Logic Synthesis for Networks of Four-terminal Switches

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Mustafa Altun

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Advisor: Marc D. Riedel

May, 2012

Acknowledgements

I would like to thank my advisor, Professor Marc Riedel, for his guidance and support through my graduate studies. Throughout the four years that he has been my advisor, he has served as an incredible inspiration and talented mentor in my educational and career development. I sincerely appreciate his patient help and consistent encouragement. It has been a pleasure working with him.

I would like to thank my great colleagues, John Backes, Hua Jiang, Aleksandra Kharam, and Weikang Qian, for providing an enjoying and enriching environment. They are the main reason why every day I am happy to go to the office. I will miss them.

I would like to thank my mother Emine Altun and my father Kemal Altun for being with me through all the ups and downs of my life; I would not earned my Ph.D. without their support. I would like to express my dearest thanks to my wife Banu Altun who has always been extremely understanding and supportive during my studies, and to my little daughter Zeynep for bringing so much happiness and joy into our life. My career and life are more meaningful because of the love and care that I have been privileged to receive from my whole family.

Dedication

To my wife, Banu, for her endless love.

Abstract

As current CMOS-based technology is approaching its anticipated limits, research is shifting to novel forms of nanoscale technologies including molecular-scale self-assembled systems. Unlike conventional CMOS that can be patterned in complex ways with lithography, self-assembled nanoscale systems generally consist of regular structures. Logical functions are achieved with crossbar-type switches. Our model, a network of four-terminal switches, corresponds to this type of switch in a variety of emerging technologies, including nanowire crossbar arrays and magnetic switch-based structures. We discuss this in the first part of the dissertation.

We consider networks of four-terminal switches arranged in rectangular *lattices*. In the second part of the dissertation, we develop a synthesis method to implement Boolean functions with lattices of four-terminal switches. Each switch of the lattice is controlled by a Boolean literal. If the literal takes the value 1, the corresponding switch is connected to its four neighbours; else it is not connected. A Boolean function is implemented in terms of connectivity across the lattice: it evaluates to 1 iff there exists a connected path between two opposing edges of the lattice. We address the synthesis problem of how best to assign literals to switches in a lattice in order to implement a given target Boolean function, with the goal of minimizing the lattice size, measured in terms of the number of switches. We present an efficient algorithm for this task. The algorithm has polynomial time complexity. It produces lattices with a size that grows linearly with the number of products of the target Boolean function. We evaluate our synthesis method on standard benchmark circuits and compare the results to a lower-bound calculation on the lattice size.

In the third part of the dissertation, we address the problem of implementing Boolean functions with lattices of four-terminal switches in the presence of defects. We assume

that such defects occur probabilistically. Our approach is predicated on the mathematical phenomenon of *percolation*. With random connectivity, percolation gives rise to a sharp non-linearity in the probability of global connectivity as a function of the probability of local connectivity. We exploit this phenomenon to compute Boolean functions robustly. We show that the *margins*, defined in terms of the steepness of the non-linearity, translate into the degree of defect tolerance. Achieving good margins entails a mapping problem. Given a target Boolean function, the problem is how to assign literals to regions of the lattice such that there are no diagonal paths of 1's in any assignment that evaluates to 0. Assignments with such paths result in poor error margins due to stray, random connections that can form across the diagonal. A necessary and sufficient condition is formulated for a mapping strategy that preserves good margins: the top-to-bottom and left-to-right connectivity functions across the lattice must be *dual* functions. Based on lattice duality, we propose an efficient algorithm to perform the mapping. The algorithm optimizes the lattice area while meeting prescribed worst-case margins. We demonstrate its effectiveness on benchmark circuits.

A significant tangent for this work is its mathematical contribution: lattice-based implementations present a novel view of the properties of Boolean function duality. In the final part of the dissertation, we study the applicability of these properties to the famous problem of testing whether a monotone Boolean function in irredundant disjunctive normal form (IDNF) is self-dual. This is one of the few problems in circuit complexity whose precise tractability status is unknown. We show that monotone self-dual Boolean functions in IDNF do not have more variables than disjuncts. We examine monotone self-dual Boolean functions in IDNF with the same number of variables and disjuncts. We propose an algorithm to test whether a monotone Boolean function in IDNF with n variables and n disjuncts is self-dual. The runtime of the algorithm is $\mathcal{O}(n^4)$.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In his seminal Master's Thesis, Claude Shannon demonstrated that any Boolean function can be implemented by a switching relay circuit [1]. Shannon's thesis became the foundation of modern digital logic design and led to the design and fabrication of digital circuits with millions of transistors. He considered two-terminal switches corresponding to electromagnetic relays. An example of a two-terminal switch is shown in the top part of Figure 1.1. The switch is either ON (closed) or OFF (open). A Boolean function can be implemented in terms of connectivity across a network of switches, often arranged in a series/parallel configuration. An example is shown in the bottom part of Figure 1.1. Each switch is controlled by a Boolean literal. If the literal is 1 then the corresponding switch is ON; if the literal is 0 then the corresponding switch is OFF. The Boolean function for the network evaluates to 1 if there is a closed path between the left and right nodes. It can be computed by taking the sum (OR) of the product (AND) of literals along each path. These products are $x_1x_2x_3$, $x_5x_1x_2x_6$, $x_5x_4x_2x_3$, and $x_5x_4x_6$.

In this dissertation, we demonstrate that any Boolean function can be implemented by a network of four-terminal switches. An example of a four-terminal switch is shown in the top part of Figure 1.2. The four terminals of the switch are all either mutually

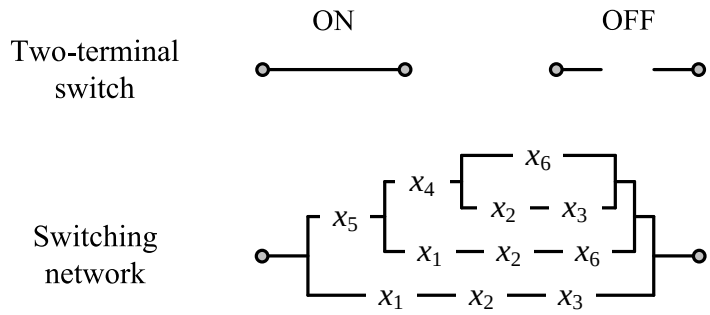


Figure 1.1: Two-terminal switching network implementing the Boolean function $x_1x_2x_3 + x_1x_2x_5x_6 + x_2x_3x_4x_5 + x_4x_5x_6$.

connected (ON) or disconnected (OFF). We consider networks of four-terminal switches arranged in rectangular *lattices*. An example is shown in the bottom part of Figure 1.2. Again, each switch is controlled by a Boolean literal. If the literal takes the value 1 then corresponding switch is ON; if the literal takes the value 0 then corresponding switch is OFF. The Boolean function for the lattice evaluates to 1 iff there is a closed path between the top and bottom edges of the lattice. Again, the function is computed by taking the sum of the products of the literals along each path. These products are $x_1x_2x_3$, $x_1x_2x_5x_6$, $x_4x_5x_2x_3$, and $x_4x_5x_6$ – the same as those in Figure 1.1. We conclude that this lattice of four-terminal switches implements the same Boolean function as the network of two-terminal switches in Figure 1.1; they both implement $x_1x_2x_3 + x_1x_2x_5x_6 + x_2x_3x_4x_5 + x_4x_5x_6$.

Throughout this dissertation, we use a "checkerboard" representation for lattices where black and white sites represent ON and OFF switches, respectively, as illustrated in Figure 1.3. We discuss the Boolean functions implemented in terms of connectivity between the top and bottom edges as well as connectivity between the left and right edges. (We refer to these edges as "plates".)

We frame our discussion at the technology-independent level. Although conceptually general, our model is applicable for a variety of nanoscale technologies, such as nanowire

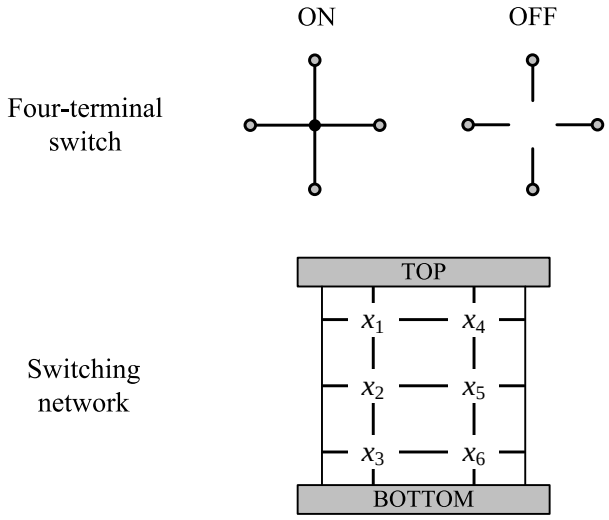


Figure 1.2: Four-terminal switching network implementing the Boolean function $x_1x_2x_3 + x_1x_2x_5x_6 + x_2x_3x_4x_5 + x_4x_5x_6$.

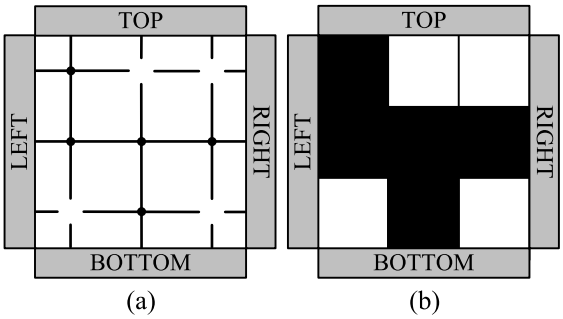


Figure 1.3: A 3×3 four-terminal switch network and its lattice form.

crossbar arrays [2, 3] and magnetic switch-based structures [4, 5]. In Chapter 2, we discuss potential technologies that fit our model of lattices of four-terminal switches.

1.1 Synthesis Problem

We address the following synthesis problem: how should we assign literals to switches in a lattice in order to implement a given target Boolean function? Suppose that we are asked to implement the function $f(x_1, x_2, x_3, x_4) = x_1x_2x_3 + x_1x_4$. We might consider

the lattice in Figure 1.4(a). The product of the literals in the first column is $x_1x_2x_3$; the product of the literals in the second column is x_1x_4. We might also consider the lattice in Figure 1.4(b). The products for its columns are the same as those for (a). In fact, the two lattices implement two different functions, only one of which is the intended target function. To see why this is so, note that we must consider all possible paths, including those shown by the red and blue lines. In (a) the product x_1x_2 corresponding to the path shown by the red line covers the product $x_1x_2x_3$ so the function is $f_a = x_1x_2 + x_1x_4$. In (b) the products $x_1x_2x_4$ and $x_1x_2x_3x_4$ corresponding to the paths shown by the red and blue lines are redundant, covered by column paths, so the function is $f_b = x_1x_2x_3 + x_1x_4$.

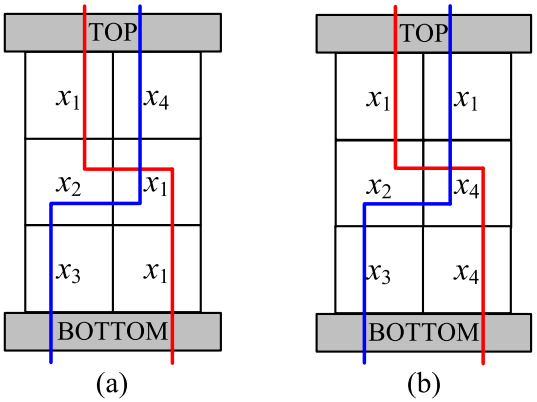


Figure 1.4: Two 3×2 lattices implementing different Boolean functions.

