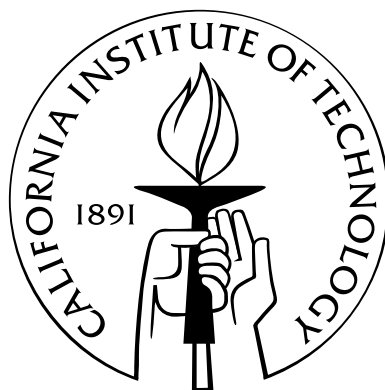


# Cyclic Combinational Circuits

Dissertation by  
Marc D. Riedel

In Partial Fulfillment of the Requirements  
for the Degree of  
Doctor of Philosophy



California Institute of Technology  
Pasadena, California

2004

(Submitted May 4, 2004)

© 2004

Marc D. Riedel

All Rights Reserved

# Acknowledgements

It has been said that the better people are at understanding mathematics the worse they are at understanding human behavior. If so, then perhaps it was a compliment when I declared that my father – Prof. Ivo Rosenberg, a mathematician – would make the worst psychologist in the universe. This characterization aside, my father is the most erudite and principled person that I have known. To him I owe my education, my values, and my passion for research.

To my advisor, Prof. Jehoshua Bruck, I owe my entire research career. Throughout these memorable and rewarding years at Caltech, he has provided unwavering guidance, support and inspiration.

My research was supported in part by a grant from the National Human Genome Research Institute (Grant no. P50 HG02370), and by the Lee Center for Advanced Networking at Caltech.

# Contents

<b>Acknowledgements</b>	<b>iii</b>
<b>Abstract</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 A New Idea . . . . .	1
1.2 Prior Work . . . . .	7
1.2.1 The Early Era . . . . .	7
1.2.2 The Later Era . . . . .	10
1.3 Overview . . . . .	12
1.3.1 Theory . . . . .	12
1.3.2 Practice . . . . .	13
<b>2 Framework</b>	<b>16</b>
2.1 Circuit Model . . . . .	16
2.1.1 Functional Behavior . . . . .	18
2.1.2 Temporal Behavior . . . . .	19
2.2 Analysis Framework . . . . .	19
2.2.1 Ternary Extension . . . . .	20
2.2.2 Fixed Point . . . . .	22
2.2.3 Explicit Analysis . . . . .	25
2.2.4 Complexity . . . . .	30
<b>3 Theory</b>	<b>32</b>

3.1	Criteria for Optimality . . . . .	33
3.2	Fan-in Lower Bound . . . . .	34
3.3	Improvement Factor . . . . .	37
3.4	Examples . . . . .	37
3.4.1	Optimality . . . . .	40
3.4.2	Acyclic Lower Bound . . . . .	40
3.4.3	A Generalization . . . . .	42
3.4.4	Variants . . . . .	43
3.5	A Minimal Cyclic Circuit with Two Gates . . . . .	45
3.6	Circuits with Multiple Cycles . . . . .	46
3.6.1	A Cyclic Circuit with Two Cycles . . . . .	46
3.6.2	Analysis in Arbitrary Terms . . . . .	47
3.6.3	A Circuit Three-Fifths the Size . . . . .	50
3.6.4	A Circuit One-Half the Size . . . . .	53
3.7	Summary . . . . .	54
<b>4</b>	<b>Analysis</b>	<b>55</b>
4.1	Decision Diagrams . . . . .	58
4.2	Controlling Values . . . . .	59
4.3	Analysis . . . . .	62
4.3.1	Symbolic Analysis Algorithm . . . . .	62
4.3.2	Examples . . . . .	66
<b>5</b>	<b>Synthesis</b>	<b>72</b>
5.1	Logic Minimization . . . . .	74
5.2	Multi-Level Logic . . . . .	76
5.3	Substitutional Orderings . . . . .	78
5.4	Branch-and-Bound Algorithms . . . . .	81
5.4.1	The “Break-Down” Approach . . . . .	82
5.4.2	The “Build-Up” Approach . . . . .	85
5.5	Example: 7-Segment Decoder . . . . .	87

<b>6 Discussion</b>	<b>93</b>
6.1 High-Level Design . . . . .	95
6.2 Data Structures . . . . .	96
<b>Appendix A: XNF Representation</b>	<b>99</b>
<b>Appendix B: Synthesis Results</b>	<b>102</b>
B-1 Optimization of Area at the Network Level . . . . .	102
B-2 Optimization of Area at the Gate Level . . . . .	103
B-3 Joint Optimization of Area and Delay at the Gate Level . . . . .	105
<b>Bibliography</b>	<b>108</b>

# Abstract

A collection of logic gates forms a *combinational* circuit if the outputs can be described as Boolean functions of the current input values only. Optimizing combinational circuitry, for instance, by reducing the number of gates (the area) or by reducing the length of the signal paths (the delay), is an overriding concern in the design of digital integrated circuits.

The accepted wisdom is that combinational circuits must have *acyclic* (i.e., loop-free or feed-forward) topologies. In fact, the idea that “combinational” and “acyclic” are synonymous terms is so thoroughly ingrained that many textbooks provide the latter as a definition of the former. And yet simple examples suggest that this is incorrect. In this dissertation, we advocate the design of *cyclic* combinational circuits (i.e., circuits with loops or feedback paths). We demonstrate that circuits can be optimized effectively for area and for delay by introducing cycles.

On the theoretical front, we discuss lower bounds and we show that certain cyclic circuits are one-half the size of the best possible equivalent acyclic implementations. On the practical front, we describe an efficient approach for *analyzing* cyclic circuits, and we provide a general framework for *synthesizing* such circuits. On trials with industry-accepted benchmark circuits, we obtained significant improvements in area and delay in nearly all cases. Based on these results, we suggest that it is time to re-write the definition: combinational might well mean cyclic.

# Chapter 1

## Introduction

New ideas pass through three periods:

1. *“It can’t be done.”*
2. *“It probably can be done, but it’s not worth doing.”*
3. *“I knew it was a good idea all along!”*

–Arthur C. Clarke (1917– )

### 1.1 A New Idea

The field of digital circuit design encompasses a broad range of topics, from semiconductor physics to system-level architecture. At the *logic level*, a circuit is viewed as a network of *gates* and *wires* that processes time-varying, discrete-valued signals – most commonly two-valued signals, designated as “0” and “1”. Open any textbook on logic design, and you will find digital circuits classified into two types:

- A **combinational** circuit has output values that depend only on the current values applied to the inputs.
- A **sequential** circuit has output values that depend on the entire sequence of values, past and current, applied to the inputs.

Thus, a sequential circuit can store information, whereas a combinational circuit cannot. Given these *behavioral* definitions, the textbooks describe a *structural* implementation of these types:



- A **combinational** circuit consists of an *acyclic* configuration of logic gates, i.e., it contains only feed-forward paths.
- A **sequential** circuit consists of a *cyclic* configuration of logic gates and memory elements, i.e., it contains loops or feedback paths.

This conforms to intuition. Logic gates are, by definition, feed-forward devices, as illustrated in Figure 1.1. In a feed-forward circuit, such as that shown in Figure 1.2,

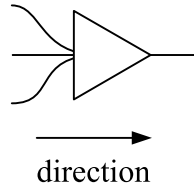


Figure 1.1: A logic gate is a feed-forward device.

the input values propagate forward and determine the values of the outputs. The outcome can be asserted regardless of the prior values of the wires, and so independently of the past sequence of inputs. The circuit is clearly combinational.

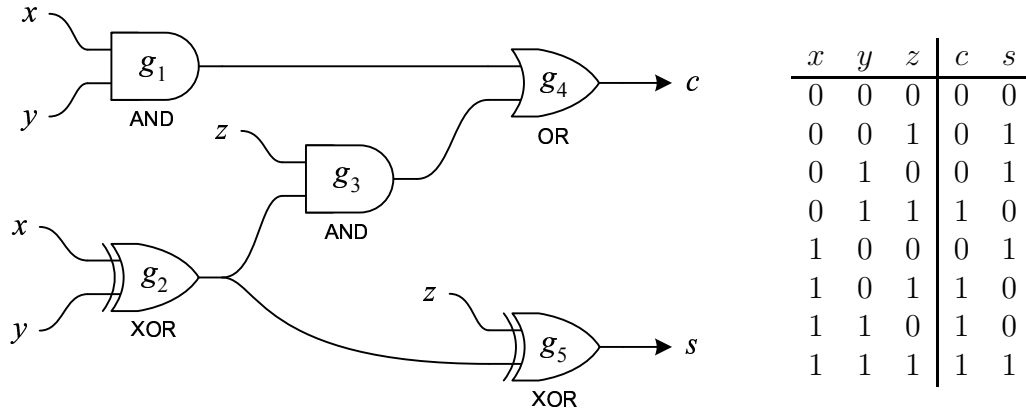


Figure 1.2: A feed-forward circuit behaves combinationaly.

In a circuit with feedback, the behavior is less transparent. A common approach is to characterize the output values and the *next* state in terms of the input values and the *current* state. The current state, in turn, depends on the prior sequence of inputs (starting from some known initial state). As an example, the cyclic circuit shown

in Figure 1.3 implements a one-bit memory element, called a *latch*. This circuit is clearly sequential.

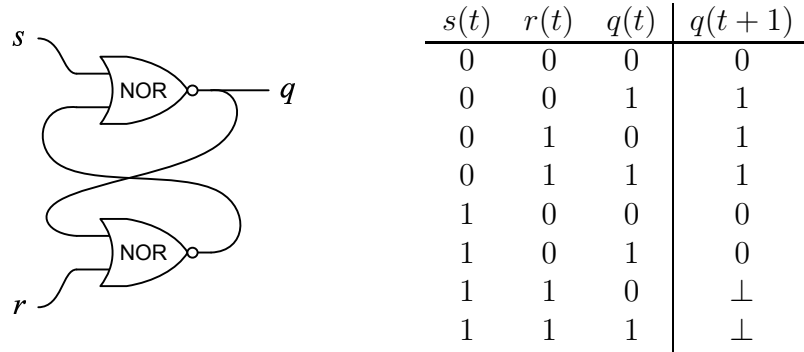


Figure 1.3: A circuit with feedback. With inputs  $s(t)$  and  $r(t)$ , and current state  $q(t)$ , the next state is  $q(t+1)$ . Here  $\perp$  indicates an indeterminate value.

Although counter-intuitive, could a combinational circuit be designed *with* feedback paths?

***“It can’t be done.”***

One might argue that with feedback, we cannot determine the output values without knowing the current state, and so the circuit must be sequential. This view is illustrated in Figure 1.4.

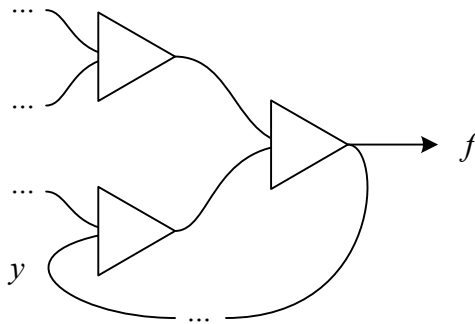


Figure 1.4: A circuit with feedback. How can we determine the output  $f$  without knowing the value of  $y$  in a feedback path?

This specious argument can easily be put to rest with the circuit in Figure 1.5. It consists of an AND gate and an OR gate connected in a cycle, both with input  $x$ .

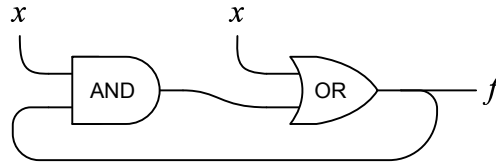


Figure 1.5: A (useless) cyclic combinational circuit.

Recall that the output of an AND gate is 0 iff either input is 0; the output of an OR gate is 1 iff either input is 1. Consider the two possible values of  $x$ . On the one hand, if  $x = 0$  then the output of the AND gate is fixed at 0; the input from the OR gate has no influence, as shown in Figure 1.6 (a). On the other hand, if  $x = 1$  then the output of the OR gate is fixed at 1; the input from the AND gate has no influence, as shown in Figure 1.6 (b). Although useless, this circuit is cyclic and combinational. The value of the output  $f$  is determined by the current input value  $x$  (actually  $f = x$ ) regardless of the prior state and independently of all timing assumptions.

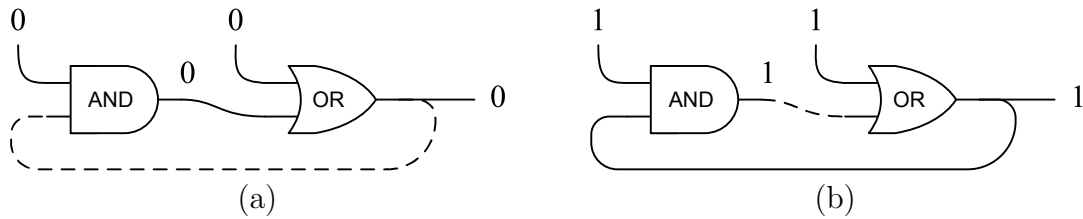


Figure 1.6: The circuit of Figure 1.5 with (a)  $x = 0$ , and (b)  $x = 1$ .

***“It probably can be done, but it’s not worth doing.”***

Although conceptually possible, one might argue that there is *no point* in designing combinational circuits with feedback. Why should one incorporate a feedback path in the computation of the output values? By definition the values fed back depend upon the prior state of the circuit, which we want to *ignore* in a combinational design.

A convincing example suggesting otherwise is shown in Figure 1.7. It consists of six alternating AND and OR gates, with inputs  $x_1, x_2, x_3$  repeated. To show that the circuit is combinational, we label the feedback path with an unknown value  $y$ , as

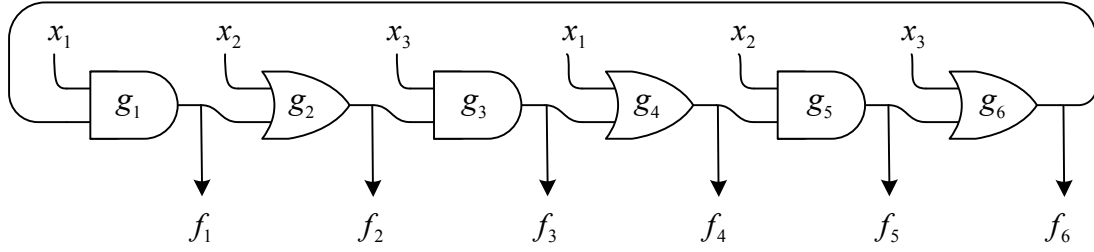


Figure 1.7: A cyclic combinational circuit due to Rivest [35].

shown in Figure 1.8. We compute

$$\begin{aligned}
 f_1 &= x_1 y \\
 f_2 &= x_2 + f_1 = x_2 + x_1 y \\
 f_3 &= x_3 f_2 = x_3(x_2 + x_1 y) \\
 f_4 &= x_1 + f_3 = x_1 + x_3(x_2 + x_1 y) = x_1 + x_2 x_3 \\
 f_5 &= x_2 f_4 = x_2(x_1 + x_2 x_3) = x_2(x_1 + x_3) \\
 f_6 &= x_3 + f_5 = x_3 + x_2(x_1 + x_3) = x_3 + x_1 x_2.
 \end{aligned}$$

(Here addition represents OR and multiplication represents AND.) We see that  $f_4$ , and consequently  $f_5$  and  $f_6$ , do not depend upon the unknown value. Thus, we compute

$$\begin{aligned}
 f_1 &= x_1 f_6 = x_1(x_3 + x_1 x_2) = x_1(x_2 + x_3) \\
 f_2 &= x_2 + f_1 = x_2 + x_1(x_2 + x_3) = x_2 + x_1 x_3 \\
 f_3 &= x_3 f_2 = x_3(x_2 + x_1 x_3) = x_3(x_1 + x_2).
 \end{aligned}$$

Each output depends on the current input values, not on the prior values, and so the circuit is combinational.

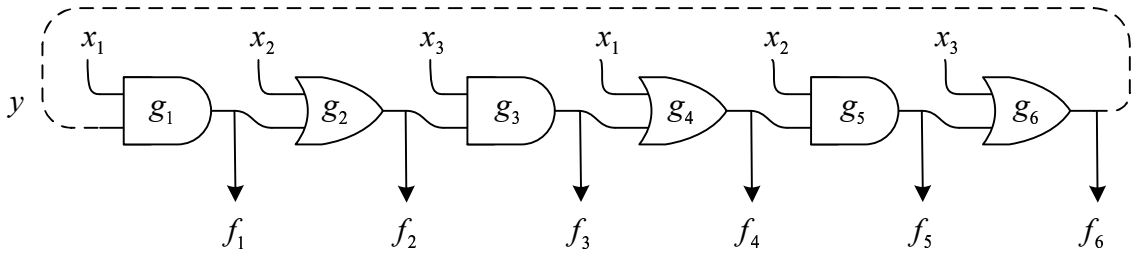


Figure 1.8: Analyzing the circuit of Figure 1.7.

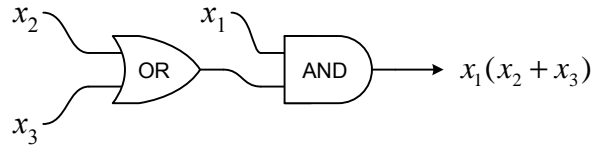


Figure 1.9: With fan-in two gates, two gates are needed to compute  $x_1(x_2 + x_3)$ .

Unlike the circuit in Figure 1.5, this one computes something useful. The six output functions are distinct, and each depends on all three input variables. Moreover, we can show that this cyclic circuit has *fewer* gates than any equivalent acyclic circuit. To see this, note that any acyclic configuration contains at least one gate producing an output function that does not depend on the output of any other gate producing an output function. (If this were not the case, then every output gate would depend upon another and so the circuit would be cyclic.) With fan-in two gates, it takes two gates to compute any one of the six functions by itself. This is illustrated in Figure 1.9. We conclude that an acyclic implementation of the six functions requires seven gates, compared to the six in the cyclic circuit.

***“I knew it was a good idea all along!”***

The circuit in Figure 1.7 was presented by Rivest in 1977, in a paper less than a page long [35]. His work on the topic, as well as that of a few others in the 1960s, seems to have gone largely unnoticed by theoreticians and practitioners alike. And yet his example hints at a fundamental misconception in the field, namely that “combinational” and “acyclic” are synonymous terms. In this dissertation, we demonstrate not only that it is feasible to design combinational circuits with cyclic topologies, but it is generally advantageous to do so.

## 1.2 Prior Work

### 1.2.1 The Early Era

Gates are a convenient abstraction, introduced for digital electronic circuits. In an earlier era, people studied switching circuits, built from electro-mechanical relays. A relay is device that conducts current if it is set to “on” (corresponding to a logical input of 1), and does not conduct current if it is set to “off” (corresponding to a logical input of 0). The device does not have an intrinsic direction; it will conduct current in either direction. The symbol for a relay is shown in Figure 1.10 A switching circuit evaluates to logical 1 if there is a conducting path between a designated “source” point and a designated “drain” point.

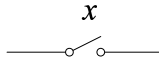


Figure 1.10: A contact relay.

Switching circuits were the subject of seminal papers by Claude Shannon: the analysis of such circuits in 1938 [39] and the synthesis of such circuits in 1949 [40]. The circuits of Shannon’s day often had cyclic topologies. Since relays are directionless, cycles do not pose any problem. Consider the *bridge* circuit shown in Figure 1.11. The logical function implemented between points S and D is

$$f(x_1, x_2, x_3, x_4, x_5) = x_1x_4 + x_1x_3x_5 + x_2x_5 + x_2x_3x_4.$$

It may be shown this circuit has fewer switches than is possible with an acyclic topology.

It was accepted that cycles were an important feature in the design of switching circuits. In 1953, Shannon described a cyclic switching circuit with 18 contacts that computes all 16 Boolean functions of two inputs, and he proved that this circuit is optimal [41].

In his Ph.D. dissertation in 1960, Short applied an abstract graphical model to the study of switching circuits [45]. Implicitly, his model imposes a direction on the

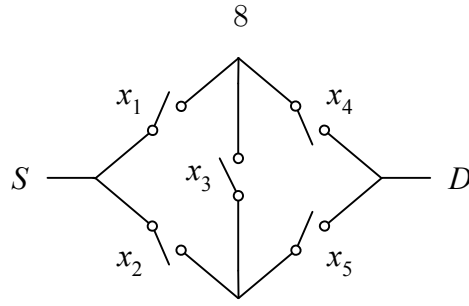


Figure 1.11: A switching circuit with a cyclic topology.

switching elements. It is equivalent to a form of binary decision diagram now known as a *zero-suppressed* decision diagram [30]. In this context, Short argued that cyclic designs are necessary for the minimal forms.

In recent years, binary decision diagrams have come to the fore as perhaps the most successful data structure for representing Boolean functions [7]. Short’s work suggests that feedback might be useful in optimizing binary decision diagrams, a topic of future research that we return to in Chapter 6.

In the 1960’s, as the research community was shifting its focus to the now-familiar model of directed logic gates (AND, OR, NOT, etc.), researchers naturally pondered the implication of cyclic designs. In 1963, McCaw presented a thesis for his Engineer’s Degree titled “Loops in Directed Combinational Switching Networks” [26]. He begins with an example, the cyclic circuit shown in Figure 1.12 consisting of two AND gates and two OR gates, with five inputs and two outputs. His argument for combinationality is in the same vein as that given above for Rivest’s circuit:

$$\begin{aligned} f_1 &= a + b + x(c + d + \bar{x}f_1) = a + b + x(c + d) \\ f_2 &= c + d + \bar{x}(a + b + xf_2) = c + d + \bar{x}(a + b). \end{aligned}$$

As with Rivest’s circuit, McCaw argues that his circuit has fewer AND/OR gates than is possible with an acyclic circuit implementing the same functions. In his thesis, he grapples with the different implications of cyclic topologies for circuits with logic gates vs. undirected switching elements. As an example, he transforms a switching circuit in Short’s dissertation, consisting of 7 switching elements, into a cyclic logic circuit consisting of 16 AND/OR gates.

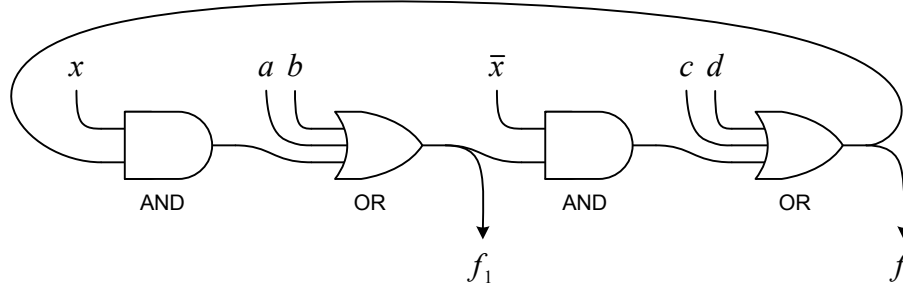


Figure 1.12: A cyclic combinational circuit due to McCaw.

In 1970, Kautz (Short’s Ph.D. advisor at Stanford) presented a short paper on the topic of feedback in circuits with logic gates [17]. He described a cyclic circuit consisting of 6 fan-in two NOR gates with three inputs and three outputs. Although plausible, his circuit is not combinational according to the rigorous model that we propose. (It assumes that all wires have definite Boolean values at the outset.)

In 1971, Huffman discussed feedback in linear threshold networks. He claimed that an arbitrarily large number of input variables can be complemented in a network containing a single NOT element, provided that feedback is used [15]. This improved upon an earlier result by Markov, demonstrating that  $k$  NOT elements suffice to generate the complements of  $2^k - 1$  variables [25]. As with Kautz’s example, Huffman’s is not combinational in the sense that we understand it. Still, in an insightful commentary on his and Kautz’s work, he hinted at the possible implications,

*“At this time, these [cyclic] examples are isolated ones. They do, however, provide tantalizing glimpses into an imaginable area of future research.”*

In 1977, Rivest presented a general version of the circuit in Figure 1.7, as well as the argument for its optimality given above [35]. For any odd integer  $n$  greater than 1, the general circuit consists of  $n$  fan-in two AND gates alternating with  $n$  fan-in two OR gates, with input variables  $x_1, \dots, x_n$  arranged as in Figure 1.7. It produces  $2n$  distinct output functions, each of which depends on all  $n$  input variables. He proved that any acyclic circuit implementing the same  $2n$  output functions requires at least  $3n - 2$  fan-in two gates. Thus, asymptotically, his cyclic implementation is at most two-thirds the size of the best possible acyclic implementation. In Section 3,



we analyze Rivest’s construction, present variants and extensions, and generalize the argument of optimality

### 1.2.2 The Later Era

More recently, practitioners observed that cycles sometimes occur in combinational circuits synthesized from high-level descriptions. In such examples, feedback either is inadvertent or else is carefully contrived. For instance, occasionally it is introduced during resource-sharing optimizations at the level of functional units [47]. In these circuits, there is explicit “control” circuitry governing the interaction between “functional” units.

Consider the example in Figure 1.13. Here we have an input *word*  $X$  (that is, a bundle of wires carrying several bits of information) and a control input  $c$ . There are two functional units,  $F$  and  $G$ , each of which performs a word-wise operation. If  $c$  is 1, then the circuit computes

$$G(F(X)),$$

while if it is 0, it computes

$$F(G(X)).$$

Suppose that  $X = (x_1, \dots, x_n)$  is an  $n$ -bit word, representing the integer

$$x_1 + 2x_2 + \dots + 2^{n-1}x_n.$$

Here  $F(X)$  might be an exponentiation

$$F(X) = 2^X \mod 2^n,$$

and  $G(X)$  might be a left-shift (division by 2),

$$G(X) = \left\lfloor \frac{X}{2} \right\rfloor.$$

The circuit either performs a left-shift followed by an exponentiation, or an exponen-

tiation followed by a left-shift.

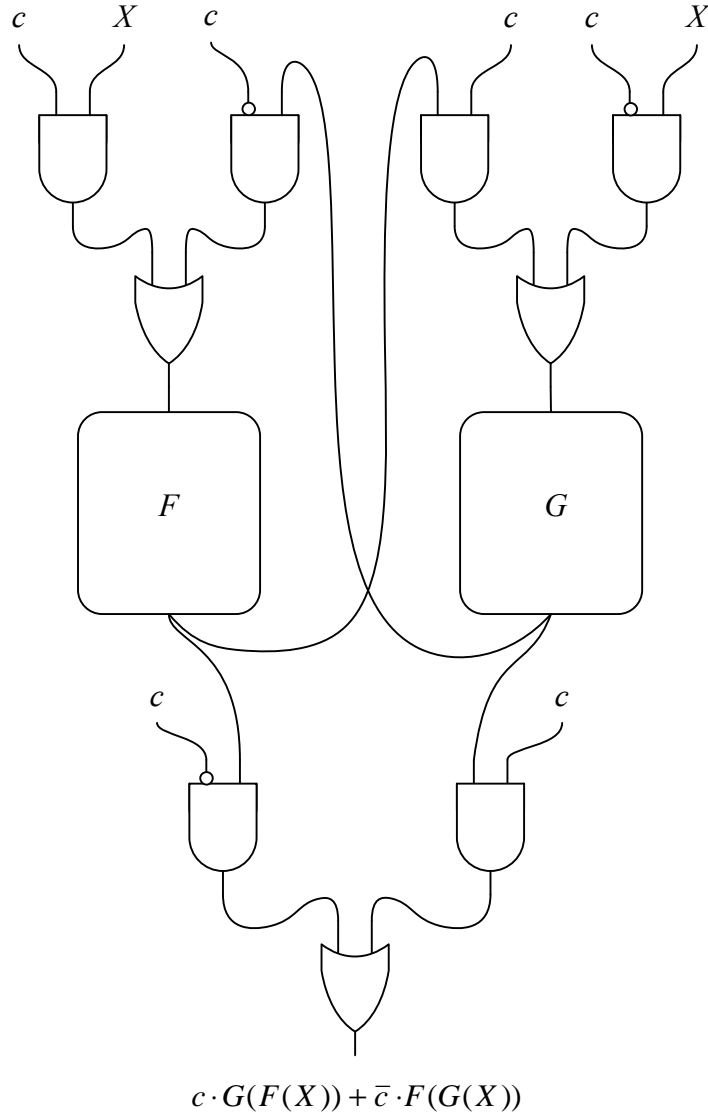


Figure 1.13: Functional units connected in a cyclic topology.

Although clearly promising, the idea of cyclic designs at the level of functional units has not been pursued, due to a lack of support in integrated circuit design packages. Indeed, nearly all logic synthesis and verification tools balk when given designs with cycles. Methods were proposed for analyzing such designs [12], [24], [42]. Nevertheless, the accepted strategy is simply to disallow cycles among functional units in the high-level phases.

## 1.3 Overview

In the realm of digital circuits, researchers seem to fall into two camps. On the one hand, there are the theoreticians, working in the field of circuit complexity. They are preoccupied with *classifying* and *characterizing* problems in general terms. They discuss the relationships among complexity classes, and prove bounds on the size of circuits. On the other hand, there are the practitioners, working in the field of electronic design automation. They strive to obtain the best circuits that they can, given the computational resources at their disposal. However, they rarely speak of *optimal* designs. The true optimum according to any criteria – be it area, delay, power – is generally unknowable to them.

### 1.3.1 Theory

In the first half of this dissertation, we wear the theoretician’s mantle. In Chapter 2 we describe our circuit model, and present a framework for analysis. In Chapter 3, we present theoretical justification for the claim that the optimal form of some circuits requires cyclic topologies. We exhibit families of cyclic circuits that are optimal in the number of gates, and we prove lower bounds on the size of equivalent acyclic circuits.

For instance, the cyclic circuit in Figure 1.14 consists of three “complex” gates, each with fan-in 6. We show that this circuit implements three distinct functions,  $f_1$ ,  $f_2$  and  $f_3$ , each depending on all 12 variables  $a, \dots, l$ . We then argue that an acyclic circuit implementing the same functions requires at least five fan-in 6 gates.

Our lower bound is based on a simple fan-in argument: in order to compute a function that depends on a certain number of variables using gates with a certain fan-in, we require a tree of at least a certain size. This is perhaps the *weakest* lower bound than one can conceive of on a circuit’s size. This suggests that feedback may be *more* powerful than we can show.

Our most notable construction is a family of cyclic circuits that have asymptotically at most one-half as many gates as equivalent acyclic circuits. We show that this is largest gap that we can prove using the fan-in lower bound technique.

