and the function f_2 depends on inputs x_3 and x_4. Hence, the two outputs are sharing the input x_3. Suppose that the output-selection procedure determines that f_1 is the best output to precompute and that inputs x_1



(a) Original Network



(b) Final Network

Figure 5-4: Logic Duplication in a Multiple-Output Function

and x_2 are the best inputs to the precomputation logic. Therefore, just as in the case of a single-output function, the inputs x_1 and x_2 feed the input register, whereas, x_3 feeds the register with the load-enable signal. However, since f_2 depends on x_3 and the register with the load-enable signal contains stale values in some clock cycles, we need to duplicate the register for x_3 and the logic from x_3 to f_2.

Chapter 6

Special Cases

6.1 Introduction

We present more examples of logic circuits that are precomputable as well as additional precomputation architectures.

In Section 6.2, we give examples of circuits that *cannot* be automatically precomputed using the input-selection and output-selection algorithms discussed earlier. This is because the circuits either have to implemented in a particular way, or, in some cases, we are precomputing *internal* signals and *not* the function's outputs. We describe new architectures and discuss their advantages and disadvantages in Section 6.3. Both the circuits and architectures described here show the power of precomputation-based optimization.

6.2 Special Circuits

We give examples illustrating how some datapath circuits can be precomputed. Such circuits, for instance, can be part of a cell library, and, hence, can be easily used and accessed when designing low power integrated circuits.

Operation	Code		
	s_0	s_1	s_2
Add	0	0	0
Subtract	0	0	1
Shift-Left	0	1	0
Shift-Right	0	1	1
AND	1	0	0
OR	1	0	1
XOR	1	1	0
NOT	1	1	1

Table 6.1: Specification of an Arithmetic Logic Unit

6.2.1 An Arithmetic Logic Unit

Table 6.1 shows the operations and instruction codes for a simple arithmetic logic unit (ALU). The operations of the ALU can be partitioned into two sets, one containing the arithmetic and shift functions and the other containing the logic functions. Note that for the first set of operations, the instruction bit $s_0 = 0$ and that for the second set $s_0 = 1$.

Figure 6-1 shows how we can implement the ALU so that it is precomputable. The circuit operates on two inputs A and B and produces a result C. When the instruction bit $s_0 = 0$, only the arithmetic and shift blocks are enabled and the logic functions are turned off. Likewise, when $s_0 = 1$, only the logic functions are evaluated while the arithmetic and shift blocks are turned off by setting the load-enable signal of their input register to a 0. The s_1 and s_2 bits drive the select line of a multiplexor that sets the output C to the correct function.

The savings in power dissipation that are obtained by turning off part of the ALU are partially offset by the register duplication. The registers had to be replicated in order for the ALU to function correctly.

The ALU operations were partitioned arbitrarily to illustrate the example. In fact, they could have been partitioned in any way. In a real implementation, for instance, the functions should be divided so that each of the blocks is equally complex. In

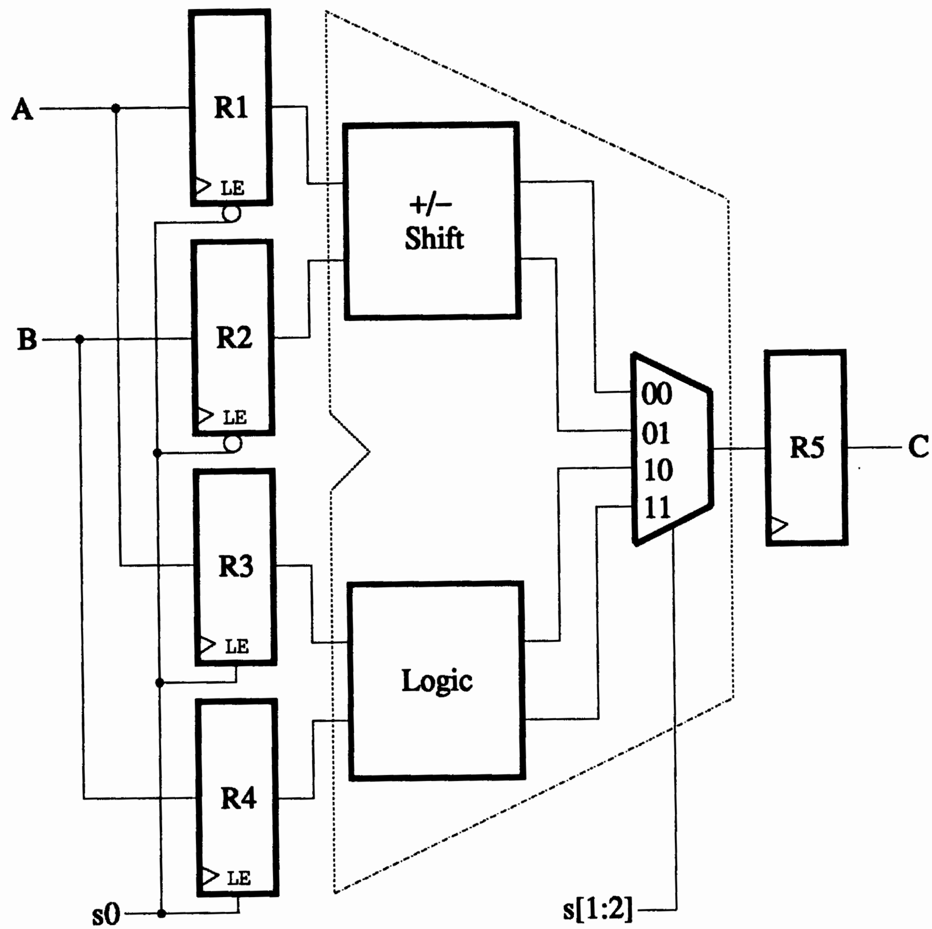


Figure 6-1: Precomputation of an Arithmetic Logic Unit

				a_3	a_2	a_1	a_0
				b_3	b_2	b_1	b_0
				$a_3 \cdot b_0$	$a_2 \cdot b_0$	$a_1 \cdot b_0$	$a_0 \cdot b_0$
			$a_3 \cdot b_1$	$a_2 \cdot b_1$	$a_1 \cdot b_1$	$a_0 \cdot b_1$	
		$a_3 \cdot b_2$	$a_2 \cdot b_2$	$a_1 \cdot b_2$	$a_0 \cdot b_2$		
	$a_3 \cdot b_3$	$a_2 \cdot b_3$	$a_1 \cdot b_3$	$a_0 \cdot b_3$			
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Figure 6-2: Partial Products of a Multiplier