In this example, the target function is implemented by a 3×2 lattice with four paths. If we were given a target function with more products, a larger lattice would likely be needed to implement it; accordingly, we would need to enumerate more paths. Here the problem is that number of paths grows exponentially with the lattice size. Any synthesis method that enumerates paths quickly becomes intractable. In Chapter 3, we present an efficient algorithm for this task – one that does not exhaustively enumerate paths but rather exploits the concept of Boolean function *duality* [6, 7]. Our synthesis algorithm produces lattices with a size that grows linearly with the number of products of the target Boolean function. It runs in time that grows polynomially.

1.2 Robust Computation

We address the problem of implementing Boolean functions with lattices of four-terminal switches in the presence of defects. We assume that defects cause switches to fail in one of two ways: they are ON when they are supposed to be OFF (OFF-to-ON defect), i.e., the controlling literal is 0; or they are OFF when they are supposed to be ON (ON-to-OFF defect), i.e., the controlling literal is 1. We allow for different defect rates in both directions, ON-to-OFF and OFF-to-ON. For example, if a switch has a larger ON-to-OFF defect rate than its OFF-to-ON defect rate then the switch works more accurately when its controlling input is 1 (the switch is ON). Crucially, we assume that all switches of the lattice fail with *independent* probability.

Defective switches can ruin the Boolean computation performed by a network. Consider the networks shown in Figure 1.5. The network in Figure 1.5(a) consists of a single switch. The networks in Figure 1.5(b) and Figure 1.5(c) consist of a pair of switches in series and in parallel, respectively. All switches are controlled by the literal x_1. Obviously, in each of these networks, the top and bottom plates are connected when $x_1 = 1$ and disconnected when $x_1 = 0$. Therefore they implement the function $f = x_1$.

Note that the three networks are not identical in their defect-tolerance capability. Suppose that exactly one switch in each network is defective when $x_1 = 1$ and exactly one is defective when $x_1 = 0$. When $x_1 = 1$, the networks in Figure 1.5(a) and Figure 1.5(b) compute the wrong value of $f = 0$; however, the network in Figure 1.5(c) computes the correct value $f = 1$. Similarly, when $x_1 = 0$, the networks in Figure 1.5(a) and Figure 1.5(c) compute the wrong value of $f = 1$. However, the network in Figure 1.5(b) computes the correct value of $f = 0$. So the series and parallel networks in Figures 1.5(b) and 1.5(c) each tolerate up to one defective switch, but they tolerate different defect types. None of these networks tolerates defects for both cases $x_1 = 1$ and $x_1 = 0$.

Now consider the network in Figure 1.6. Compared to the networks in Figure 1.5, it has more switches. We expect that it will be superior in terms of its defect tolerance,

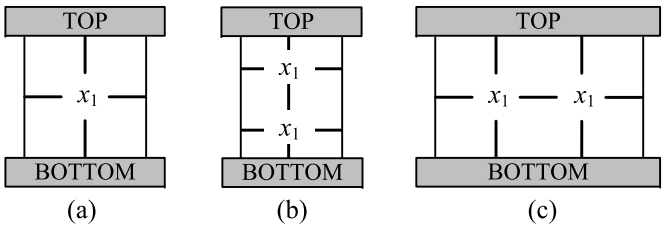


Figure 1.5: Switching networks

for both the cases $x_1 = 1$ and $x_1 = 0$. But what is the relationship between the amount of redundancy and the defect tolerance that is achieved? As shown in Figure 1.7, the relationship is non-linear. The explanation hinges on percolation.

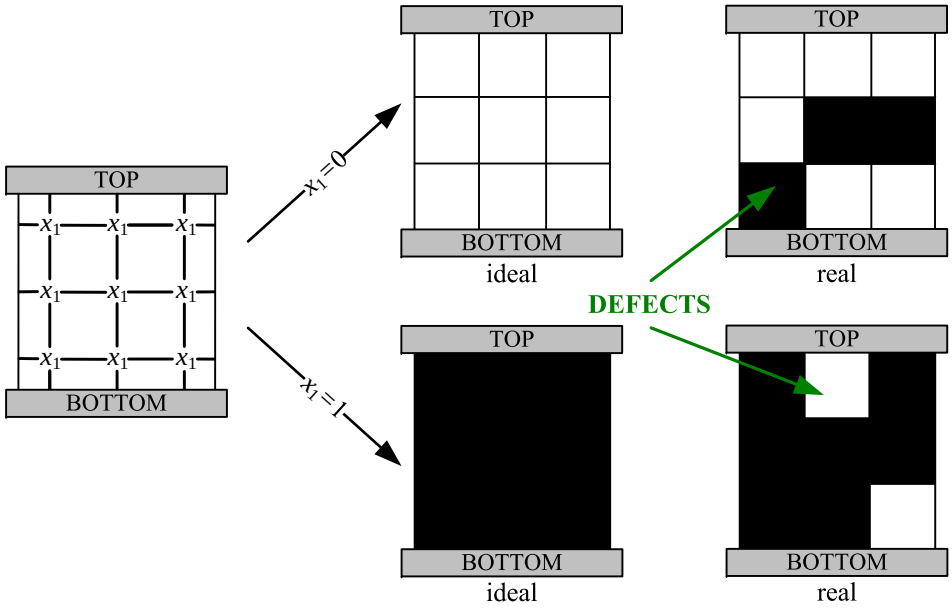


Figure 1.6: Switching network with defects.

We exploit the percolation phenomenon to compute Boolean functions robustly. We show that the *margins*, defined in terms of the steepness of the non-linearity, translate into the degree of defect tolerance. Achieving good margins entails a mapping problem. Given a target Boolean function, the problem is how to assign literals to regions of the lattice such that there are no diagonal paths of 1's in any assignment that evaluates

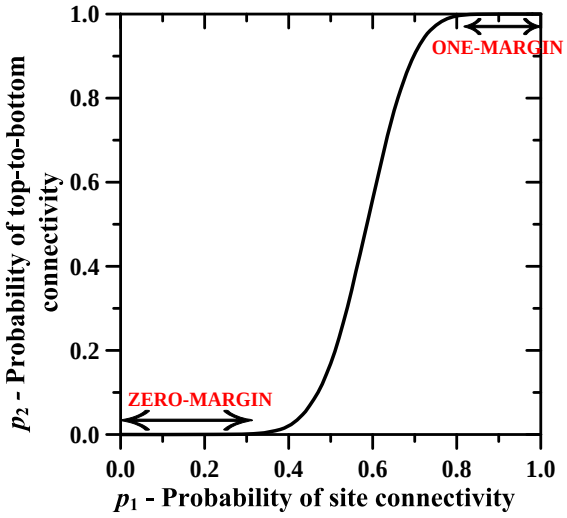


Figure 1.7: Non-linearity through percolation in random media.

to 0. Assignments with such paths result in poor error margins due to stray, random connections that can form across the diagonal. A necessary and sufficient condition is formulated for a mapping strategy that preserves good margins: the top-to-bottom and left-to-right connectivity functions across the lattice must be *dual* functions. In Chapter 4, we propose an efficient algorithm, based on lattice duality, to perform the mapping. The algorithm optimizes the lattice area while meeting prescribed worst-case margins.

1.3 Boolean Duality

We exploit the concept of Boolean function duality in our synthesis methodology. Our method, presented in Chapter 3, implements Boolean functions with lattices of four-terminal switches such that top-to-bottom and left-to-right Boolean functions across the lattice are dual pairs. Our mapping strategy for robust computation, presented in Chapter 4, is also constructed on lattice duality. These lattice-based implementation techniques present a novel view of the properties of Boolean function duality. We

study the applicability of these properties to the famous problem of testing whether a monotone Boolean function in irredundant disjunctive normal form (IDNF) is self-dual. This is one of the few problems in circuit complexity whose precise tractability status is unknown. This famous problem is called the *monotone self-duality problem* [8].

Consider a monotone Boolean function f in IDNF. Suppose that f has k variables and n disjuncts:

$$f(x_1, x_2, \ldots, x_k) = D_1 \vee D_2 \vee \cdots \vee D_n$$

where each disjunct D_i is a prime implicant of f, $i = 1, \ldots n$. The relationship between k and n is a key aspect of the monotone self-duality problem. Prior work has shown that if f is self-dual then $k \leq n^2$ [6, 9]. We improve on this result. In Chapter 5, we show that if f is self-dual then $k \leq n$. We consider the monotone self-duality problem for Boolean functions with the same number of variables and disjuncts (i.e., $n = k$). For such functions, we propose an algorithm that runs in $\mathcal{O}(n^4)$ time.

1.4 Outline

In Chapter 2, we discuss potential technologies that fit our model of lattices of four-terminal switches.

In Chapter 3, we present our general synthesis method that implements any target function with a lattice of four-terminal switches. We also discuss the implementation of a specific function, the *parity function*. We evaluate our synthesis method on standard benchmark circuits and compare the results to the derived lower-bound on the lattice size.

In Chapter 4, we present our defect tolerance method based on percolation. We discuss the mathematics of percolation and how this phenomenon can be exploited for tolerating defects in our method. We evaluate our method on benchmark circuits.

In Chapter 5, we study the applicability of the properties of lattice duality to the famous problem of testing whether a monotone Boolean function in IDNF is self-dual.

We show that monotone self-dual Boolean functions in IDNF do not have more variables than disjuncts. We propose a polynomial-time algorithm to test whether a monotone Boolean function in IDNF with the same number of variables and disjuncts is self-dual.

Chapter 2

Applicable Technologies

In this chapter, we discuss potential technologies that fit our model of lattices of four-terminal switches. The concept of regular two-dimensional arrays of four-terminal switches is not new; it dates back to a seminal paper by Akers in 1972 [10]. With the advent of a variety of types of emerging nanoscale technologies, the model has found renewed interest [11, 12]. Unlike conventional CMOS that can be patterned in complex ways with lithography, self-assembled nanoscale systems generally consist of regular structures [13, 14]. Logical functions are achieved with crossbar-type switches [15, 5]. Although conceptually general, our model corresponds to exactly this type of switch in a variety of emerging technologies.

A schematic for the realization of our circuit model is shown in Figure 2.1. Each site of the lattice is a four-terminal switch, controlled by an input voltage. When a high (logic 1) or low (logic 0) voltage is applied, the switch is ON or OFF, respectively. The output of the circuit depends upon the top-to-bottom connectivity across the lattice. If the top and bottom plates are connected, then the lattice allows signals to flow; accordingly, the output is logic 1. Otherwise the output is logic 0. One can sense the output with a resistor connected to the bottom plate while a high voltage applied to the top plate. Below, we discuss two potential technologies that fit this circuit model.

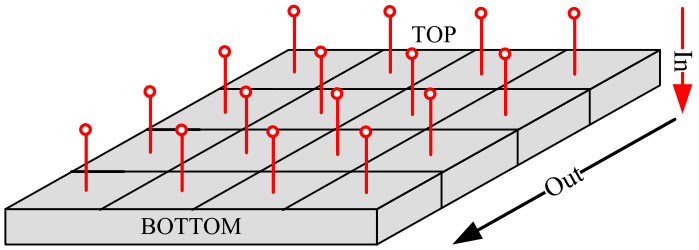


Figure 2.1: 3D realization of our circuit model with the inputs and the output.

2.1 Nanowire Crossbar Arrays

In their seminal work, Yi Cui and Charles Lieber investigated crossbar structures for different types of nanowires including n-type and p-type nanowires [16]. They achieved the different types of junctions by crossing different types of nanowires.

By crossing an n-type nanowire and a p-type nanowire, they achieved a diode-like junction. By crossing two n-types or two p-types, they achieved a resistor-like junction (with a very low resistance value). They showed that the connectivity of nanowires can be controlled by an insulated input voltage V-in. A high V-in makes the p-type nanowires conductive and the n-type nanowires resistive; a low V-in makes the p-type nanowires resistive and the n-type nanowires conductive. So they showed that, based on a controlling voltage, nanowires can behave either like short circuits or like open circuits.