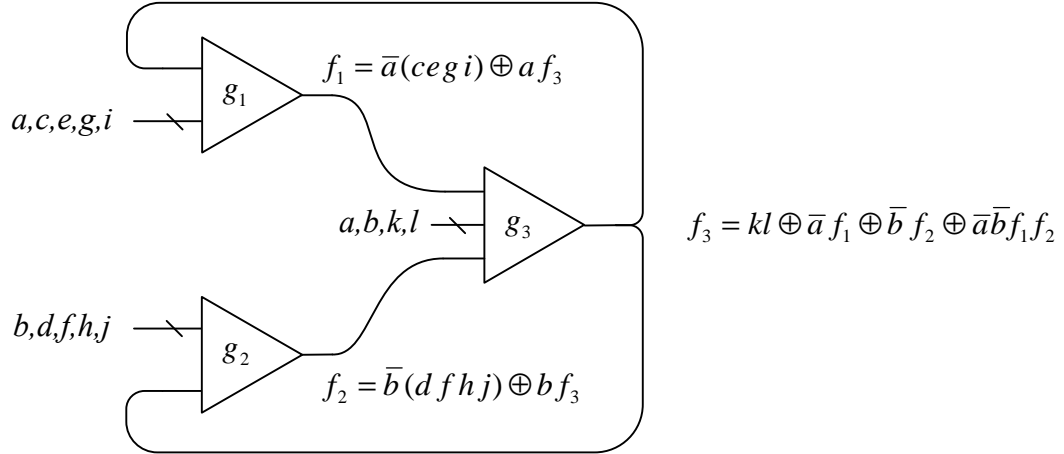


Figure 1.14: Cyclic circuit with inputs  $a, \dots, l$  and outputs  $f_1, f_2, f_3$ . ( $\oplus$  represents XOR.)

### 1.3.2 Practice

In the second half of the dissertation, we wear the practitioner's mantle. In Chapter 5 we describe a general methodology for synthesizing cyclic combinational circuits, and compare our results to those produced by state-of-the art logic synthesis tools.

Consider the example shown in Figure 1.15, ubiquitous in introductory logic design courses: a 7-segment display decoder. The inputs are four bits,  $x_0, x_1, x_2, x_3$ , specifying a number from 0 to 9. The outputs are 7 bits,  $a, b, c, d, e, f, g$ , specifying which segments to light up in a display – such as that of a digital alarm clock – to form the image of this number.

With our synthesis methodology, we arrive at the network shown in Figure 1.16, with the ordering illustrated. This network translates into a cyclic circuit with 27 fan-in two gates. In contrast, standard synthesis techniques produce an acyclic circuit with 32 fan-in two gates.

Note that the network in Figure 1.16 contains cyclic dependencies; in fact, all the functions except  $d$  form a strongly connected component. How can we establish that this network computes what it is supposed to, namely the output functions for the 7-segment decoder? We refer to this task as *functional analysis*. Given an upper bound on the time that it takes each gate to compute a value – the *gate delay* – how can we establish an upper bound on the time that it takes for the circuit to compute the

inputs				Digit	outputs						
$x_3$	$x_2$	$x_1$	$x_0$		a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	0	1	1	1
0	0	0	1	1	0	0	0	0	0	1	1
0	0	1	0	2	0	1	1	1	1	1	0
0	0	1	1	3	0	0	1	1	1	1	1
0	1	0	0	4	1	0	0	1	0	1	1
0	1	0	1	5	1	0	1	1	1	0	1
0	1	1	0	6	1	1	0	1	1	0	1
0	1	1	1	7	0	0	1	0	0	1	1
1	0	0	0	8	1	1	1	1	1	1	1
1	0	0	1	9	1	0	1	1	0	1	1

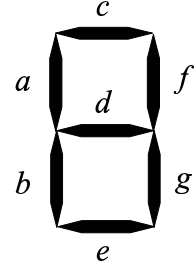


Figure 1.15: 7-Segment Display Decoder.

values of the functions – the *circuit delay*? We refer to this task as *timing analysis*.

Khrapchenko was the first to recognize that depth and delay in a circuit are not equivalent concepts [18]. There may exist *false paths*, that is to say, topological paths that are never sensitized. So-called “exact” algorithms for timing analysis consider the presence of false paths; these provide the requisite tool for the analysis of cyclic circuits. For a cyclic circuit, we can say that it is combinational if all cycles are false; the sensitized paths in the circuit never bite their own tail to form true cycles.

Our synthesis program can routinely tackle designs with, say 50 inputs and 30 outputs. For circuits of this size, a exhaustive approach to analysis – that is to say, checking every input assignment – is not feasible: with  $n$  variables there would be  $2^n$  input combinations. In Chapter 4 we describe efficient algorithms for analysis based on *symbolic* techniques, using flexible data structures called *binary decision diagrams*. Our analysis considers topological aspects of the design, for instance sub-dividing the problem into strongly connected components.

Our synthesis strategy is to introduce feedback in the re-structuring and minimization phases. A branch-and-bound search is performed, with analysis used to validated and rank potential solutions. Although general, our methodology is of immediate practical interest. For instance, we optimized the area of the ALU Decoder of a 8051 microprocessor design by 20%. In trials with benchmark circuits, nearly all

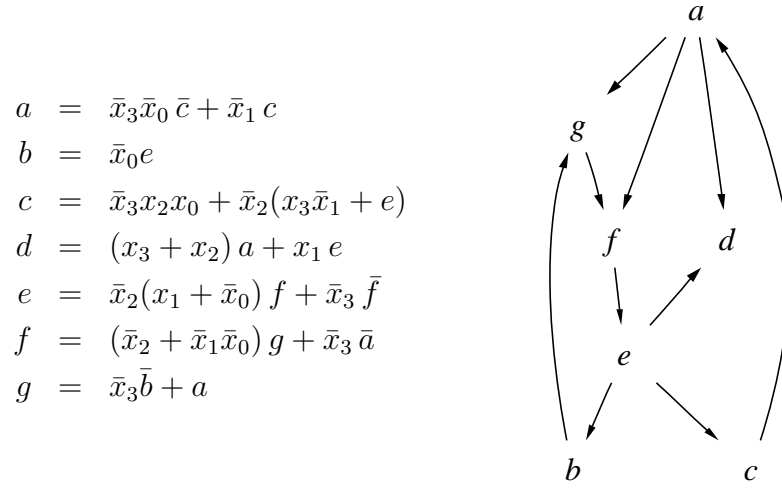


Figure 1.16: A cyclic network for the example in Figure 1.15.

were optimized significantly, with improvements of up to 30% in the area and up to 25% in the delay.

Theoreticians may dismiss optimizations of this sort as inconsequential:

*“Saving a few gates in the design of a 7-segment decoder doesn’t prove anything”.*

Practitioners may dismiss the theoretical results as contrived:

*“Asymptotic bounds don’t help me one bit in designing real circuits.”.*

However, taken together our results should convince both camps. It is time to re-write the definition: in both theory and practice, combinational might well mean cyclic.

# Chapter 2

## Framework

*Make everything as simple as possible without making anything too simple.*

– Albert Einstein (1879–1955)

The concepts discussed in this dissertation are not tied to any particular physical model or computing substrate. For the core ideas in Chapter 4 and Chapter 5, the exposition is at a *symbolic* level, that is to say, in terms of Boolean expressions. However, we first postulate an underlying *structural* model, consisting of *gates* and *wires*, and discuss analysis in an explicit sense – in terms of signal values.

### 2.1 Circuit Model

We work with digital abstraction of 0’s and 1’s. Nevertheless, our model recognizes that the underlying signals are, in fact, analog: each signal is a continuous real-valued function of time  $s(t)$ , corresponding to a voltage level. For analysis, we adopt a *ternary* framework, extending the set of *Boolean* values  $\mathbb{B} = \{0, 1\}$  to the set of ternary values  $\mathbb{T} = \{0, 1, \perp\}$ . The logical value of an analog signal is obtained by the mapping

$$\text{logical}[s(t)] = \begin{cases} 0 & \text{if } s(t) < V_{\text{low}} \\ 1 & \text{if } s(t) > V_{\text{high}} \\ \perp & \text{otherwise,} \end{cases}$$

where  $V_{\text{low}}$  and  $V_{\text{high}}$  are real values demarcating the range corresponding to Boolean 0 and Boolean 1, respectively. Clearly,  $V_{\text{high}}$  must be strictly greater than  $V_{\text{low}}$ . The third value,  $\perp$ , indicates that the signal is ambiguous. For the purposes of analysis,  $\perp$  is used in a broader sense: it denotes a signal value that is *unknown*. This signal may be Boolean 0, Boolean 1, or some ambiguous value – we simply do not know.

The idea of three-valued logic for circuit analysis is well established. It was originally proposed for the analysis of *hazards* in combinational logic [13], [50]. Bryant popularized its use for verification [8], and it has been widely adopted for the analysis of asynchronous circuits [9]. For a theoretical treatment, see [29].

A circuit consists of **gates** connected by **wires**. Each gate has one or more inputs and a single output. The symbols for common gates are shown in Figure 2.1. A bubble is used to indicate that an input or output is negated, as illustrated in Figure 2.2.

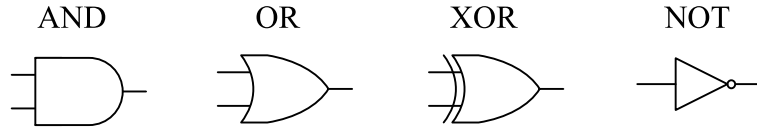


Figure 2.1: Symbols for different types of gates.

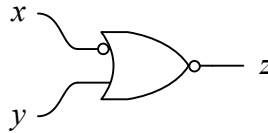


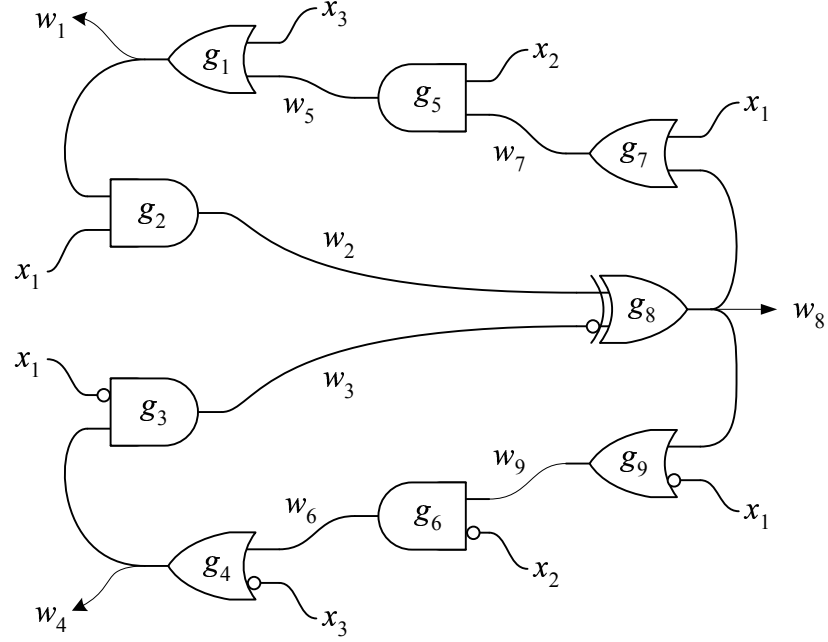
Figure 2.2: Bubbles on the inputs or the output of a gate indicate negation. Here  $z = \text{NOT}(\text{OR}(\text{NOT}(x), y))$ .

An example of a circuit is shown in Figure 2.3. Even though a wire may split in our diagrams, as is the case with wire  $w_8$  in Figure 2.3, conceptually there is a single instance of it.

- The circuit accepts signals  $x_1, \dots, x_m$ , ranging over  $\{0, 1\}$ , called the **primary inputs**. Each primary input is fed into one or more gate inputs. Even though the symbol for a primary input may appear in several places, as is the case with  $x_1$ ,  $x_2$  and  $x_3$  in Figure 2.3, conceptually there is a single instance of it.



- The gates in the circuit produce **internal signals**,  $w_1, \dots, w_n$  ranging over  $\{0, 1, \perp\}$ .
- A subset of the set of internal signals is designated as the set of **primary outputs**.



input signals:  $x_1, x_2, x_3$   
 internal signals:  $w_1, \dots, w_9$   
 output signals:  $w_1, w_4, w_8$   
 gates:  $g_1, \dots, g_9$

Figure 2.3: An example of a circuit, consisting of gates and wires.

### 2.1.1 Functional Behavior

In the digital realm, a gate implements a Boolean function, i.e., a mapping from Boolean inputs to a Boolean output value,

$$g : \{0, 1\}^k \rightarrow \{0, 1\}.$$

The set of inputs to a gate are called its **fan-in** set. When we say a “fan-in  $k$ ” gate, we mean a gate with fan-in set of cardinality  $k$ . The set of gates that are attached to a gate output are called its **fan-out** set. The truth tables for fan-in two AND, OR and XOR gates, as well as a fan-in one NOT gate, are shown Figure 2.4.

$x$	$y$	AND( $x, y$ )	OR( $x, y$ )	XOR( $x, y$ )
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

$x$	NOT( $x$ )
0	1
1	0

Figure 2.4: Truth table for common gates.

### 2.1.2 Temporal Behavior

We characterize the temporal behavior of a gate by a single parameter, a bound on its delay  $t_d$ .

For a gate characterized by a mapping  $g$ , if the inputs assume the values  $y_1(t), \dots, y_k(t)$  at time  $t$ , and subsequently do not change, then the output assumes the value  $g[y_1(t), \dots, y_k(t)]$  at time no later than  $t + t_d$ , and does not change.

Further, we assume that the wires have zero propagation delay. More realistic models for timing analysis can readily be incorporated within our framework; we neglect such details here in order to focus on the conceptual aspects.

## 2.2 Analysis Framework

Our analysis characterizes the functional behavior of circuits according to the so-called “floating-mode” assumption [9], [49]: at the outset of each interval, all wires in a circuit are assumed to have unknown or possibly undefined values ( $\perp$ ). We apply definite values to the inputs, and track the propagation of signal values.

Analysis of an *acyclic* circuit is transparent. We first evaluate the gates connected only to primary inputs, and then gates connected to these and primary inputs, and so on, until we have evaluated all gates. For instance, in the circuit of Figure 1.2 in the Introduction, we first evaluate  $g_1$  and  $g_2$ , then  $g_3$ , then  $g_4$  and  $g_5$ . At each step, we only evaluate a gate when all of its input signals are known. The previous values of the internal signals do not enter into play.

In a cyclic circuit, there are one or more *strongly connected components*. Recall that in a directed graph  $G$ , a strongly connected component is an induced subgraph  $S \subseteq G$  such that

- there exists a directed path between every pair of nodes in  $S$ ;
- for every node  $s$  in  $S$  and every node  $n$  outside of  $S$ , if there exists a path from  $s$  to  $n$  (from  $n$  to  $s$ ) then there is no path from  $n$  to  $s$  (from  $s$  to  $n$ , respectively).

We analyze each strongly connected component separately.

At the outset, with only the primary inputs fixed at definite values, each gate in a strongly connected component has some unknown/undefined inputs (valued  $\perp$ ). Nevertheless, for each such gate we can ask: is there sufficient information to conclude that the gate output is 0 or 1, in spite of the  $\perp$  values? If yes, we assign this value as the output; otherwise, the value  $\perp$  persists. For instance, with an AND gate, if the inputs include a 0, then the output is 0, regardless of other  $\perp$  inputs. If the inputs consist of 1 and  $\perp$  values, then the output is  $\perp$ . Only if all the inputs are 1 is the output 1. This is illustrated in Figure 2.5. Input values that determine the gate output are called *controlling*.

### 2.2.1 Ternary Extension

For the set  $\{0, 1, \perp\}$ , we define a **partial ordering**

$$\perp \sqsubseteq 0 \quad \text{and} \quad \perp \sqsubseteq 1,$$

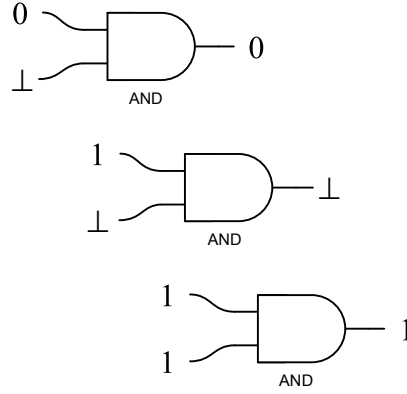


Figure 2.5: An AND gate with 0, 1, and  $\perp$  inputs.

with 0 and 1 not comparable. For vectors  $Y = (y_1, \dots, y_n)$  and  $Z = (z_1, \dots, z_n)$ , we define the ordering coordinate-wise:

$$Y \sqsubseteq Z \quad \text{if} \quad y_i \sqsubseteq z_i \quad \text{for all } i = 1, \dots, n.$$

For instance, if  $Y = (\perp, 1, \perp, 0)$ , and  $Z = (1, 1, 1, 0)$  then  $Y \sqsubseteq Z$ . However, if  $Y = (\perp, 1, \perp, 0)$  and  $Z = (1, 1, 1, \perp)$  then  $Y$  and  $Z$  are not comparable.

We define the **partial join**  $V = (v_1, \dots, v_n) = Y \sqcup Z$  as:

$$v_i = \begin{cases} a & \text{if } y_i = z_i = a \text{ for some } a \in \{0, 1\}, \\ b & \text{if } \{y_i, z_i\} = \{b, \perp\} \text{ for some } b \in \{0, 1\}, \\ \perp & \text{else.} \end{cases}$$

for all  $i = 1, \dots, n$ . For instance, if  $Y = (\perp, 1, \perp, 0)$ , and  $Z = (1, 1, 1, \perp)$  then  $Y \sqcup Z = (1, 1, 1, 0)$ .

Within the ternary framework, a gate performs a mapping from ternary values to ternary values,

$$g' : \{0, 1, \perp\}^k \rightarrow \{0, 1, \perp\}.$$

We call this mapping the *ternary extension* of  $g$ . Given a Boolean mapping  $g$ , the **ternary extension**  $g'$  is defined as follows. For a vector of ternary values  $Y \in$

$\{0, 1, \perp\}^k,$ 

$$g'(Y) = \begin{cases} 0 & \text{if } g(Z) = 0 \text{ for each } Z \in \{0, 1\}^k, \text{ where } Y \sqsubseteq Z, \\ 1 & \text{if } g(Z) = 1 \text{ for each } Z \in \{0, 1\}^k, \text{ where } Y \sqsubseteq Z, \\ \perp & \text{else.} \end{cases}$$

A similar definition of the ternary extension is found in [9]. The truth-tables for the ternary extensions of fan-in two AND, OR and XOR gates, as well as a fan-in one NOT gate, are shown in Figure 2.6.

$x$	$y$	AND( $x, y$ )	OR( $x, y$ )	XOR( $x, y$ )
0	0	0	0	0
0	1	0	1	1
0	$\perp$	0	$\perp$	$\perp$
1	0	0	1	1
1	1	1	1	0
1	$\perp$	$\perp$	1	$\perp$
$\perp$	0	0	$\perp$	$\perp$
$\perp$	1	$\perp$	1	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

$x$	NOT( $x$ )
0	1
1	0
$\perp$	$\perp$

Figure 2.6: Ternary extensions for common gates.

### 2.2.2 Fixed Point

The goal of functional analysis is to determine what output values a circuit produces in response to Boolean input values. Of course, if the circuit is cyclic, we cannot be sure that it settles to a stable state. Consider the inverter ring shown in Figure 2.7. With  $x = 1$ , the ring will probably oscillate, with the output  $z$  alternating between 0 and 1, as shown in Figure 2.8. Within the ternary framework, all instability is hidden beneath the  $\perp$  values. This is illustrated with the inverter chain in Figure 2.9.

The following theorem shows that once a definite value is assigned to an internal wire, this value persists for the duration of the interval (so long as the input values are held constant). Furthermore, the order of gate evaluations is irrelevant; the final

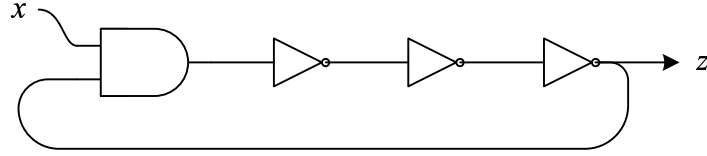


Figure 2.7: An inverter ring.

outcome – which internal wires are assigned definite values, and what these values are – is the same regardless. The analysis terminates at a **fixed point**: in this state, every gate evaluation agrees with the value on its output wire, so there are no further changes. Of course, the term “fixed point” is somewhat paradoxical: with  $\perp$  values, the state includes signals that are potentially unstable.

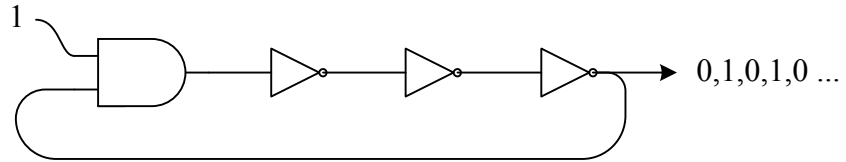


Figure 2.8: In the Boolean framework, the inverter ring oscillates.

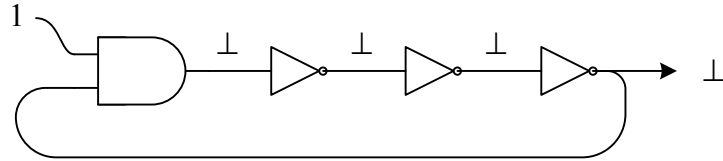


Figure 2.9: In the ternary framework, the values are unknown/undefined.

**Theorem 2.1** *With all the internal signals assigned an initial value  $\perp$ , for a given set of Boolean values applied to the inputs and held constant, the analysis terminates at a unique fixed point.*

**Proof:** Call the values assumed by the internal variables  $W = (w_1, \dots, w_n)$  the *state*. Beginning from the initial state  $W_0 = (\perp, \dots, \perp)$ , the circuit evolves through a sequence,

$$W_0, W_1, W_2, \dots$$

Since each gate update consists of a change  $\perp \rightarrow \{0, 1\}$ , the sequence of states is

ordered,

$$W_0 \subseteq W_1 \subseteq W_2 \subseteq \dots$$

Since the number of states is finite, clearly the computation terminates at some fixed point. This is illustrated in Figure 2.10.

It remains to show that this fixed point is unique. To do so, we argue that the order of updates is irrelevant. Indeed, from a given state  $W$ , if we have a choice of immediate successor states  $W_i$  and  $W_j$ , then the partial join  $W_k = W_i \sqcup W_j$  exists and is an immediate successor state to both  $W_i$  and  $W_j$ . This is illustrated in Figure 2.11. With the initial state  $(\perp, \dots, \perp)$  as the base case, a simple inductive argument suffices to show that all states have a common successor. This common successor must be a fixed point.  $\square$

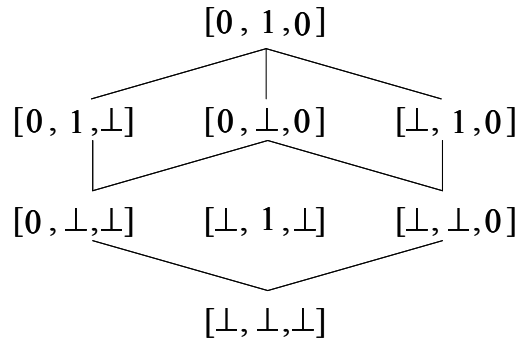


Figure 2.10: The computation terminates at a fixed point.

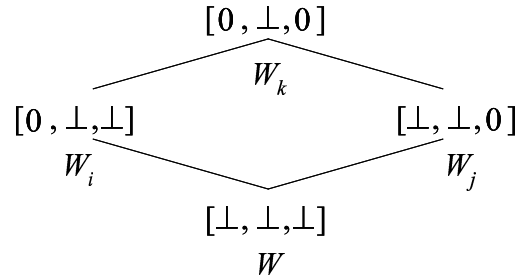


Figure 2.11: The order of updates is irrelevant.

### 2.2.3 Explicit Analysis

The analysis strategy for specific Boolean input values might be termed *simulation*: we apply inputs and follow the evolution of the circuit. The goal of **functional analysis** is to determine *what* values appear; the goal of **timing analysis** is to determine *when* these values appear.

For functional analysis, Theorem 2.1 tells us that the gates may be evaluated in any order. We simply apply the inputs and follow the signals as they propagate; gates are evaluated when new signals arrive. Once a gate evaluates to a definite Boolean value, it is not evaluated again. Once the analysis terminates, if there are  $\perp$  values on the outputs, we conclude that the circuit does not behave combinationaly.

For timing analysis, we establish an upper bound on the *arrival times* of definite Boolean values for internal signals. We always evaluate gates in the order that signals arrive, ensuring that we know the earliest time that a signal value becomes known. When evaluating a gate, we use only present and past input values, not future values.

We illustrate analysis with a collection of examples: first two (acyclic) circuit fragments; then a non-combinational cyclic circuit; and finally a combinational cyclic circuit. We assume that all gates have unit delay, and that the primary inputs arrive at time 0.

#### Example 2.1

Consider the circuit fragment shown in Figure 2.12. It consists of four gates: an AND gate  $g_1$ , an OR gate  $g_2$ , an AND gate  $g_3$ , and an OR gate  $g_4$ :

$$\begin{aligned} g_1(x_1, y_1) &= x_1 y_1, \\ g_2(x_2, y_2) &= x_2 + y_2, \\ g_3(x_3, y_3) &= x_3 y_3, \\ g_4(x_1, y_4) &= x_1 + y_4. \end{aligned}$$

This circuit illustrates the concept of *false paths*. From a topological perspective,



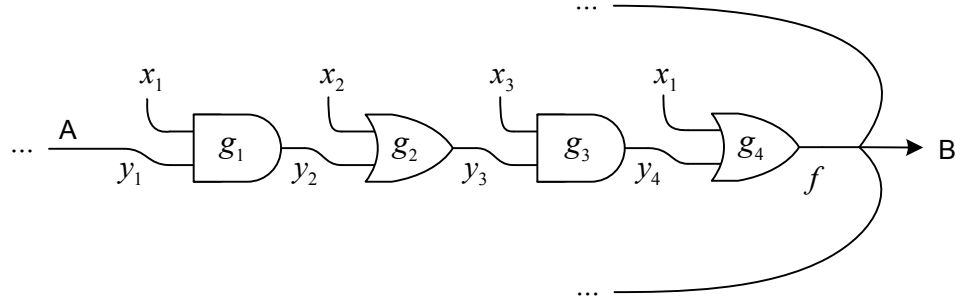


Figure 2.12: A circuit fragment.

there exists a path from point  $A$  to point  $B$  in this circuit. However, from a functional standpoint, this path is never *sensitized*. To see this, consider specific input values:

- With  $x_1 = 0$ , the path is blocked at gate  $g_1$ .
- With  $x_2 = 1$ , the path is blocked at gate  $g_2$ .
- With  $x_3 = 0$ , the path is blocked at gate  $g_3$ .
- With  $x_1 = 1$ , the path is blocked at gate  $g_4$ .

For any combination of input values, we know what value appears at point  $B$ , and how long it takes for this value to appear, regardless of the signal at point  $A$ . Assuming that the gates  $g_1$ ,  $g_2$ ,  $g_3$  and  $g_4$  have uniform delay bounds of  $t_1$ ,  $t_2$ ,  $t_3$ , and  $t_4$ , respectively, we can assert:

- With  $x_1 = 1$ , a value of 1 appears after  $t_4$  time units.
- With  $x_1 = 0$ , and  $x_3 = 0$ , a value of 0 appears after  $t_3 + t_4$  time units.
- With  $x_1 = 0$ ,  $x_3 = 1$ , and  $x_2 = 1$ , a value of 1 appears after  $t_2 + t_3 + t_4$  time units.
- With  $x_1 = 0$ ,  $x_3 = 1$ , and  $x_2 = 0$ , a value of 0 appears after  $t_1 + t_2 + t_3 + t_4$  time units.

Further assuming a unit delay model (i.e.,  $t_1 = t_2 = t_3 = t_4 = 1$ ), we obtain the analysis results in Table 2.1. Subscripts on the values indicate the arrival times.

$x_1$	$x_2$	$x_3$	$f$
0	0	0	$0_2$
0	0	1	$0_4$
0	1	0	$0_2$
0	1	1	$1_3$
1	0	0	$1_1$
1	0	1	$1_1$
1	1	0	$1_1$
1	1	1	$1_1$

Table 2.1: Analysis of the circuit fragment in Figure 2.12.

**Example 2.2**

Consider the circuit shown in Figure 2.13, consisting of an AND gate  $g_1$ , an OR gate  $g_2$ , and an AND gate  $g_3$ , in a cycle. By inspection, note that if  $x_1 = 0$  then  $f_1$  assumes value 0 after one time unit; if  $x_2 = 1$  then  $f_2$  assumes value 1 after one time unit; and if  $x_3 = 0$  then  $f_3$  assumes value 0 after one time unit. But what happens if  $x_1 = 1$ ,  $x_2 = 0$  and  $x_3 = 1$ ? In this case, all the outputs equal  $\perp$ , as illustrated in Figure 2.14. The outcome for all eight cases is shown in Table 2.2.

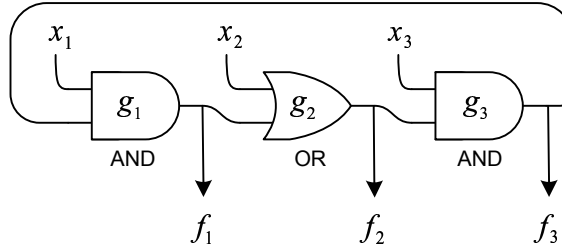
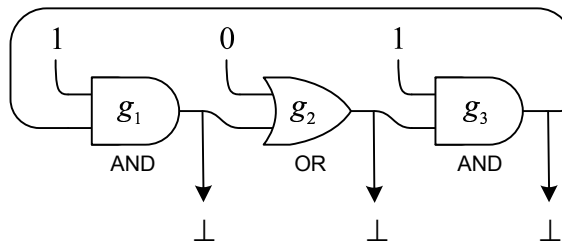


Figure 2.13: A non-combinational cyclic circuit.

Figure 2.14: The circuit of Figure 2.13 with  $x_1 = 1$ ,  $x_2 = 0$  and  $x_3 = 1$ .