other words, we are not interested in turning off a small part of the ALU a fraction of the time. We want to disable a large part of the logic most of the time. If some operations share a lot of logic, it would not be worthwhile to place those operations in different sets as logic would have to be duplicated. The instruction codes should be tailored keeping in mind how frequently they occur and how much logic is needed to implement them in order to get the best power reduction. Furthermore, for ALUs that perform a large number of functions, the operations can be partitioned into several blocks and the same technique can be used to get a savings in power dissipation.

6.2.2 An Array Multiplier

Figure 6-2 shows how the partial products of a $4 - bit$ multiplier with inputs $A = \{a_0, \cdots, a_3\}$ and $B = \{b_0, \cdots, b_3\}$ are calculated.

In Figure 6-3, we show the first three stages of a pipelined array multiplier. This circuit performs the calculation in the same manner as in Figure 6-2. At each stage, $1 - bit$ of the input B is ANDed with the vector A. The result is shifted and added to the partial product from the previous stage. Note that the shift operation can be implicit in the way the AND gates and adders are connected.

Figure 6-4 shows how the ith stage of the multiplier can be precomputed. The observation here is that when $b_i = 0$, the partial product at the ith stage is 0, and, hence, it does not contribute to the final product. Therefore, when $b_i = 0$, that stage is turned off and the partial product from the previous stage is propagated to the