Cui and Lieber implemented a four-terminal device with crossed n- and p-type nanowires, illustrated in Figure 2.2. The device works as follows. When a high V-in is applied, a p-type nanowire (green) behaves like a short circuit, so the N and S terminals are connected, and an n-type nanowire (red) behaves like an open circuit, so the W and E terminals are disconnected. When a low V-in is applied, a p-type nanowire behaves like an open circuit, so the N and S terminals are disconnected, and an n-type nanowire behaves like a short circuit, so the W and E terminals are connected.

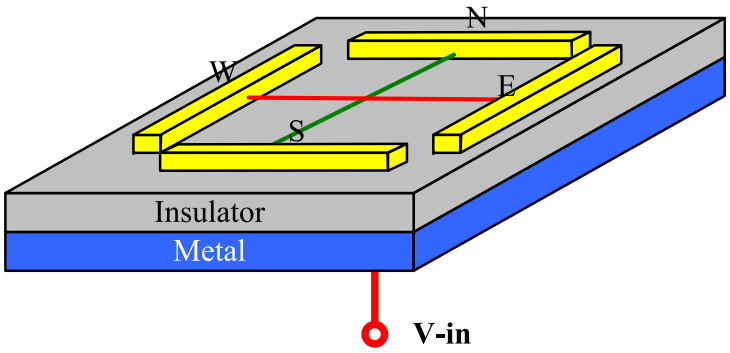


Figure 2.2: Nanowire four-terminal device with crossed n- and p-type nanowires.

A four-terminal switch can be implemented with the techniques of Cui and Lieber, as illustrated in Figure 2.3. The switch has crossed p-type nanowires. When a high V-in is applied, the nanowires behave like short circuits. A resistor-like junction is formed, with low resistance. Thus, all four terminals are connected; the switch is ON. When a low V-in is applied, the nanowires behave like open circuits: all four terminals are disconnected; the switch is OFF. The result is a four-terminal switch that corresponds to our model. It is also apparent that a four-terminal switch with negative logic can be achieved by using crossed n-type nanowires. In this case, when a high V-in is applied the switch is OFF, and when a low V-in is applied the switch is ON.

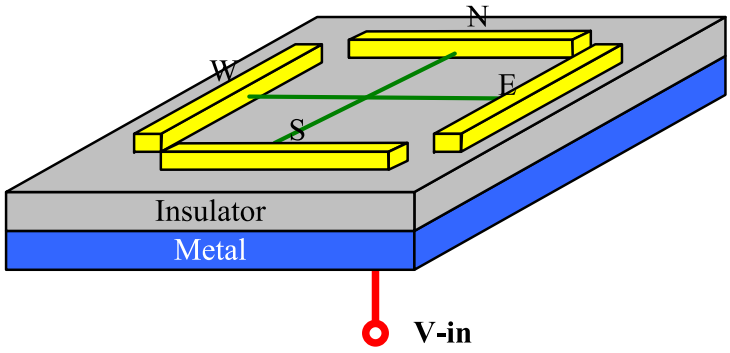


Figure 2.3: Nanowire four-terminal device (switch) with crossed p-type nanowires.

Nanowire switches, of course, are assembled in large arrays. Indeed, the impetus for nanowire-based technology is the potential density, scalability and manufacturability [17, 3, 2]. Consider a p-type nanowire array, where each crosspoint is controlled by an input voltage. From the discussion above, we know that each such crosspoint behaves like a four-terminal switch. Accordingly, the nanowire crossbar array can be modeled as a lattice of four-terminal switches as illustrated in Figure 2.4. Here the black and white sites represent crosspoints that are ON and OFF, respectively.

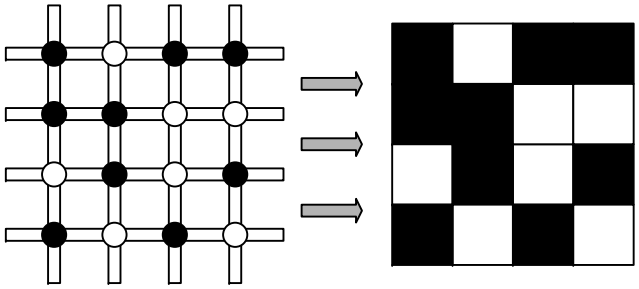


Figure 2.4: Nanowire crossbar array with random connections and its lattice representation.

Nanowire crossbar arrays may offer substantial advantages over conventional CMOS when used to implement programmable architectures. Conventional implementations typically employ SRAMs for programming crosspoints. However, for nanoscale technologies, relative to the size of the switches, SRAMs would be prohibitively costly. A variety of techniques have been suggested for fabricating programmable nanowire crosspoints based on bistable switches that form memory cores [2, 18]. Also, molecular switches and solid-electrolyte nanoswitches could be used to form programmable crosspoints [19].

2.2 Arrays of Spin Wave Switches

Other novel and emerging technologies fit our model of four-terminal switches. For instance, researchers are investigating *spin waves* [20]. Unlike conventional circuitry such as CMOS that transmits signals electrically, spin-wave technology transmits signals as

propagating disturbances in the ordering of magnetic materials. Potentially, spin-wave based logic circuits could compute with significantly less power than conventional CMOS circuitry.

Spin wave switches are four-terminal devices, as illustrated in Figure 2.5. They have two states ON and OFF, controlled by an input voltage V-in. In the ON state, the switch transmits all spin waves; all four terminals are connected. In the OFF state, the switch reflects any incoming spin waves; all four terminals are disconnected. Spin-wave switches, like nanowire switches, are also configured in crossbar networks [4].

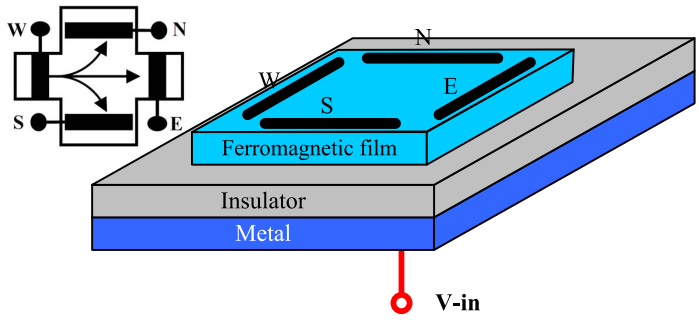


Figure 2.5: Spin-wave switch.

Chapter 3

Lattice-Based Computation

In this chapter, we consider the general problem of implementing Boolean functions with lattices of four-terminal switches. We seek to minimize the lattice size represented by the number of switches. Also, we aim for an efficient algorithm for this task – one that does not exhaustively enumerate paths in a lattice. In our synthesis strategy, we exploit the concept of lattice and Boolean function duality. This forms a novel and rich framework for Boolean computation.

This chapter is organized as follows. In Section 3.1, we present definitions that are used throughout this chapter. In Section 3.2, we present our general synthesis method that implements any target function with a lattice of four-terminal switches. In Section 3.3, we discuss the implementation of a specific function, the *parity function*. In Section 3.4, we derive a lower bound on the size of a lattice required to implement a Boolean function. In Section 3.5, we evaluate our general synthesis method on standard benchmark circuits. In Section 3.6, we discuss extensions and future directions.

3.1 Definitions

Definition 1

Consider k independent **Boolean variables**, $x_1, x_2, \ldots, x_k$. **Boolean literals** *are Boolean variables and their complements, i.e., $x_1, \bar{x}_1, x_2, \bar{x}_2, \ldots, x_k, \bar{x}_k$.* □

Definition 2

A **product (P)** *is an AND of literals, e.g., $P = x_1\bar{x}_3x_4$. A* **set of a product (SP)** *is a set containing all the product's literals, e.g., if $P = x_1\bar{x}_3x_4$ then $SP = \{x_1, \bar{x}_3, x_4\}$. A* **sum-of-products (SOP)** *expression is an OR of products.* □

Definition 3

A **prime implicant (PI)** *of a Boolean function f is a product that implies f such that removing any literal from the product results in a new product that does not imply f.* □

Definition 4

An **irredundant sum-of-products (ISOP)** *expression is an SOP expression, where each product is a PI and no PI can be deleted without changing the Boolean function f represented by the expression.* □

Definition 5

f and g are **dual Boolean functions** *iff*

$$f(x_1, x_2, \ldots, x_k) = \bar{g}(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_k).$$

Given an expression for a Boolean function in terms of AND, OR, NOT, 0, and 1, its dual can also be obtained by interchanging the AND and OR operations as well as interchanging the constants 0 and 1. For example, if $f(x_1, x_2, x_3) = x_1x_2 + \bar{x}_1x_3$ then $f^D(x_1, x_2, x_3) = (x_1 + x_2)(\bar{x}_1 + x_3)$. A trivial example is that for $f = 1$, the dual is $f^D = 0$. □

Definition 6

A **parity function** *is a Boolean function that evaluates to* 1 *iff the number of variables assigned to* 1 *is an odd number. The parity function* f *of* k *variables can be computed by the exclusive-OR (XOR) of the variables:* $f = x_1 \oplus x_2 \oplus \ldots \oplus x_k$. □

3.2 Synthesis Method

In our synthesis method, a Boolean function is implemented by a lattice according to the connectivity between the top and bottom plates. In order to elucidate our method, we will also discuss connectivity between the left and right plates. Call the Boolean functions corresponding to the top-to-bottom and left-to-right plate connectivities f_L and g_L, respectively. As shown in Figure 3.1, each Boolean function evaluates to 1 if there exists a path between corresponding plates, and evaluates to 0 otherwise. Thus, f_L can be computed as the OR of all top-to-bottom paths, and g_L as the OR of all left-to-right paths. Since each path corresponds to the AND of inputs, the paths taken together correspond to the OR of these AND terms, so implement sum-of-products expressions.

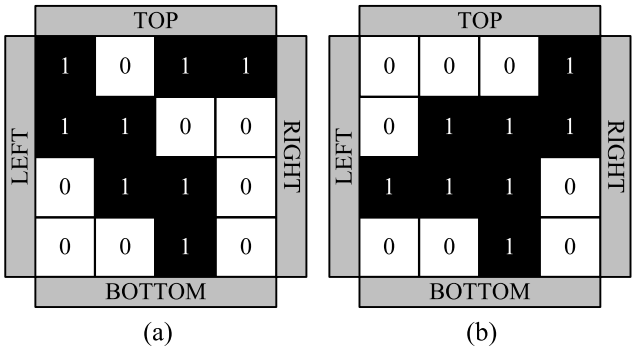


Figure 3.1: Relationship between Boolean functionality and paths. (a): $f_L = 1$ and $g_L = 0$. (b): $f_L = 1$ and $g_L = 1$.

Example 1

Consider the lattice shown in Figure 3.2. It consists of six switches. Consider the three top-to-bottom paths x_1x_4, x_2x_5, and x_3x_6. Consider the four left-to-right paths $x_1x_2x_3$, $x_1x_2x_5x_6$, $x_4x_5x_2x_3$, and $x_4x_5x_6$. While there are other possible paths, such as the one shown by the dashed line, all such paths are covered by the paths listed above. For instance, the path $x_1x_2x_5$ shown by the dashed line is covered by the path x_2x_5 shown by the solid line, and so is redundant. We conclude that the top-to-bottom function is the OR of the three products above, $f_L = x_1x_4 + x_2x_5 + x_3x_6$, and the left-to-right function is the OR of the four products above, $g_L = x_1x_2x_3 + x_1x_2x_5x_6 + x_2x_3x_4x_5 + x_4x_5x_6$.

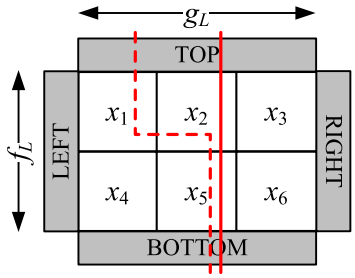


Figure 3.2: A 2×3 lattice with assigned literals.

□

We address the following logic synthesis problem: given a target Boolean function f_T, how should we assign literals to the sites in a lattice such that the top-to-bottom function f_L equals f_T? More specifically, how can we assign literals such that the OR of all the top-to-bottom paths equals f_T? In order to solve this problem we exploit the concept of lattice duality, and work with both the target Boolean function and its dual.