$x_1$	$x_2$	$x_3$	$f_1$	$f_2$	$f_3$
0	0	0	0 <sub>1</sub>	0 <sub>2</sub>	0 <sub>1</sub>
0	0	1	0 <sub>1</sub>	0 <sub>2</sub>	0 <sub>3</sub>
0	1	0	0 <sub>1</sub>	1 <sub>1</sub>	0 <sub>1</sub>
0	1	1	0 <sub>1</sub>	1 <sub>1</sub>	1 <sub>2</sub>
1	0	0	0 <sub>2</sub>	0 <sub>3</sub>	0 <sub>1</sub>
1	0	1	$\perp$	$\perp$	$\perp$
1	1	0	0 <sub>2</sub>	1 <sub>1</sub>	0 <sub>1</sub>
1	1	1	1 <sub>3</sub>	1 <sub>1</sub>	1 <sub>2</sub>

Table 2.2: Analysis of circuit in Figure 2.13.

In general, we would reject this circuit, since its outputs are not defined for the input assignment  $x_1 = 1$ ,  $x_2 = 0$  and  $x_3 = 1$ . However, if this particular assignment is in the “don’t care” set, then the design would be valid.

### Example 2.3

Consider the circuit in Figure 2.15 (a).

- Of the three gates, we see that initially only  $g_1$  evaluates to a definite value,

$$w_1 = g_1(x_1, x_2) = \text{OR}(1, 0) = 1.$$

We set the arrival time of  $w_1$  to be

$$t_1 = 1.$$

- With  $w_1$  defined, we see that  $g_2$  evaluates to a definite value,

$$w_2 = g_2(w_1, x_2) = \text{AND}(\text{NOT}(1), 1) = 0.$$

We set arrival time of  $w_2$  to be

$$t_2 = 1 + t_1 = 2.$$

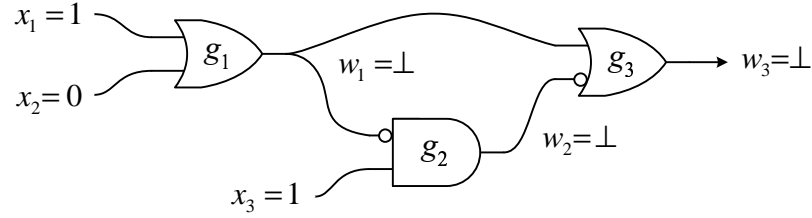
- At this point in the execution of the algorithm,  $w_1$  and  $w_2$  have been assigned definite values. However,  $w_1$  has an earlier arrival time. Evaluating  $g_3$  at time 1,

$$w_3 = g_3(w_1, w_2) = \text{OR}(1, \text{NOT}(\perp)) = 1.$$

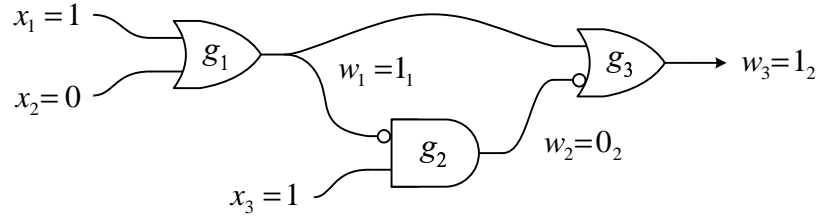
We set the arrival time of  $w_3$  to be

$$t_3 = 1 + t_1 = 2.$$

The final values of  $w_1, w_2, w_3$  are shown in Figure 2.15 (b). Subscripts indicate the arrival times.



(a) Initial state.



(b) Final state.

Figure 2.15: Example 2.3 Subscripts on the values of the internal variables indicate the arrival times.

The salient point of this example is that the algorithm tracks the arrival times of signals, and establishes the earliest possible set of controlling signals. If gate  $g_3$  had been evaluated after both  $w_1$  and  $w_2$  had been determined, we might have concluded that its arrival time was 3 time units instead of 2.

### Example 2.4

Consider the circuit in Figure 2.16. Suppose that we apply inputs  $x_1 = 1, x_2 = 0, x_3 = 1$ , as shown in Part (a). Gates  $g_1, g_3, g_5$  and  $g_7$  produce outputs of 1, 0, 0, and 1, respectively, after one time unit. Gate  $g_2$  produces an output of 1 after two time units. Gate  $g_8$  produces an output of 0 after three time units. Gate  $g_9$  produces

an output of 0 after four time units. Gate  $g_6$  produces an output of 0 after five time units. Finally, gate  $g_4$  produces an output of 0 after six time units. The circuit after six time units is shown in Figure 2.16 (b).

The analysis for all eight input combinations is summarized in Table 2.3. We see that the maximum delay of the circuit is six time units.

$x_1$	$x_2$	$x_3$	$g_1$	$g_2$	$g_3$	$g_4$	$g_5$	$g_6$	$g_7$	$g_8$	$g_9$
0	0	0	0 <sub>2</sub>	0 <sub>1</sub>	1 <sub>2</sub>	1 <sub>1</sub>	0 <sub>1</sub>	1 <sub>2</sub>	0 <sub>4</sub>	0 <sub>3</sub>	1 <sub>1</sub>
0	0	1	1 <sub>1</sub>	0 <sub>1</sub>	1 <sub>4</sub>	1 <sub>3</sub>	0 <sub>1</sub>	1 <sub>2</sub>	0 <sub>6</sub>	0 <sub>5</sub>	1 <sub>1</sub>
0	1	0	0 <sub>6</sub>	0 <sub>1</sub>	1 <sub>2</sub>	1 <sub>1</sub>	0 <sub>5</sub>	0 <sub>1</sub>	0 <sub>4</sub>	0 <sub>3</sub>	1 <sub>1</sub>
0	1	1	1 <sub>1</sub>	0 <sub>1</sub>	0 <sub>3</sub>	0 <sub>2</sub>	1 <sub>6</sub>	0 <sub>1</sub>	1 <sub>5</sub>	1 <sub>4</sub>	1 <sub>1</sub>
1	0	0	0 <sub>2</sub>	0 <sub>3</sub>	0 <sub>1</sub>	1 <sub>1</sub>	0 <sub>1</sub>	1 <sub>6</sub>	1 <sub>1</sub>	1 <sub>4</sub>	1 <sub>5</sub>
1	0	1	1 <sub>1</sub>	1 <sub>2</sub>	0 <sub>1</sub>	0 <sub>6</sub>	0 <sub>1</sub>	0 <sub>5</sub>	1 <sub>1</sub>	0 <sub>3</sub>	0 <sub>4</sub>
1	1	0	1 <sub>3</sub>	1 <sub>4</sub>	0 <sub>1</sub>	1 <sub>1</sub>	1 <sub>2</sub>	0 <sub>1</sub>	1 <sub>1</sub>	0 <sub>5</sub>	0 <sub>6</sub>
1	1	1	1 <sub>1</sub>	1 <sub>2</sub>	0 <sub>1</sub>	0 <sub>2</sub>	1 <sub>2</sub>	0 <sub>1</sub>	1 <sub>1</sub>	0 <sub>3</sub>	0 <sub>4</sub>

Table 2.3: Analysis summary for the circuit of Figure 2.3. Subscripts on the output values indicate arrival times.

## 2.2.4 Complexity

In the analysis, we evaluate a gate whenever a new Boolean signal arrives on one of its inputs. In the worst case, we could evaluate a gate with fan-in  $d$  as many as  $d$  times. Given a circuit with  $m$  primary inputs and  $n$  gates, each with fan-in  $d$ , there are  $O(dn)$  gate evaluations. In addition, for timing analysis, we must maintain a sorted list of arrival times. This contributes a complexity factor of  $O(n \log_2 n)$ .

We could perform the analysis explicitly for every assignment of input values. However, such an exhaustive approach is simply not tractable for most real circuits: with  $n$  variables there would be  $2^n$  input combinations to analyze separately. In Chapter 4 we describe an efficient analysis algorithm based on *symbolic* techniques.

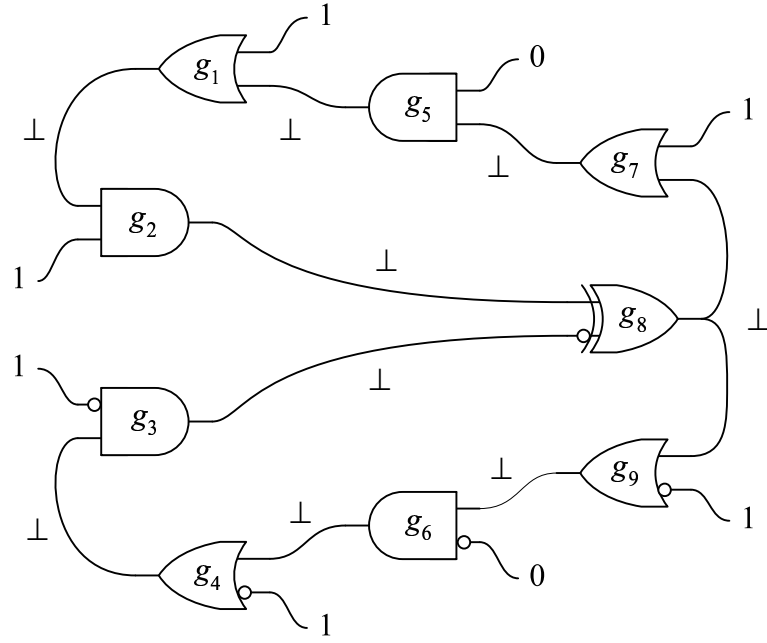
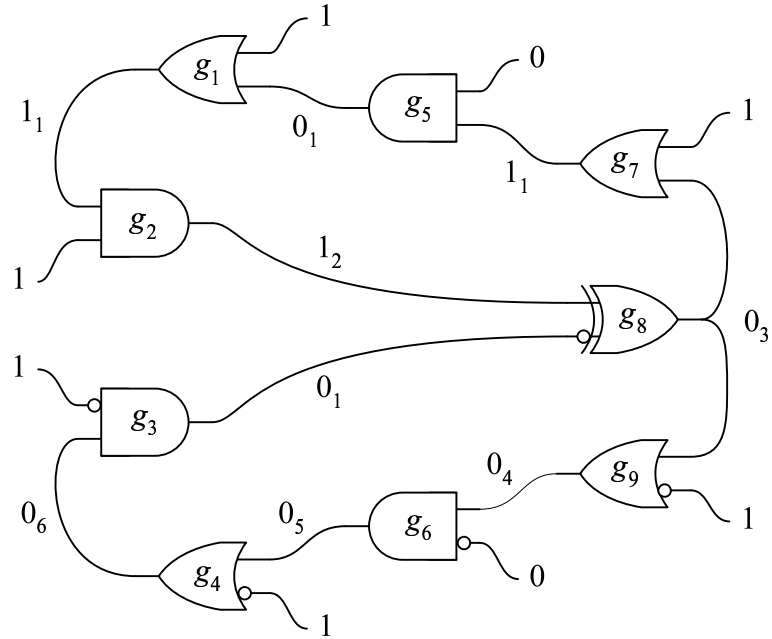
(a) time  $t = 0$ (b) time  $t = 6$ 

Figure 2.16: The circuit of Figure 2.3 with inputs  $x_1 = 1, x_2 = 0, x_3 = 1$ . Subscripts on the output values indicate arrival times.

# Chapter 3

## Theory

*In theory there is no difference between theory and practice, but in practice there is.* – Yogi Berra (1925– )

Theoreticians are preoccupied with *classifying* and *characterizing* problems in general terms. They discuss the relationships among complexity classes, and prove bounds on the size of circuits. However, somewhat to their embarrassment, they offer very little help in proving or disproving the optimality of specific circuits. There have been a handful of papers, dating back to the 1960's, describing approaches for finding optimal multi-level circuit designs [10], [11], [20], but these have limited applicability: the largest circuits that these methods can hope to tackle have 4 (or perhaps 5) input variables.

Lower bounds on circuit size are notoriously difficult to establish. In fact, such proofs are related to fundamental questions in computer science, such as the separation of the  $P$  and  $NP$  complexity classes. (To prove that  $P \neq NP$  it would suffice to find a class of problems in  $NP$  that cannot be computed by a polynomially sized circuit.) Much of the recent work in circuit complexity has been spurred by these open problems [1].

All existing lower bounds on circuit size are linear in the number of variables [1]. In 1949, Shannon showed by a straight forward counting argument that nearly *all* functions require circuits with an exponential number of gates [40]. Yet there is no known *explicit* example [48].

Given these limitations, how can we hope to justify our general claim that feedback can be used to optimize circuits? In this section, we assume the theoretician’s mantle and *prove* that some cyclic designs are smaller than equivalent acyclic ones, based on the best lower-bound techniques that we know.

### 3.1 Criteria for Optimality

Any assertion of optimality rests on a restricted circuit model. Indeed, with gates of arbitrary size and complexity, any function can be implemented with a single “gate.” We restrict the scope of gates in two ways. The first way is to bound the *fan-in*, as shown in Figure 3.1 (a). Each gate can have at most  $d$  inputs, for some finite  $d$ . The second way is to restrict the type of gate. For instance, we can limit ourselves to so-called AON gates: AND gates with the inputs and output possibly negated. An example of such a gate is shown in Figure 3.1 (b). The general form of the Boolean function realized by an AON gate is

$$g(x_1, x_2, \dots, x_d) = (x_1 \oplus c_1) \cdot (x_2 \oplus c_2) \cdots (x_d \oplus c_d) \oplus c_{d+1},$$

where  $c_1, \dots, c_{d+1}$  are arbitrary choices of 0 and 1. (Multiplication represents AND, addition represents OR, and  $\oplus$  represents XOR.) The fan-in  $d$  may or may not be limited.

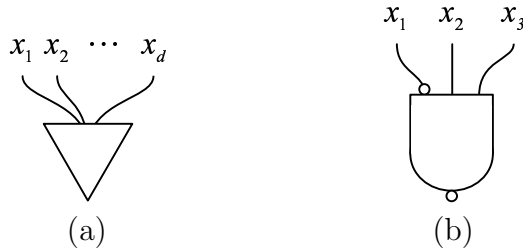


Figure 3.1: Restricting the scope of gates. (a) Bound the fan-in. (b) Use AND gates (with the inputs and output possibly negated).

In this chapter, we often represent functions in **XNF**, a canonical form consisting of XOR and AND operations. This form has advantages: since it is canonical,



we need not concern ourselves with simplifying the expressions. Furthermore, the dependence of a function on its variables is explicit. See Appendix A for a discussion of this representation.

Our general strategy in the following constructions is to present a cyclic circuit that is optimal in the number of gates, and then prove a lower bound on the size of any acyclic circuit implementing the same functions. The argument for the optimality of the cyclic circuit rests on two properties:

**Property 3.1** *Each of the output functions depends on all its variables.*

**Property 3.2** *The output functions are distinct.*

The cyclic circuit is shown to be optimal according to the following trivial claim (true regardless of the gate model):

**Claim 3.1** *A circuit implementing  $m$  distinct functions consists of at least  $m$  gates.*

## 3.2 Fan-in Lower Bound

Our lower bound on the size of an acyclic circuit is formulated as a fan-in argument. The essence of the argument was presented by Rivest [35], although we present it in a more general form.

A circuit can only compute a function of a given set of input variables if it “sees” all of them. For example, in Figure 3.2, gate  $g_2$  can compute a function of  $x_1, x_2$  and  $x_3$ ;  $g_1$  cannot compute a function of  $x_3$  since it does not see  $x_3$ . In an acyclic circuit, there is a partial ordering among the gates: if a gate  $g_i$  depends on a gate  $g_j$ , directly or indirectly, then  $g_j$  cannot depend on  $g_i$ , directly or indirectly. With a partial ordering on the output functions, there must be at least one output function at the top which depends upon no other. If this function depends on  $v$  input variables, the gate producing it must be the root of a tree that sees all these  $v$  variables as leaves. The lower bound is based on a calculation of the minimum number of gates in this tree.

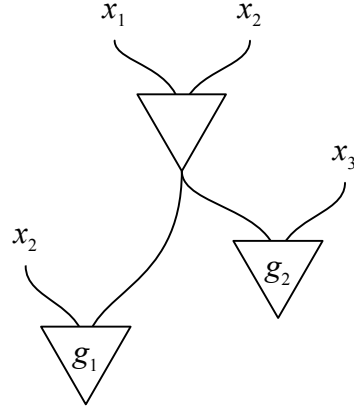


Figure 3.2: A gate can only compute functions of variables that it “sees”.

**Claim 3.2** *An acyclic circuit implementing  $m$  distinct output functions, each depending on  $v$  input variables, consisting of gates with fan-in at most  $d$  has at least*

$$\left\lceil \frac{v-1}{d-1} \right\rceil + m - 1$$

*gates.*

**Proof:** Consider a connected directed acyclic graph (DAG). Call nodes with no in-coming edges *leaves*, and all other nodes *internal* nodes. We show, by a simple inductive argument, that a connected DAG with  $k$  internal nodes, each with in-degree at most  $d$ , has at most  $k(d-1) + 1$  leaves. Obviously, a graph consisting of a single such internal node has at most  $d$  leaves. Suppose an internal node with in-degree at most  $d$  is added to a connected DAG. If the resulting graph is to be a *connected* DAG, the new node can replace an existing leaf or it can be attached to an existing internal node. The former case is illustrated with node  $g_1$  in Figure 3.3, and the latter with node  $g_2$ . In both cases there is a net gain of at most  $d-1$  leaves. We conclude that connected DAG with  $k$  internal nodes has at most

$$\begin{aligned} & d + (k-1)(d-1) \\ = & k(d-1) + 1 \end{aligned}$$

leaves, as expected. Suppose that a connected DAG has  $v$  leaves. Since

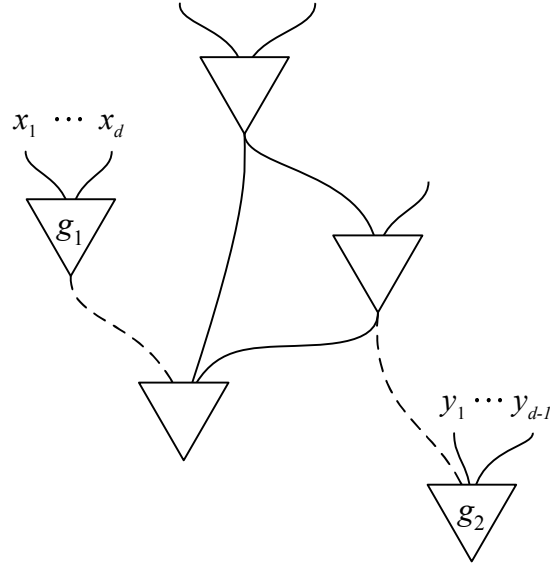


Figure 3.3: Adding a node with in-degree  $d$  to a connected DAG results in net gain of at most  $d - 1$  leaves.

$$v \leq k(d - 1) + 1,$$

the number of internal nodes  $k$  is bounded by

$$k \geq \left\lceil \frac{v - 1}{d - 1} \right\rceil.$$

Now, in an acyclic circuit implementing  $m$  output functions, at least one of the output functions depends on no other. By the argument above, this output function requires at least

$$\left\lceil \frac{v - 1}{d - 1} \right\rceil$$

gates. With *distinct* output functions, each output function must emanate from a different gate, so at least  $m - 1$  gates are required to implement the remaining  $m - 1$  functions.  $\square$

### 3.3 Improvement Factor

Suppose that we have a cyclic circuit with  $m$  gates, each with fan-in at most  $d$ , that implements  $m$  distinct functions, each of which depends on all  $v$  input variables. Call the *improvement factor* the ratio of size of the cyclic circuit to the lower bound on the size of the acyclic circuit:

$$\frac{\text{size of cyclic}}{\text{size of acyclic}} = \frac{m}{\left\lceil \frac{v-1}{d-1} \right\rceil + m - 1}.$$

With an improvement factor of  $C$ , we can say that our cyclic circuit is  $C$  times the size of any equivalent acyclic circuit.

**Claim 3.3** *The improvement factor is bounded below by  $\frac{1}{2}$ .*

**Proof:** For a given  $d$ , the improvement factor is minimized if the term

$$\frac{v-1}{d-1}$$

in the denominator is maximized. Now, the number of variables  $v$  in a cyclic circuit is at most  $m(d-1)$ , and this is achieved if all the gates have fan-in  $d$ . For such a circuit,

$$\frac{m}{\left\lceil m - \frac{1}{d-1} \right\rceil + m - 1} = \frac{m}{2m-1} \geq \frac{1}{2}.$$

□

### 3.4 Examples

Consider the example shown in Figure 3.4, due to Rivest [35]. We first verify that this circuit is combinational. For gate  $g_1$ , an AND gate,  $x_1 = 0$  is a controlling value.

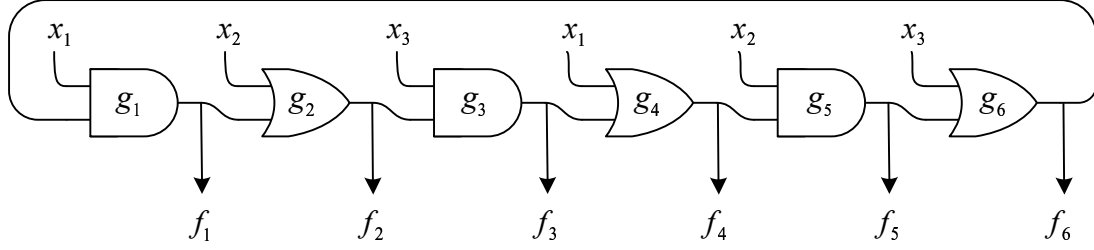


Figure 3.4: A cyclic combinational circuit with 3 inputs, due to Rivest [35].

Setting  $x_1 = 0$  we have

$$\begin{aligned}
 f_1 |_{\bar{x}_1} &= 0, \\
 f_2 |_{\bar{x}_1} &= f_1 + x_2 = x_2, \\
 f_3 |_{\bar{x}_1} &= f_2 x_3 = x_2 x_3, \\
 f_4 |_{\bar{x}_1} &= f_3 + 0 = x_2 x_3, \\
 f_5 |_{\bar{x}_1} &= f_4 x_2 = x_2 x_3, \\
 f_6 |_{\bar{x}_1} &= f_5 + x_3 = x_3.
 \end{aligned}$$

All outputs assume definite Boolean values. For gate  $g_4$ , an OR gate,  $x_1 = 1$  is a controlling value. Setting  $x_1 = 1$ , we have

$$\begin{aligned}
 f_4 |_{x_1} &= 1, \\
 f_5 |_{x_1} &= f_4 x_2 = x_2, \\
 f_6 |_{x_1} &= f_5 + x_3 = x_2 + x_3, \\
 f_1 |_{x_1} &= f_6 1 = x_2 + x_3, \\
 f_2 |_{x_1} &= f_1 + x_2 = x_2 + x_3, \\
 f_3 |_{x_1} &= f_2 x_3 = x_3.
 \end{aligned}$$

Again, all outputs assume definite Boolean values. Since  $x_1$  must either have value 0 or value 1, we conclude that the network is combinational. We assemble the output

functions from these two cases:

$$\begin{aligned}
f_1 &= \bar{x}_1 \cdot f_1|_{\bar{x}_1} + x_1 \cdot f_1|_{x_1} = \bar{x}_1 \cdot 0 + x_1 \cdot (x_2 + x_3) = x_1(x_2 + x_3) \\
f_2 &= \bar{x}_1 \cdot f_2|_{\bar{x}_1} + x_1 \cdot f_2|_{x_1} = \bar{x}_1 \cdot x_2 + x_1 \cdot (x_2 + x_3) = x_2 + x_1x_3 \\
f_3 &= \bar{x}_1 \cdot f_3|_{\bar{x}_1} + x_1 \cdot f_3|_{x_1} = \bar{x}_1 \cdot x_2x_3 + x_1 \cdot x_3 = x_3(x_1 + x_2) \\
f_4 &= \bar{x}_1 \cdot f_4|_{\bar{x}_1} + x_1 \cdot f_4|_{x_1} = \bar{x}_1 \cdot x_2x_3 + x_1 \cdot 1 = x_1 + x_2x_3 \\
f_5 &= \bar{x}_1 \cdot f_5|_{\bar{x}_1} + x_1 \cdot f_5|_{x_1} = \bar{x}_1 \cdot x_2x_3 + x_1 \cdot x_2 = x_2(x_1 + x_3) \\
f_6 &= \bar{x}_1 \cdot f_6|_{\bar{x}_1} + x_1 \cdot f_6|_{x_1} = \bar{x}_1 \cdot x_3 + x_1 \cdot (x_2 + x_3) = x_3 + x_1x_2.
\end{aligned}$$

Rivest presented a more general version of this circuit. For any odd integer  $n$  greater than 1, the general circuit consists of  $n$  two-input AND gates alternating with  $n$  two-input OR gates in a single cycle, with inputs  $x_1, \dots, x_n$  repeated, as shown in Figure 3.5. Analyzing the general circuit in the same manner as above, we find that

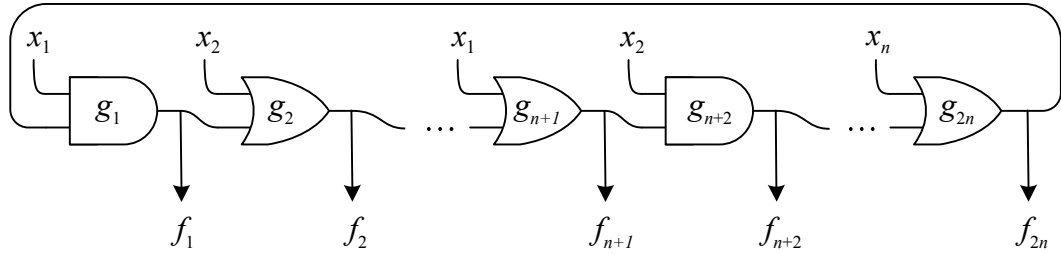


Figure 3.5: A cyclic combinational circuit with  $n$  inputs (for any odd  $n \geq 3$ ) due to Rivest.

it implements the functions

$$\begin{aligned}
f_1 &= x_1(x_n + x_{n-1}(\cdots(x_3 + x_2)\cdots)) \\
f_2 &= x_2 + x_1(x_n + \cdots(x_4x_3)\cdots) \\
&\vdots \\
f_{2n} &= x_n + x_{n-1}(x_{n-2} + \cdots(x_2x_1)\cdots).
\end{aligned}$$

Note that the functions are symmetrical with respect to a cyclic permutation of the variables.

### 3.4.1 Optimality

To show that circuit is optimal, we must show that it satisfies Properties 3.1 and 3.2.

1. To show that each function depends on all  $n$  input variables, we note that in the parenthesized expression, each variable appears exactly once. Without loss of generality, consider the  $i$ -th function  $f_i$  in the list, for an odd  $i$ , and consider the  $j$ -th variable appearing in its expression, from the left-hand side. To show the dependence on this variable, set each variable preceding a product to 1, and each variable preceding a sum to zero, beginning on the left-hand side, until we arrive at  $x_j$ . Set the variable following  $x_j$  to 1 and all variables following that to 0. The result is

$$f_i = 1(0 + 1(0 + \cdots + x_j(1 + 0(0 + 0(\cdots)))))) = x_j.$$

2. To show that all the functions are distinct, we exhibit an assignment that sets any chosen function to 0 if it is odd-numbered (to 1 if it is even-numbered), while setting all the other functions to 1 (to 0, respectively). Without loss of generality, consider function  $f_i$ , for an odd  $i \leq n$ . This function is the output of an AND gate with input  $x_i$ . Set  $x_i$  to 0 and set all the other the variables to 1. Clearly,  $f_i$  has value 0 while all the other functions have value 1 in this case.

(Rivest stated these conditions without proof.)

### 3.4.2 Acyclic Lower Bound

Note that the Rivest circuit has  $n$  input variables and implements  $2n$  distinct output functions with  $2n$  fan-in 2 gates. According to Claim 3.2, an acyclic circuit implementing the same functions requires at least

$$\left\lceil \frac{n-1}{2-1} \right\rceil + 2n - 1 = 3n - 2$$

fan-in 2 gates. For large  $n$ , the improvement factor is

$$\frac{\text{size of cyclic}}{\text{size of acyclic}} = \frac{2n}{3n-2} \approx \frac{2}{3}.$$

Rivest's cyclic circuit is two-thirds the size of any acyclic circuit implementing the same functions.

Given a circuit with a single cycle, we can always obtain a corresponding acyclic version by breaking the feedback and doubling the length of the chain, as shown in Figure 3.6. (The input  $\perp$  indicates any constant value.)

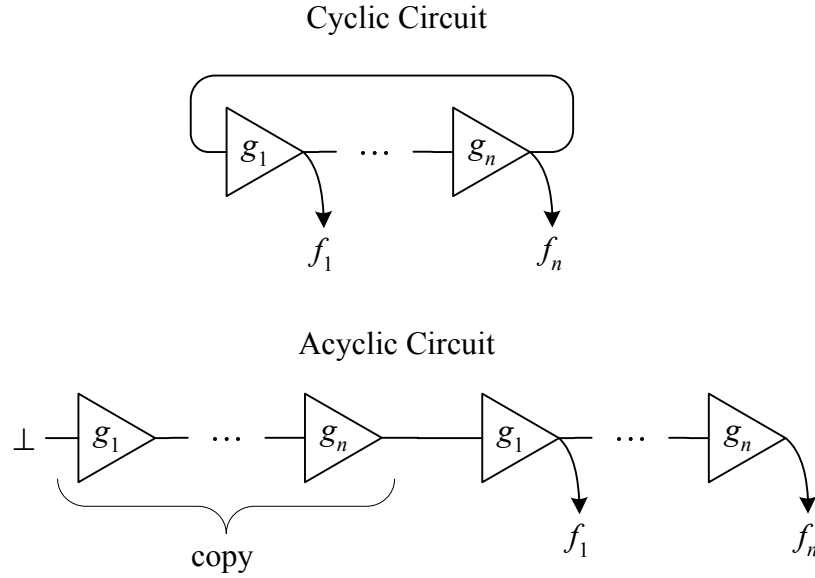


Figure 3.6: Obtaining an equivalent acyclic circuit from a cyclic circuit.

In general, this will not yield an optimal acyclic circuit. However, in the case of Rivest's circuit, the bound of  $3n - 2$  is, in fact, tight. To obtain an acyclic circuit with  $3n - 2$  gates, we break the cycle and prepend a copy of the last  $n - 2$  gates. For  $n = 3$ , we simply prepend an OR gate with inputs  $x_2$  and  $x_3$ , as shown in Figure 3.7.

Rivest's circuit is also optimal seen from a different perspective. The circuit consists of AON gates, and yet none of the output functions are implementable with a single AON gate, regardless of the fan-in. Thus, any acyclic circuit implementing the functions requires at least one more gate.