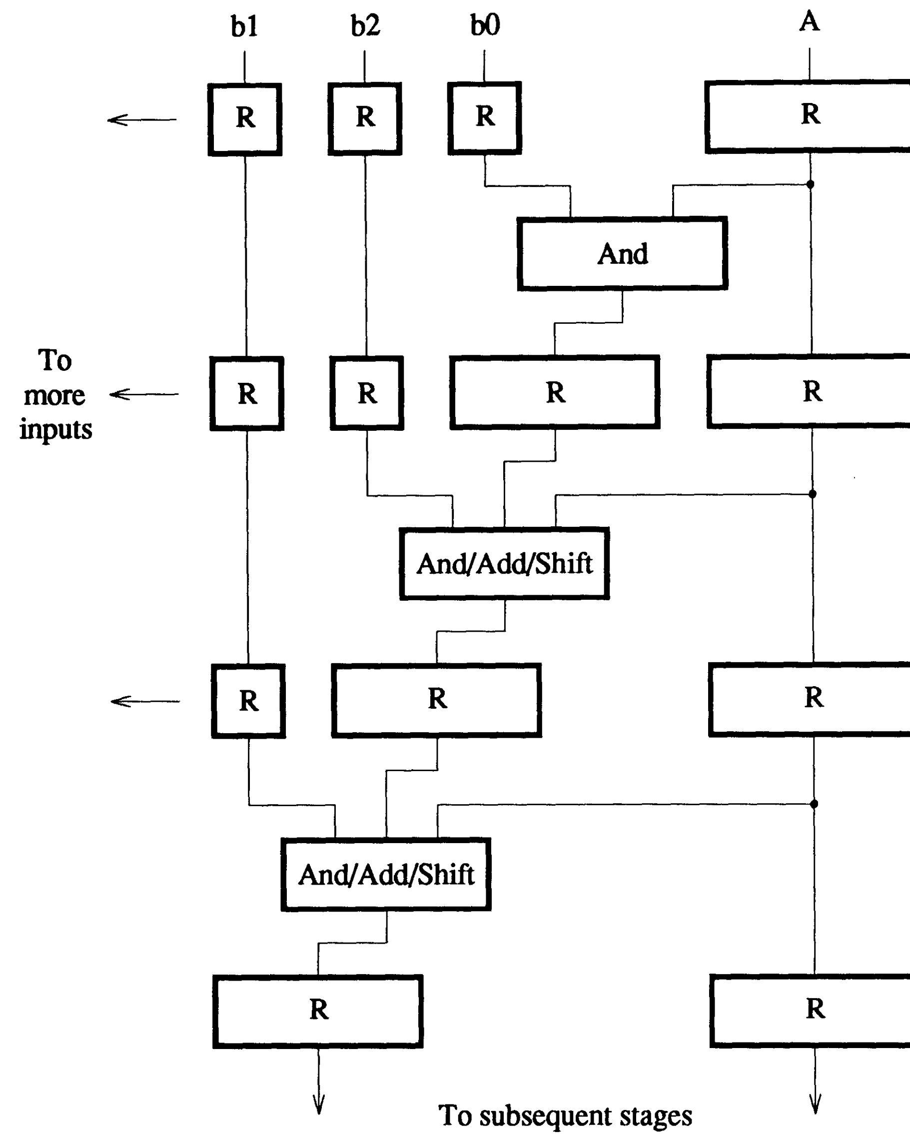


Figure 6-3: The First 3 Stages of an Array Multiplier

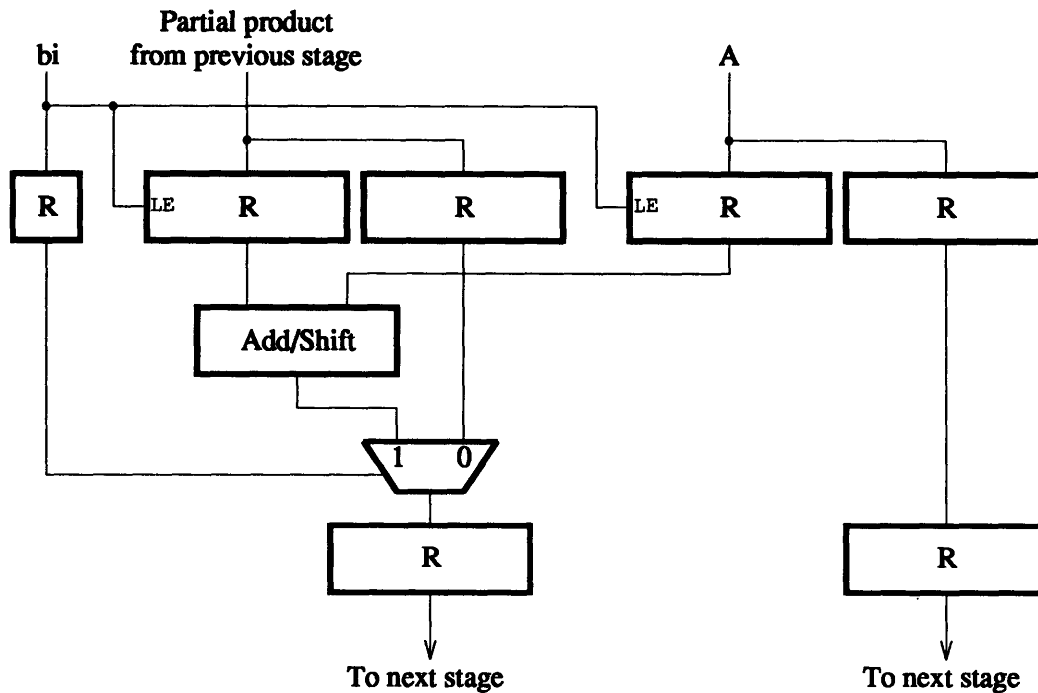


Figure 6-4: Precomputing the ith Stage of an Array Multiplier

next stage. For the case when $b_i = 1$, a shifted version of the input A is added to the partial product of the previous stage.

As is shown in Figure 6-4, the original And/Add/Shift block is now just an Add/Shift block, and it calculates a partial product only when $b_i = 1$. The input b_i drives the load-enable signal of the input registers so that when it is a 0, the input A and the partial product from the previous stage are prevented from propagating into the Add/Shift block. A multiplexor selects the correct partial product depending on the value of b_i and passes it to the next stage.

Assuming that the inputs to the multiplier have equal probability of occuring, a particular stage can be turned off 50% of the time. To get a 50% reduction in power for the entire multiplier, each stage must be precomputed. The savings in power, once again, is reduced because of the register duplication. Note that the addition of the multiplexors does not affect the power savings as they replaced the AND gates

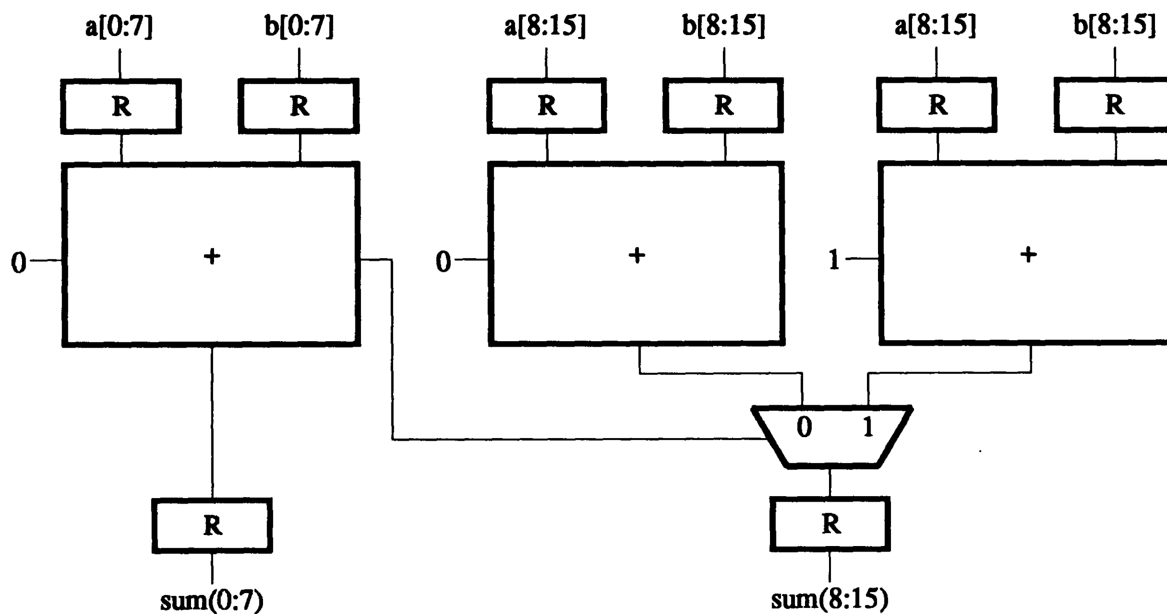


Figure 6-5: A Carry-Select Adder

that were in the original circuit.

The power dissipation of the array multiplier can be improved substantially if a combinational logic precomputation technique is used. This type of architecture is described in Section 6.3. In that architecture, the duplicated registers can be replaced with pass transistors or transmission gates, which are much smaller, and, hence do not dissipate as much power.

6.2.3 A Carry-Select Adder

Figure 6-5 shows a $16 - bit$ carry-select adder. The low-order $8 - bits$ are added just as in a normal adder. The remaining high-order bits are added in parallel, once assuming a carry-in of a 0 and once assuming a carry-in of a 1. When the carry of the low-order sum is finally computed, it selects the correct $8 - bit$ sum through a multiplexor. The carry-select adder is typically used in designs which require a fast adder. For the example shown, if the delay of the multiplexor is ignored, the

$16 - bit$ sum is computed with the same delay as an $8 - bit$ adder. The speed increase, however, comes at the expense of area.