Suppose that we are given a target Boolean function f_T and its dual f_T^D, both in ISOP form such that

$$f_T = P_1 + P_2 + \cdots + P_n \qquad \text{and}$$
$$f_T^D = P_1' + P_2' + \cdots + P_m'$$

where each P_i is a prime implicant of f_T, $i = 1, \ldots n$, and each P'_j is a prime implicant of f_T^D, $j = 1, \ldots m$. [1] We use a set representation for the prime implicants:

$$P_i \rightarrow SP_i, \quad i = 1, 2, \ldots, n$$
$$P'_j \rightarrow SP'_j, \quad j = 1, 2, \ldots, m$$

where each SP_i is the set of literals in the corresponding product P_i and each SP'_j is the set of literals in the corresponding product P'_j.

3.2.1 Algorithm

We first present the synthesis algorithm; then we illustrate it with examples; then we explain why it works.

Above we argued that, in establishing the Boolean function that a lattice implements, we must consider all possible paths. Paradoxically, our method allows us to consider only the *column paths* and the *row paths*, that is to say, the paths formed by straight-line connections between the top and bottom plates and between the left and right plates, respectively. Our algorithm is formulated in terms of the set representation of products and their intersections.

1. Begin with f_T and its dual f_T^D, both in ISOP form. Suppose that f_T and f_T^D have n and m products, respectively.

2. Start with an $m \times n$ lattice. Assign each product of f_T to a column and each product of f_T^D to a row.

3. Compute intersection sets for every site, as shown in Figure 3.3.

4. Arbitrarily select a literal from an intersection set and assign it to the corresponding site.

[1] Here $'$ is used to distinguish symbols. It does *not* indicate negation.

The proposed implementation technique is illustrated in Figure 3.3. The technique implements f_T with an $m \times n$ lattice where n and m are the number of products of f_T and f_T^D, respectively. Each of the n column paths implements a product of f_T and each of the m row paths implements a product of f_T^D. As we explain in the next section, the resulting lattice implements f_T and f_T^D as the top-to-bottom and left-to-right functions, respectively. None of the paths other than the column and row paths need be considered.

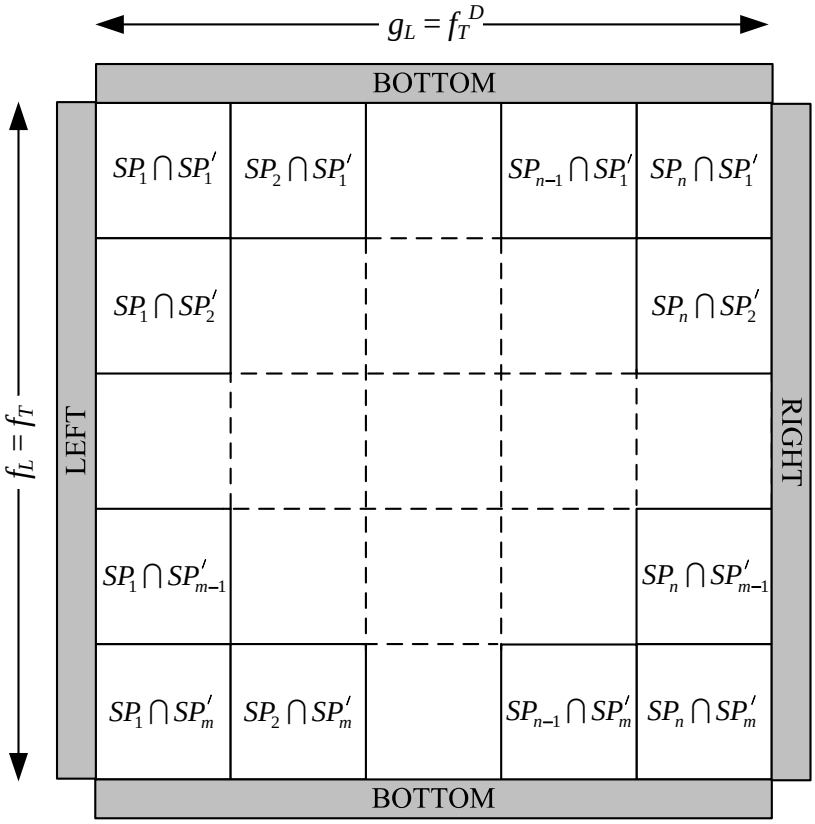


Figure 3.3: Proposed implementation technique: f_T and f_T^D are implemented by top-to-bottom and left-to-right functions of the lattice, respectively.

We present a few examples to elucidate our algorithm.

Example 2

Suppose that we are given the following target function f_T in ISOP form:

$$f_T = x_1x_2 + x_1x_3 + x_2x_3.$$

We compute its dual f_T^D in ISOP form:

$$f_T^D = (x_1 + x_2)(x_1 + x_3)(x_2 + x_3),$$
$$f_T^D = x_1x_2 + x_1x_3 + x_2x_3.$$

We have:

$$SP_1 = \{x_1, x_2\}, \quad SP_2 = \{x_1, x_3\}, \quad SP_3 = \{x_2, x_3\},$$
$$SP_1' = \{x_1, x_2\}, \quad SP_2' = \{x_1, x_3\}, \quad SP_3' = \{x_2, x_3\}.$$

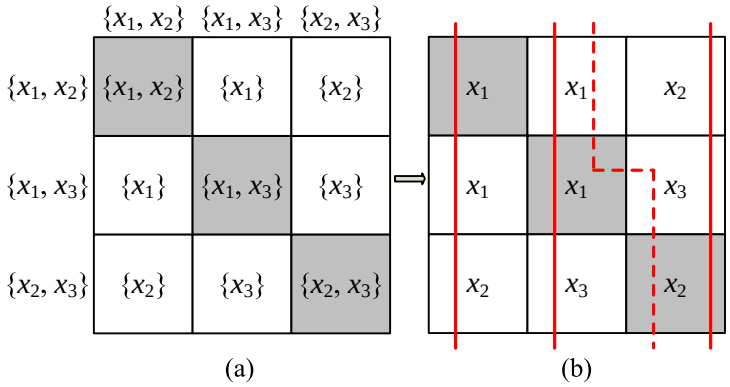


Figure 3.4: Implementing $f_T = x_1x_2+x_1x_3+x_2x_3$. (a): Lattice sites with corresponding sets. (b): Lattice sites with corresponding literals.

Figure 3.4 shows the implementation of the target function. Grey sites represent sets having more than one literal; which literal is selected for these sites is arbitrary. For example, selecting x_2, x_3, x_3 instead of x_1, x_1, x_2 does not change f_L and g_L. In order to implement the target function, we only use column paths; these are shown by the solid lines. All other paths are, in fact, redundant. Indeed there are a total of 9 top-to-bottom

paths: the 3 column paths and 6 other paths; however all other paths are covered by the column paths. For example, the path $x_1x_2x_3$ shown by the dashed line is a redundant path covered by the column paths. The lattice implements the top-to-bottom and left-to-right functions $f_L = f_T = x_1x_2 + x_1x_3 + x_2x_3$ and $g_L = f_T^D = x_1x_2 + x_1x_3 + x_2x_3$, respectively. □

Example 3

Suppose that we are given the following target function f_T in ISOP form:

$$f_T = x_1x_2x_3 + x_1x_4 + x_1x_5.$$

We compute its dual f_T^D in ISOP form:

$$f_T^D = (x_1)(x_2 + x_4 + x_5)(x_3 + x_4 + x_5).$$
$$f_T^D = x_1 + x_2x_4x_5 + x_3x_4x_5.$$

We have:

$$SP_1 = \{x_1, x_2, x_3\}, \quad SP_2 = \{x_1, x_4\}, \quad SP_3 = \{x_1, x_5\},$$
$$SP_1' = \{x_1\}, \quad SP_2' = \{x_2, x_4, x_5\}, \quad SP_3' = \{x_3, x_4, x_5\}.$$

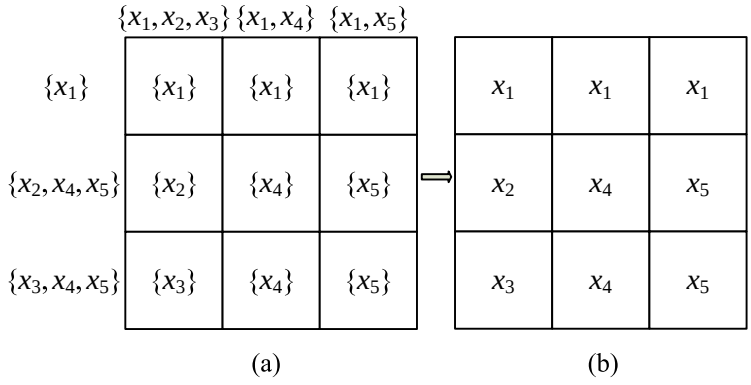


Figure 3.5: Implementing $f_T = x_1x_2x_3 + x_1x_4 + x_1x_5$. (a): Lattice sites with corresponding sets. (b): Lattice sites with corresponding literals.

Figure 3.5 shows the implementation of the target function. In this example, all the intersection sets are singletons, so the choice of which literal to assign is clear. The lattice implements $f_L = f_T = x_1x_2x_3 + x_1x_4 + x_1x_5$ *and* $g_L = f_T^D = x_1 + x_2x_4x_5 + x_3x_4x_5$. □

We give another example, this one somewhat more complicated.

Example 4

Suppose that f_T *and* f_T^D *are both given in ISOP form as follows:*

$$f_T = x_1\bar{x}_2x_3 + x_1\bar{x}_4 + x_2x_3\bar{x}_4 + x_2x_4x_5 + x_3x_5 \qquad \textit{and}$$

$$f_T^D = x_1x_2x_5 + x_1x_3x_4 + x_2x_3\bar{x}_4 + \bar{x}_2\bar{x}_4x_5.$$

	x_1 $\bar{x}_2$ x_3	x_1 $\bar{x}_4$	x_2 x_3 $\bar{x}_4$	x_2 x_4 x_5	x_3 x_5
$x_1\ x_2\ x_5$	x_1	x_1	x_2	x_2	x_5
$x_1\ x_3\ x_4$	x_1	x_1	x_3	x_4	x_3
$x_2\ x_3\ \bar{x}_4$	x_3	$\bar{x}_4$	x_2	x_2	x_3
$\bar{x}_2\ \bar{x}_4\ x_5$	$\bar{x}_2$	$\bar{x}_4$	$\bar{x}_4$	x_5	x_5

Figure 3.6: Implementing $f_T = x_1\bar{x}_2x_3 + x_1\bar{x}_4 + x_2x_2\bar{x}_4 + x_2x_4x_5 + x_3x_5$.

Figure 3.6 shows the implementation of the target function. Grey sites represent intersection sets having more than one literal. For these sites, selection of the final literal is arbitrary. The result is $f_L = f_T = x_1\bar{x}_2x_3 + x_1\bar{x}_4 + x_2x_2\bar{x}_4 + x_2x_4x_5 + x_3x_5$ *and* $g_L = f_T^D = x_1x_2x_5 + x_1x_3x_4 + x_2x_3\bar{x}_4 + \bar{x}_2\bar{x}_4x_5$. □

3.2.2 Proof of Correctness

We present a proof of correctness of the synthesis method. Since our method does not enumerate paths, we must answer the question: for the top-to-bottom lattice function, how do we know that all paths other than the column paths are redundant? The following theorem answers this question. It pertains to the lattice functions and their duals.

Theorem 1

If we can find two dual functions f and f^D that are implemented as subsets of all top-to-bottom and left-to-right paths, respectively, then $f_L = f$ and $g_L = f^D$. □

Before presenting the proof, we provide some examples to elucidate the theorem.

Example 5

We analyze the two lattices shown in Figure 3.7.