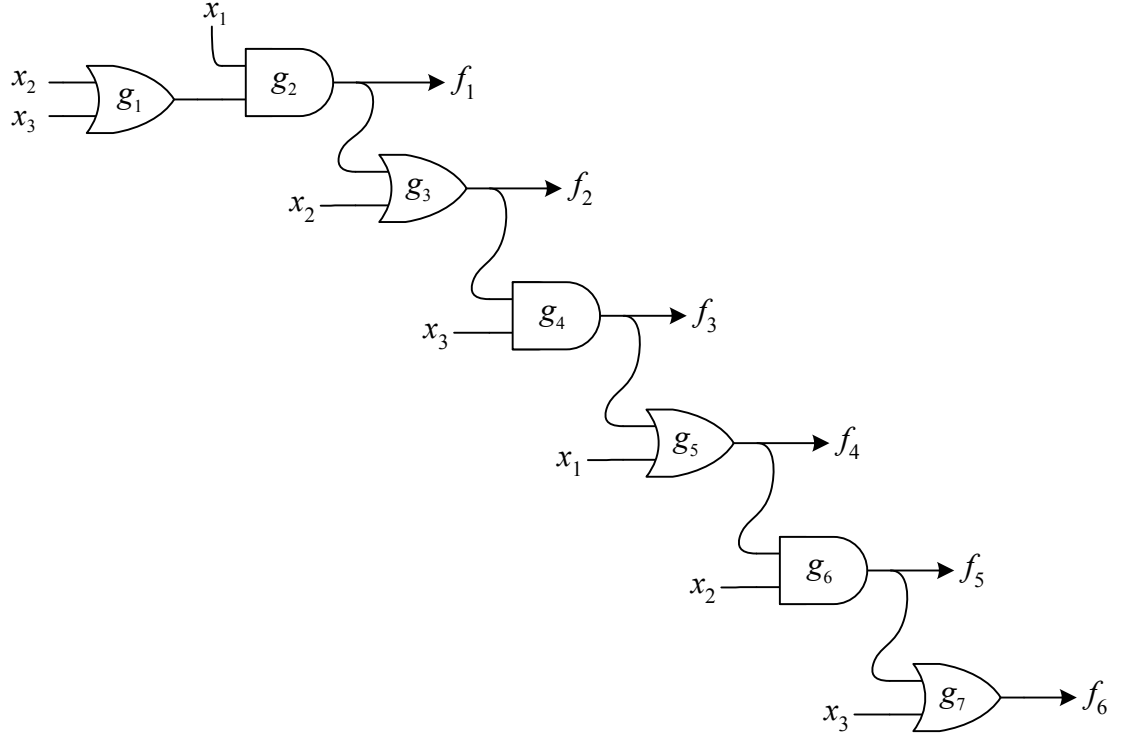


Figure 3.7: An acyclic circuit implementing the same functions as the circuit in Figure 3.4.

### 3.4.3 A Generalization

We note that Rivest's circuit can be generalized to AND and OR gates with arbitrary fan-in. The circuit, shown in Figure 3.8, consists of  $2n$  fan-in  $d$  AND/OR gates, with  $n(d-1)$  inputs repeated, for  $n \geq 3$ ,  $n$  odd, and  $d \geq 2$ .

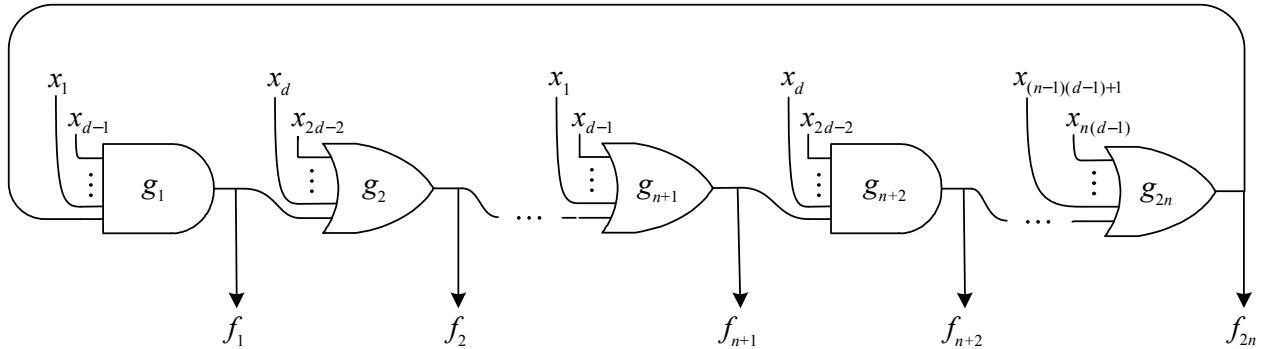


Figure 3.8: A generalization of Rivest's circuit to gates with fan-in greater than 2.

This circuit produces outputs

$$\begin{aligned}
 f_1 &= y_1(y_n + y_{n-1}(\cdots(y_3 + y_2)\cdots)) \\
 f_2 &= y_2 + y_1(y_n + \cdots(y_4 y_3)\cdots) \\
 &\vdots \\
 f_{2n} &= y_n + y_{n-1}(y_{n-2} + \cdots(y_2 y_1)\cdots),
 \end{aligned}$$

where

$$\begin{aligned}
 y_1 &= x_1 \cdots x_{d-1} \\
 y_2 &= x_d + \cdots + x_{2d-2} \\
 &\vdots \\
 y_n &= x_{(n-1)(d-1)+1} + \cdots + x_{n(d-1)}.
 \end{aligned}$$

It may be shown that all  $2n$  functions are distinct, and that each depends on all  $n(d-1)$  input variables.

### 3.4.4 Variants

We note that many different circuits of the same general form as Rivest's example exist. In Figure 3.9, we show a circuit with 4 variables and 8 gates in a single cycle. As with Rivest's circuit, this one produces distinct output functions, each of which depends on all the variables.

A more intriguing example is shown in Figure 3.10. It consists of two copies of Rivest's circuit with the outputs of the first fed as inputs into the second. Although not shown here, we assert that this circuit produces 20 functions are distinct, and each depends on all 5 variables.

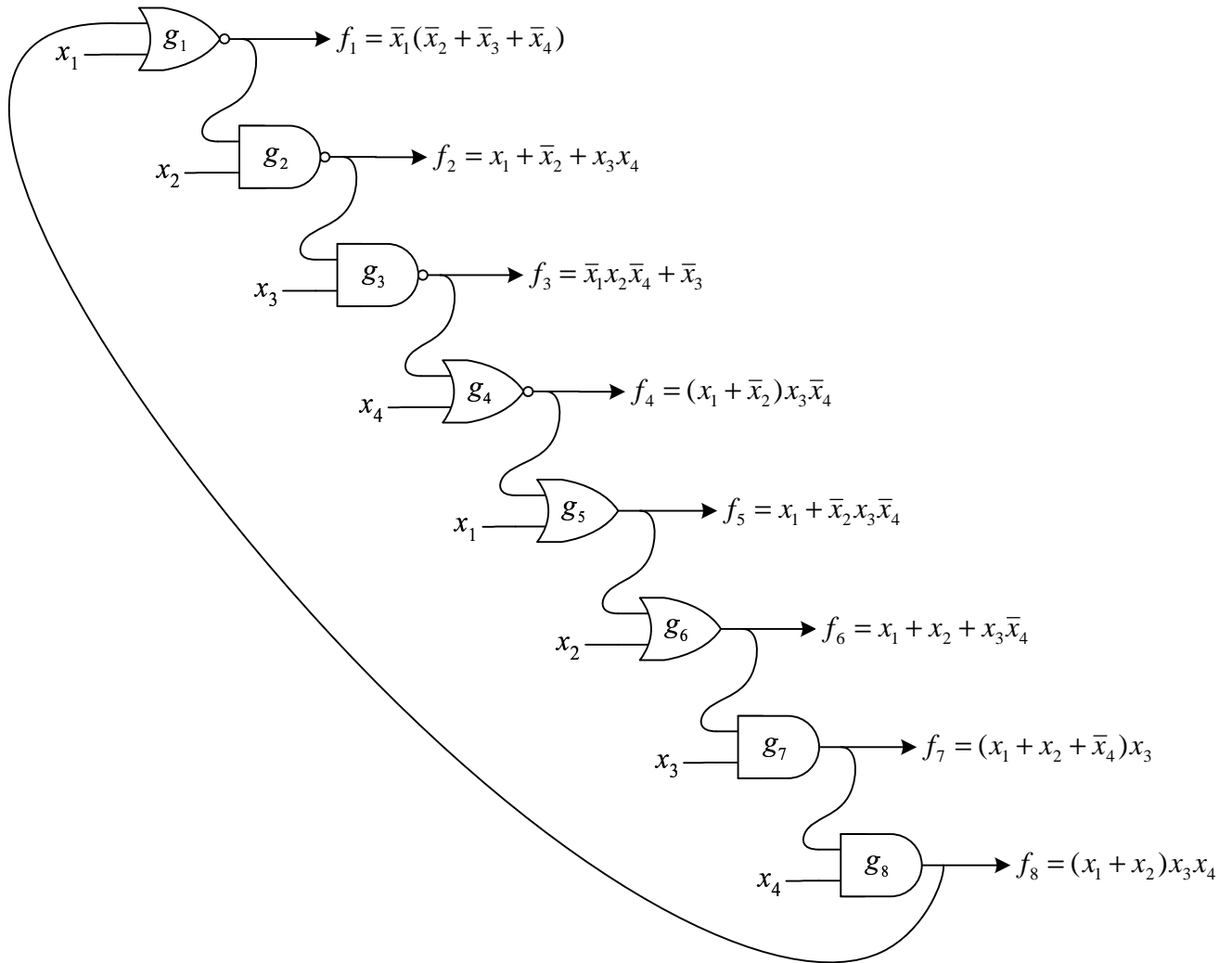


Figure 3.9: A circuit with the same properties as Rivest's example.

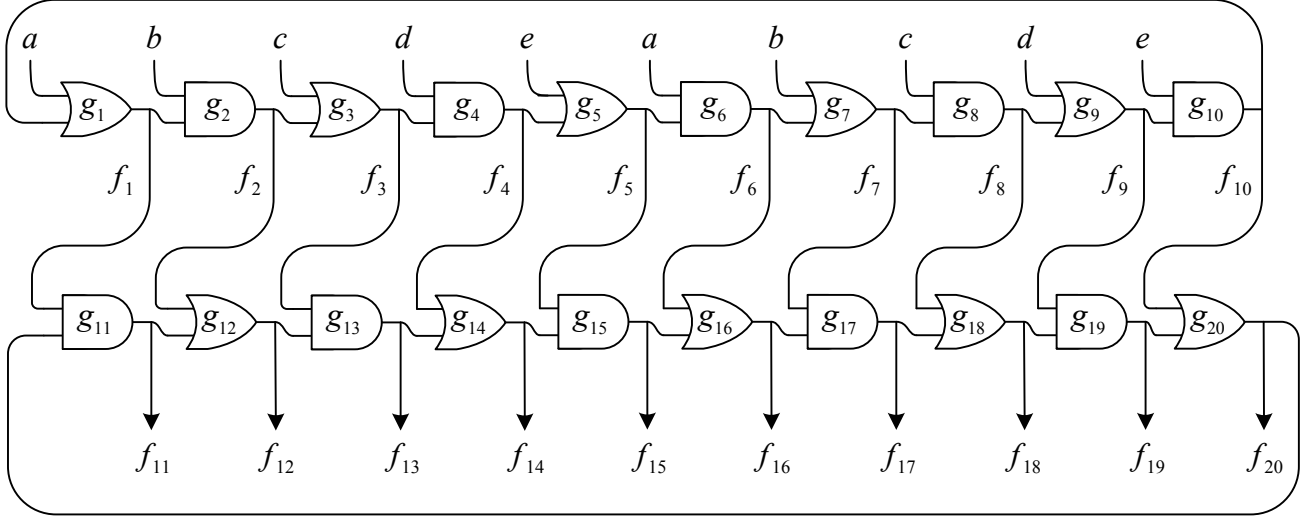


Figure 3.10: A pair of Rivest circuits,  $n = 5$ , stacked.

### 3.5 A Minimal Cyclic Circuit with Two Gates

We provide an example of a circuit with the same property as Rivest's circuit, but with only two gates. The circuit, shown in Figure 3.11, consists of two fan-in 4 gates of the form

$$g(w, x, y, z) = wx \oplus yz.$$

connected in a cycle with 5 inputs,  $a, b, c, d, e$ . The circuit computes  $f$  and  $g$ :

$$f = ab \oplus gc$$

$$g = f\bar{c} \oplus de.$$

To verify that the circuit is combinational, note that if  $c = 0$ ,  $f$  assumes a definite value. We have

$$f|_{\bar{c}} = ab \oplus g0 = ab$$

$$g|_{\bar{c}} = f1 \oplus de = ab \oplus de.$$

Similarly, if  $c = 1$ , then  $g$  also assumes a definite value. We have

$$g|_c = f0 \oplus de = de$$

$$f|_c = ab \oplus g1 = ab \oplus de.$$

Assembling the output functions, we obtain

$$\begin{aligned} f &= \bar{c} \cdot f|_{\bar{c}} + c \cdot f|_c = ab \oplus cde \\ g &= \bar{c} \cdot g|_{\bar{c}} + c \cdot g|_c = ab\bar{c} \oplus de. \end{aligned}$$

With the functions thus written in XNF form, we can readily assert that  $f$  and  $g$  are distinct and that each depends on all 5 variables. Now, consider an acyclic circuit, also with fan-in 4 gates, that computes the same functions. Since a single fan-in 4 gate cannot possibly compute a function of 5 variables, we conclude that the acyclic circuit must have 3 gates.

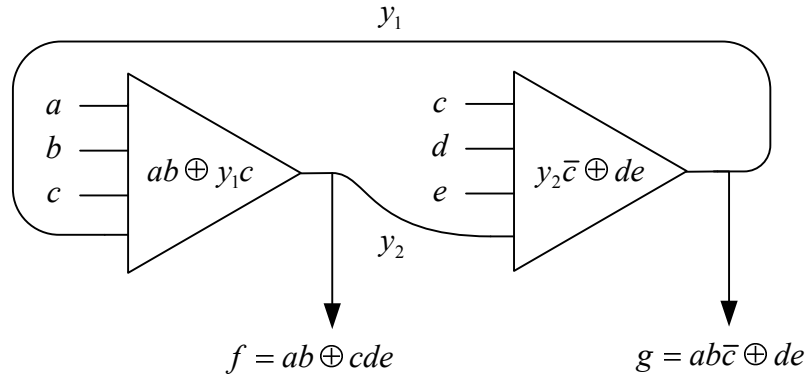


Figure 3.11: A cyclic circuit with two gates.

## 3.6 Circuits with Multiple Cycles

In this section, we present examples of cyclic circuits with multiple cycles, culminating with the main result of this section: a cyclic circuit that is one-half the size of any equivalent acyclic circuit.

### 3.6.1 A Cyclic Circuit with Two Cycles

Consider the circuit shown in Figure 3.12, written in a general form. The inputs are  $x_1, \dots, x_n$ , grouped together in the figure as  $X$ . (A diagonal line across a line indicates that it represents multiple wires.) There are three gates, connected in a

configuration consisting of two cycles:

$$\begin{aligned} f_1 &= \alpha_1 \oplus \beta_1 f_3 \\ f_2 &= \alpha_2 \oplus \beta_2 f_3 \\ f_3 &= \alpha_3 \oplus \beta_3 f_1 \oplus \gamma_3 f_2 \oplus \delta_3 f_1 f_2 \end{aligned}$$

where the  $\alpha$ 's,  $\beta$ 's,  $\gamma$ 's, and  $\delta$ 's are arbitrary functions of the input variables.

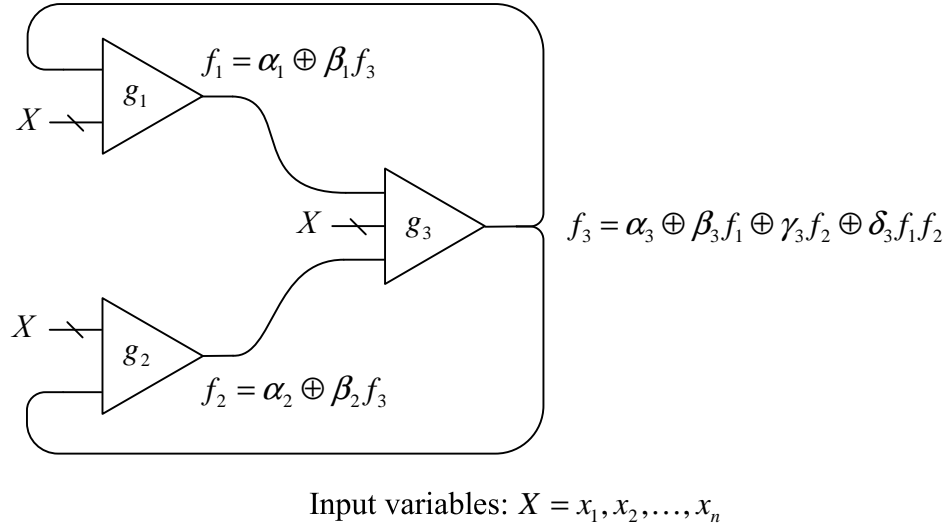


Figure 3.12: A cyclic circuit with two cycles.

### 3.6.2 Analysis in Arbitrary Terms

We analyze this circuit with the goal of obtaining a necessary and sufficient condition for combinationality, as well as expressions for the gate outputs in terms of the inputs. We proceed on a case basis.

#### Case I

Suppose that for some  $X$ ,  $\beta_1 = 0$ . In this case  $f_1$  assumes the definite value  $\alpha_1$ . This situation is shown in Figure 3.13. Now suppose further that  $\gamma_3 \oplus \delta_3 \alpha_1 = 0$ . In this case,  $f_3$  assumes a definite value of  $\alpha_3 \oplus \beta_3 \alpha_1$ . Given this value for  $f_3$ ,  $f_2$  assumes a definite value of  $\alpha_2 \oplus \beta_2 \alpha_3 \oplus \beta_2 \beta_3 \alpha_1$ . This situation is shown in Figure 3.14. We conclude that the functions assume definite values if  $\beta_1 = 0$  and  $\gamma_3 \oplus \delta_3 \alpha_1 = 0$ .

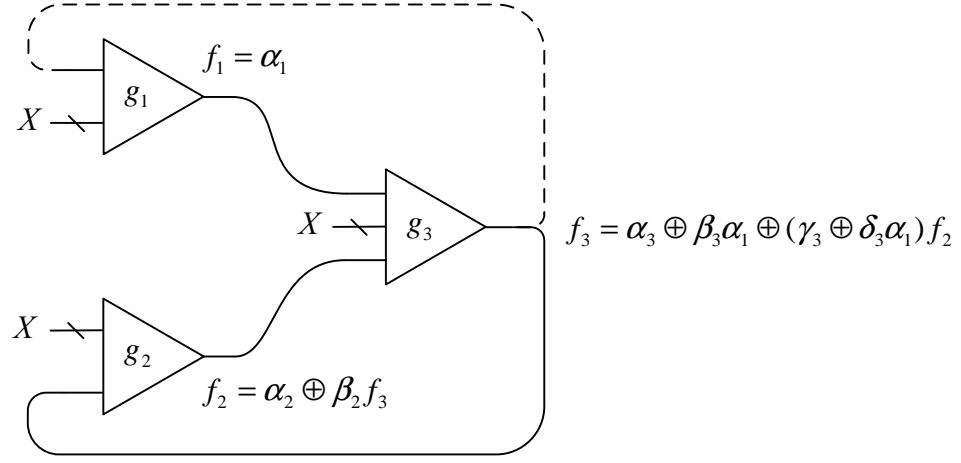


Figure 3.13: The circuit of Figure 3.12 if  $\beta_1 = 0$ .

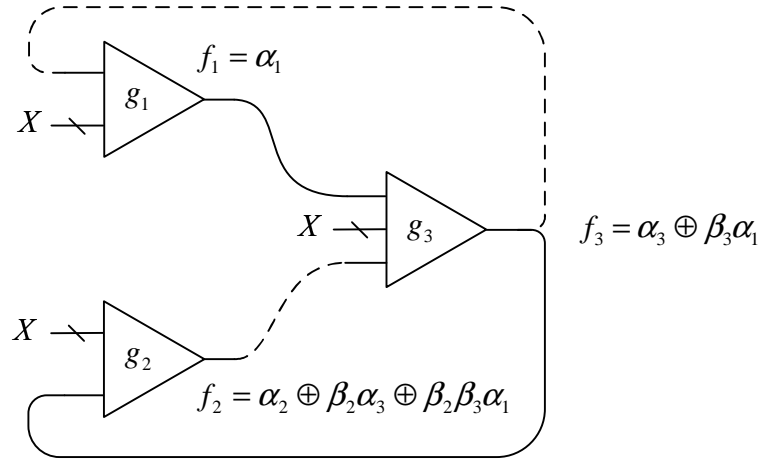


Figure 3.14: The circuit of Figure 3.12 if  $\beta_1 = 0$  and  $\gamma_3 \oplus \delta_3 \alpha_1 = 0$ .

### Case II

A symmetrical analysis shows that the functions  $f_1, f_2$  and  $f_3$  assume definite values if  $\beta_2 = 0$  and  $\beta_3 \oplus \delta_3 \alpha_2 = 0$ .

### Case III

Suppose that for some  $X$ , we have  $\beta_1 = 0$  and  $\beta_2 = 0$ . In this case  $f_1$  and  $f_2$  assume definite values of  $\alpha_1$  and  $\alpha_2$ , respectively. Given these values for  $f_1$  and  $f_2$ ,  $f_3$  assumes a definite value of  $\alpha_3 \oplus \beta_3 \alpha_1 \oplus \gamma_3 \alpha_2 \oplus \delta_3 \alpha_1 \alpha_2$ . This situation is shown in Figure 3.15.

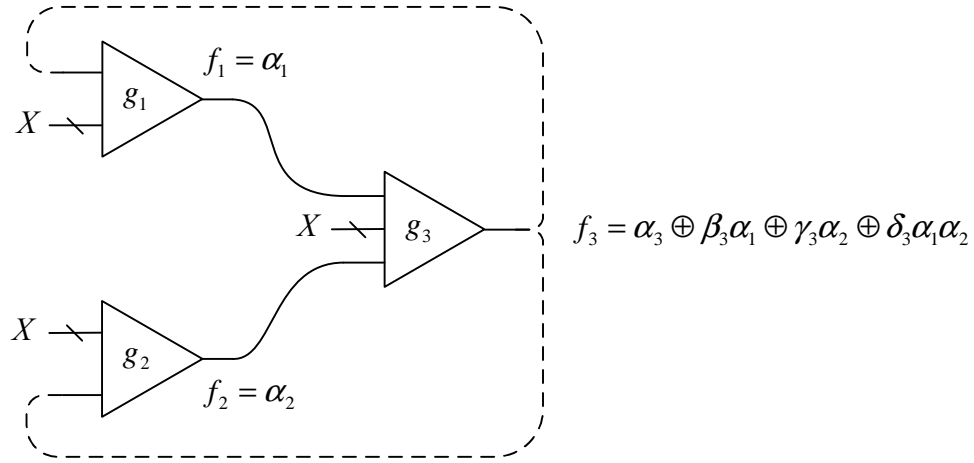


Figure 3.15: The circuit of Figure 3.12 if  $\beta_1 = 0$  and  $\beta_2 = 0$ .

### Case IV

Suppose that  $\beta_3 = \gamma_3 = \delta_3 = 0$ . In this case  $f_3$  assumes the definite value  $\alpha_3$ . Given this value for  $f_3$ ,  $f_1$  and  $f_2$  assume the definite values  $\alpha_1 \oplus \beta_1 \alpha_3$  and  $\alpha_2 \oplus \beta_2 \alpha_3$ , respectively. This situation is shown in Figure 3.16.  $\square$

Let

$$c_1 = \overline{\beta_1} \cdot \overline{(\gamma_3 \oplus \delta_3 \alpha_1)}, \quad (3.1)$$

$$c_2 = \overline{\beta_2} \cdot \overline{(\beta_3 \oplus \delta_3 \alpha_2)}, \quad (3.2)$$

$$c_3 = \overline{\beta_1} \cdot \overline{\beta_2}, \quad (3.3)$$

$$c_4 = \overline{\beta_3} \cdot \overline{\gamma_3} \cdot \overline{\delta_3}. \quad (3.4)$$



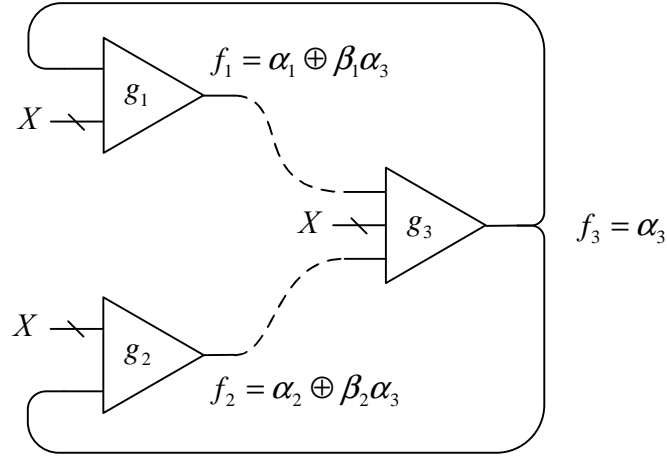


Figure 3.16: The Circuit of Figure 3.12 if  $\beta_3 = \gamma_3 = \delta_3 = 0$ .

We conclude that the circuit is combinational iff

$$C = c_1 + c_2 + c_3 + c_4$$

holds. If  $C$  holds, then the functions have values

$$f_1 = c_1 \alpha_1 + c_2 (\alpha_1 \oplus \beta_1 \alpha_3 \oplus \beta_1 \gamma_3 \alpha_2) + c_3 \alpha_1 + c_4 (\alpha_1 \oplus \beta_1 \alpha_3) \quad (3.5)$$

$$f_2 = c_1 (\alpha_2 \oplus \beta_2 \alpha_3 \oplus \beta_1 \beta_3 \alpha_1) + c_2 \alpha_2 + c_3 \alpha_2 + c_4 (\alpha_2 \oplus \beta_2 \alpha_3) \quad (3.6)$$

$$f_3 = c_1 (\alpha_3 \oplus \beta_3 \alpha_1) + c_2 (\alpha_3 \oplus \gamma_3 \alpha_2) + c_3 (\alpha_3 \oplus \beta_3 \alpha_1 \oplus \gamma_3 \alpha_2 \delta_3 \alpha_1 \alpha_2) + c_4 \alpha_3 \quad (3.7)$$

### 3.6.3 A Circuit Three-Fifths the Size

Let's make the circuit of Figure 3.12 somewhat more concrete. Suppose that the inputs are  $a, b, x_1, x_2, \dots, y_1, y_2, \dots, z_1, z_2, \dots$ . Suppose that the gates are defined by

$$\alpha_1 = \bar{a}X, \quad \alpha_2 = \bar{b}Y, \quad \alpha_3 = Z,$$

where

$$X = x_1 x_2 \cdots, \quad Y = y_1 y_2 \cdots, \quad Z = z_1 z_2 \cdots,$$

and

$$\begin{aligned}\beta_1 &= a, & \beta_2 &= b \\ \beta_3 &= \bar{a}, & \gamma_3 &= \bar{b}, & \delta_3 &= \bar{a}\bar{b}.\end{aligned}$$

The resulting circuit is shown in Figure 3.17. For this circuit, the conditions defined

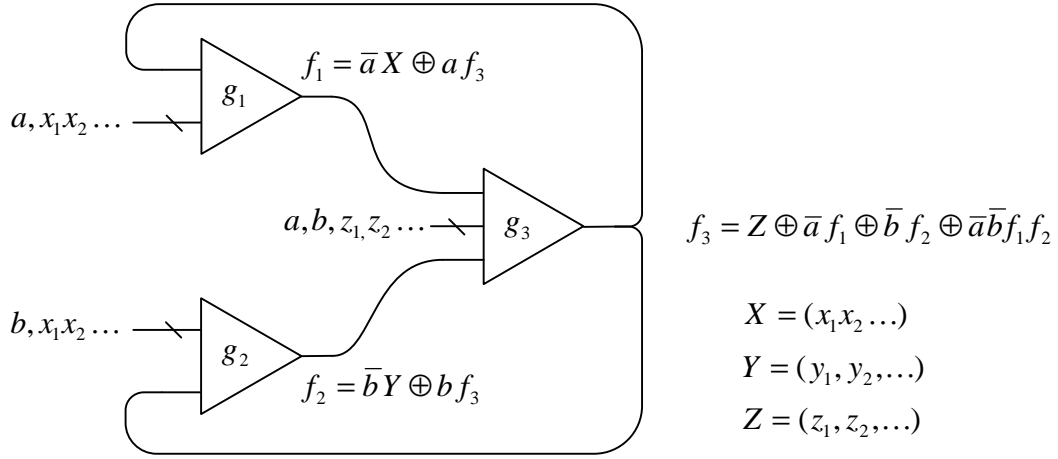


Figure 3.17: Variant of the circuit of Figure 3.12.

in Equations 3.1– 3.4 evaluate to

$$\begin{aligned}c_1 &= \bar{a}(b + X) \\ c_2 &= \bar{b}(a + Y) \\ c_3 &= \bar{a}\bar{b} \\ c_4 &= ab\end{aligned}$$

It may easily be verified that for every combination of values assigned to  $a$  and  $b$ , one of  $c_1, c_2, c_3, c_4$  is true. The functions defined in Equations 3.5– 3.7 become

$$\begin{aligned}f_1 &= X \oplus a(X \oplus Y \oplus Z) \oplus abY \\ f_2 &= Y \oplus b(X \oplus Y \oplus Z) \oplus abX \\ f_3 &= X \oplus Y \oplus Z \oplus XY \oplus a(X \oplus XY) \oplus b(Y \oplus XY) \oplus abXY\end{aligned}$$

With the functions expressed in XNF notation, we can assert that they are distinct and that each depends on all the variables.

To make the situation more concrete, suppose that

$$X = c e g i, \quad Y = d f h j, \quad Z = k l.$$

There are a total of 12 variables ( $a$  through  $l$ ). Each gate has fan-in 6. This situation is shown in Figure 3.18.