We can precompute the carry that drives the select line of the multiplexor. Note that this signal is *not* an output of the circuit. The precomputation logic is as follows:

$$g_1 = a_7 \cdot b_7 \quad (6.1)$$

$$g_2 = \overline{a_7} \cdot \overline{b_7} \quad (6.2)$$

Clearly, when a_7 and b_7 are both high, the carry will be high, and we can turn off the high-order $8 - bit$ adder with a carry-in of a 0, as eventually, the multiplexor will select the other sum. Similarly, when a_7 and b_7 are both 0, the carry will be a 0 regardless of whether there were carries generated from the previous stages. Hence, we can turn off the adder with a carry-in of a 1 by setting its input register's load-enable signal to a 0 (not explicitly shown in the figure).

Assuming a uniform probability for the inputs, we can see from Equations 6.1 and 6.2 that we can turn off the adder with the carry-in of a 0 25% of the time, and, similarly, we can turn off the other adder another 25% of the time. Like the previous examples, we can include a_6 and b_6 in the precomputation logic and increase the percent of the time that we can turn off the adders. The precomputation logic equations can be determined directly from the generate and propagate terms of a carry-lookahead adder [19].

We also need to duplicate some of the input registers, in particular, those for inputs $a_8, b_8, \cdots, a_{15}, b_{15}$. If only one set of registers is used to drive both the high-order $8 - bit$ adders, then, when we disable those registers, the sum will be incorrect.

6.2.4 A Maximum Function

Figure 6-6 shows how one should implement a maximum function in order for it to be precomputable. The function computes $MAX(K, L)$. The two inputs K and L are compared and the output of the comparator drives the select line of a multiplexor

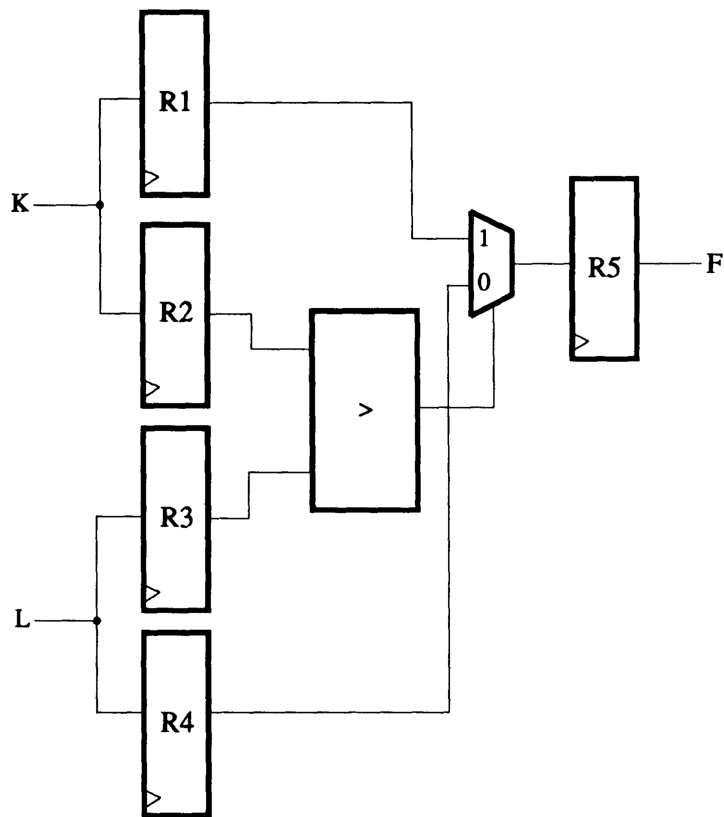


Figure 6-6: Precomputation of a Maximum Function

which chooses the correct input. The comparator can be precomputed, as was shown in Figure 4-5. Note, once again, we need to duplicate the input registers as they contain old or stale values during particular clock cycles.

6.3 Special Architectures

In this section, we describe additional precomputation architectures. We first present an architecture that is applicable to *all* logic circuits and does not require, for instance, that the inputs should be in the observability don't-care set in order to be disabled. This was the case for the architectures shown in Chapter 4. We also extend precomputation so that it can be used in combinational logic circuits.

6.3.1 Multiplexor-Based Precomputation

All logic functions can be written in a Shannon expansion as was shown in Chapter 2. For the function f with inputs $X = \{x_1, \cdots, x_n\}$, we can write:

$$f \; = \; x_1 \cdot f_{x_1} \; + \; \overline{x_1} \cdot f_{\overline{x_1}} \tag{6.3}$$

where f_{x_1} and $f_{\overline{x_1}}$ are the cofactors of f with respect to x_1.

Figure 6-7 shows an architecture based on Equation 6.3. We implement the functions f_{x_1} and $f_{\overline{x_1}}$. Depending on the value of x_1, only one of the cofactors is computed while the other is disabled by setting the load-enable signal of its input register. The input x_1 drives the select line of a multiplexor which chooses the correct cofactor.

The main advantage of this architecture is that it applies to *all* logic functions. The input x_1 in the example was chosen for the purpose of illustration. In fact, any input $x_1, \cdots, x_n$ could have been selected. Unlike the architectures described earlier, we do not require that the inputs being disabled should be don't-cares for the input conditions which we are precomputing. In other words, the inputs being disabled do not have to be in the observability don't-care set. A disadvantage of this architecture