Lattice (a): *The top-to-bottom paths shown by the red lines implement $f = x_1x_2 + \bar{x}_1x_3$. The left-to-right paths shown by the blue lines implement $g = x_1x_3 + \bar{x}_1x_2$. Since $g = f^D$, we can apply Theorem 1: $f_L = f = x_1x_2 + \bar{x}_1x_3$ and $g_L = f^D = x_1x_3 + \bar{x}_1x_2$. Relying on the theorem, we obtain the functions* without *examining all possible paths. Let us check the result by using the formal definition of f_L and g_L, namely the OR of all corresponding paths. Since there are 9 total top-to-bottom paths, $f_L = x_1x_1\bar{x}_1 + x_1x_1x_2x_2 + x_1x_1x_2x_3\bar{x}_1 + x_3x_2x_1\bar{x}_1 + x_3x_2x_2 + x_3x_2x_3\bar{x}_1 + x_3x_3\bar{x}_1 + x_3x_3x_2x_2 + x_3x_3x_2x_1\bar{x}_1$, which is equal to $x_1x_2 + \bar{x}_1x_3$. Thus all the top-to-bottom paths but the paths shown by the red lines are redundant. Since there are 9 total left-to-right paths, $g_L = x_1x_3x_3 + x_1x_3x_2x_3 + x_1x_3x_2x_2\bar{x}_1 + x_1x_2x_3x_3 + x_1x_2x_3 + x_1x_2x_2\bar{x}_1 + \bar{x}_1x_2x_2x_3x_3 + \bar{x}_1x_2x_2x_3 + \bar{x}_1x_2\bar{x}_1$, which is equal to $x_1x_3 + \bar{x}_1x_2$. Thus all the left-to-right paths but the paths shown by the blue lines are redundant. So Theorem 1 holds for this example.*

Lattice (b): *The top-to-bottom paths shown by the red lines implement* $f = x_1x_2x_3+x_1x_4+x_1x_5$. *The left-to-right paths shown by the blue lines implement* $g = x_1+x_2x_4x_5+x_3x_4x_5$. *Since* $g = f^D$, *we can apply Theorem 1:* $f_L = f = x_1x_2x_3+x_1x_4+x_1x_5$ *and* $g_L = f^D = x_1 + x_2x_4x_5 + x_3x_4x_5$. *Again, we see that Theorem 1 holds for this example.* □

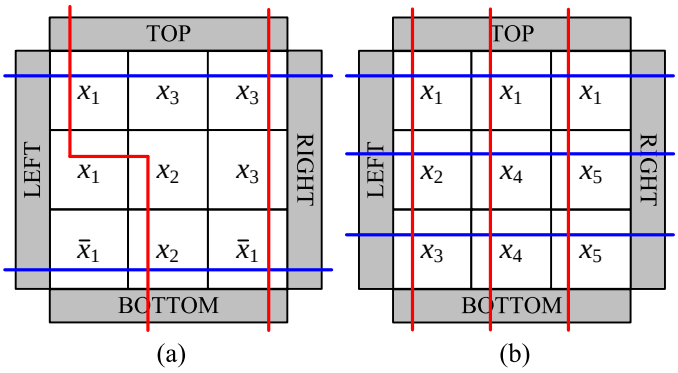


Figure 3.7: Examples to illustrate Theorem 1. (a): $f_L = x_1x_2 + \bar{x}_1x_3$ *and* $g_L = x_1x_2 + \bar{x}_1x_3$. *(b):* $f_L = x_1x_2x_3 + x_1x_4 + x_1x_5$ *and* $g_L = x_1 + x_2x_4x_5 + x_3x_4x_5$.

Proof of Theorem 1: If $f(x_1, x_2, \ldots, x_k) = 1$ then $f_L = 1$. From the definition of duality, if $f(x_1, x_2, \ldots, x_k) = 0$ then $f^D(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_k) = \bar{f}(x_1, x_2, \ldots, x_k) = 1$. This means that there is a left-to-right path consisting of all 0's; accordingly, $f_L = 0$. Thus, we conclude that $f_L = f$. If $f^D(x_1, x_2, \ldots, x_k) = 1$ then $g_L = 1$. From the definition of duality, if $f^D(x_1, x_2, \ldots, x_k) = 0$ then $f(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_k) = \bar{f^D}(x_1, x_2, \ldots, x_k) = 1$. This means that there is a top-to-bottom path consisting of all 0's; accordingly, $g_L = 0$. Thus, we conclude that $g_L = f^D$. □

Theorem 1 provides a constructive method for synthesizing lattices with the requisite property, namely that the top-to-bottom and left-to-right functions f_T and f_T^D are duals, and each column path of the lattice implements a product of f_T and each row path implements a product of f_T^D.

We begin by lining up the products of f_T as the column headings and the products of f_T^D as the row headings. We compute intersection sets for every lattice site. We arbitrarily select a literal from each intersection set and assign it to the corresponding site. The following lemma and theorem explain why we can make such an arbitrary selection.

Suppose that functions $f(x_1, x_2, \ldots, x_k)$ and $f^D(x_1, x_2, \ldots, x_k)$ are both given in ISOP form such that

$$f = P_1 + P_2 + \cdots + P_n \qquad \text{and}$$
$$f^D = P'_1 + P'_2 + \cdots + P'_m$$

where each P_i is a prime implicant of f, $i = 1, \ldots n$, and each P'_j is a prime implicant of f^D, $j = 1, \ldots m$. Again, we use a set representation for the prime implicants:

$$P_i \rightarrow SP_i, \quad i = 1, 2, \ldots, n$$
$$P'_j \rightarrow SP'_j, \quad j = 1, 2, \ldots, m$$

where each SP_i is the set of literals in the corresponding product P_i and each SP'_j is the set of literals in the corresponding product P'_j. Suppose that SP_i and SP'_j have z_i and z'_j elements, respectively. We first present a property of dual Boolean functions from [6]:

Lemma 1

Dual pairs f and f^D must satisfy the condition

$SP_i \cap SP'_j \neq \emptyset$ *for every* $i = 1, 2, \ldots, n$ *and* $j = 1, 2, \ldots, m$. □

Proof of Lemma 1: The proof is by contradiction. Suppose that we focus on one product P_i from f and assign all its literals, namely those in the set SP_i, to 0. In this case $f^D = 0$. However if there is a product P'_j of f^D such that $SP'_j \cap SP_i = \emptyset$, then we can always make P'_j equal 1 because SP'_j does not contain any literals that have been previously assigned to 0. If follows that $f^D = 1$, a contradiction. □

Lemma 2

Consider a product P with a corresponding set representation SP. Consider a Boolean function $f = P_1 + P_2 + \cdots + P_n$ with a corresponding set representation SP_i for each of its products P_i, $i = 1, 2, \ldots, n$. If SP has non-empty intersections with every SP_i, $i = 1, 2, \ldots, n$, then P is a product of f^D. □

Proof of Lemma 2: To prove that P is a product of f^D we assign 1's to all the variables of P and see if this always results in $f^D = 1$. Since SP has non-empty intersections with every SP_i, $i = 1, 2, \ldots, n$, each product of f should have at least one assigned 1. From the definition of duality, these assigned 1's always result in $f^D = (1 + \ldots)(1 + \ldots)\ldots(1 + \ldots) = 1$. □

Theorem 2

Assume that f and f^D are both in ISOP form. For any product P_i of f, there exist m non-empty intersection sets, $(SP_i \cap SP'_1), (SP_i \cap SP'_2), \ldots, (SP_i \cap SP'_m)$. Among these m sets, there must be at least z_i single-element disjoint sets. These single-element sets include all z_i literals of P_i.

We can make the same claim for products of f^D: for any product P'_j of f^D there exist n non-empty intersection sets, $(SP'_j \cap SP_1), (SP'_j \cap SP_2), \ldots, (SP'_j \cap SP_n)$. Among these n sets there must be at least z'_j single-element disjoint sets that each represents one of the z'_j literals of P'_j. □

Before proving the theorem we elucidate it with examples.

Example 6

Suppose that we are given a target function f_T and its dual f_T^D, both in ISOP form such that

$$f_T = x_1\bar{x}_2 + \bar{x}_1x_2x_3 \text{ and } f_T^D = x_1x_2 + x_1x_3 + \bar{x}_1\bar{x}_2.$$

Thus,

$$SP_1 = \{x_1, \bar{x}_2\}, \quad SP_2 = \{\bar{x}_1, x_2, x_3\},$$
$$SP'_1 = \{x_1, x_2\}, \qquad SP'_2 = \{x_1, x_3\}, \qquad SP'_3 = \{\bar{x}_1, \bar{x}_2\}.$$

Let us apply Theorem 2 for SP_2 *(*$z_2 = 3$*).*

$$SP_2 \cap SP'_1 = \{x_2\},\ SP_2 \cap SP'_2 = \{x_3\},\ SP_2 \cap SP'_3 = \{\bar{x}_1\}.$$

Since these three sets are all the single-element disjoint sets of the literals of SP_2*, Theorem 2 is satisfied.* □

Example 7

Suppose that we are given a target function f_T *and its dual* f_T^D*, both in ISOP form such that*

$$f_T = x_1x_2 + x_1x_3 + x_2x_3 \text{ and } f_T^D = x_1x_2 + x_1x_3 + x_2x_3.$$

Thus,

$$SP_1 = \{x_1, x_2\}, \quad SP_2 = \{x_1, x_3\}, \quad SP_3 = \{x_2, x_3\},$$
$$SP'_1 = \{x_1, x_2\}, \quad SP'_2 = \{x_1, x_3\}, \quad SP'_3 = \{x_2, x_3\}.$$

Let us apply Theorem 2 for SP'_1 *(*$z'_1 = 2$*).*

$$SP'_1 \cap SP_1 = \{x_1, x_2\},\ SP'_1 \cap SP_2 = \{x_1\},\ SP'_1 \cap SP_3 = \{x_2\}.$$

Since $\{x_1\}$ *and* $\{x_2\}$*, the single-element disjoint sets of the literals of* SP'_1*, are among these sets, Theorem 2 is satisfied.* □

Proof of Theorem 2: The proof is by contradiction. Consider a product P_i of f such that $SP_i = \{x_1, x_2, \ldots, x_{z_i}\}$. From Lemma 1 we know that SP_i has non-empty intersections with every SP'_j, $j = 1, 2, \ldots, m$. For one of the elements of SP_i, say x_1, assume that none of the intersection sets $(SP_i \cap SP'_1), (SP_i \cap SP'_2), \ldots, (SP_i \cap SP'_m)$ is

$\{x_1\}$. This means that if we extract x_1 from SP_i then the new set $\{x_2, \ldots, x_{z_i}\}$ also has non-empty intersections with every SP'_j, $j = 1, 2, \ldots, m$. From Lemma 2 we know that the product $x_2 x_3 \ldots x_{z_i}$ must be a product of f. This product covers P_i. However, in an ISOP expression, all products including P_i are irredundant, not covered by a product of f. So we have a contradiction. □

From Lemma 1 we know that none of the lattice sites will have an empty intersection set. Theorem 2 states that the intersection sets of a product include single-element sets for *all* of its literals. So the corresponding column or row has always all literals of the product regardless of the final literal selections from multiple-element sets. Thus we obtain a lattice whose column paths and row paths implement f_T and f_T^D, respectively.

In the next section, we present a method to implement a specific function, the parity function.

3.3 Parity Functions

The algorithm proposed in Section 3.2 provides a general method for implementing any type of Boolean function with an $m \times n$ lattice, where n and m are the number of products of the function and its dual, respectively. In this section, we discuss a method for implementing a specific function, the *parity function*, with a $(log(m)+1) \times n$ lattice. Compared to the general method, we improve the lattice size by a factor of $m/(log(m)+1)$ for this function.

As defined in Section 3.1, a k-variable parity function can be represented as a k-variable XOR operation. We exploit the following properties of XOR functions:

$$XOR_k = x_k \overline{XOR}_{k-1} + \overline{x}_k XOR_{k-1}$$
$$\overline{XOR}_k = x_k XOR_{k-1} + \overline{x}_k \overline{XOR}_{k-1}.$$

These properties allow us to implement both XOR_k and its complement $\overline{XOR}_k$ recursively. The approach for the k-variable parity function is illustrated in Figure 3.8.