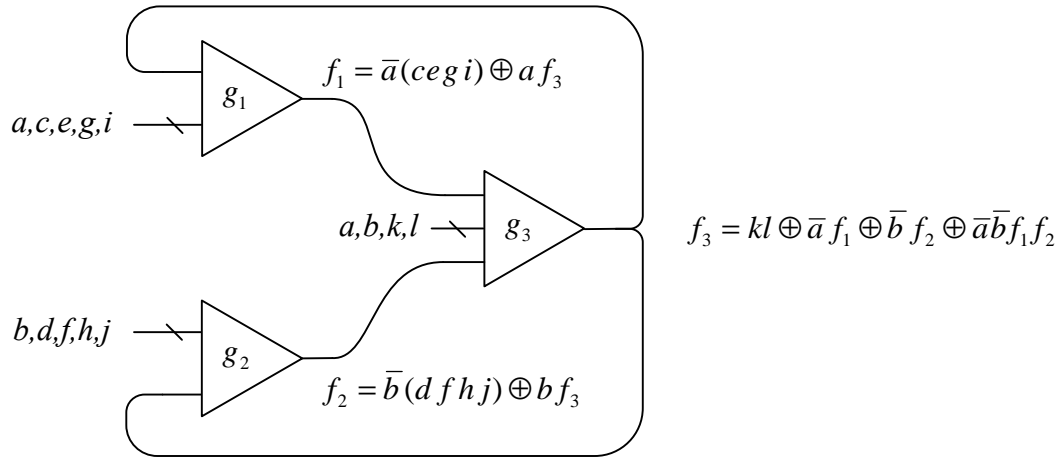


Figure 3.18: Circuit of Figure 3.17 with 12 variables.

According to Claim 3.2, an acyclic circuit implementing the same functions requires at least

$$\left\lceil \frac{v-1}{d-1} \right\rceil + m - 1$$

gates, where  $v = 12$  (the number of variables),  $d = 6$  (the fan-in) and  $m = 3$  (the number of functions). Thus

$$\left\lceil \frac{v-1}{d-1} \right\rceil + m - 1 = \left\lceil \frac{12-1}{6-1} \right\rceil + 3 - 1 = 5.$$

We conclude that the circuit in Figure 3.18 is at most  $\frac{3}{5}$  the size of any equivalent acyclic circuit.

### 3.6.4 A Circuit One-Half the Size

Consider the circuit shown in Figure 3.19, a generalization of the circuit in Figure 3.12 to  $k$  gates. We argue the validity of this circuit informally. On the one hand, for each

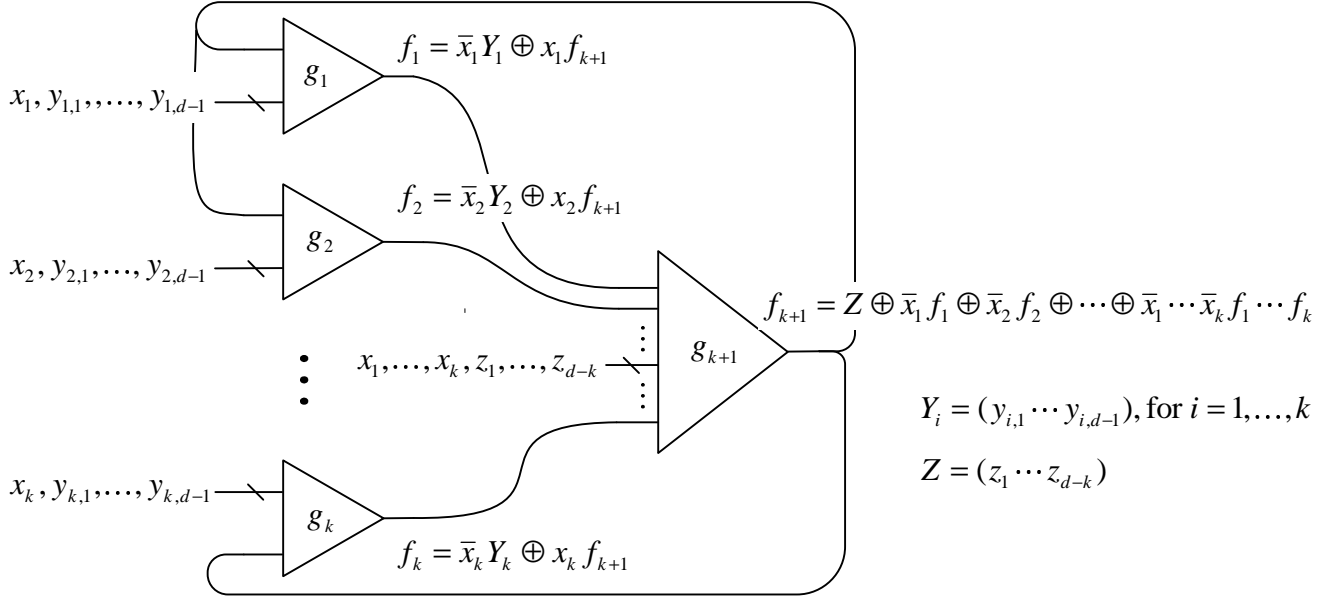


Figure 3.19: A generalization of the circuit of Figure 3.12.

variable  $x_i$ , if  $x_i = 0$  then the function  $f_i$  does not depend on  $f_{k+1}$ . On the other hand, if  $x_i = 1$ , then  $f_{k+1}$  does not depend on  $f_i$ . We conclude that none of the  $k$  cycles can be sensitized, and so the circuit is combinational. Now consider the function  $f_i$  implemented by each gate. With  $x_i = 0$ ,  $f_i$  depends on the variables  $y_{1,1}, \dots, y_{1,d-1}$ . Since  $f_{k+1}$  depends on  $f_i$ , it also depends on these variables. Thus  $f_{k+1}$  depends on all the variables  $y_{i,j}$  for  $i = 1, \dots, k, j = 1, \dots, d-1$ . With  $x_i = 1$ ,  $f_i$  depends on  $f_{k+1}$ ; hence it also depends on all these variables. We conclude that each function depends on all the variables.

With a fan-in of  $d$ , we have a total of

$$v = k(d-1) + d - 2k = (k+1)d - 3k$$

variables. We have

$$m = k + 1$$

gates. According to Claim 3.2, an acyclic circuit implementing the same functions requires at least

$$\left\lceil \frac{v-1}{d-1} \right\rceil + m - 1$$

gates. The improvement factor is,

$$\frac{m}{\left\lceil \frac{v-1}{d-1} \right\rceil + m - 1} = \frac{k+1}{\left\lceil \frac{(k+1)d-3k}{d-1} \right\rceil + k}$$

gates. Suppose that  $d = 3k$ , and that  $k$  is large. Then the ratio is

$$\frac{k+1}{\left\lceil \frac{3k^2-1}{3k-1} \right\rceil + k} \approx \frac{k}{k+k} = \frac{1}{2}.$$

We conclude that the circuit of Figure 3.19 is at most one-half the size of any equivalent acyclic circuit. According to Claim 3.3, this is the best possible improvement factor that we can obtain with the fan-in lower bound of Section 3.2.

### 3.7 Summary

Admittedly, the circuits in latter half of this chapter are a bit contrived. The constructions assume “gates” with arbitrarily large fan-in. The gates in this context should properly be described as sub-circuits. And yet the fact remains that there exist families of functions that can be implemented by cyclic circuits with 50% fewer gates than is possible with equivalent acyclic circuits. Feedback is an inescapable feature of the Boolean circuit model. Hereafter, complexity theorists must stop speaking of Boolean circuits as directed acyclic graphs (DAG’s). Even in an abstract setting, the optimal circuit under consideration may well be cyclic.

# Chapter 4

## Analysis

*All are lunatics, but he who can analyze his delusions is called a philosopher.* – Ambrose Pierce (1869–1950)

For logic design, as with any engineering construct, analysis provides the underpinnings to synthesis. In Chapter 2, we characterized combinational circuits in terms of explicit signal values (0, 1, and  $\perp$ ). For most real circuits, an exhaustive analysis of every input combination is intractable. Instead, we turn to symbolic techniques to analyze and validate cyclic designs.

In what has been described as the most influential Master’s Thesis ever, in 1938 Claude Shannon proposed the use of symbolic logic for the analysis of relay circuits [39]. What had been until then a desultory, ad hoc process was now logical and systematic. His brilliant work brought digital systems into the realm of mathematics.

Symbolic analysis derives *formulas* that describe the logic values of signals in a circuit in terms of its input signals. Instead of working with explicit values, the analysis takes place over a domain consisting of a set of *functions*. In a Boolean setting, this domain is the set  $\mathcal{B}$  of Boolean-valued functions of Boolean variables, i.e., of maps

$$\{f : \{0, 1\}^n \rightarrow \{0, 1\}\}.$$

for all positive integers  $n$ .

Boolean operations, such as AND, OR and NOT, are applied to functions. For instance, given functions  $f$  and  $g$

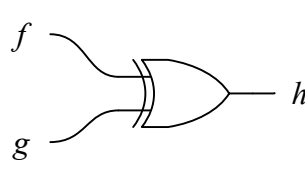
$$f = x_1(x_2 + x_3),$$

$$g = x_1 + x_2x_3,$$

the XOR operation yields a new function  $h$  of these input variables,

$$h = x_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2x_3.$$

In a sense, symbolic computation is equivalent to processing all input combinations in parallel. For the gate on the left of Figure 4.1, the computation is equivalent to processing the truth tables shown on the right.



$x_1$	$x_2$	$x_3$	$f$	$g$	$h$
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	1	0

Figure 4.1: An example of symbolic computation.

Throughout, we often make statements such as

“if  $x_1 + x_2x_3$ , then ...”

By this we mean, implicitly,

“for  $(x_1, x_2, x_3) \in \{0, 1\}^3$  such that  $x_1 + x_2x_3 = 1$ , ...”

For cyclic circuits, we must allow for the possibility that signals never settle to definite values. Symbolically, the analysis takes place over a domain  $\mathcal{T}$  of the set of

ternary-valued functions of ternary variables, i.e., maps

$$\{f : \{0, 1, \perp\}^n \rightarrow \{0, 1, \perp\}\}.$$

for all positive integers  $n$ . For instance, given ternary-valued functions  $f$  and  $g$

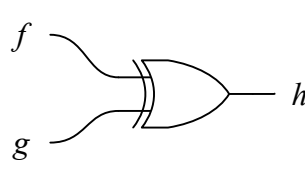
$$f = \begin{cases} 0 & \text{if } \bar{x}_1 + \bar{x}_2\bar{x}_3 \\ 1 & \text{if } x_1(x_2 \oplus x_3) \\ \perp & \text{else} \end{cases}$$

$$g = \begin{cases} 0 & \text{if } \bar{x}_1\bar{x}_2 + \bar{x}_1\bar{x}_3 \\ 1 & \text{if } x_1\bar{x}_2 + \bar{x}_1x_2x_3 \\ \perp & \text{else} \end{cases}$$

the XOR operation yields a new ternary-valued function  $h$

$$h = \begin{cases} 0 & \text{if } \bar{x}_1\bar{x}_3 + \bar{x}_2x_3 \\ 1 & \text{if } x_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2x_3 \\ \perp & \text{else} \end{cases}$$

The corresponding truth tables are shown in Figure 4.2.



$x_1$	$x_2$	$x_3$	$f$	$g$	$h$
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	0	1	1	1	0
1	1	0	1	$\perp$	$\perp$
1	1	1	$\perp$	$\perp$	$\perp$

Figure 4.2: An example of ternary-valued symbolic computation.



Note that  $\text{XOR}(\perp, x) = \text{XOR}(x, \perp) = \perp$ , for all  $x \in \{0, 1, \perp\}$ .

## 4.1 Decision Diagrams

From a computational standpoint, there would be little advantage to the symbolic approach if each function were represented as a truth table of all  $2^n$  combinations of Boolean values for  $n$  inputs. We could instead analyze the circuit for each input combination separately, in the manner described in Chapter 2.

Symbolic analysis gets its traction from efficient data structures. Chief among these are Binary Decision Diagram (BDDs). A BDD consists of a directed graph. Each node either has an associated input variable or is designated as a constant (“0” or “1”). Variables nodes have out-degree two: one edge designated as “0” (represented by a dashed line in diagrams) and the other designated as “1” (represented by a solid line in diagrams). Constant nodes have out-degree zero. To evaluate a function represented by BDD, we begin at a designated source node, and follow a path dictated by the values of the variables. When we terminate at a constant node, the function evaluates to this constant. An example of a BDD is given in Figure 4.3.

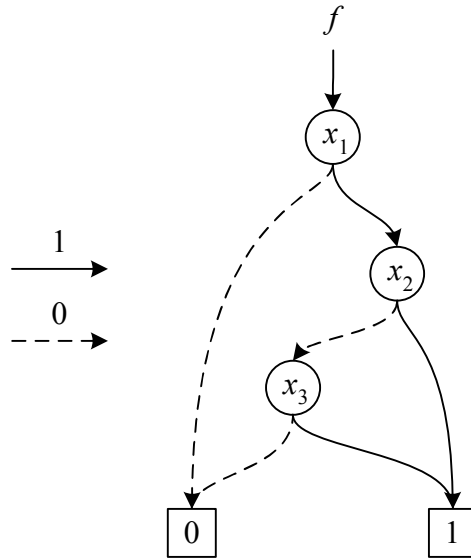


Figure 4.3: A binary decision diagram (BDD) implementing the function:  $f = x_1(x_2 + x_3)$ .

First proposed in 1959 by Lee [21], BDDs were popularized in 1986 by a seminal paper by Bryant [7]. (As of this writing, Bryant’s paper is ranked as the most widely cited paper in the history of Computer Science.) Although comparable in size to a truth table in the worst case, BDDs are surprisingly compact for most of the functions encountered in practice. Functions of up to 50 variables can often be represented with ease. The strength of the representation resides in the fact that it is canonical, and that it can be manipulated efficiently. Most logical operations have linear complexity in the number of variables.

Decision diagrams can readily be adapted to the multi-valued case. Multi-terminal binary decision diagrams (MTBDD), also known as “algebraic” decision diagrams, share all of the features of their two terminal counter parts: the representation is canonical, compact and efficient [2]. MTBDDs seem tailor-made for timing analysis. We use multiple terminal nodes to represent the combination of Boolean values and associated arrival times.

Consider the function specified by the truth table on the left-hand side in Figure 4.4 (implemented by the circuit fragment in Figure 2.12 in Chapter 2). Subscripts on the logical values indicate the arrival times. The corresponding MTBDD is shown on the right-hand side.

## 4.2 Controlling Values

Central to timing analysis is the concept of *controlling* values. Recall that 0 is the controlling value for an AND gate, as shown in Figure 2.5. Similarly, 1 is the controlling value for an OR gate. We propose a symbolic technique, called the *marginal* operator, for computing the set of controlling values for an arbitrary logic function . Before explaining the marginal, we define some ancillary operations.

The **restriction** operation (also known as the cofactor) of a function  $f$  with respect to a variable  $x$ ,

$$f|_{x=v},$$

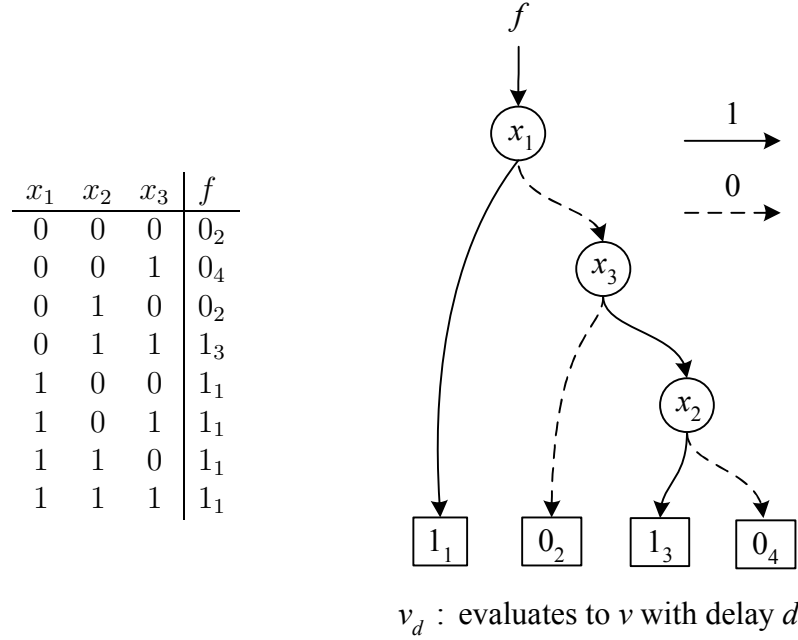


Figure 4.4: A multi-terminal binary decision diagram (MTBDD).

refers to the assignment of the constant value  $v \in \{0, 1\}$  to  $x$ . The **composition** operation of a function  $f$  with respect to a variable  $x$  and a function  $g$ ,

$$f|_{x=g},$$

refers to the substitution of  $g$  for  $x$  in  $f$ . A function  $f$  **depends** upon a variable  $x$  iff  $f|_{x=0}$  is not identically equal to  $f|_{x=1}$ . Call the variables that a function depends upon its **support set**.

The **universal quantification** operation (also known as consensus) yields a function

$$\forall(y_1, \dots, y_n)f$$

that equals 1 iff the given function  $f$  equals 1 for all  $2^n$  assignments of Boolean values to the variables  $y_1, \dots, y_n$ . The **existential quantification** operation (also known as smoothing) yields a function

$$\exists(y_1, \dots, y_n)f$$

that equals 1 iff the given function  $f$  equals 1 for *some* assignment of Boolean values to the variables  $y_1, \dots, y_n$ .

Now, the **marginal** operation yields a function

$$f \downarrow (y_1, \dots, y_n)$$

that equals 1 iff the given function  $f$  is invariant for all  $2^n$  assignments of Boolean values to  $y_1, \dots, y_n$ . For a single variable  $y$ , it equals 1 iff  $f|_{y=0}$  agrees with  $f|_{y=1}$ :

$$f \downarrow y = f|_{y=0} \cdot f|_{y=1} + \overline{f|_{y=0}} \cdot \overline{f|_{y=1}}.$$

(For a single variable, the marginal is the complement of what is known as the *Boolean difference*.) For several variables  $y_1, \dots, y_n$ , the marginal is computed as the universal quantification of the product of the marginals:

$$f \downarrow (y_1, \dots, y_n) = \forall y_1, \dots, y_n [(f \downarrow y_1) \cdots (f \downarrow y_n)].$$

(With several variables, the marginal is *not* the same as the complement of the Boolean difference, in general.) For example, with

$$f = x_1 + x_2y_1 + x_3y_2 + x_4y_1y_2,$$

we have

$$\begin{aligned} f \downarrow y_1 &= x_1 + x_3y_2 + \bar{x}_2(\bar{x}_4 + \bar{y}_2), \\ f \downarrow y_2 &= x_1 + x_2y_1 + \bar{x}_3(\bar{x}_4 + \bar{y}_1), \\ f \downarrow (y_1, y_2) &= x_1 + \bar{x}_2\bar{x}_3\bar{x}_4. \end{aligned}$$

Note that computing a marginal of  $n$  variables requires  $O(n)$  symbolic operations.

## 4.3 Analysis

Conceptually, the analysis is just an algorithmic implementation of procedure described in Chapter 2, Section 2.2.3. We apply definite values to the primary inputs, and track the propagation of signal values. Once we have established that a definite value has appeared on a gate output, this value persists for the duration of the analysis. The arrival time of a well-defined value at a gate output is determined either:

- by the arrival time of the earliest *controlling* input value;
- or by the arrival time of the latest *non-controlling* input value.

The analysis proceeds in time intervals. If the gates have fixed delay bounds, we can choose the interval length to match the shortest delay bound. In each time interval, we evaluate all the gates that received new input values in the previous interval. In this manner, we are assured that we know the earliest time that signal values becomes known. If definite Boolean values never arrive at one or more of the primary outputs, then we conclude that the circuit is not combinational.

We state the algorithm – somewhat informally – and illustrate its execution on examples.

### 4.3.1 Symbolic Analysis Algorithm

Let  $X = (x_1, \dots, x_n)$  be the primary inputs. We maintain a pair of *characteristic sets* for the output of each gate  $g_i$ . The first

$$C_i^{(0)}(X),$$

consists of the set of input assignments for which the gate evaluates to 0; the second,

$$C_i^{(1)}(X),$$

the set for which it evaluates to 1. Implicitly, the complement of the union of these two sets is the set of assignments for which the gate evaluates to  $\perp$ .

At the outset, all wires are assumed to have undefined values, so the characteristic sets are empty,

$$C_i^{(0)} := C_i^{(1)} := 0.$$

As the analysis proceeds, input assignments that induce gates to produce definite output values are added to these sets. Call the addition of input assignments to the set  $C_i^{(v)}$ , for some  $v \in \{0, 1\}$ , an *arrival event* valued  $v$  at gate  $g_i$ .

### Initialization

The initial arrival events occur at gates controlled by primary inputs. For instance, suppose that an AND gate  $g_i$  is connected to the primary input  $x$ . We have an initial arrival event

$$C_i^{(0)} := \bar{x}.$$

Similarly, suppose that an OR gate  $g_j$  is connected to the primary input  $y$ . We have an initial arrival event

$$C_j^{(1)} := y.$$

We compute such arrival events for all gates attached to the primary inputs.

### Propagation

Given the initial set of arrival events, we begin propagating events forward: to the fan-outs of the gates at which the signals arrived, and then to the fan-outs of these gates, and so on. In each interval, we compute new arrival events for gates based on the antecedent arrival events on their inputs.

1. Suppose that in the previous interval there was an arrival event valued  $v$  at gate  $g_i$ ; that  $g_i$  is a fan-in to gate  $g_j$ ; and that  $v$  is a controlling input value for  $g_j$ ,

producing an output value  $w$ . We compute

$$C_j^{(w)} := C_j^{(w)} + C_i^{(v)}.$$

If  $C_j^{(w)}$  changes as a result (i.e.,  $C_i^{(v)}$  was not contained in  $C_j^{(w)}$ ), then we have a new arrival event valued  $w$  at  $g_j$ .

2. Suppose that in the previous interval there was an arrival event valued  $v$  at gate  $g_i$ ; that  $g_i$  is a fan-in to gate  $g_j$ ; and that  $v$  is a non-controlling input value for  $g_j$ . Let  $g_{i_1}, \dots, g_{i_k}$  be *all* the gates that fan-in to  $g_j$ , and let  $v_{i_1}, \dots, v_{i_k}$  be the non-controlling values for these fan-in gates. Suppose that these non-controlling inputs produce an output value  $w$  for  $g_j$ . We compute

$$C_j^{(w)} := C_j^{(w)} + [C_{i_1}^{(v_{i_1})} \dots C_{i_k}^{(v_{i_k})}].$$

Again, if  $C_j^{(w)}$  changes as a result, then we have a new arrival event valued  $w$  at  $g_j$ .

To illustrate these propagation conditions, suppose that we have an AND gate  $g_3$  with fan-in gates  $g_1$  and  $g_2$ , as shown in Figure 4.5.

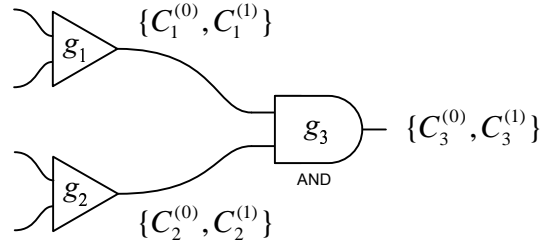


Figure 4.5: An illustration of the propagation conditions.

Suppose that the characteristic sets are

$$\begin{aligned} C_1^{(0)} &= x_1, & C_1^{(1)} &= x_2, \\ C_2^{(0)} &= x_3, & C_2^{(1)} &= x_4, \\ C_3^{(0)} &= x_1 + x_3, & C_3^{(1)} &= x_2 x_4. \end{aligned}$$

Now suppose that there is an arrival event valued 0 at  $g_1$  setting

$$C_1^{(0)} = x_1 + x_5.$$

In the next interval, we compute

$$C_3^{(0)} := C_3^{(0)} + C_1^{(0)} = x_1 + x_3 + x_5.$$

Now suppose that there is an arrival event valued 1 at  $g_1$  setting

$$C_1^{(1)} = x_2 + x_6.$$

In the next interval, we compute

$$C_3^{(1)} := C_3^{(1)} + [C_1^{(1)} C_2^{(1)}] = (x_2 + x_6)x_4.$$

## Termination

Termination is guaranteed since the cardinality of the characteristic sets either increases or remains unchanged with arrival events. A characteristic set cannot grow beyond the size of the full set of input assignments.

When the algorithm terminates, the union of the characteristic sets

$$C_i^{(0)} + C_i^{(1)}$$

for each gate  $g_i$  specifies the input assignments for which  $g_i$  produces definite values. If the complement of this union includes input assignments not in the “don’t care” set for any gate producing a primary output, then we conclude that the circuit is *not* combinational. In particular, if there are no “don’t care” input assignments, then the circuit is combinational if and only if the union consists of *all* input assignments for every gate producing a primary output.



Also, when the algorithm terminates, the time that has lapsed – the number of intervals times the interval length – gives a bound on the circuit delay.

### 4.3.2 Examples

We illustrate symbolic analysis on two detailed examples.

#### Example 4.1

Consider the circuit shown in Figure 4.6. It consists of six AND and OR gates, with two primary outputs,  $f_1$  and  $f_2$ , and five primary inputs  $a$ ,  $b$ ,  $c$ ,  $d$  and  $x$  (note that the input  $x$  is repeated).

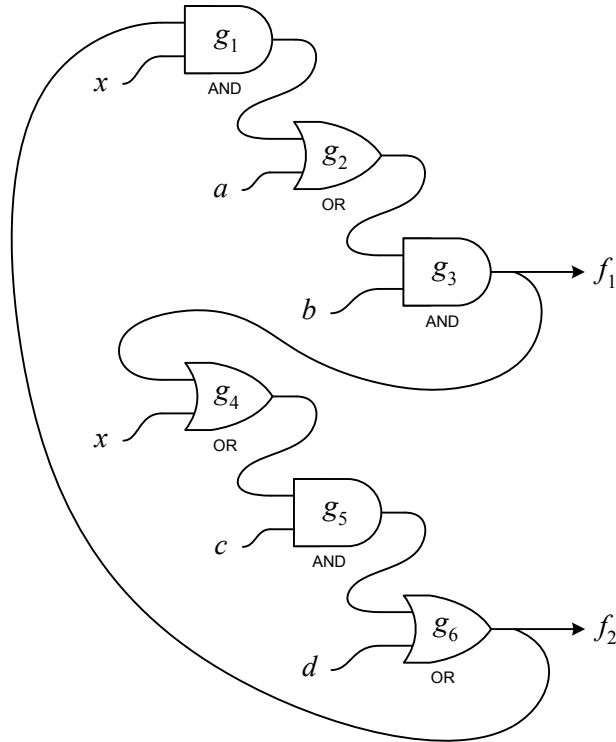


Figure 4.6: A cyclic combinational circuit.

We step through a symbolic analysis of this circuit. We assume that each gate has a delay bound of 1 time unit, and that the primary inputs arrive at time 0.