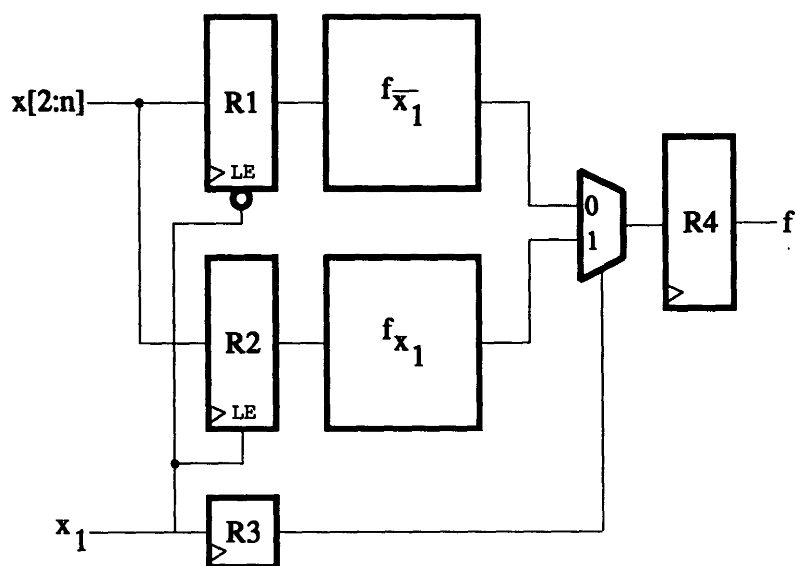


Figure 6-7: Precomputation Using the Shannon Expansion

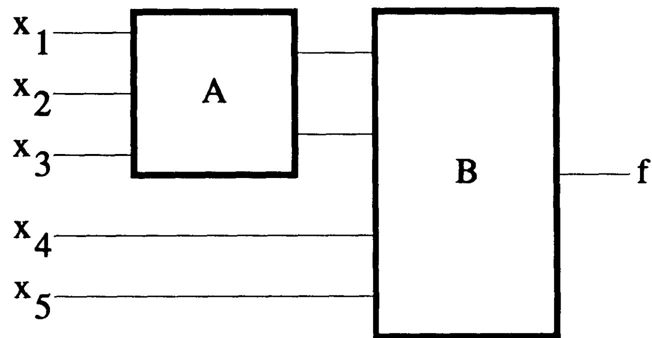
is that we need to duplicate the registers for the inputs not being used to turn off part of the logic. On the other hand, no precomputation logic functions have been added to the circuit.

The algorithm to select the best input for this architecture is also quite different. We will not discuss this algorithm in detail, except to mention that in this case, we are interested in finding the input that yields the most area efficient f_{x_1} and $f_{\overline{x_1}}$ functions.

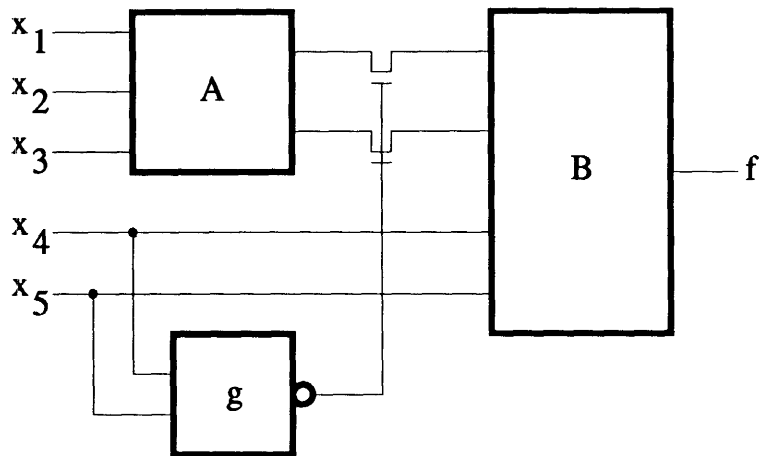
6.3.2 Combinational Logic Precomputation

The architectures described so far apply only to sequential circuits. We now describe precomputation of combinational circuits.

Suppose we have some combinational logic function f composed of two sub-functions **A** and **B** as shown in Figure 6-8(a). Suppose we also want to precompute this function with the inputs x_4 and x_5. Figure 6-8(b) shows how this can be accomplished. For simplicity, pass transistors, instead of transmission gates, are shown.



(a) Original Network



(b) Final Network

Figure 6-8: Combinational Logic Precomputation

The function g with inputs x_4 and x_5 drives the gates of the pass transistors. As in the previous architectures, $g = \overline{g_1 + g_2}$. Hence, when g is a 0, the pass transistors are turned off and the new values of logic block **A** are prevented from propagating into logic block **B**. The inputs x_4 and x_5 are also inputs to the logic block **B** just as in the original network in order to ensure that the output is set correctly.

For the combinational architecture, there is an implied delay constraint i.e., the pass transistors should be off *before* the new values of **A** are computed. In the example shown, the worst-case delay of the g block plus the arrival time of inputs x_4 or x_5 should be less than the best-case delay of logic block **A** plus the arrival time of the inputs x_1, x_2, or x_3. The *arrival time* of an input is defined as the time at which the input settles to its steady state value [7]. If the delay constraint is not met, then it may be necessary to delay the x_1, x_2, and x_3 inputs with respect to the x_4 and x_5 inputs in order to get the switching activity reduction in logic block **B**.

Chapter 7

Multiple-Cycle Precomputation

7.1 Introduction

The architectures presented in Chapter 4 can be referred to as single-cycle precomputation as they predict the outputs of the circuit one clock cycle before they are required. In this chapter, we present multiple-cycle precomputation.

In Section 7.2, we describe the basic idea behind multiple-cycle precomputation and discuss its advantages. We show some examples in Section 7.3.

7.2 Basic Strategy

It is possible to precompute output values that are not required in the next clock cycle, but are required two or more clock cycles later.

Consider the architecture of Figure 7-1. If the outputs of R_3 are not used except to compute f, then we can precompute the value of f using a subset of the inputs to the logic block **A**. If f can be precomputed for a set of input conditions, then for these conditions we can set the load-enable signal of R_2 to a 0. This will reduce switching activity not only in logic block **A**, but also in logic block **B**. Furthermore, we can single-cycle precompute the logic block **B** to get additional savings in power. Another