The approach for 1, 2, and 3 variable parity functions is shown in Figure 3.9. As in our general method, we implement each product of the target function with a separate column path; in this construction, all paths other than column paths are redundant. The following lemma explains why this configuration works. Figure 3.10 illustrates the lemma.

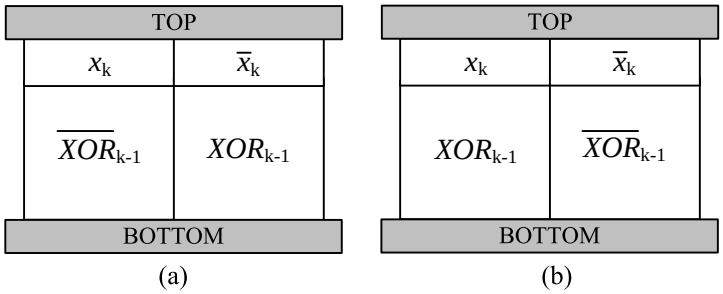


Figure 3.8: (a): Implementation of XOR_k. (b): Implementation of $\overline{XOR}_k$.

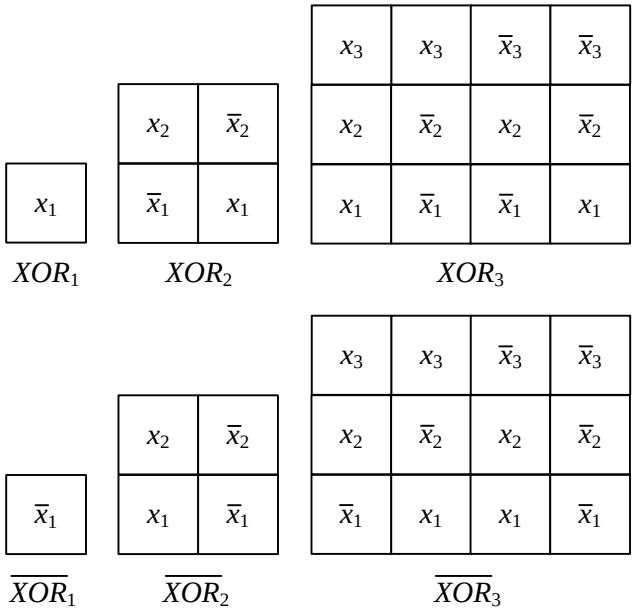


Figure 3.9: Implementation of XOR_1, $\overline{XOR}_1$, XOR_2, $\overline{XOR}_2$, XOR_3, and $\overline{XOR}_3$.

Lemma 3

Consider two lattices with the same number of rows. Suppose that the lattices implement the Boolean functions f_{L1} and f_{L2}. Construct a new lattice with the two lattices side by side. If the attached columns of the lattices have negated variables facing each other for all rows except the first and the last, then the new lattice implements the Boolean function $f_{L3} = f_{L1} + f_{L2}$. □

Proof of Lemma 3: The new lattice has three types of top-to-bottom paths: paths having all sites from the first lattice that implement f_{L1}, paths having all sites from the second lattice that implement f_{L2}, and paths having sites from both the first and the second lattices that implement f_{L1-2}. The Boolean function f_{L3} implemented by the third lattice is OR of the all paths; $f_{L3} = f_{L1} + f_{L2} + f_{L1-2}$. The paths having sites from both the first and the second lattices should cross the attached columns. This means that such paths both include a variable and its negation; negated variables in attached columns result in $f_{L1-2} = 0$. We conclude that $f_{L3} = f_{L1} + f_{L2}$. □

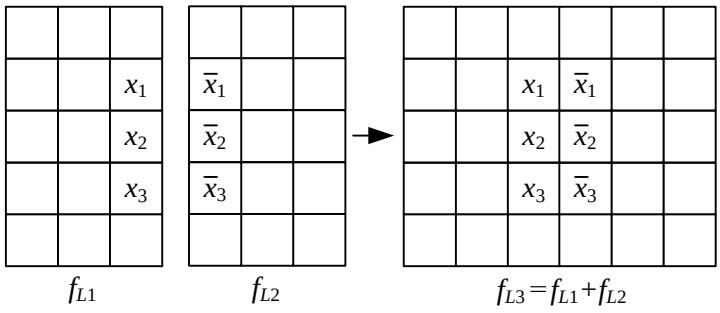


Figure 3.10: An example illustrating Lemma 3: from left to right, the top-to-bottom Boolean functions of the lattices are f_{L1}, f_{L2}, and $f_{L1} + f_{L2}$.

We exploit Lemma 3 to compute the parity function as follows (please refer back to Figure 3.8). We attach the lattices implementing $f_{L1} = x_k\overline{XOR}_{k-1}$ and $f_{L2} = \overline{x}_k XOR_{k-1}$ to implement $f_{L3} = XOR_k$. We attach the lattices implementing $f_{L1} = x_k XOR_{k-1}$ and $f_{L2} = \overline{x}_k\overline{XOR}_{k-1}$ to implement $f_{L3} = \overline{XOR}_k$. One can easily see that

attached columns always have the proper configuration of negated variables to ensure that $f_{L1-2} = 0$.

3.4 A Lower Bound on the Lattice Size

In this section, we propose a lower bound on the size of any lattice implementing a Boolean function. Although it is a weak lower bound, it allows us to gauge the effectiveness of our synthesis method. The bound is predicated on the maximum length of any path across the lattice. The length of such a path is bounded from below by the maximum number of literals in terms of an ISOP expression for the function.

We present preliminaries including definitions that are used throughout this section as follows.

3.4.1 Preliminaries

Definition 7

Let the **degree** *of an SOP expression be the maximum number of literals in terms of the expression.* □

A Boolean function might have several different ISOP expressions and these might have different degrees. Among all the different expressions, we need the one with the smallest degree for our lower bound. (We need only consider ISOP expressions; every SOP expression is covered by an ISOP expression of equal or lesser degree.)

Consider a target Boolean function f_T and its dual f_T^D, both in ISOP form. We will use v and y to denote the minimum degrees of f_T and f_T^D, respectively. For example, if $v = 3$ and $y = 5$, this means that every ISOP expression for f_T includes terms with 3 literals or more, and every ISOP expression for f_T^D includes terms with 5 literals or more. Our lower bound, described in the next section by Theorem 4, consists of inequalities on v and y. We first illustrate how it works with an example.

Example 8

Consider two target Boolean functions $f_{T1} = x_1x_2x_3 + x_1x_4 + x_1x_5$ *and* $f_{T2} = x_1x_2x_3 + \bar{x}_1\bar{x}_2x_4 + x_2x_3x_4$, *and their duals* $f_{T1}^D = x_1 + x_2x_4x_5 + x_3x_4x_5$ *and* $f_{T2}^D = x_1x_4 + \bar{x}_1x_2 + \bar{x}_2x_3$. *These expressions are all in ISOP form with minimum degrees. Since each expressions consists of three products, the synthesis method described in Section 3.2 implements each target function with a* 3×3 *lattice.*

Examining the expressions, we see that the degrees of f_{T1} *and* f_{T2} *are* $v_1 = 3$ *and* $v_2 = 3$, *respectively, and the degrees of* f_{T1}^D *and* f_{T2}^D *are* $y_1 = 3$ *and* $y_2 = 2$, *respectively. Our lower bounds based on these values are* 3×3 *for* f_{T1} *and* 3×2 *for* f_{T2}. *Thus, the lower bound for* f_{T2} *suggests that our synthesis method might not be producing optimal results. Indeed, Figure 3.11 shows minimum-sized lattices for for* f_{T1} *and* f_{T2}. *Here the* 3×2 *lattice for* f_{T2} *was obtained through exhaustive search.* □

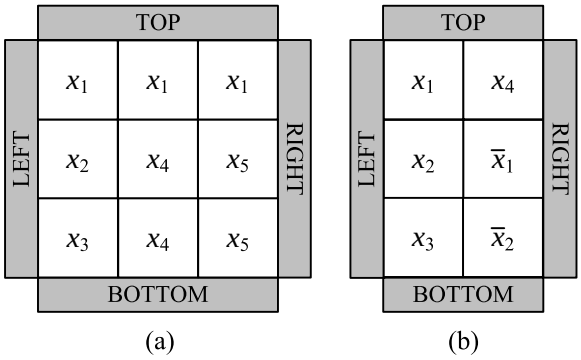


Figure 3.11: Minimum-sized lattices (a): $f_L = f_{T1} = x_1x_2x_3 + x_1x_4 + x_1x_5$. (b): $f_L = f_{T2} = x_1x_2x_3 + \bar{x}_1\bar{x}_2x_4 + x_2x_3x_4$.

Since we implement Boolean functions in terms of top-to-bottom connectivity across the lattice, it is apparent that we cannot implement a target function f_T with top-to-bottom paths consisting of fewer than v literals, where v is the minimum degree of an

ISOP expression for f_T. The following theorem explains the role of y, the minimum degree of f_T^D. It is based on *eight-connected* paths.[2]

Definition 8

An **eight-connected path** *consists of both directly and diagonally adjacent sites.* □

An example is shown in Figure 3.12. Here the paths $x_1x_4x_8$ and $x_3x_6x_5x_8$ shown by red and blue lines are both eight-connected paths; however only the blue one is four-connected.

Recall that f_L and g_L are defined as the OR of all four-connected top-to-bottom and left-to-right paths, respectively. (A lattice implements a given target function f_T if $f_L = f_T$.) We define f_{L-8} and g_{L-8} to be the OR of all eight-connected top-to-bottom and left-to-right paths, respectively.

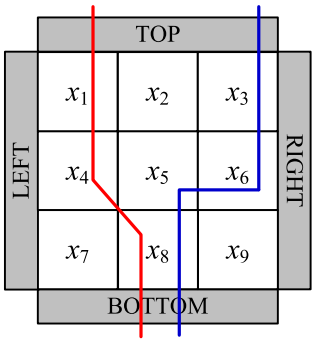


Figure 3.12: A lattice with eight-connected paths.

Theorem 3

The functions f_L and g_{L-8} are duals. The functions f_{L-8} and g_L duals. □

Before proving the theorem, we elucidate it with an example.

[2] Note that because our synthesis methodology is based on lattices of four-terminal switches, the target function f_T is always implemented by four-connected paths. We discuss eight-connected paths only because it is helpful to do so in order to prove our lower bound.

Example 9

Consider the lattice shown in Figure 3.13. Here f_L is the OR of 3 top-to-bottom four-connected paths x_1x_4, x_2x_5, and x_3x_6; g_L is the OR of 4 left-to-right four-connected paths $x_1x_2x_3$, $x_1x_2x_5x_6$, $x_4x_5x_2x_3$, and $x_4x_5x_6$; f_{L-8} is the OR of 7 eight-connected top-to-bottom paths x_1x_4, x_1x_5, x_2x_4, x_2x_5, x_2x_6, x_3x_5, and x_3x_6; and g_{L-8} is the OR of 8 eight-connected left-to-right paths $x_1x_2x_3$, $x_1x_2x_6$, $x_1x_5x_3$, $x_1x_5x_6$, $x_4x_2x_3$, $x_4x_2x_6$, $x_4x_5x_3$, and $x_4x_5x_6$. We can easily verify that $f_L = g^D_{L-8}$ and $f_{L-8} = g^D_L$. Accordingly, Theorem 3 holds true for this example.

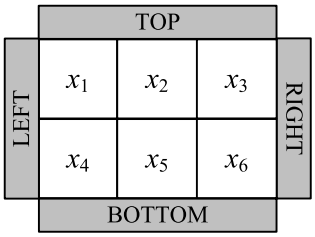


Figure 3.13: A 2×3 lattice with assigned literals.

□

Proof of Theorem 3: We consider two cases, namely $f_L = 1$ and $f_L = 0$.