**Time 1**

For the AND gates, controlling values of 0 on the primary inputs result in

$$C_1^{(0)} = \bar{x}, \quad C_3^{(0)} = \bar{b}, \quad C_5^{(0)} = \bar{c}.$$

For the OR gates, controlling values of 1 on the primary inputs result in

$$C_2^{(1)} = a, \quad C_4^{(1)} = x, \quad C_6^{(1)} = d.$$

**Time 2**

For the AND gates, non-controlling values of 1 from the preceding OR gates result in

$$C_1^{(1)} = x d, \quad C_3^{(1)} = b a, \quad C_5^{(1)} = c x.$$

For the OR gates, non-controlling values of 0 from the preceding AND gates result in

$$C_2^{(0)} = \bar{a} \bar{x}, \quad C_4^{(0)} = \bar{x} \bar{b}, \quad C_6^{(0)} = \bar{d} \bar{c}.$$

**Time 3**

For the AND gates, controlling values of 0 from the preceding OR gates result in

$$C_1^{(0)} = \bar{x} + \bar{d} \bar{c}, \quad C_3^{(0)} = \bar{b} + \bar{a} \bar{x}, \quad C_5^{(0)} = \bar{c} + \bar{x} \bar{b}.$$

For the OR gates, controlling values of 1 from the preceding AND gates result in

$$C_2^{(1)} = a + x d, \quad C_4^{(1)} = x + b a, \quad C_6^{(1)} = d + c x.$$

**Time 4**

For the AND gates, non-controlling values of 1 from the preceding OR gates result in

$$\begin{aligned} C_1^{(1)} &= x(d + c), \\ C_3^{(1)} &= b(a + x d), \\ C_5^{(1)} &= c(x + b a). \end{aligned}$$

For the OR gates, non-controlling values of 0 from the preceding AND gates result in

$$\begin{aligned} C_2^{(0)} &= \bar{a}(\bar{x} + \bar{d} \bar{c}), \\ C_4^{(0)} &= \bar{x}(\bar{a} + \bar{b}), \\ C_6^{(0)} &= \bar{d}(\bar{c} + \bar{x} \bar{b}). \end{aligned}$$

**Time 5**

For the AND gates, controlling values of 0 from the preceding OR gates result in

$$\begin{aligned} C_1^{(0)} &= \bar{x} + \bar{d} \bar{c}, \\ C_3^{(0)} &= \bar{b} + \bar{a}(\bar{x} + \bar{d} \bar{c}), \\ C_5^{(0)} &= \bar{c} + \bar{x}(\bar{b} + \bar{a}). \end{aligned}$$

For the OR gates, controlling values of 1 from the preceding AND gates result in

$$\begin{aligned} C_2^{(1)} &= a + x(d + c), \\ C_4^{(1)} &= x + b a, \\ C_6^{(1)} &= d + c(x + b a). \end{aligned}$$

**Time 6**

For the AND gates, non-controlling values of 1 from the preceding OR gates result in

$$\begin{aligned} C_1^{(1)} &= x(d + c), \\ C_3^{(1)} &= b(a + x(d + c)), \\ C_5^{(1)} &= c(x + ba). \end{aligned}$$

For the OR gates, non-controlling values of 0 from the preceding AND gates result in

$$\begin{aligned} C_2^{(0)} &= \bar{a}(\bar{x} + \bar{d}\bar{c}), \\ C_4^{(0)} &= \bar{x}(\bar{b} + \bar{a}). \\ C_6^{(0)} &= \bar{d}(\bar{c} + \bar{x}(\bar{b} + \bar{a})). \end{aligned}$$

At this point, there are no new arrival events. The characteristic sets are:

$$\begin{array}{ll} C_1^{(0)} = \bar{x} + \bar{d}\bar{c}, & C_1^{(1)} = x(d + c), \\ C_2^{(0)} = \bar{a}(\bar{x} + \bar{d}\bar{c}), & C_2^{(1)} = a + x(d + c), \\ C_3^{(0)} = \bar{b} + \bar{a}(\bar{x} + \bar{d}\bar{c}), & C_3^{(1)} = b(a + x(d + c)), \\ C_4^{(0)} = \bar{x}(\bar{b} + \bar{a}), & C_4^{(1)} = x + ba, \\ C_5^{(0)} = \bar{c} + \bar{x}(\bar{b} + \bar{a}), & C_5^{(1)} = c(x + ba), \\ C_6^{(0)} = \bar{d}(\bar{c} + \bar{x}(\bar{b} + \bar{a})), & C_6^{(1)} = d + c(x + ba). \end{array}$$

Note that for each  $i = 1, \dots, 6$

$$C_i^{(0)} + C_i^{(1)} = 1.$$

Hence, all input assignments produce definite values at the outputs, and so we conclude that the circuit is combinational. Since we propagated events for 6 time units, we conclude that the circuit has delay 6.  $\square$

**Example 4.2**

Consider the circuit shown in Figure 4.7. It computes four output functions,  $f_1, f_2, f_3$ , and  $f_4$  of three input variables  $a, b$ , and  $c$ . The corresponding equations are

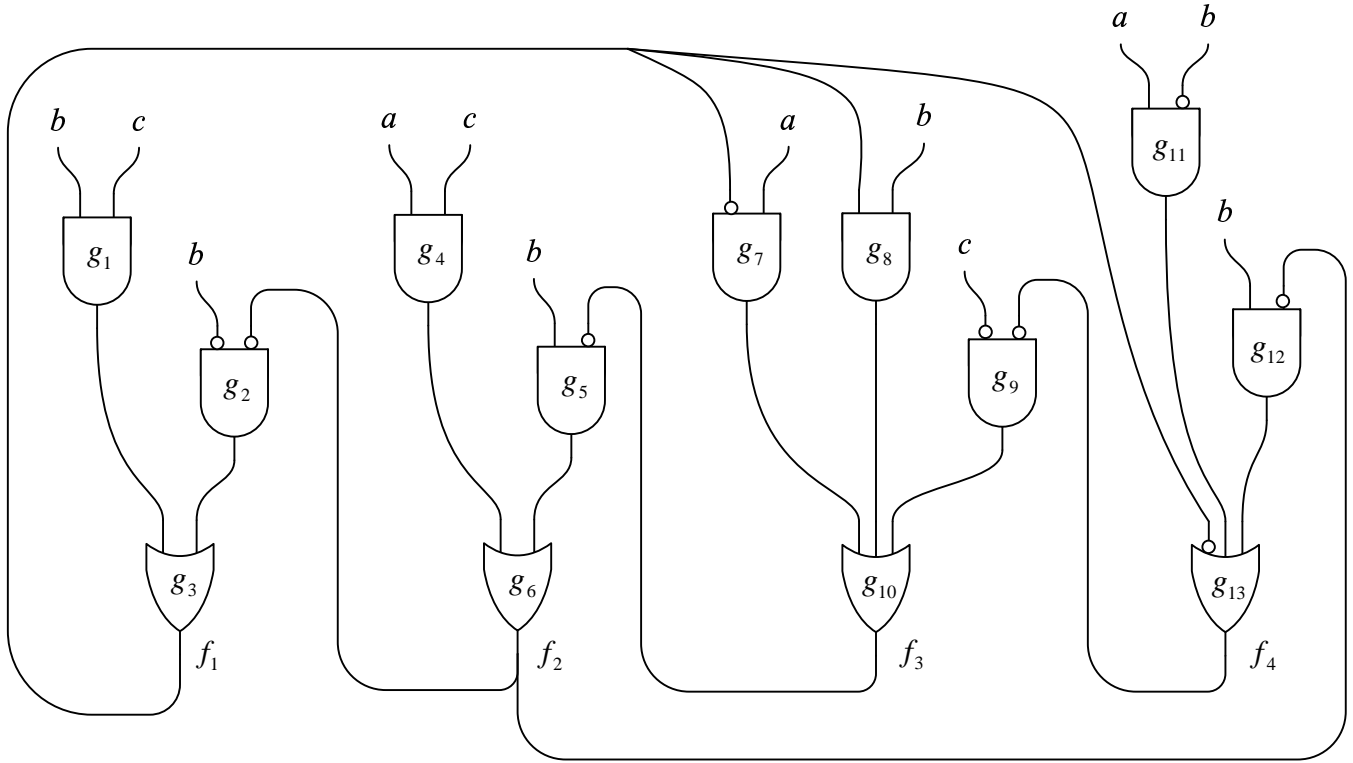
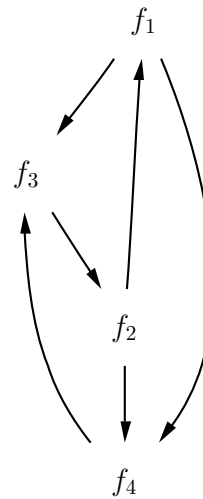


Figure 4.7: A cyclic combinational circuit.

$$\begin{aligned}
 f_1 &= bc + \bar{b} \bar{f}_2 \\
 f_2 &= ac + b \bar{f}_3 \\
 f_3 &= a \bar{f}_1 + b f_1 + \bar{c} \bar{f}_4 \\
 f_4 &= a \bar{b} + \bar{f}_1 + b \bar{f}_2
 \end{aligned}$$



Note that there are cyclic dependencies:  $f_1$  depends on  $f_2$ ;  $f_2$  depends on  $f_3$ ;  $f_3$  depends on  $f_1$  and  $f_4$ ; and  $f_4$  depends on  $f_1$  and  $f_2$ . Nevertheless, this circuit is combinational with delay 8.

We do not trace through the analysis this time. The table in Figure 4.8 summarizes the results. It gives the characteristic sets  $C_i^{(0)}$  and  $C_i^{(1)}$  for the output gates.  $\square$

$g_3$		
$[b\bar{c},$	$bc$	$]_2$
$[a\bar{b}c + b\bar{c},$	$c(b + \bar{a}) + \bar{b}\bar{c}$	$]_4$
$g_6$		
$[\bar{b}(\bar{c} + \bar{a}),$	$ac$	$]_2$
$[\bar{a}(c + \bar{b}) + a\bar{c},$	$ac$	$]_6$
$[\bar{a}(c + \bar{b}) + a\bar{c},$	$\bar{a}b\bar{c} + ac$	$]_7$
$g_{10}$		
$[\bar{a}\bar{b}c,$	$0$	$]_2$
$[\bar{a}\bar{b}c,$	$b(c + a)$	$]_4$
$[\bar{a}(\bar{b}\bar{c} + \bar{b}c),$	$b(c + a)$	$]_5$
$[\bar{a}\bar{b}c + \bar{c}(a\bar{b} + \bar{a}b),$	$c(b + a) + ab$	$]_6$
$[\bar{a}\bar{b}c + \bar{c}(a\bar{b} + \bar{a}b),$	$\bar{a}\bar{b}\bar{c} + c(b + a) + ab$	$]_7$
$g_{13}$		
$[0,$	$a\bar{b}$	$]_2$
$[0,$	$b\bar{c} + a\bar{b}$	$]_3$
$[abc,$	$b\bar{c} + a\bar{b}$	$]_4$
$[abc + a\bar{b},$	$b\bar{c} + a\bar{b}$	$]_5$
$[abc + a\bar{b},$	$b(\bar{c} + \bar{a}) + a\bar{b}$	$]_8$

Figure 4.8: Characteristic sets  $[C_i^{(0)}, C_i^{(1)}]_j$  for the circuit of Figure 4.7, for gates  $g_i$ ,  $i = 3, 6, 10, 13$ , at time intervals  $j = 2, \dots, 8$ .

Timing analysis with such an idealized model is transparent. However, the devil is in the details – and with realistic timing models there are *many* detailed aspects to consider. Nevertheless, we conclude that, at least in a conceptual sense, the analysis of cyclic circuits is no more complicated than that of acyclic circuits. We can perform this task efficiently through symbolic event propagation, within the ternary framework.

## Chapter 5

# Synthesis

*To invent, all you need is a pile of junk and a good imagination.* – Thomas A. Edison (1847–1931)

Logic synthesis is the task of designing circuits at the level of gates and wires to meet a specification. As a research area, it is at once mature and wide-open. It is mature in the sense that great intellectual effort has been expended in the development of sophisticated tools with a dizzying array of heuristics; it is wide-open in the sense that even the best available tools produce results that everyone admits are probably far from optimal.

For combinational logic, synthesis begins with a set of *target functions*, each expressed in terms of the primary input variables. This is sometimes called a *register-transfer* level specification, since it specifies what must be computed by blocks of combinational logic situated between memory registers, as shown in Figure 5.1.

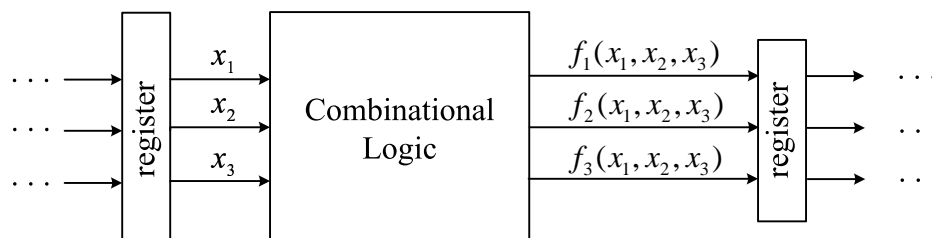


Figure 5.1: A register-transfer level specification.

Suppose that we are given target functions, one for each gate, and asked to verify that the circuit implements these functions. The first step is to check that the specification is *consistent*. By this we mean: for each gate – viewed in isolation – if we apply the specified target functions at its inputs, do we obtain the specified target function at its output? This is illustrated in Figure 5.2. Here we check whether a gate  $g$  computes the target function  $f_3$  at its output given the target functions  $f_1$  and  $f_2$  at its inputs.

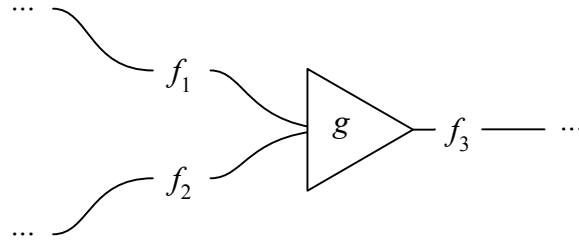


Figure 5.2: Consistent specifications.

In an acyclic circuit, consistency is sufficient to guarantee correctness. In a cyclic circuit, however, we may have a consistent specification, and yet the computation may be spurious. Consider the circuit in Figure 5.3. Although absurd, the circuit is consistent for an arbitrary function  $f_1$  and its complement  $\bar{f}_1$ .

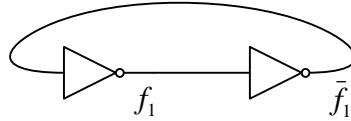


Figure 5.3: A *consistent*, yet spurious circuit.

We will use the following target functions to illustrate the concepts in this section:

**Example 5.1** Target Functions

$$f_1 = \bar{x}_1 x_2 \bar{x}_3 + x_1 \bar{x}_2 \bar{x}_3 + x_1 \bar{x}_2 x_3 + \bar{x}_1 \bar{x}_2 x_3,$$

$$f_2 = \bar{x}_1 \bar{x}_2 \bar{x}_3 + x_1 x_2 x_3 + x_1 x_2 \bar{x}_3 + x_1 \bar{x}_2 x_3,$$

$$f_3 = \bar{x}_1 x_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 \bar{x}_3 + x_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3.$$



The end product of synthesis is a set of simplified expressions that can be translated into a network of gates. The goals may vary greatly depending on the technology and the setting. Two obvious criteria for optimization are

- **Area**, as a measure of the number of transistors in the final silicon implementation. This correlates to the *cost* of manufacturing the circuit.
- **Delay**, as a measure of the time it takes for the circuit to produce outputs, given stable inputs. This determines the *performance* of the circuit – in synchronous designs, the clock speed achievable.

Our cost measure for area is the number of the *literals* in the algebraic expressions – that is to say, the number of appearances of variables, without regard to negations. For instance, in the expressions above, each function has a cost of 12, for a total cost of 36. The literal count seems to correlate well with the silicon area of the final implementation [6].

Our cost measure for delay is the propagation time from inputs to outputs. In most examples, we assume a *unit delay* model for the gates. Accordingly, the propagation delay is simply the number of hops along the longest sensitized path.

## 5.1 Logic Minimization

The first step in logic design is to simplify the Boolean expressions individually, if possible. In the *sum-of-products* (S-of-P) form, a Boolean expression is formulated as the OR (disjunction) of AND (conjunctive) terms. Every student of logic design learns the Karnaugh Map method, and perhaps the tabular Quine-McCluskey algorithm for finding the minimal sum-of-products expression of a logic function [16], [27], [31]. Minimal in this context means with the fewest conjunctive terms, and the fewest literals per conjunctive term. Using such a technique, the functions in Example 5.1

simplify to

$$\begin{aligned} f_1 &= x_1\bar{x}_2 + \bar{x}_1x_2\bar{x}_3 + \bar{x}_2x_3, \\ f_2 &= x_1x_2 + x_1x_3 + \bar{x}_1\bar{x}_2\bar{x}_3, \\ f_3 &= \bar{x}_1\bar{x}_3 + \bar{x}_1\bar{x}_2 + \bar{x}_2\bar{x}_3, \end{aligned}$$

with a total cost of 20.

For some technologies, such as programmable logic arrays (PLAs), so-called “two-level” designs are required. In two-level designs, the functions are expressed in S-of-P form, but conjunctive terms may be *shared* among the expressions. For Example 5.1, if we select the terms,

$$\begin{aligned} c_1 &= x_1\bar{x}_2x_3, \\ c_2 &= x_1\bar{x}_2\bar{x}_3, \\ c_3 &= \bar{x}_1\bar{x}_2x_3, \\ c_4 &= \bar{x}_1x_2\bar{x}_3, \\ c_5 &= \bar{x}_1\bar{x}_2\bar{x}_3, \\ c_6 &= x_1x_2, \end{aligned}$$

we obtain the expressions

$$\begin{aligned} f_1 &= c_1 + c_2 + c_3 + c_4, \\ f_2 &= c_1 + c_5 + c_6, \\ f_3 &= c_2 + c_3 + c_4 + c_5, \end{aligned}$$

with a total cost of 17. A two-level design can be viewed as a layer of AND gates on top of a layer of OR gates, as shown in Figure 5.4.

As a research area, two-level minimization is evidently quite mature. Sophisticated algorithms – some exact, some heuristic – can routinely find the optimal or near

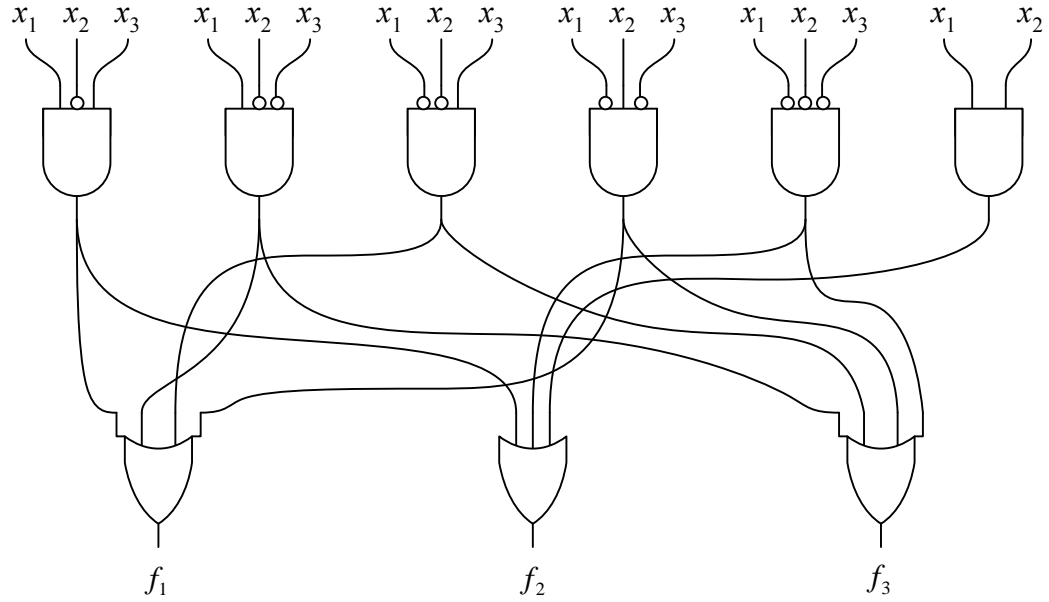


Figure 5.4: A two-level implementation for the target functions of Example 5.1.

optimal solution for networks of hundreds of functions, each depending on hundreds of variables [4].

## 5.2 Multi-Level Logic

In multi-level designs, an arbitrary structure is permitted. Many of the examples presented thus are, in fact, multi-level. Since there is greater freedom of structure, the search space of potential solutions is correspondingly much larger. In fact, for anything more complex than a single function of 5 input variables, an exhaustive search of all solutions is generally intractable.

In spite of the daunting complexity, practitioners can claim considerable success with heuristic optimization methods. For a survey of the topic, see [6]. Although there have been innumerable approaches suggested, the most widely adopted paradigm is that championed by the University of California at Berkeley group. It consists of an iterative application of minimization, decomposition, and restructuring operations [5].

In a multi-level setting, expressions are often manipulated in *factored form*. Alge-

braically, a factored form is a parenthesized expression of OR and AND operations. For Example 5.1, factored forms of the functions are

$$\begin{aligned} f_1 &= \bar{x}_1 x_2 \bar{x}_3 + \bar{x}_2 (x_1 + x_3), \\ f_2 &= \bar{x}_1 \bar{x}_2 \bar{x}_3 + x_1 (x_2 + x_3), \\ f_3 &= \bar{x}_3 (\bar{x}_1 + \bar{x}_2) + \bar{x}_1 \bar{x}_2, \end{aligned}$$

with a total cost of 17.

The factored forms of a collection of Boolean functions  $\{f_1, \dots, f_n\}$  are in one-to-one correspondence with multi-level structures of gates. For the example above, the circuit is shown in Figure 5.5. In general, a factored form is not unique. Determining the optimal factorization is a difficult (NP-complete) problem. Nevertheless, algorithms such as those in the Berkeley SIS package [38] work efficiently in practice. Although we use the factored form in our examples, we do not discuss the algorithms or the implementation. (The interested reader is referred to [6].)

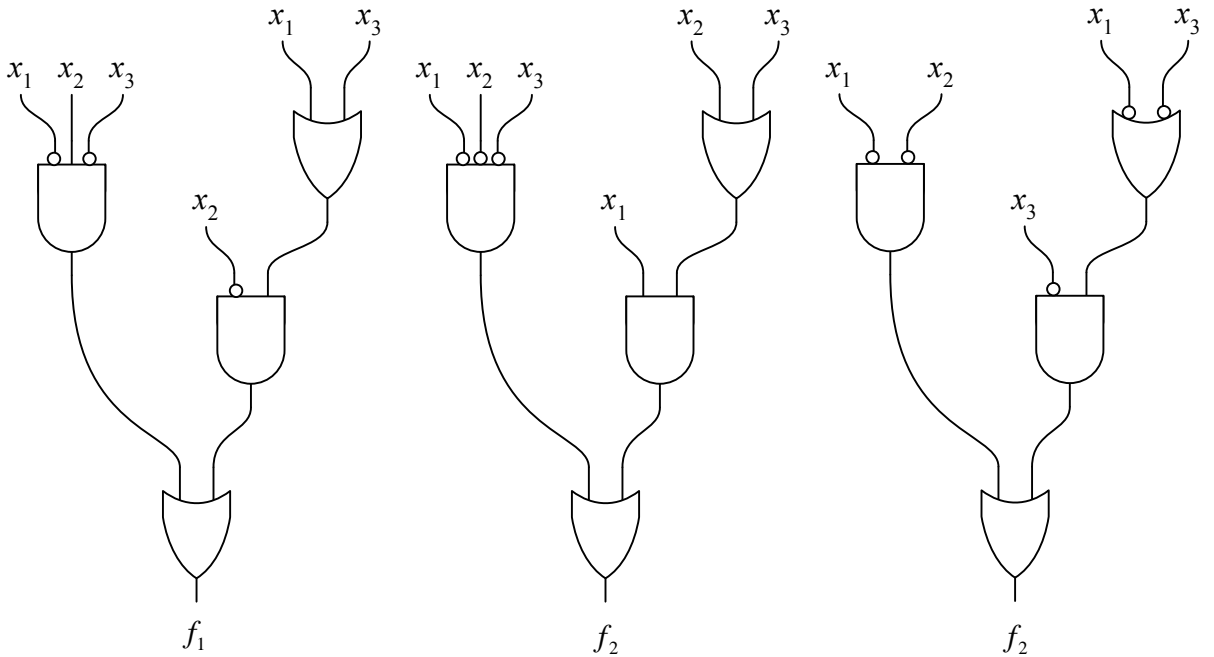


Figure 5.5: A factored form for the target functions in Example 5.1.

An important operation in multi-level synthesis is **substitution** (also sometimes

called “re-substitution”). In the substitution phase, node functions are expressed, or re-expressed, in terms of other node functions as well as of the original inputs. For instance, with the target functions in Example 5.1, we can express  $f_1$  in terms of  $f_2$  and  $f_3$ ,

$$f_1 = \bar{x}_2 x_3 + \bar{f}_2 f_3.$$

The substitution and minimization steps are often performed jointly. The algorithms for this task form the corner-stone of multi-level synthesis, and are key to our exposition; however, it is beyond the scope of this dissertation to delve into the details. The reader is referred to [6]. In our implementation, we use the `simplify` command of the Berkeley SIS package [38].

### 5.3 Substitutional Orderings

In general, for a given collection of target functions we have a choice of substitutions that can be performed. For each target function, call the set of other target functions that it is expressed in terms of its **substitutional set**. Different substitutional sets yield alternative expression of varying cost.

With the functions in Example 5.1, substituting  $f_3$  into  $f_1$ , we obtain

$$f_1 = f_3(x_1 + x_2) + \bar{x}_2 x_3.$$

Substituting  $f_2$  and  $f_3$  into  $f_1$ , we obtain

$$f_1 = \bar{x}_2 x_3 + \bar{f}_2 f_3.$$

Substituting  $f_3$  into  $f_2$ , we obtain

$$f_2 = \bar{x}_1 \bar{x}_2 \bar{x}_3 + x_1 \bar{f}_3.$$

Substituting  $f_1$  and  $f_3$  into  $f_2$ , we obtain

$$f_2 = \bar{f}_1 \bar{x}_3 + \bar{f}_3 x_1.$$

Substituting  $f_1$  into  $f_3$ , we obtain

$$f_3 = f_1 \bar{x}_1 + \bar{x}_2 \bar{x}_3.$$

Finally, substituting  $f_1$  and  $f_2$  into  $f_3$ , we obtain

$$f_3 = f_1 \bar{f}_2 + \bar{x}_2 \bar{x}_3.$$

Other combinations of substitutions are not helpful.

In existing methodologies, an ordering is enforced among the functions in the substitution phase to ensure that no cycles occur. This choice can influence the cost of the solution. With the ordering shown on the right in Figure 5.6, substitution yields the network shown on the left with a cost of 14. The corresponding circuit is shown in Figure 5.7.

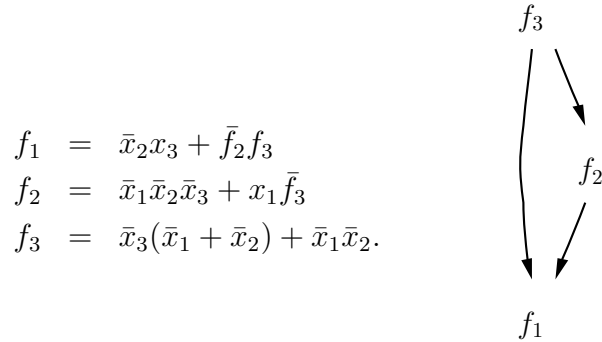


Figure 5.6: Acyclic substitution order.

Enforcing an ordering is limiting since functions near the top cannot be expressed in terms of very many others (the one at the very top cannot be expressed in terms of *any* others). Dropping this restriction can lower the cost. However, cyclic substitutions can result in spurious computation.

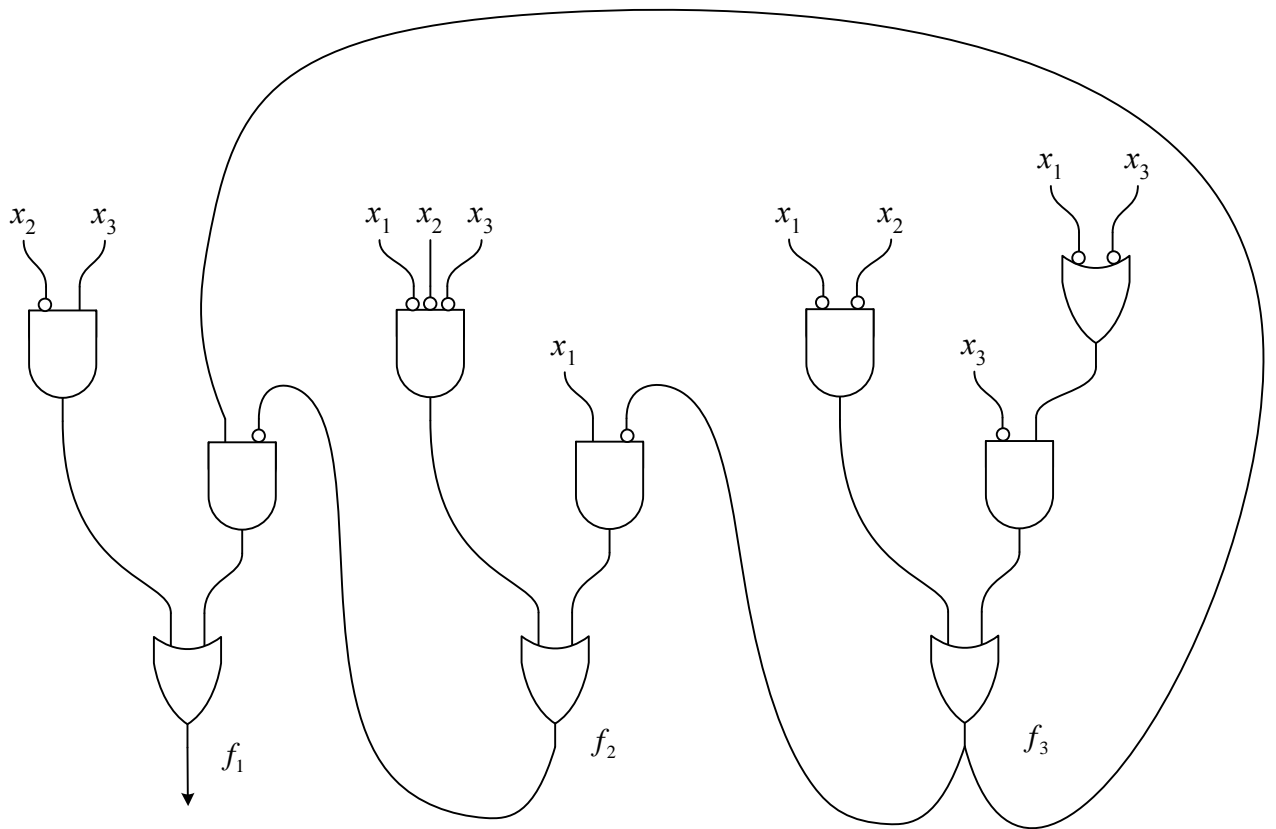


Figure 5.7: Implementation of the acyclic solution in Figure 5.6.

Similarly, for the target functions of Example 5.1, if we allow every function to be substituted into every other, then we obtain the network shown on the left in Figure 5.8, with a cost of 12. This network is cyclic, with the dependency shown on the right.

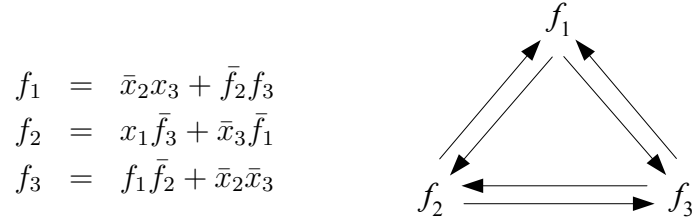


Figure 5.8: A network obtained with unordered substitutions (it is *not* combinational).

Unfortunately, analysis according to the techniques in Chapter 4 tells us that this solution is *not* combinational. Indeed, setting  $x_1 = 1$ ,  $x_2 = 1$  and  $x_3 = 1$  yields

$$\begin{aligned} f_1 &= \bar{f}_2 f_3, \\ f_2 &= \bar{f}_3 + \bar{f}_1, \\ f_3 &= f_1 \bar{f}_2, \end{aligned}$$

which results in indeterminate values. The key is to select a substitutional order that minimizes the cost, and yet results in a combinational solution. Given a candidate network, we analyze it using the analysis techniques described in Chapter 4 to decide whether it is valid.

## 5.4 Branch-and-Bound Algorithms

For each node, we expect the lowest cost expression to be obtained with the full substitutional set (i.e., all other node functions) and the highest cost expression to be obtained with the empty set. For a network with a non-trivial number of nodes, a brute-force exhaustive search is evidently intractable. With  $n$  nodes, there are  $2^{n-1}$  substitutional sets for each node, for a total of  $n \cdot 2^{n-1}$  possibilities.

We describe a branch-and-bound approach, as well as various heuristics.



### 5.4.1 The “Break-Down” Approach

With this approach, the search is performed *outside* the space of combinational solutions. A branch terminates when it hits a combinational solution. The search begins with a densely connected network, such as that in Figure 5.8. This initial branch provides a lower bound on the cost. As edges are excluded in the branch-and-bound process, the cost of the network remains unchanged or increases. (Again, since the substitution step is heuristic, this may not be strictly true.)

#### Algorithm 5.1 Break-Down Synthesis

1. Analyze the current branch for combinationality. If it is combinational, add it to a solution list. If it is not, select a set of edges to exclude based on the analysis.
2. For each edge in the set, create a new branch. Create a node expression, excluding the incident node from the substitutional set. If the cost of the new branch equals or exceeds that of a solution already found, kill the branch.
3. Mark the current branch as “explored”.
4. Set the current branch to be the lowest cost unexplored branch.
5. Repeat steps 1 – 4 until the cost goal is met, or until the cost of all unexplored branches exceeds that of a solution.