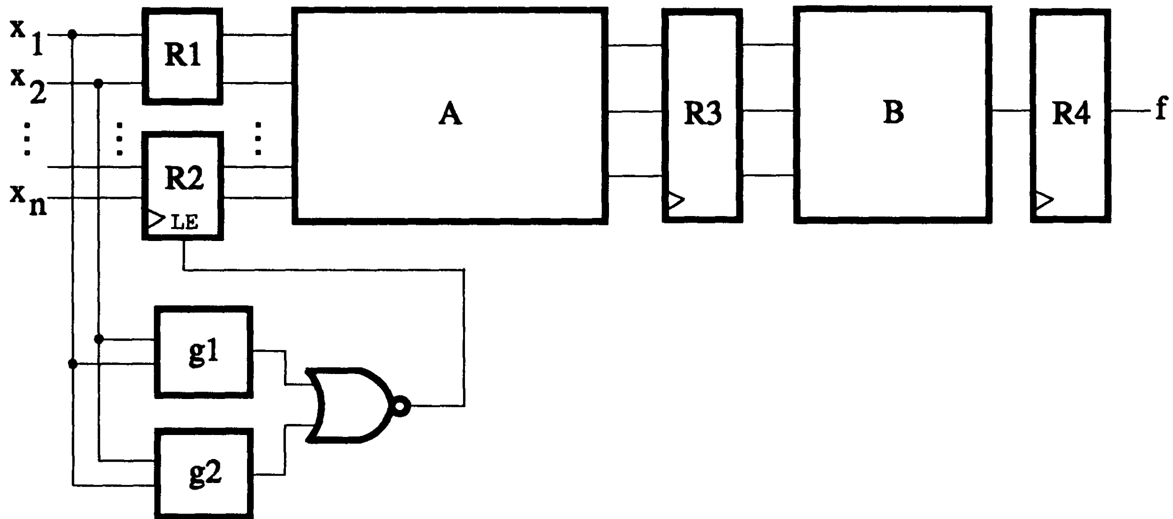


Figure 7-1: Multiple-Cycle Precomputation

advantage of multiple-cycle precomputation is that while the logic block **A** may not be precomputable, the overall function f may be precomputable, hence, resulting in a more powerful optimization.

7.3 Examples

We give examples illustrating multiple-cycle precomputation.

Consider the circuit of Figure 7-2. The function f computes $(C + D) > (X + Y)$ in two clock cycles. [1] Attempting to precompute $C + D$ or $X + Y$ using the methods described previously do not result in any savings because there are too many outputs to consider. However, two-cycle precomputation can reduce switching activity by 12.5% if the precomputation logic functions are as follows:

$$g_1 = C\langle n-1\rangle \cdot D\langle n-1\rangle \cdot \overline{X\langle n-1\rangle} \cdot \overline{Y\langle n-1\rangle} \quad (7.1)$$

$$g_2 = \overline{C\langle n-1\rangle} \cdot \overline{D\langle n-1\rangle} \cdot X\langle n-1\rangle \cdot Y\langle n-1\rangle \quad (7.2)$$

[1] + in the figure stands for addition.