Case 1: If $f_L(x_1, x_2, \ldots, x_k) = 1$, there must be a four-connected path of 1's between the top and bottom plates. If we complement all the inputs $(1 \to 0, 0 \to 1)$, these four-connected 1's become 0's and vertically separate the lattice into two parts. Therefore no eight-connected path of 1's exists between the left and right plates; accordingly, $g_{L-8}(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_k) = 0$. As a result $\bar{g}_{L-8}(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_k) = f_L(x_1, x_2, \ldots, x_k) = 1$

Case 2: If $f_L(x_1, x_2, \ldots, x_k) = 0$, there must be an eight-connected path of 0's between the left and right plates. If we complement all the inputs, these eight-connected 0's become 1's; accordingly, $g_{L-8}(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_k) = 1$. As a result $\bar{g}_{L-8}(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_k) =$

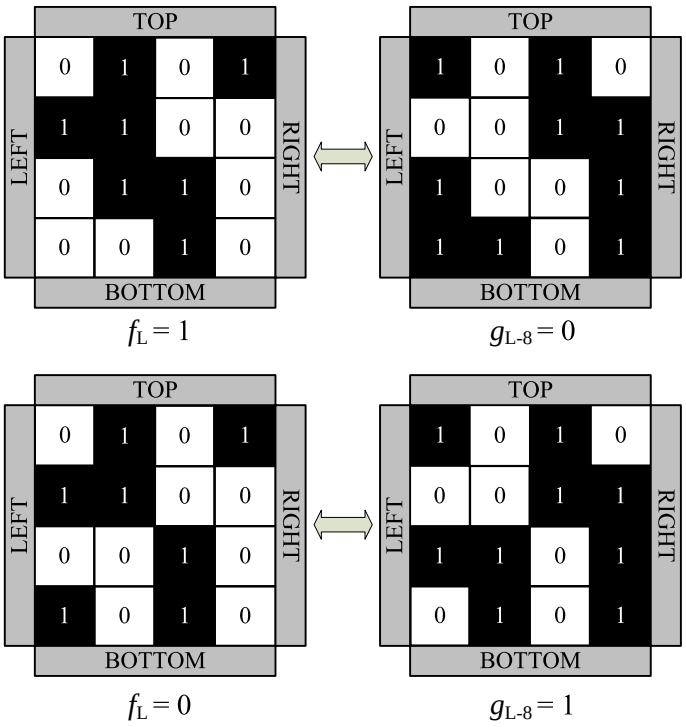


Figure 3.14: Conceptual proof of Theorem 3.

$f_L(x_1, x_2, \ldots, x_k) = 0$

Figure 3.14 illustrates the two cases. Taken together, the two cases prove that f_L and g_{L-8} are duals. With inverse reasoning we can prove that f_{L-8} and g_L are duals. □

Theorem 3 tells us that the products of f_T^D are implemented with eight-connected left-to-right paths. Now consider y, the degree of f_T^D. We know that we cannot implement f_T^D with eight-connected right-to-left paths having fewer than y literals. Consider v, the degree of f_T. We know that we cannot implement f_T with four-connected top-to-bottom paths having fewer than v literals.

Returning to the functions in Example 8, we can now prove that lower bounds on the lattice sizes are 9 (3×3) for f_{T1}, and 6 (3×2) for f_{T2}. Since $v_1 = 3$ and $y_1 = 3$ for f_{T1}, a

3×3 lattice is a minimum-size lattice that has four-connected top-to-bottom and eight-connected left-to-right paths of at least 3 literals, respectively. Since $v_2 = 3$ and $y_2 = 2$ for f_{T2}, a 3×2 lattice is a minimum-size lattice that has four-connected top-to-bottom and eight-connected left-to-right paths of at least 3 and 2 literals, respectively.

Based on these preliminaries, we now formulate the lower bound.

3.4.2 Lower Bound

Consider a target Boolean function f_T and its dual f_T^D, both in ISOP form. Recall that v and y are defined as the minimum degrees of f_T and f_T^D, respectively. Our lower bound is based on the observation that a minimum-size lattice must have a four-connected top-to-bottom path with at least v literals and an eight-connected left-to-right path with at least y literals. Since the functions are in ISOP form, all products of f_T and f_T^D are irredundant, i.e., not covered by other products. Therefore, we need only to consider irredundant paths:

Definition 9

A four-connected (eight-connected) path between plates is **irredundant** *if it is not covered by another four-connected (eight-connected) path between the corresponding plates.* □

We bound the length of irredundant paths. For example, the length of an eight-connected left-to-right path in a 3×3 lattice is at most 3. Accordingly, no Boolean function with y greater than 3 can be implemented by a 3×3 lattice. Figure 3.15 shows eight-connected left-to-right paths in a 3×3 lattice. The path in (a) consists of 3 sites. The path in (b) consists of 4 sites; however it is a redundant path – it is covered by the path in (a).

The following simple lemmas pertain to irredundant paths of a lattice.

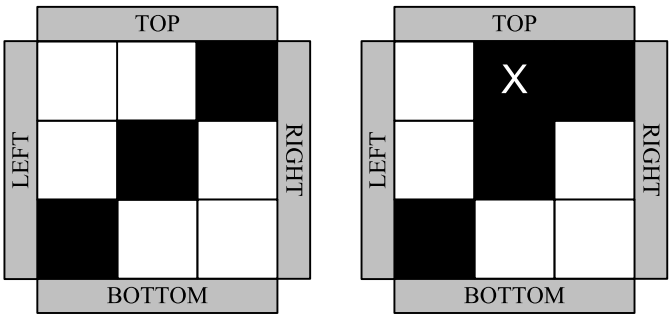


Figure 3.15: Lattices with (a) an irredundant path and (b) a redundant path. The site marked with $\times$ is redundant.

Lemma 4

An irredundant top-to-bottom path of a lattice contains exactly one site from the top-most row and exactly one site from the bottommost row. An irredundant left-to-right path of a lattice contains exactly one site from the leftmost column and exactly one site from the rightmost column. □

Proof of Lemma 4: All sites in the first row of a lattice are connected through the top plate. Therefore we do not need a path to connect any two sites in this row; such a path is redundant. Similarly for the last row. Similarly for the first and last columns. □

Lemma 5

An irredundant four-connected path of a lattice contains at most 3 of 4 sites in any 2×2 *sub-lattice. An irredundant eight-connected path of a lattice contains at most 2 of 4 sites in any* 2×2 *sub-lattice.* □

Proof of Lemma 5: In order to connect any 2 sites of a 2×2 sub-lattice with a four-connected path, we need at most 3 sites of the sub-lattice. Similarly, in order to connect any 2 sites of a 2×2 sub-lattice with an eight-connected path, we need at most 2 sites of the sub-lattice. □

Figure 3.16 shows examples illustrating Lemma 5. The lattice in (a) has a four-connected top-to-bottom path. This path contains 4 of the 4 sites in the 2×2 sub-lattice encircled in red. Lemma 5 tells us that the path in (a) is redundant. Indeed, it is covered by the path achieved by removing the site marked by $\times$. The lattice in (b) has an eight-connected left-to-right path. This path contains 3 of 4 sites in the 2×2 sub-lattice encircled in red. Lemma 5 tells us that the path in in (b) is redundant. Indeed it is covered by the path achieved by removing the site marked by $\times$.

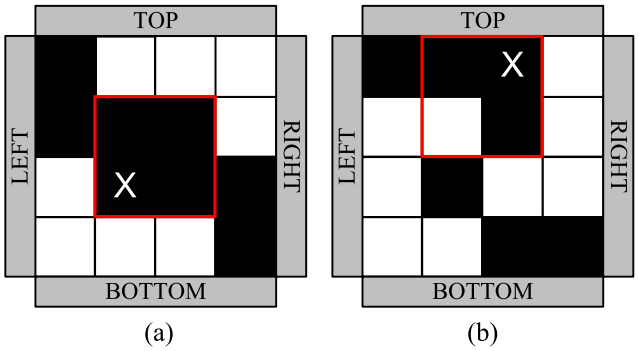


Figure 3.16: Examples to illustrate Lemma 5. (a): a four-connected path with a redundant site marked with $\times$. (b): an eight-connected path with a redundant site marked with $\times$.

From Lemmas 4 and 5, we have the following theorem consisting of two inequalities. The first inequality states that the degree of f_T is equal to or less than the maximum number of sites in any four-connected top-to-bottom path. The second inequality states that the degree of f_T^D is less than or equal to the maximum number of sites in any eight-connected left-to-right path.

Theorem 4

If a target Boolean function f_T is implemented by an $R \times C$ lattice then the following inequalities must be satisfied:

$$v \leq \begin{cases} R, & \text{if } R \leq 2 \text{ or } C \leq 1 \\ 3\left\lceil \frac{R-2}{2} \right\rceil \left\lceil \frac{C}{2} \right\rceil + \frac{2+(-1)^R+(-1)^C}{2}, & \text{if } R > 2 \text{ and } C > 1, \end{cases}$$

$$y \leq \begin{cases} C, & \text{if } R \leq 3 \text{ or } C \leq 2 \\ 2\left\lceil \frac{R}{2} \right\rceil \left\lceil \frac{C-2}{2} \right\rceil + \frac{2+(-1)^R+(-1)^C}{2}, & \text{if } R > 3 \text{ and } C > 2, \end{cases}$$

where v and y are the minimum degrees of f_T and its dual f_T^D, respectively, both in ISOP form. □

Proof of Theorem 4: If R and C are both even then all irredundant top-to-bottom and left-to-right paths contain at most $\frac{3}{4}(R-2)C+2$ and $\frac{2}{4}R(C-2)+2$ sites, respectively; this follows directly from Lemmas 4 and 5. If R or C are odd then we first round these up to the nearest even number. The resulting lattice contains at least one extra site (if either R or C but not both are odd) or two extra sites (if both R and C are odd). Accordingly, we compute the maximum number of sites in top-to-bottom and left-to-right paths and subtract 1 or 2. This calculation is reflected in the inequalities. □

The theorem proves our lower bound. Table 3.1 shows the calculation of the bound for different values of v and y up to 10.

3.5 Experimental Results

In Table 3.2 and Table 3.3 we report synthesis results for a few standard benchmark circuits [21]. We treat each output of a benchmark circuit as a separate target function.

The values for n and m represent the number of products for each target function f_T and its dual f_T^D, respectively. We obtained these through sum-of-products minimization using the program Espresso [22]. The lattice size, representing the number of switches, is computed as a product of n and m.

Table 3.1: Lower bounds on the lattice size for different values of v and y the minimum degrees.

v \ y	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	12	15	20	20	20	24
4	4	6	9	12	12	15	20	20	20	24
5	5	8	9	12	12	15	20	20	20	24
6	6	9	9	12	12	15	20	20	20	24
7	7	10	12	12	12	15	20	20	20	24
8	8	12	15	15	15	15	20	20	20	24
9	9	14	15	15	15	15	20	20	20	24
10	10	14	15	15	15	15	20	20	20	24

For the lower bound calculation, we obtained values of v and y, the minimum degrees of f_T and f_T^D, as follows: first we generated prime implicant tables for the target functions and their duals using Espresso with the "`-Dprimes`" option; then we deleted prime implicants one by one, beginning with those that had the most literals, until we obtained an expression of minimum degree. Given values of v and y, we computed the lower bound from the inequalities in Theorem 4.

Table 3.2 and Table 3.3 list the runtimes for the lattice size and the lower bound calculations. The runtimes for the lattice size consist of the time for obtaining the functions' duals and for SOP minimization of both the functions and their duals. The runtimes for the lower bound consist of the time for generating the prime implicant tables and for obtaining the minimum degrees from the tables. We performed trials on an AMD Athlon 64 X2 6000+ Processor (at 3Ghz) with 3.6GB of RAM running Linux.

Examining the numbers in Table 3.2 and Table 3.3, we see that, often, the synthesized lattice size matches the lower bound. In these cases, our results are optimal. However for most of the Boolean functions, especially those with larger values of n and m, the lower bound is much smaller than the synthesized lattice size. This is not surprising since the lower bound is weak, formulated based on path lengths.