□

The branching point of the algorithm occurs in Step 1. Symbolic analysis informs us whether the current branch is combinational or not. If not, the analysis suggests which new branches to form. Consider the example in Figure 5.9. Analysis begins by dividing the network into strongly connected components. We focus on the non-combinational components, selecting edges from them to exclude. Suppose that component II is combinational, while component I is not. Accordingly, we form new branches with the edges  $e_1$ ,  $e_2$  and  $e_3$  excluded. We preserve edges that do not belong

to strongly connected components, as such dependencies do not affect combinationality. Thus, we do not cut the edges  $e_{12}$ ,  $e_{13}$  and  $e_{14}$  for any of the newly formed branches.

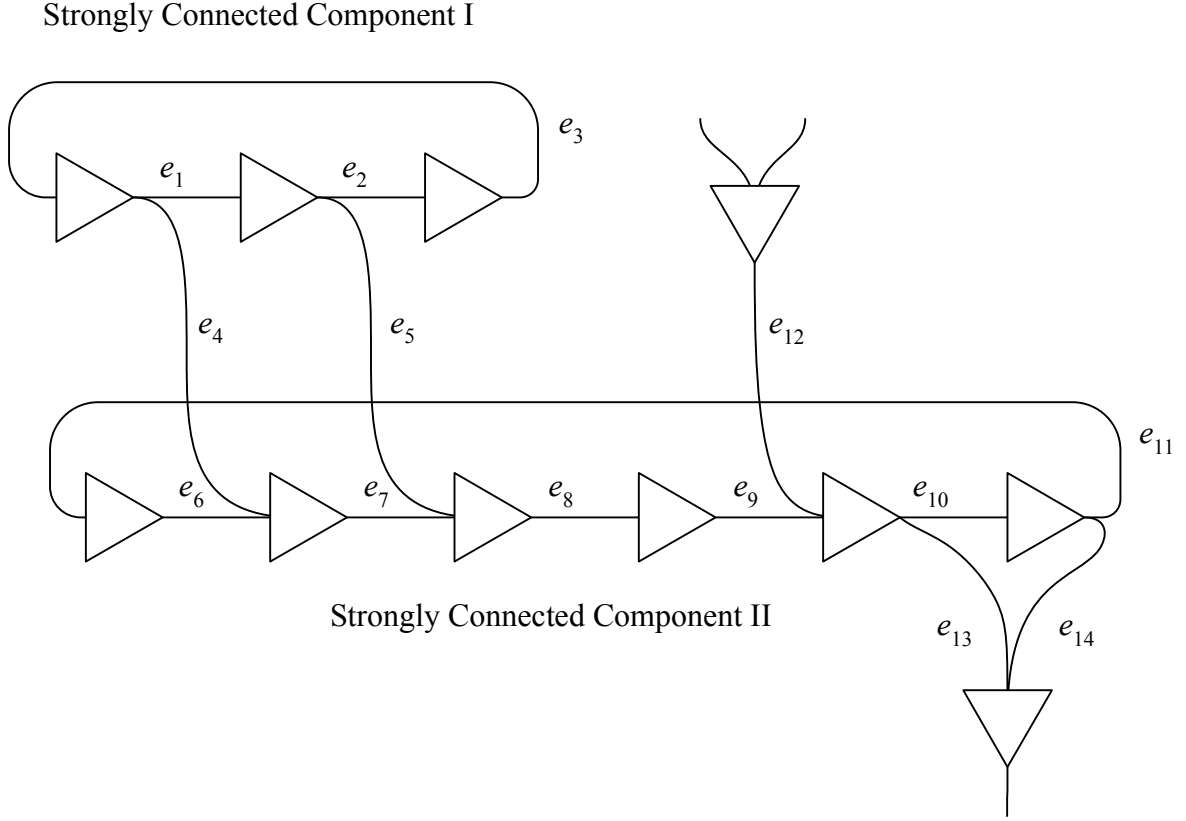


Figure 5.9: Example illustrating edge selection for “break-down” search strategy.

A sketch of the algorithm is shown in Figure 5.12. (This is *not* a complete trace of the search; only the trajectory to a solution is shown.) For the target functions of Example 5.1, the algorithm yields a cyclic combinational solution with a cost of 13. The expressions are shown in Figure 5.10, and the corresponding circuit in Figure 5.11.

We argue that Algorithm 5.1 produces the *optimal* solution. Indeed, as the algorithm proceeds, the cardinality of the substitutional sets decrease along each branch. This produces networks with monotonically non-decreasing cost. The algorithm always explores the lowest cost open branch first. Therefore, when the algorithm termi-

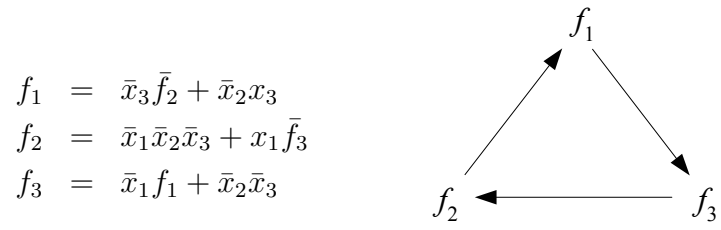


Figure 5.10: Cyclic solution for target functions of Example 5.1.

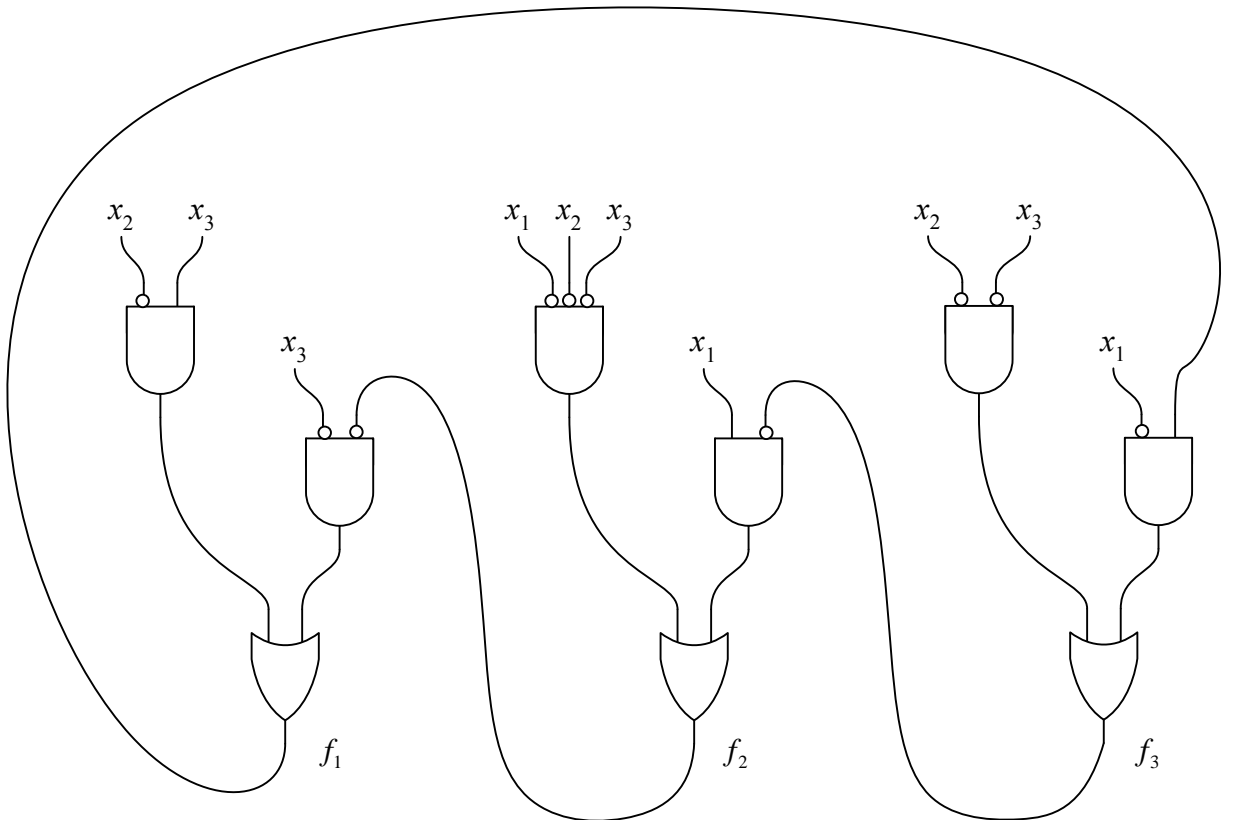


Figure 5.11: Implementation of the cyclic solution in Figure 5.10.

nates on a combinational solution, we can assert that this is the lowest cost solution possible.

This bold claim must be tempered with an important qualification. The substitute/minimize operations that we use are based on heuristics, and themselves provide no guarantee of optimality. The quality of our solutions is only as good as that of these operations. For target functions with large support sets, it would appear that the substitute/minimize step often produces results that are very far from optimal.

Many ideas immediately suggest themselves for expediting the search heuristically. We can prioritize progress slightly, at the expense of quality, for instance by choosing branches with fewer gates in strongly connected components. Also, we can limit the density of edges a priori, or prune the set of edges before creating new branches.

### 5.4.2 The “Build-Up” Approach

With this approach, the search is performed *inside* the space of combinational solutions. A branch terminates when it hits a non-combinational solution. The search begins with an empty edge set. Edges are added as the substitutional sets of nodes are augmented. As edges are included, the cost of the network remains the same or decreases.

#### Algorithm 5.2 Build-Up Synthesis

1. Analyze the current branch for combinationality. If it is not combinational discard it. If it is combinational, select a set of edges to include based on the analysis.
2. For each edge in the set, create a new branch. Create a new node expression, including the incident node from the substitution set.
3. Mark the current branch as “explored.”
4. Set the current branch to be the lowest cost unexplored branch.
5. Repeat steps 1 – 4 until the cost goal is met.

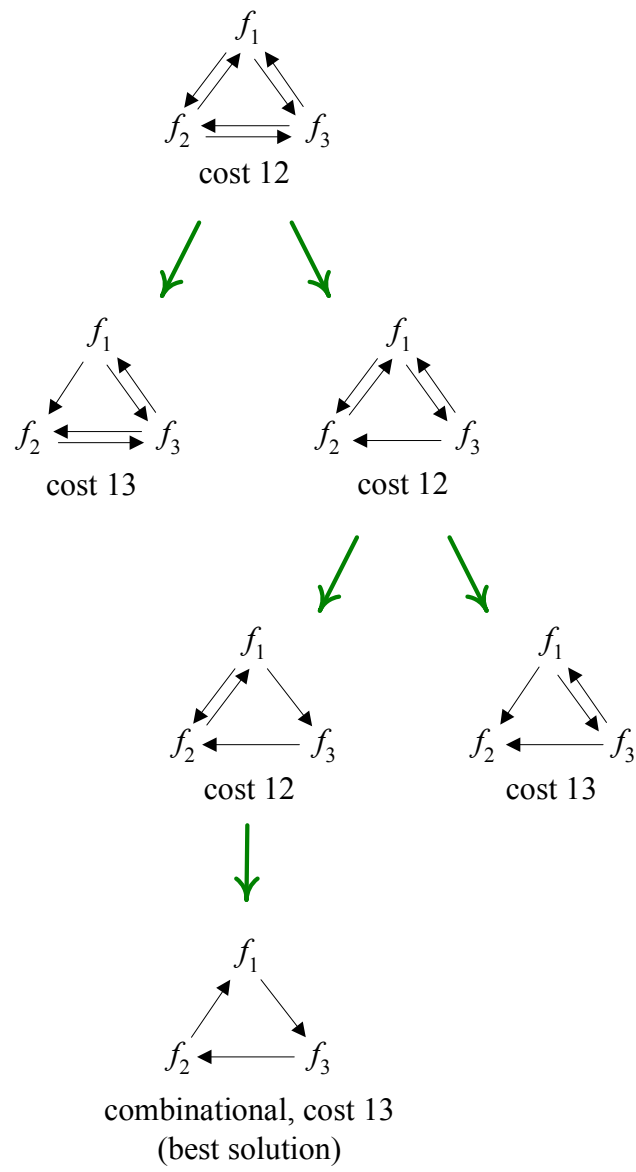


Figure 5.12: "Break-down" search strategy.

□

A sketch of the algorithm is shown in Figure 5.13. (Again, *not* a complete trace.) For the target functions of Example 5.1, the algorithm yields the same cyclic combinational solution, that shown in Figure 5.10.

With this method, we cannot prune branches through a lower-bound analysis. However, exploring within the space of combinational solutions ensures that incrementally better solutions are found as the computation proceeds. As an alternative starting point, we can use an existing acyclic solution. Adding edges reduces the cost, while potentially introducing cycles.

## 5.5 Example: 7-Segment Decoder

As a final example, we illustrate the design of the 7-Segment Decoder circuit shown in Figure 1.15 of Chapter 1. As described there, the inputs to this circuit are four bits,  $x_0, x_1, x_2, x_3$ , specifying a number from 0 to 9. The outputs are 7 bits,  $a, b, c, d, e, f, g$ , specifying which segments to light up in a 7-segment LED display in order to form the image of this number.

Our goal is to design a circuit that implements the following functions:

$$\begin{aligned}
 a &= \bar{x}_0 x_2 \bar{x}_3 + \bar{x}_1 (\bar{x}_2 (x_3 + \bar{x}_0) + x_2 \bar{x}_3) \\
 b &= \bar{x}_0 (x_1 \bar{x}_3 + \bar{x}_1 \bar{x}_2) \\
 c &= \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_3 (x_0 (x_2 + x_1) + \bar{x}_0 \bar{x}_2) \\
 d &= \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_3 (x_2 (\bar{x}_1 + \bar{x}_0) + x_1 \bar{x}_2) \\
 e &= \bar{x}_0 \bar{x}_1 \bar{x}_2 + \bar{x}_3 (x_0 \bar{x}_1 x_2 + x_1 (\bar{x}_2 + \bar{x}_0)) \\
 f &= \bar{x}_3 (\bar{x}_0 \bar{x}_1 + x_0 x_1 + \bar{x}_2) + \bar{x}_1 \bar{x}_2 \\
 g &= \bar{x}_3 (x_2 + x_0) + \bar{x}_1 \bar{x}_2.
 \end{aligned}$$

With the break-down approach, we begin with the network shown in Figure 5.14, with the ordering illustrated. This network was obtained by permitting the substi-

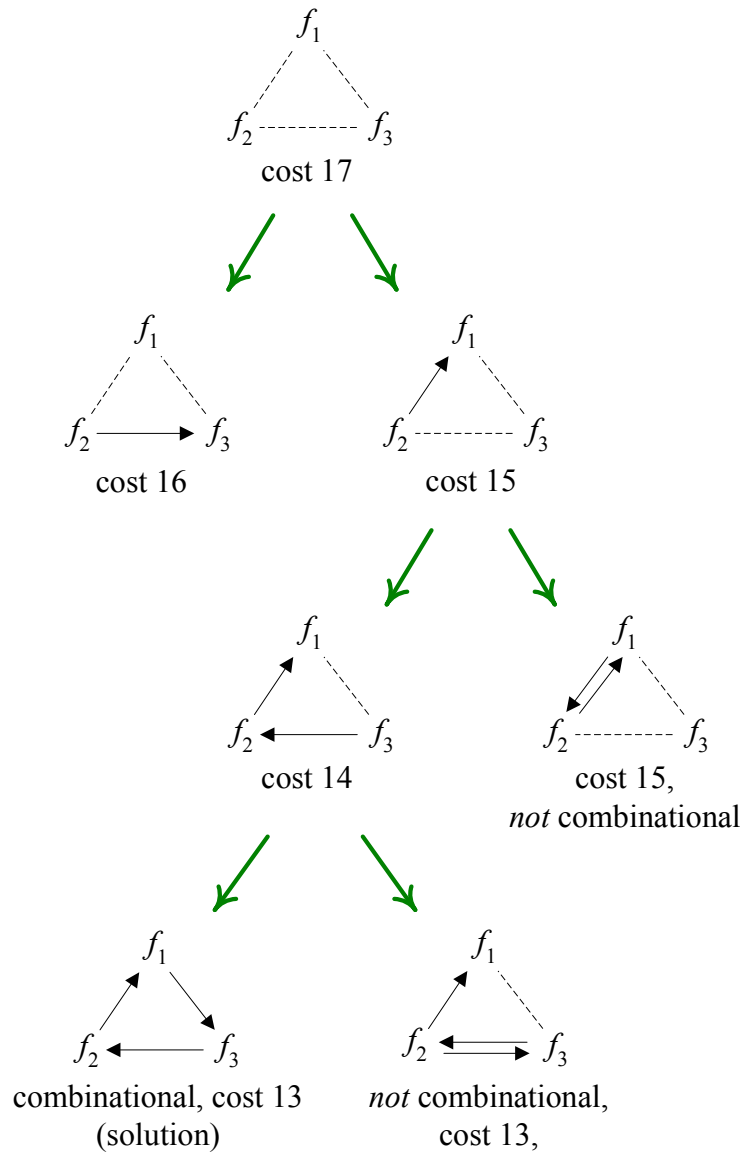


Figure 5.13: “Build-up” search strategy.

tution of every function into every other. The result is a network with cost 30, as measured by the literal count. Not surprisingly, analysis tells us that this circuit is *not* combinational.

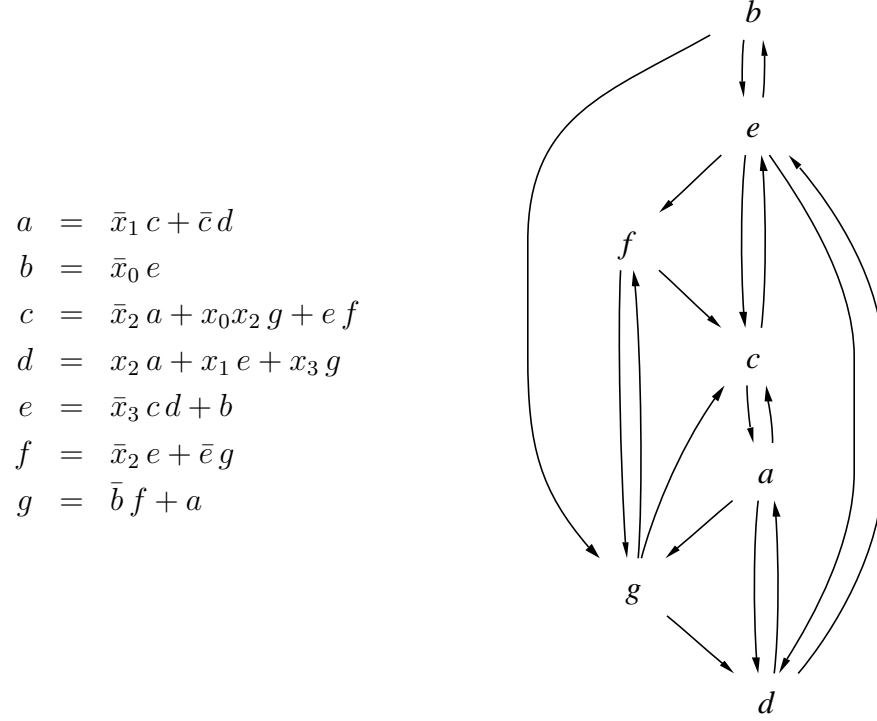


Figure 5.14: An invalid cyclic network for the example in Figure 1.15.

Through the branch-and-bound process, the algorithms prunes the following dependencies:

$$d \rightarrow a$$

$$a \rightarrow c, g \rightarrow c, f \rightarrow c$$

$$g \rightarrow d$$

$$c \rightarrow e, b \rightarrow e, d \rightarrow e$$

$$e \rightarrow f, g \rightarrow f$$

$$f \rightarrow g$$



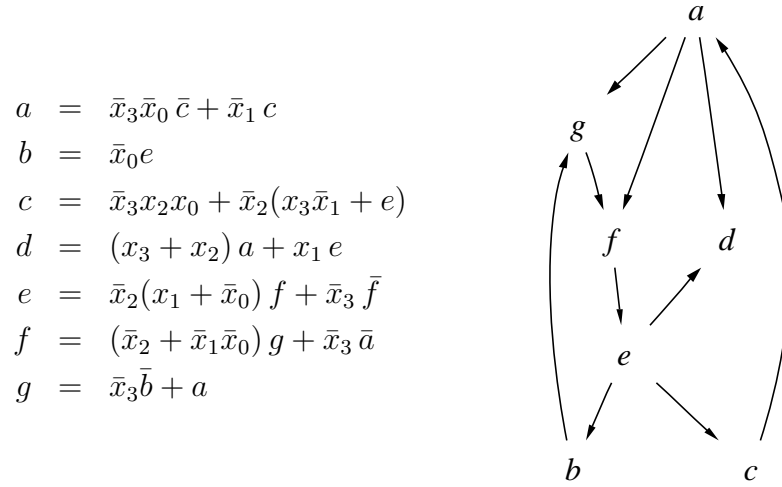


Figure 5.15: A valid cyclic network for the example in Figure 1.15.

Paradoxically, pruning these dependencies introduces new ones:

$$f \rightarrow e$$

$$a \rightarrow f$$

Here  $f$  only becomes helpful in expressing  $e$  when the dependencies on  $b, c$  and  $d$  are excluded. Similarly,  $a$  only becomes helpful in expressing  $f$  when the dependency on  $e$  is excluded.

The result is the network shown in Figure 5.15, with the ordering illustrated. Note that this network contains cyclic dependencies; in fact, all the functions except  $d$  form a strongly connected component. However, analysis tells us that this network is combinational. Note that it has cost 34. In contrast, using existing methods, we would obtain an acyclic network with a higher cost of 37.

In Appendix B, we present synthesis results obtained with our program CYCLIFY. We have run trials on a range of randomly generated examples and benchmark circuits. In Section B-1 we present results for optimizations of area at the *network level*, that is to say, in terms of functional dependencies only. In Section B-2 we present

area optimizations at the gate level, that is to say, for circuits decomposed into primitive gates (2-input NAND and NOR gates). Nearly all trials produced significant optimizations, with improvements of up to 30% in the area. We note that solutions for many of the circuits have dense strongly connected components. For example, the dependency graph for the cyclic solution of one of the benchmark circuits, called **exp**, is shown in Figure 5.16. (The circuit performs binary exponentiation.)

In this chapter, we have focused on area as our cost measure. Indeed, the case for using feedback to optimize area is the most compelling. However, our branch-and-bound algorithm can readily be adapted for optimization according to other metrics, provided that analysis techniques exist to measure these. The symbolic algorithm in Chapter 4 provides not only functional validation, but also timing information. Accordingly, we can use our synthesis strategy to optimize for *delay*. While this is a topic of ongoing research, we present preliminary results in Appendix B.

We incorporated a sliding scale for the relative weight of the two criteria, area and delay, in the cost metric for the branch-and-bound algorithm. In Section B-3, we present synthesis results for optimizations at the gate level, with area weighted at one-third, and delay weighted at two-thirds. Again, nearly all designs were optimized significantly, with simultaneous improvements of up to 10% in the area and 25% in the delay.

The most salient result to report, and one of the main messages of this dissertation, is that cyclic solutions are not a rarity; they can readily be found for nearly all of the circuits that we encountered. Our search algorithms, while heuristic in nature, can effectively tackle circuits of sizes that are of practical importance. The synthesis results are sufficiently convincing to warrant a bold claim: practitioners should explore feedback optimizations for all types of combinational circuits, ranging from the examples that we find in textbooks, to the benchmark circuits referenced by the research community, to real-world circuits. Perhaps the ALU of the next Intel Pentium chip will be designed with feedback...

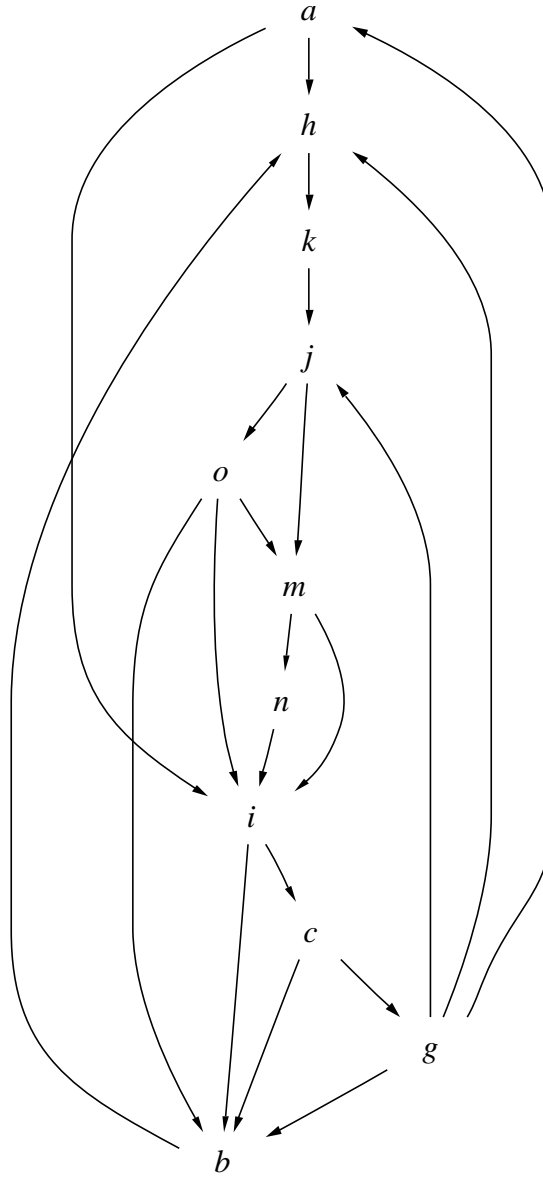


Figure 5.16: Topology of the cyclic solution for the benchmark circuit **exp**, with 8 inputs, 18 outputs, and cost 262. (Only nodes in the strongly connected component are shown.)

# Chapter 6

## Discussion

*A person with a new idea is a crank until the idea succeeds.*

– Mark Twain (1835–1910)

In theoretical computer science, one of the main goals is to prove lower bounds on the resources (e.g., time and/or space) required for computation. A combinational circuits is often postulated as a representative model. In this context, a circuit is viewed not as an abstract device, such as a Turing machine, but as a bit-level implementation of a computational procedure – in a sense, the most basic way that one can compute something [22]. A lower bound on the circuit size is taken as a true measure of the computational requirements of a problem. For instance, lower bounds on circuit size have been used to justify the security of cryptographic algorithms [36].

Complexity theorists invariably define a combinational circuit as a directed acyclic graph (DAG); see, for instance, the textbook by Papadimitriou [23]. It is conceivable that some of the proofs of lower bounds on circuit size depend on this definition. We hope that this dissertation will help promulgate the view that a Boolean circuit is not necessarily a DAG; rather it is a directed graph that may have cycles, as long as it is combinational.

In spite of the synthesis results in Chapter 5, the contemplative reader might still ask: How much do we really gain with feedback? What is the true potential of this idea? In a sense, feedback is a boundary optimization. In an acyclic circuit, there

is a topological ordering among the output functions. A function at the top of this ordering does not depend on any other. A function at the bottom of the ordering depends on all the others. On average, each function depends on about half the others.

In a cyclic circuit, every gate producing an output function may depend on all the others. Thus, reasoning in a very loose manner, feedback yields at most a 50% increase in the overlap of the computational resources. Viewed this way, the example in Chapter 3 – a cyclic circuit that is one-half the size of the smallest equivalent acyclic circuit – may be representative of the best improvement that can be achieved.

Feedback is clearly effective when implementing a collection of output functions. Could there be a benefit in implementing a single output function with feedback? The structure shown in Figure 6.1 is certainly plausible. The output function  $f$  is decomposed into sub-functions, and these are implemented in a cyclic configuration. Of course, we cannot use the fan-in lower bound to argue the optimality of such a cyclic circuit.

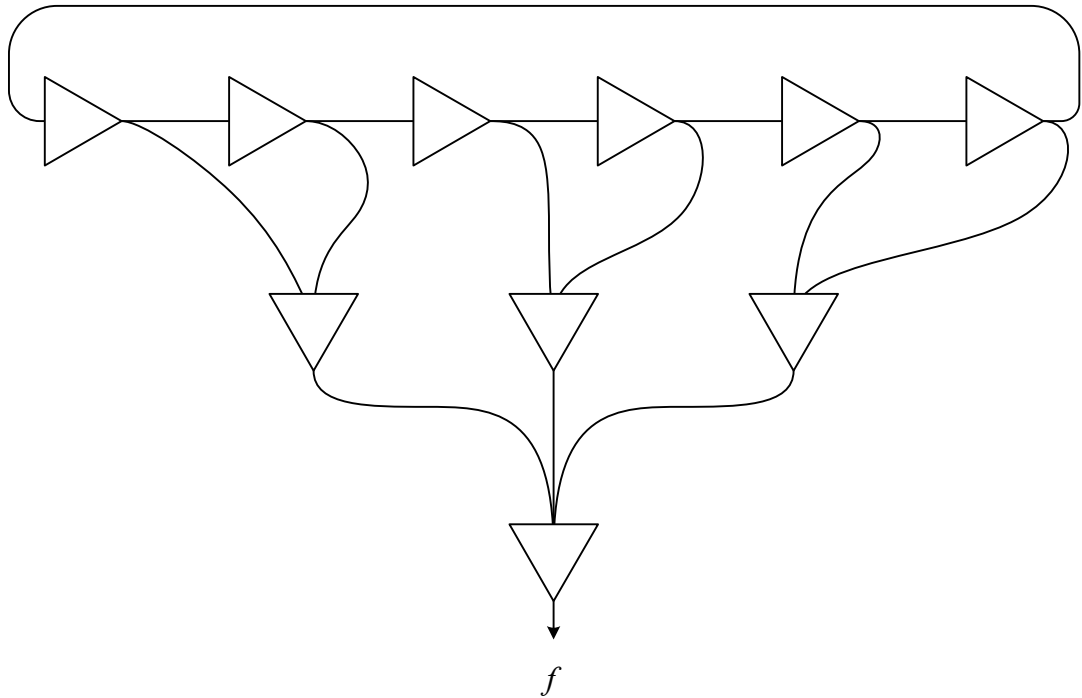


Figure 6.1: A cyclic circuit with a single output.

## 6.1 High-Level Design

In this dissertation, we discussed combinational logic design at the gate level. The circuit design process typically begins at higher, more abstract level in the form of a behavioral specification. The design is approached in a *hierarchical* fashion: a solution is given in terms of modules, initially viewed as “black boxes”. Each of these modules is further refined, and perhaps broken down into simpler modules. Finally, with modules of manageable size and complexity defined, a gate-level solution is synthesized. This is illustrated in Figure 6.2.