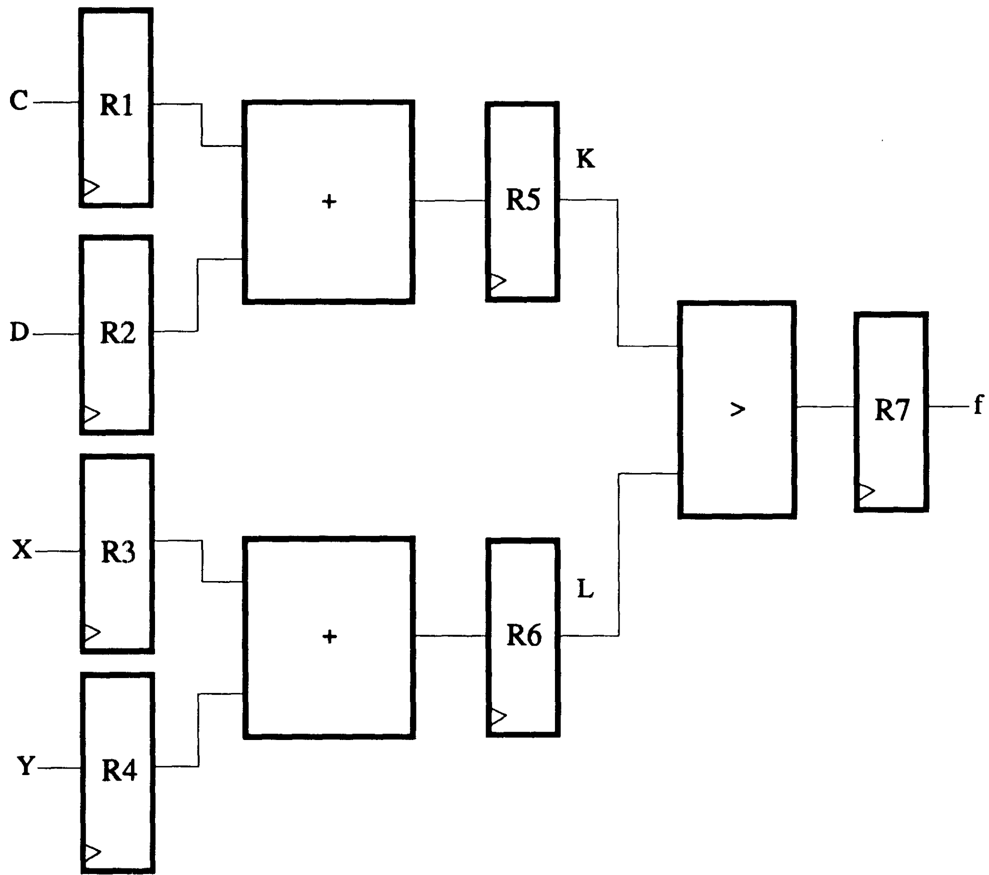


Figure 7-2: An Add-Compare Function

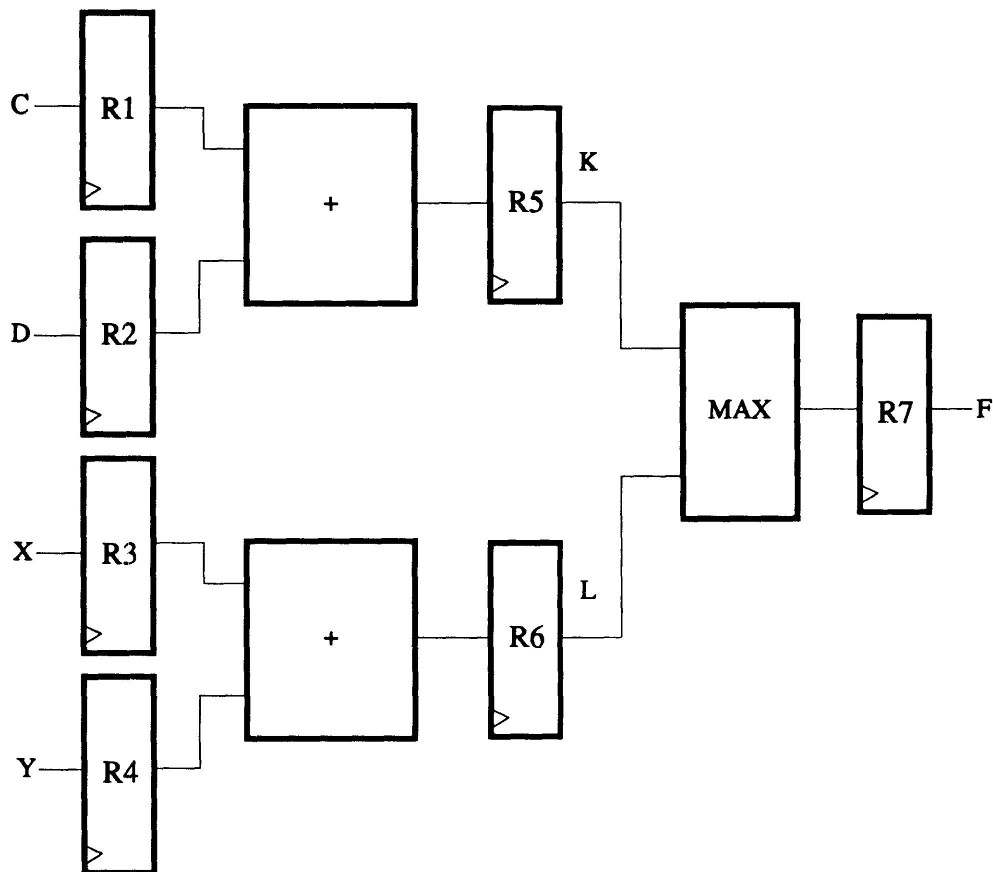


Figure 7-3: An Add-Maximum Function

where g_1 and g_2 satisfy the constraints of Equations 4.1 and 4.2. Since $prob(g_1 + g_2) = \frac{2}{16} = 0.125$, we can disable the loading of registers $C\langle n-2:0\rangle$, $D\langle n-2:0\rangle$, $X\langle n-2:0\rangle$, and $Y\langle n-2:0\rangle$ 12.5% of the time. This percent can be increased to over 45% by using $C\langle n-2\rangle$ through $Y\langle n-2\rangle$. We can, in addition, use single-cycle precomputation (as illustrated in Figure 4-5) to further reduce the switching activity in the comparator of Figure 7-2.

Next, consider the circuit of Figure 7-3. The multiple-output function F computes $MAX(C+D,\ X+Y)$ in two clock cycles. We can use exactly the same g_1 and g_2 functions as those for the add-compare function, except that g_1 is used to disable the loading of registers $X\langle n-2:0\rangle$ and $Y\langle n-2:0\rangle$, and g_2 is used to disable the loading

of $C\langle n-2:0\rangle$ and $D\langle n-2:0\rangle$. We exploit the fact that if $C+D>X+Y$, there is no need to compute $X+Y$, and vice versa. Finally, we can implement the MAX function as shown in Figure 6-6 and precompute it.

Chapter 8

Experimental Results

8.1 Introduction

In this chapter, we present experimental results on various sequential circuits. The examples that are shown in Section 8.2 were described using the Berkeley Logic Interchange Format, a language to specify combinational and sequential logic functions. The algorithms presented in Chapter 5 were implemented using **sis** [16], a C-based programming environment for sequential logic synthesis and optimization. All experiments were performed on a Sun SPARC-10 workstation.

8.2 Results

We present results for datapath circuits in Table 8.1. We show results for random logic circuits in Table 8.2. For each circuit, the number of literals, logic levels, and power of the original circuit, the number of bits, literals, and logic levels of the precomputation logic, the final power, and the percent reduction in power are shown.

All power estimates are in micro-Watts and are computed using the techniques described in Chapter 3. A supply voltage of 5V and a clock frequency of 20MHz were assumed. Load capacitances were obtained by sizing the transistors in each logic

CKT	Original			Precompute Logic			Optimized	
	Lits	Levs	Pwr	Bits	Lits	Levs	Pwr	% Red
comp16	286	7	1281	2	4	2	965	25
				4	8	2	683	47
				6	12	2	550	57
				8	16	2	518	60
				10	20	2	538	58
priority16	126	16	455	1	1	1	381	16
				2	3	2	270	41
				3	6	2	209	54
				4	10	2	190	58
				5	15	2	187	59
				6	21	2	196	57
add_comp16	3026	8	6941	4/0	8	2	6346	9
				4/8	24	4	5711	18
				8/0	51	4	4781	31
				8/8	67	6	3933	43
max16	350	9	1744	8	16	2	1281	27
csa16	975	10	2945	2	4	2	2958	0
				4	11	4	2775	6
				6	18	4	2676	9
				8	25	5	2644	10
add_max16	3090	9	7370	4/0	8	2	7174	3
				4/8	24	4	6751	8
				8/0	51	4	6624	10
				8/8	67	6	6116	17

Table 8.1: Power Reductions for Datapath Circuits

gate so that its delay was roughly equal to the delay of a minimum-sized inverter. Capacitance values were based on a 2μ CMOS technology. The switching activity of each gate was computed using symbolic simulation and was based on a zero delay model. The precomputation logic was optimized for area using the rugged script of sis.

Power dissipation decreases for almost all cases. For circuit **comp16**, a 16-bit parallel comparator, the power decreases by as much as 60% when 8 of the 32 inputs are used in the precomputation logic. For all examples, as is indicated by the literal count, the precomputation logic is much smaller than the original circuit.