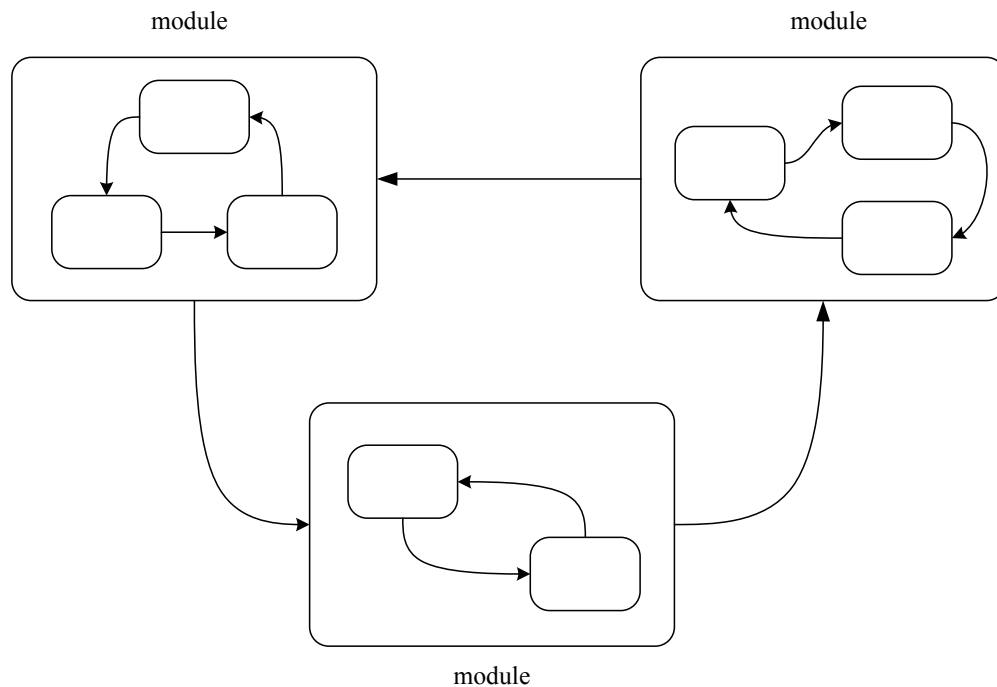


Figure 6.2: Cycles at various levels within a hierarchy.

Of course, internally the modules should be designed with feedback. What can we say about feedback between modules? Conceptually, the requirements are the same and the potential benefits just as compelling.

The methodology that we have proposed and the software tools that we have developed for the analysis and synthesis of cyclic circuits at the logic-level should obviate these concerns. The development of cyclic design strategies at the behavioral level, for instance within the framework of hardware description languages such as

Verilog and VHDL, is an ambitious but exciting direction of future research.

## 6.2 Data Structures

Throughout this dissertation, the discussion has pertained to circuits: physical devices, computing by means of electrical current. The concept of a combinational circuit with cycles required careful justification – indeed, electrical current running in a loop could render the computation invalid.

By definition, a combinational circuit is a device that accepts inputs and produces outputs, as shown in Figure 6.3. This definition could equally apply at a higher level

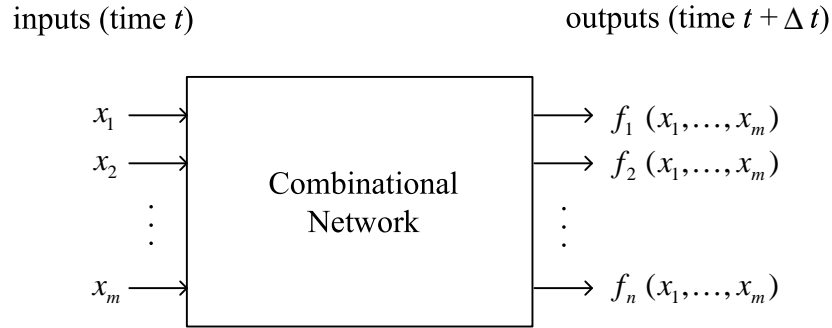


Figure 6.3: A combinational network.

of abstraction: that of a data structure. For instance, with a database the input consists of a search *key*, and the output consists of the corresponding *record*. Data structures are often represented as graphs. To extract data from the structure, we begin at the leaves, follow a path dictated by the search criteria, and terminate at a root node where the data is stored.

Consider binary decision diagrams (BDD), the preeminent data structure for circuit design, described in Chapter 4. An example is shown in Figure 6.4. BDDs are defined as acyclic graphs. Indeed, the accepted wisdom is that BDDs cannot contain cycles. If a BDD contained a cycle, then we might loop indefinitely when evaluating a function. Or maybe not?

Consider the example in Figure 6.5. To see that this BDD is valid, note that we drop out of the cycle at the first node if  $x_1 = 0$ , and we drop out of the cycle at the

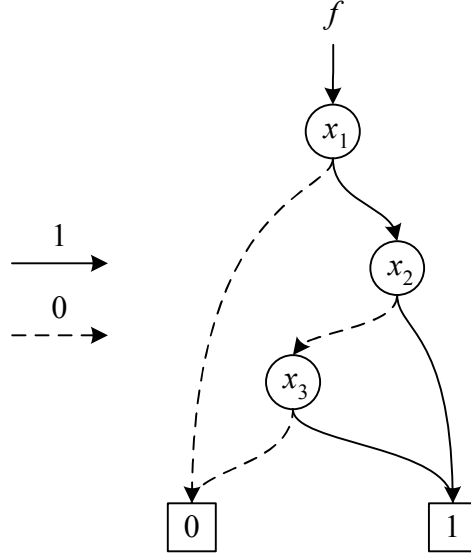


Figure 6.4: A binary decision diagram (BDD) implementing the function:  $f = x_1(x_2 + x_3)$ .

fourth node if  $x_1 = 1$ .

It may be shown that this cyclic BDD implements the the six functions

$$\begin{aligned}
 f_1 &= x_1(x_2 + x_3), \\
 f_2 &= x_2 + x_1x_3, \\
 f_3 &= x_3(x_1 + x_2), \\
 f_4 &= x_1 + x_2x_3, \\
 f_5 &= x_2(x_1 + x_3), \\
 f_6 &= x_3 + x_1x_2.
 \end{aligned}$$

Clearly we cannot implement the six distinct functions with a BDD consisting of fewer than six non-terminal nodes, so this BDD is optimal with respect to the number of nodes.

In an acyclic BDD implementing the same functions, there is an ordering among the variable nodes. A node at the bottom of the ordering must have both out-going edges leading to constant nodes. Consequently, it cannot be a source node for any of these functions, since each function depends on three variables. We conclude that any



acyclic BDD implementing these functions has more than six non-terminal nodes.

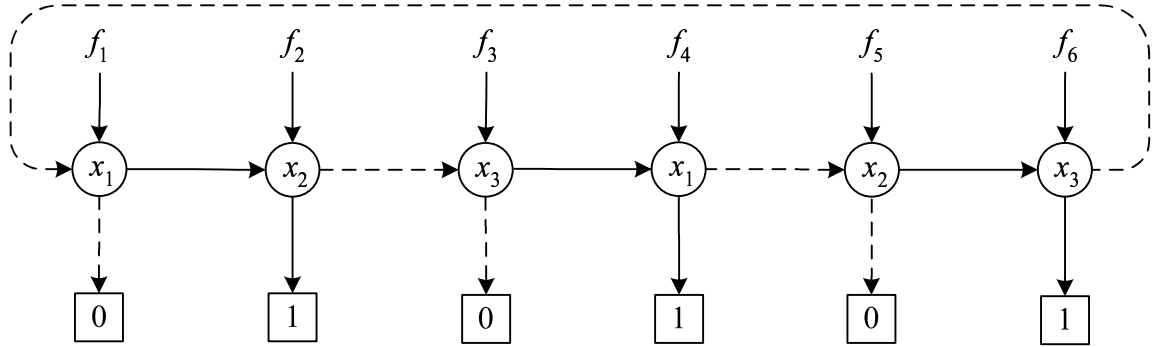


Figure 6.5: A cyclic BDD.

Thus, we see that the concept of a cyclic data structure is not only viable, but also offers the possibility for size improvements over acyclic structures. This example enlarges the scope of the concepts in this thesis: feedback is a general phenomenon in computation, whether at a physical level or an abstract level. Future research awaits...

## Appendix A: XNF Representation

Boolean functions are most commonly represented with the operations AND, OR and NOT. We refer to this representation as AON. For instance, the function specified by the truth table

$x_1$	$x_2$	$x_3$	$f$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

can be represented as

$$f = \bar{x}_1(\bar{x}_2 + \bar{x}_3).$$

Here addition denotes OR, multiplication denotes AND, and an overbar denotes NOT.

A less common, but equally general, representation is based on the AND and XOR operations. Together, these operations form the Galois Field  $GF(2)$ . In  $GF(2)$ , we may use all the arithmetic properties valid in the familiar fields, e.g., the fields of rational, real and complex numbers. In addition,  $GF(2)$  satisfies  $x \cdot x = x$  and  $x \oplus x = 0$ , where multiplication denotes AND, and  $\oplus$  denotes XOR. In this representation, the function above is

$$f = 1 \oplus x_1 \oplus x_2x_3 \oplus x_1x_2x_3.$$

Note that  $1 \oplus x = \bar{x}$ .

As early as 1929, Zhegalkin showed that this representation is canonical [51]: if we multiply out all parentheses, perform the simplifications  $x \oplus x = 0$  and  $x \cdot x = x$ , and sort the product terms, the resulting expression is unique. Accordingly, we call the representation XNF, for XOR Normal Form. It is also sometimes known as the Reed-Muller form.

The XNF representation has distinct advantages when manipulating expressions algebraically. Since it is canonical, we need not concern ourselves with simplifying the expressions, as we would working with the AON representation. Furthermore, unlike AON, the dependence of a function on its variables is explicit in XNF. If a variable appears in an expression, then there exists some assignment of values to the other variables such the value of the expression depends on the value of that variable.

What follows is a proof of the uniqueness of the representation. Denote

$$x_1 \oplus \cdots \oplus x_n \quad \text{by} \quad \sum_{i=1}^n x_i,$$

and

$$x_1 \cdots x_n \quad \text{by} \quad \prod_{i=1}^n x_i.$$

Let  $f$  be an  $n$ -ary Boolean function.

**Proposition 6.1** *To every  $n$ -ary Boolean function  $f$  there exists a unique family  $F$  of subsets of  $N = \{1, \dots, n\}$  such that*

$$f(x_1, \dots, x_n) = \sum_{I \in F} \prod_{i \in I} x_i. \tag{6.1}$$

For example,

$$x_1 + x_2 = x_1 \oplus x_2 \oplus x_1 x_2$$

with  $F = \{\{1\}, \{2\}, \{1, 2\}\}$ . Call the right-hand side of Equation 6.1 a *Boolean polynomial*.

**Proof:** Let  $\mathcal{P}(N)$  be the family of all subsets of  $N$ , and let  $O^{(n)}$  be the set of  $n$ -ary

Boolean functions. For every  $F \subseteq \mathcal{P}(N)$ , denote by  $\phi(F)$  the corresponding Boolean polynomial. Clearly  $\phi$  is a map from the set  $\mathcal{P}(\mathcal{P}(N))$  of families of subsets of  $N$  into  $O^{(n)}$ .

**Claim** The map  $\phi$  is injective.

**Proof:** Let  $\phi(F) = \phi(G)$  for some  $F, G \subseteq \mathcal{P}(N)$ . By the way of contradiction, suppose that  $F \setminus G \neq \emptyset$ . Choose  $I \in F \setminus G$  of the least possible cardinality. Put  $a_i = 1$  for  $i \in I$  and  $a_i = 0$  otherwise. For  $A = (a_1, \dots, a_n)$ , we have  $\phi(F)(A) = 1$ , since  $\prod_{i \in I} a_i = 1$  and all the other monomials vanish at  $A$ ; while  $\phi(G)(A) = 0$ , since every  $J \in G$  meets  $N \setminus I$ . This contradiction shows that  $F \subseteq G$ . By symmetry  $G \subseteq F$  and so  $G = F$ . Now  $|\mathcal{P}(\mathcal{P}(N))| = 2^{2^n} = |O^{(n)}|$  and hence  $\phi$  is a bijection from  $\mathcal{P}(\mathcal{P}(N))$  onto  $O^{(n)}$ , proving the uniqueness.  $\square$

# Appendix B: Synthesis Results

## B-1 Optimization of Area at the Network Level

We present a simple comparison between the cost of cyclic versus acyclic substitutions. The substitution/minimization operation is performed with the `simplify` procedure in the Berkeley SIS package [38], with parameters:

- `method = snocomp`,
- `dctype = all`,
- `filter = exact`,
- `accept = fct_lits`.

The cost given is that of the resulting network, as measured by the literal count of the nodes expressed in factored form. This is compared to the cost of the network obtained by executing `simplify` directly with the same parameters. For the larger circuits, the amount of improvement drops off due to time limits imposed on the search.

For benchmark circuits, we used the usual suspects, namely the Espresso [52] suite, as well as the International Workshop on Logic Synthesis [53] suite. Examples were selected based on size and suitability (generally, circuits with fewer than 30 inputs and fewer than 30 outputs). For circuits with latches, we extracted the combinational part. In Tables 6.1 and 6.2, we present those circuits for which cyclic solutions were found. Column 4 gives the improvement, and Column 5 the computation time.

Since randomly generated functions are very dense, they are not generally representative of functions encountered in practice. Nevertheless, it is interesting to

Espresso Benchmarks				
	Simplify	CYCLIFY	Improvement	Time (H:M:S)
p82	104	90	13.5%	00:02:03
t4	109	89	18.3%	00:00:02
dc2	130	123	5.4%	00:01:34
apla	185	131	29.2%	00:00:31
tms	185	158	14.6%	00:01:17
m2	231	207	10.4%	00:06:02
t1	273	206	24.5%	00:21:40
b4	292	281	3.8%	00:09:50
exp	320	260	18.8%	00:33:26
in3	361	333	7.8%	00:22:06
in2	397	291	26.7%	00:00:45
b10	398	359	9.8%	00:08:29
gary	421	404	4.0%	00:18:15
m4	439	412	6.2%	00:07:22
in0	451	434	3.8%	00:05:53
max1024	793	774	2.4%	00:00:29

Table 6.1: Cost (literals in factored form) of Berkeley SIS **Simplify** vs. **CYCLIFY** for Espresso Benchmarks.

examine the performance of the **CYCLIFY** program on these. We present results from random trials in Table 6.3. Each row lists the results of 25 trials. Cyclic solutions were found in nearly all cases (3rd column). The average improvement is given in the 4th column, and the range of improvement in the 5th column.

## B-2 Optimization of Area at the Gate Level

Here we compare the results of cyclic vs. acyclic optimizations for designs carried through to the decomposition and mapping phases. The optimizations are performed according to the standard “**script.rugged**” sequence and then mapped to fan-in 2 NAND/NOR gates and inverters.

Table 6.4 shows the results on some of the same benchmark collection. Again, the sequence begins with a collapsed specification of the circuit.

IWLS 93 Benchmarks				
	Simplify	CYCLIFY	Improvement	Time (H:M:S)
ex6	85	76	10.6%	00:00:06
inc	116	112	3.4%	00:00:04
bbsse	118	106	10.2%	00:00:08
sse	118	106	10.2%	00:00:10
5xp1	123	109	11.4%	00:00:01
s386	131	113	13.7%	00:00:08
bw	171	163	4.7%	00:15:41
s400	179	165	7.8%	00:02:12
s382	180	165	8.3%	00:02:30
s526n	194	189	2.6%	00:00:29
s526	196	188	4.1%	00:00:25
cse	212	177	16.5%	00:00:05
clip	213	193	9.4%	00:00:01
pma	226	211	6.6%	00:04:30
dk16	248	233	6.0%	00:00:53
s510	260	227	12.7%	00:00:05
ex1	309	276	10.7%	00:09:11
s1	332	322	3.0%	00:03:34
duke2	415	397	4.3%	00:02:58
styr	474	443	6.5%	00:03:24
planet1	550	517	6.0%	05:09:19
planet	555	504	9.2%	02:57:47
s1488	622	589	5.3%	00:47:04
s1494	659	634	3.8%	05:19:41
table3	1287	1175	8.7%	12:39:20
table5	1059	1007	4.9%	14:10:10
s298	2598	2445	5.9%	10:15:03
ex1010	3703	3593	3.0%	10:57:58

Table 6.2: Cost (literals in factored form) of Berkeley SIS **Simplify** vs. **CYCLIFY** for the Workshop on Logic and Synthesis Benchmarks.

Randomly Generated Networks				
# In.	# Out.	Cyclic Solns. Found	Average. Improvement	Range
5	5	100%	8.5%	3% – 17%
5	7	96%	9.1%	0% – 18%
5	10	100%	12.0%	2% – 20%
5	15	100%	13.4%	7% – 23%
5	20	100%	14.2%	8% – 18%
7	10	96%	5.6%	0% – 11%
7	15	88%	3.6%	0% – 10%

Table 6.3: Cost improvement (literals in factored form) of CYCLIFY over Berkeley SIS `Simplify` for randomly generated networks (25 trials per row).

- For the results in column 2, we apply the “`script.rugged`” sequence.
- For the results in column 3, we apply the “`script.rugged`” sequence, but using our CYCLIFY command in the place of the SIS `simplify` command.

In both cases, the results are mapped with the command “`map -m 0`” to a NAND2/NOR2 library, specified in Figure 6.6. The area improvements obtained with the cyclic solutions, as a percentage of the area of the acyclic solutions, are given in column 4. All cyclic solutions were validated in their final form, at the gate-level.

```

GATE zero    0  0=CONST0;
GATE one     0  0=CONST1;
GATE inv1    1  0=!a;      PIN * INV 1 999 1.0 0.0 1.0 0.0
GATE nand2   2  0=!(a*b);  PIN * INV 1 999 2.0 0.0 2.0 0.0
GATE nor2    2  0=!(a+b);  PIN * INV 1 999 2.0 0.0 2.0 0.0

```

Figure 6.6: NAND2/NOR2 library, in “`genlib`” format.

## B-3 Joint Optimization of Area and Delay at the Gate Level

Finally, we compare gate-level optimization according to the standard “*script.delay*” sequence. In the optimizations with CYCLIFY, we assigned an arbitrary weighting



LGSynth93 Benchmarks			
	SIS Area	CYCLIFY Area	Improvement
5xp1	203	182	10.34%
ex6	194	152	21.65%
planet	943	889	5.73 %
s386	231	222	3.90 %
bw	302	255	15.56%
cse	344	329	4.36 %
pma	409	393	3.91 %
s510	514	483	6.03 %
duke2	847	673	20.54%
styr	858	758	11.66%
s1488	1084	1003	7.47 %

Table 6.4: Area of SIS solutions vs. CYCLIFY solutions for Benchmarks Optimized with “`script.rugged`”, and mapped to NAND2/NOR2 gates and inverters.

of one-third to the area and two-thirds to the delay. Again, the results were mapped to two-input NAND/NOR gates and inverters. We assume an area of 2 units per gate, and 1 unit per inverter; and a delay of bound of 1 time unit per gate, and 0.5 time units per inverter.

Table B-3 shows the benchmarks for which cyclic solutions were found offering improvement in either area or delay. The area and delay of the SIS solutions are given in columns 2 and 3, respectively. The area and delay of the CYCLIFY solutions are given in columns 4 and 6, respectively. The improvements in area and delay, as percentages of the SIS solutions, are given in Columns 5 and 7, respectively.

Espresso Benchmarks						
	SIS		CYCLIFY			
	Area	Delay	Area		Delay	
p82	175	19.0	167	4.6 %	15.0	21.1 %
apla	242	26.0	243	-0.4 %	25.0	3.8 %
tms	302	31.0	292	3.3 %	30.0	3.2 %
t1	343	17.0	327	4.6 %	14.0	17.6 %
b4	474	30.0	464	2.1 %	29.0	3.4 %
exp	502	31.0	480	4.4 %	29.0	6.4 %
in3	599	40.0	593	1.0 %	33.0	17.5 %
in2	590	34.0	558	5.4 %	29.0	14.7 %
b10	681	37.0	691	-1.5 %	35.0	5.4 %
in0	751	42.0	777	-3.5 %	37.0	11.9 %
LGSynth93 Benchmarks						
	SIS		CYCLIFY			
	Area	Delay	Area		Delay	
5xp1	210	23.0	180	14.3 %	22.0	4.3 %
planet	964	40.0	938	2.7 %	38.0	5.0 %
s386	222	21.0	217	2.2 %	20.0	4.7 %
bw	280	28.0	254	9.3 %	20.5	26.8 %
cse	337	29.5	333	1.2 %	27.5	6.7 %
clip	356	28.0	342	3.9 %	27.0	3.5 %
s510	452	28.0	444	1.8 %	24.0	14.3 %
ex1	526	40.0	522	0.7 %	34.0	15.0 %
s1	566	36.0	542	4.2 %	31.0	13.9 %
duke2	742	38.0	716	3.5 %	34.0	10.5 %
styr	821	39.0	827	-0.7 %	36.0	7.7 %
s1488	1016	43.0	995	2.1 %	34.0	20.9 %
s1494	1090	46.0	1079	1.0 %	39.0	15.2 %

Table 6.5:

Area and Delay of Berkeley SIS vs. CYCLIFY for Benchmarks with “*script.delay*” optimizations, and mapping to NAND2/NOR2 gates and inverters.

# Bibliography

- [1] E. Allender, “Circuit Complexity before the Dawn of the New Millennium,” in *Conf. Foundations of Software Technology and Theoretical Computer Science*, published as *Springer Lecture Notes in Computer Science*, Vol. 1180, pp. 1–18, 1996.
- [2] R. I. Bahar, H. Cho, G. D. Hachtel, E. Macii, F. Somenzi, “Timing Analysis of Combinational Circuits Using ADDs,” *European Conf. Design Automation*, pp. 625–629, 1994.
- [3] R. I. Bahar, E.A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, F. Somenzi, “Algebraic Decision Diagrams and their Applications,” *Int’l Conf. Computer-Aided Design*, 1993.
- [4] R. K. Brayton, G. D. Hachtel, C. T. McMullen, A. L. Sangiovanni-Vincentelli, “Logic Minimization Algorithms for VLSI Synthesis,” Kluwer Academic, 1984.
- [5] R. K. Brayton, R. Rudell, A. L. Sangiovanni-Vincentelli, A. Wang, “MIS: Multiple-level Interactive Logic Optimization System,” *IEEE. Trans. Computer-Aided Design*, Vol. 6, No. 6, pp. 1062–1081, 1987.
- [6] R. K. Brayton, G. D. Hachtel, C. T. McMullen, A. L. Sangiovanni-Vincentelli, “Multilevel Logic Synthesis,” *Proceedings IEEE*, Vol. 78, No. 2, pp. 264–300, 1990.
- [7] R. E. Bryant, “Graph-Based Algorithms For Boolean Function Manipulation,” *IEEE Trans. Computers*, Vol. C-35, No. 6, pp. 677–691, 1986.

- [8] R. E. Bryant, "Boolean Analysis of MOS Circuits," *IEEE Trans. Computer-Aided Design*, pp. 634–649, 1987.
- [9] J. A. Brzozowski, C.-J. H. Seger, "Asynchronous Circuits," Springer-Verlag, 1995.
- [10] E. S. Davidson, "An Algorithm for NAND Decomposition under Network Constraints," *IEEE Trans. Computers*, Vol. C-18, No. 12, pp. 1098–1109, 1969.
- [11] R. Drechsler, W. Gunter, "Exact Circuit Synthesis," *Int'l Conf. Advanced Computer Systems*, pp. 517–524, 1998.
- [12] S. A. Edwards, "Making Cyclic Circuits Acyclic," *Design Automation Conf.*, pp. 159–162, 2003.
- [13] E. B. Eichelberger, "Hazard Detection in Combinational and Sequential Switching Circuits," *IBM J. Research & Development*, Vol. 9, pp. 90–99, 1965.
- [14] M. R. Garey, D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-completeness," W. H. Freeman & Co., 1979.
- [15] D. A. Huffman, "Combinational Circuits with Feedback," Recent Developments in Switching Theory, A. Mukhopadhyay, ed., pp. 27–55, 1971.
- [16] M. Karnaugh, "The Map Method for Synthesis of Combinational Logic Circuits," *Trans. AIEE*, Pt. I, Vol. 79, No. 9, pp. 593–599, 1953.
- [17] W. H. Kautz, "The Necessity of Closed Circuit Loops in Minimal Combinational Circuits," *IEEE Trans. Computers*, Vol. C-19, pp. 162–166, 1970.
- [18] V. Khrapchenko, "Depth and Delay in a Network," *Soviet Math. Dokl.*, No. 19, pp. 1006–1009, 1978.
- [19] Y. Kukimoto, R. Brayton, "Exact Required Time Analysis via False Path Detection," *Design Automation Conf.*, pp. 220–225, 1997.
- [20] E. L. Lawler, "An Approach to Multilevel Boolean Minimization," *Journal of the ACM*, Vol. 11, No. 3, pp. 283–295, 1964.

- [21] C. Y. Lee, “Representation of Switching Circuits by Binary-Decision Programs,” *Bell System Technical Journal*, Vol. 38, pp. 985–999, 1959.
- [22] L. Lovász, D. B. Shmoys, É. Tardos, “Combinatorics in Computer Science, ” in *Handbook of Combinatorics*, R. L. Graham, M. Grötschel, L. Lovász, eds., Elsevier Science, pp. 2012, 1995.
- [23] C. H. Papadimitriou, “Computational Complexity,” Addison-Wesley, pp. 80, 1995.
- [24] S. Malik, “Analysis of Cyclic Combinational Circuits,” *IEEE Trans. Computer-Aided Design*, Vol. 13, No. 7, pp. 950–956, 1994.
- [25] A. Markov, “On the Inversion Complexity of a System of Functions,” *Soviet Math. Dokl.*, No. 116, pp. 917–919, 1957; also published in *Journal of the ACM*, Vol. 5, pp. 331–334, 1958.
- [26] C. R. McCaw, “Loops in Directed Combinational Switching Networks,” Engineer’s Thesis, Stanford University, 1963.
- [27] E. McCluskey, “Minimization of Boolean Functions,” *Bell System Technical Journal*, Vol. 35, pp. 437–457, 1956.
- [28] C. Mead, L. Conway, “Introduction to VLSI Systems,” Addison-Wesley, 1980.
- [29] M. Mendler, M. Fairlough, “Ternary Simulation: A Refinement of Binary Functions or an Abstraction of Real-Time Behavior, ” *Workshop on Designing Correct Circuits*, 1996.
- [30] S. Minato, “Zero-suppressed BDDs for Set Manipulation in Combinational Problems,” *Design Automation Conf.*, pp. 272–277, 1993.
- [31] W. Quine, “The Problem of Simplifying Truth Functions,” *American Math. Monthly*, Vol. 59, No. 8, pp. 521–531, 1952.

- [32] A. Raghunathan, P. Ashar, S. Malik, “Test Generation for Cyclic Combinational Circuits,” *IEEE Trans. Computer-Aided Design*, Vol. 14, No. 11, pp. 1408–1414, 1995.
- [33] M. Riedel, J. Bruck, “Cyclic Combinational Circuits: Analysis for Synthesis,” *Int’l Workshop Logic and Synthesis*, pp. 105–112, 2003.
- [34] M. Riedel, J. Bruck, “The Synthesis of Cyclic Combinational Circuits,” *Design Automation Conf.*, pp. 163–168, 2003.
- [35] R. L. Rivest, “The Necessity of Feedback in Minimal Monotone Combinational Circuits,” *IEEE Trans. Computers*, Vol. C-26, No. 6, pp. 606–607, 1977.
- [36] J. Rothe, “Some Facets of Complexity Theory and Cryptography,” *ACM Computing Surveys*, Vol. 34, No. 4, pp. 504–549, 2002.
- [37] D. Scott, “A Type-Theoretical Alternative to CUCH, ISWIM, OWHY,” *Theoretical Computer Science*, Vol. 121, pp. 411–440. (Published version of unpublished notes, Oxford, 1969.)
- [38] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, A. Sangiovanni-Vincentelli, “SIS: A System for Sequential Circuit Synthesis,” Tech. Rep., UCB/ERL M92/41, Electronics Research Lab, University of California, Berkeley, 1992.
- [39] C. E. Shannon, “A Symbolic Analysis of Relay and Switching Circuits,” *Trans. AIEE*, Vol. 57, pp. 713–723, 1938.
- [40] C. E. Shannon, “The Synthesis of Two Terminal Switching Circuits,” *Bell System Technical Journal*, Vol. 28, pp. 59–98, 1949.
- [41] C. E. Shannon, “Realization of All 16 Switching Functions of Two Variables Requires 18 Contacts,” Memorandum MM 53-1400-40, Bell Laboratories, 1953.
- [42] T. R. Shiple, “Formal Analysis of Synchronous Circuits,” Ph.D. Dissertation, University of California, Berkeley, 1996.

- [43] T. R. Shiple, V. Singhal, R. K. Brayton, A. L. Sangiovanni-Vincentelli, "Analysis of Combinational Cycles in Sequential Circuits," *IEEE Int'l Symp. Circuits and Systems*, Vol. 4, pp. 592–595, 1996.
- [44] T. R. Shiple, G. Berry, H. Touati, "Constructive Analysis of Cyclic Circuits," *European Design and Test Conf.*, pp. 328–333, 1996.
- [45] R. A. Short, "A Theory of Relations Between Sequential and Combinational Realizations of Switching Functions," Ph.D. Dissertation, Stanford University, 1961.
- [46] A. Srinivasan, S. Malik, "Practical Analysis of Cyclic Combinational Circuits," *IEEE Custom Integrated Circuits Conf.*, pp. 381–384, 1996.
- [47] L. Stok, "False Loops Through Resource Sharing," *Int'l Conf. Computer-Aided Design*, pp. 345–348, 1992.
- [48] I. Wegener, "The Complexity of Boolean Functions," John Wiley & Sons, 1987.
- [49] H. Yalcin, J. Hayes, "Event Propagation Conditions in Circuit Delay Computation," *ACM Trans. Design Automation of Electronic Systems*, Vol. 2, No. 3, pp. 249—280, 1997.
- [50] M. Yoeli, S. Rinon, "Application of Ternary Algebra to the Study of Static Hazards," *Journal of the ACM*, Vol. 11, No. 1, pp. 84–97, 1964.
- [51] I. Zhegalkin, "The Arithmetization of Symbolic Logic," *Math. Sbornik*, Vol. 36, pp. 205–338, 1929.
- [52] Benchmarks from "Logic Minimization Algorithms for VLSI Synthesis," by R. K. Brayton et al., available at <ftp://ic.eecs.berkeley.edu/>.
- [53] Benchmarks from the 1993 *Int'l Workshop on Logic and Synthesis*, available at <http://www.cbl.ncsu.edu/>.