CKT	Original			Precompute Logic			Optimized	
	Lits	Levs	Pwr	Bits	Lits	Levs	Pwr	% Red
9symml	267	8	1452	7	41	8	1429	2
cm150a	61	5	744	1	1	1	552	26
cm152a	28	4	370	9	2	1	261	29
i2	230	4	5606	22	30	3	2324	59
majority	12	4	173	3	4	2	124	28
mux	54	6	715	1	1	1	533	25
parity	60	5	187	0	0	0	187	0
t481	1028	11	1562	8	16	3	1393	11

Table 8.2: Power Reductions for Random Logic Circuits

In Table 8.1, we also show the improvement in power dissipation as more inputs are included in the precomputation logic. As expected, however, the power gains diminish once we reach a certain point. If more inputs are added to the precomputation logic, the power savings are offset by the increased complexity of the precomputation logic and the fact that fewer inputs are being disabled.

Multiple-cycle precomputation results are given for circuits **add_comp16** and **add_max16**. For circuit **add_comp16**, for instance, 4/8 bits for the precomputation logic indicate that 4 bits were used to precompute the adders in the first cycle and 8 bits were used to precompute the comparator in the next cycle.

The random logic circuits in Table 8.2 are from the MCNC benchmark set. For circuit **i2**, we get a 59% reduction in power dissipation. The **parity** function, also shown in Table 8.2, is a circuit that is not precomputable. This function counts the number of ones in a bit string, and its output is a 0 if there are an even number of ones in the string or a 1 if there are an odd number of ones. One can never predict the output of the parity function just by looking at a few of its inputs. *All* of the inputs must be known to determine its output.

Chapter 9

Conclusion and Future Work

We presented a method of precomputing the outputs of a sequential circuit one clock cycle before they were required and used this knowledge to reduce power dissipation in the succeeding clock cycle. Different architectures that exploited precomputation were presented.

Precomputation increases circuit area and adversely affects circuit performance. In order to keep the area and delay increases small, we synthesized the precomputation logic so that it depended on a small set of inputs. When the logic block had a large number of outputs, it was worthwhile to selectively apply precomputation-based optimization on a small set of complex outputs. This selective partitioning entailed a duplication of combinational logic and registers, and the savings in power was offset by this duplication.

We presented special circuits and architectures that showed the power of this optimization method. We also extended the idea of precomputation in order to predict the outputs of a circuit two clock cycles ahead of time.

Although circuit area and delay increased, significant reductions in power dissipation were obtained. This suggests that synthesis, optimization, and even design for low power may be fundamentally different from the traditional problem of synthesis, optimization, and design for area and delay. Clearly, precomputation is not good in

terms of area and delay, but it does extremely well for low power.

Several issues presented in this work are further being researched. For instance, new precomputation architectures are being explored. Algorithms that can automate the multiplexor-based and multiple-cycle precomputation architectures are also being studied. Furthermore, ensuring that precomputation-based logic circuits are fully testable is another important problem. Finally, more work is needed in trying to find special types of circuits that are precomputable.

Bibliography

[1] S. B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

[2] P. Ashar, S. Devadas, and K. Keutzer. Path-Delay-Fault Testability Properties of Multiplexor-Based Networks. *INTEGRATION, the VLSI Journal*, 15(1):1–23, July 1993.

[3] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A Multiple-Level Logic Optimization System. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1062-1081, November 1987.

[4] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[5] A. Chandrakasan, T. Sheng, and R. W. Brodersen. Low Power CMOS Digital Design. *IEEE Journal of Solid State Circuits*, 27(4):473-484, April 1992.

[6] J. Cong, C. K. Koh, and K. S. Leung. Wiresizing and Driver Sizing for Performance and Power Optimization. In *Proceedings of the 1994 Int'l Conference on Low Power Design*, pages 81–86, April 1994.

[7] S. Devadas, A. Ghosh, and K. Keutzer. *Logic Synthesis.* McGraw-Hill, 1994.

[8] A. Ghosh, S. Devadas, K. Keutzer, and J. White. Estimation of Average Switching Activity in Combinational and Sequential Circuits. In *Proceedings of the* 29th *Design Automation Conference*, pages 253–259, June 1992.

[9] L. Glasser and D. Dobberpuhl. *The Design and Analysis of VLSI Circuits.* Addison-Wesley, 1985.

[10] C. Y. Lee. Representation of Switching Circuits by Binary-Decision Programs. *Bell Systems Technical Journal*, 38(4):985–999, July 1959.

[11] J. Monteiro, S. Devadas, and A. Ghosh. Retiming Sequential Circuits for Low Power. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 398–402, November 1993.

[12] J. Monteiro, S. Devadas, and B. Lin. A Methodology for Efficient Estimation of Switching Activity in Sequential Logic Circuits. In *Proceedings of the* 31st *Design Automation Conference*, June 1994.

[13] F. Najm. Transition Density, A Stochastic Measure of Activity in Digital Circuits. In *Proceedings of the* 28th *Design Automation Conference*, pages 644–649, June 1991.

[14] A. Papoulis. *Probability, Random Variables and Stochastic Processes.* McGraw-Hill, third edition, 1991.

[15] K. Roy and S. Prasad. SYCLOP: Synthesis of CMOS Logic for Low Power Applications. In *Proceedings of the Int'l Conference on Computer Design: VLSI in Computers and Processors*, pages 464–467, October 1992.

[16] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *Proceedings of the Int'l Conference on Computer Design: VLSI in Computers and Processors*, pages 328–333, October, 1992.

[17] A. Shen, S. Devadas, A. Ghosh, and K. Keutzer. On Average Power Dissipation and Random Pattern Testability of Combinational Logic Circuits. In *Proceedings*

of the Int'l Conference on Computer-Aided Design, pages 402–407, November 1992.

[18] C. H. Tan and J. Allen. Minimization of Power in VLSI Circuits Using Transistor Sizing, Input Ordering, and Statistical Power Estimation. In *Proceedings of the 1994 Int'l Conference on Low Power Design*, pages 75–80, April 1994.

[19] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design.* Addison-Wesley, 1988.