In the final columns of Table 3.2 and Table 3.3, we list the number of transistors required in a CMOS implementation of the functions. We obtained the transistor counts through synthesis trials with the Berkeley tool ABC [23]. We applied the standard synthesis script "resyn2" in ABC and then mapped to a generic library consisting of NAND2 gates and inverters. We assume that each NAND2 gate requires four transistors and each inverter requires two transistors.

The number of switches needed by our method compares very favorably to the number of transistors required in a CMOS implementation. Of course, a rigorous comparison would depend on the specifics of the types of technology used. Each four-terminal switch might equate to more than one transistor. Then again, in some nanoscale technologies, it might equate to much less: the density of crosspoints in nanowire arrays is generally much higher than the density achievable with CMOS transistors.

3.6 Discussion

The two-terminal switch model is fundamental and ubiquitous in electrical engineering [24]. Either implicitly or explicitly, nearly all logic synthesis methods target circuits built from independently controllable two-terminal switches (i.e., transistors). And yet, with the advent of novel nanoscale technologies, synthesis methods targeting lattices of multi-terminal switches are *apropos*. Our model consists of a regular lattice of four-terminal switches.

Our treatment is at a technology-independent level; nevertheless we comment that our synthesis results are applicable to a range of emerging technologies, including

Table 3.2: Proposed lattice sizes, lower bounds on the lattice sizes, and CMOS transistor counts for standard benchmark circuits. Each row lists the numbers for a separate output function of the benchmark circuit.

Circuit	n	m	**Lattice size**	**Run time**	v	y	**Lower bound**	**Run time**	**CMOS circuit size**
alu1	3	2	6		2	3	6		18
alu1	2	3	6	< 0.01	3	2	6	0.02	26
alu1	1	3	3		3	1	3		16
clpl	4	4	16		4	4	12		42
clpl	3	3	9		3	3	9		26
clpl	2	2	4	< 0.01	2	2	4	0.01	10
clpl	6	6	36		6	6	15		74
clpl	5	5	25		5	5	12		64
newtag	8	4	32	< 0.01	3	6	15	< 0.01	60
dc1	4	4	16		3	3	9		38
dc1	2	3	6		3	2	6		24
dc1	4	4	16	< 0.01	3	4	12	< 0.01	40
dc1	4	5	20		4	3	9		42
dc1	3	3	9		2	3	6		26
misex1	2	5	10		4	2	6		64
misex1	5	7	35		4	4	12		84
misex1	5	8	40		5	4	12		64
misex1	4	7	28	< 0.01	5	3	9	0.01	58
misex1	5	5	25		4	4	12		76
misex1	6	7	42		4	4	12		64
misex1	5	7	35		4	3	9		36
b12	4	6	24		4	3	9		50
b12	7	5	35		4	4	12		54
b12	7	6	42		5	4	12		70
b12	4	2	8		2	2	4		16
b12	4	2	8	0.01	2	4	8	0.41	28
b12	5	1	5		1	5	5		30
b12	9	6	54		6	4	12		332
b12	6	4	24		4	6	15		60
b12	7	2	14		2	7	14		62
newbyte	1	5	5	< 0.01	5	1	5	< 0.01	26
c17	3	3	9	< 0.01	2	3	6	< 0.01	16
c17	4	2	8		2	2	4		18

Table 3.3: Proposed lattice sizes, lower bounds on the lattice sizes, and CMOS transistor counts for standard benchmark circuits. Each row lists the numbers for a separate output function of the benchmark circuit.

Circuit	n	m	**Lattice size**	**Run time**	v	y	**Lower bound**	**Run time**	**CMOS circuit size**
ex5	1	3	3		3	1	3		16
ex5	1	5	5		5	1	5		24
ex5	1	4	4		4	1	4		18
ex5	1	7	7		7	1	7		36
ex5	1	8	8		8	1	8		40
ex5	1	6	6		6	1	6		34
ex5	8	4	32		3	6	15		46
ex5	10	4	40		3	8	20		52
ex5	7	3	21		3	7	20		44
ex5	7	3	21		3	6	15		48
ex5	8	2	16		2	8	16		42
ex5	9	4	36		3	8	20		56
ex5	8	2	16	0.26	2	7	14	3.17	42
ex5	12	6	72		4	7	20		70
ex5	14	8	112		4	7	20		388
ex5	7	2	14		2	7	14		38
ex5	6	3	18		3	6	15		40
ex5	6	2	12		2	6	12		36
ex5	10	7	70		3	7	20		76
ex5	6	6	36		3	6	15		64
ex5	12	10	120		4	8	20		318
ex5	14	8	112		5	7	20		350
ex5	8	5	40		3	7	20		86
ex5	10	8	80		3	7	20		116
ex5	12	7	84		4	7	20		356
ex5	9	3	27		3	8	20		60
ex5	5	2	10		2	5	10		44
mp2d	11	1	11		1	11	11		46
mp2d	8	6	48		5	8	20		82
mp2d	10	5	50		4	10	24		102
mp2d	6	10	60	0.01	9	3	15	0.61	318
mp2d	1	5	5		5	1	5		26
mp2d	3	6	18		5	2	8		46
mp2d	1	8	8		8	1	8		36
mp2d	5	1	5		1	5	5		28

nanowire crossbar arrays [16, 2] and magnetic switch-based structures [4] with independently controllable crosspoints. We are investigating its applicability to DNA nanofabrics [25, 26].

In this chapter, we present a synthesis method targeting regular lattices of four-terminal switches. Significantly, our method assigns literals to lattice sites without enumerating paths. It produces lattice sizes that are linear in the number of products of the target Boolean function. The time complexity of our synthesis algorithm is polynomial in the number of products. Comparing our results to a lower bound, we conclude that the synthesis results are not optimal. However, this is hardly surprising: at their core, most logic synthesis problems are computationally intractable; the solutions that are available are based on heuristics. Furthermore, good 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 [27].)

The results on benchmarks illustrate that our method is effective for Boolean functions of practical interest. We should note, however, we would not expect it to be effective on some specific types of Boolean functions. In particular, our method will not be effective for Boolean functions such that the functions' duals have much more products than the functions do have.

The lattices for such functions will be inordinately large. For example, consider the function $f = x_1x_2x_3 + x_4x_5x_6 + x_7x_8x_9$. It has only three products, but its dual has $3^3 = 27$ products. With our method, a lattice with 27 rows and 3 columns would be required. The cost of implementing such functions could be mitigated by decomposing and implementing Boolean functions with separate lattices (or physically separated regions in a single lattice). For example, $f_T = x_1x_2x_3 + x_4x_5x_6$ can be implemented by two lattices each of which is for each product, so the target function is implemented by two 3×1 lattices. An apparent disadvantage of this technique is the necessity of using

multiple lattices rather than a single lattice to implement a target function. We do not get into further details of the technique of decomposition and sharing; Techniques for functional decomposition are well established [22, 28]. We would like to keep it as a future work.

Another future direction is to extend the results to lattices of eight-terminal switches, and then to 2^k-terminal switches, for arbitrary k.

Another direction is to study methods for synthesizing robust computation in lattices with *random connectivity*. In Chapter 4, we explore methods based on the principle of *percolation* [29].

A significant tangent for this work is its mathematical contribution: lattice-based implementations present a novel view of the properties of Boolean functions. In Chapter 5, we study the applicability of these properties to the famous problem of testing whether two monotone Boolean functions in ISOP form are dual. This is one of the few problems in circuit complexity whose precise tractability status is unknown [8].

Chapter 4

Robust Computation Through Percolation

In Chapter 3, we discuss strategies for implementing Boolean functions with lattices of four-terminal switches. We address the synthesis problem of how best to assign literals to switches in a lattice in order to implement a given target Boolean function, with the goal of minimizing the lattice size, measured in terms of the number of switches.

In this chapter, we address the problem of implementing Boolean functions with lattices of four-terminal switches in the presence of defects. We assume that such defects occur probabilistically. Our approach is predicated on the mathematical phenomenon of percolation. With random connectivity, percolation gives rise to a sharp non-linearity in the probability of global connectivity as a function of the probability of local connectivity. We exploit this phenomenon to compute Boolean functions robustly, within prescribed error margins.

This chapter is organized as follows. In Section 4.1, we discuss our defect model. In Section 4.1.1, we discuss the mathematics of percolation and how this phenomenon can be exploited for tolerating defects. In Section 4.2, we present our main technical result: a method for assigning Boolean literals to sites in a switching lattice that optimizes the

lattice area while meeting prescribed defect tolerances. In Section 4.3, we evaluate our method on benchmark circuits.

4.1 Defect Model

We assume that defects cause switches to fail in one of two ways: they are ON when they are supposed to be OFF (OFF-to-ON defect), i.e., the controlling literal is 0; or they are OFF when they are supposed to be ON (ON-to-OFF defect), i.e., the controlling literal is 1. We allow for different defect rates in both directions, ON-to-OFF and OFF-to-ON. For example, if a switch has a larger OFF-to-ON defect rate than its ON-to-OFF defect rate then the switch works more accurately when its controlling input is 1 (the switch is ON). Crucially, we assume that all switches of the lattice fail with *independent* probability.

Defective switches can ruin the Boolean computation performed by a network. Consider the network in Figure 4.1. White and black sites represent OFF and ON switches, respectively. If $x_1 = 1$, each four-terminal switch is ideally ON and represented by a black site. If $x_1 = 0$, each four-terminal switch is ideally OFF and represented by a white site. Due to defects, not all switches will behave in this way. Defective switches are represented by white and black sites while the switch is supposed to be ON and OFF, respectively. This is illustrated in Figure 4.1. Note that in spite of defects, the network in Figure 4.1 computes correctly for both the cases $x_1 = 0$ and $x_1 = 1$.

4.1.1 Percolation

Percolation theory is a rich mathematical topic that forms the basis of explanations of physical phenomena such as diffusion and phase changes in materials. It tells us that in media with random local connectivity, there is a critical threshold for global connectivity: below the threshold, the probability of global connectivity quickly drops to zero; above it, the probability quickly rises to one.

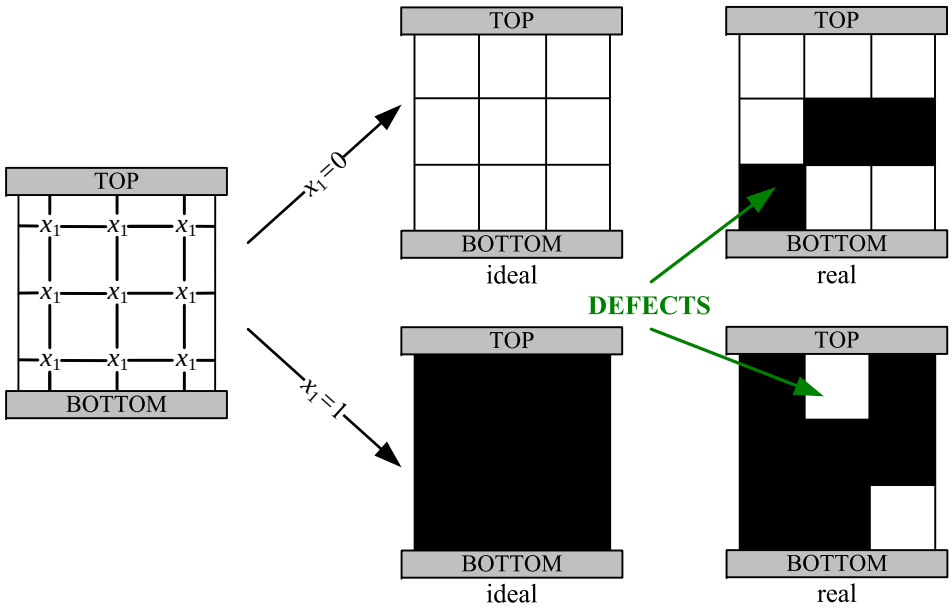
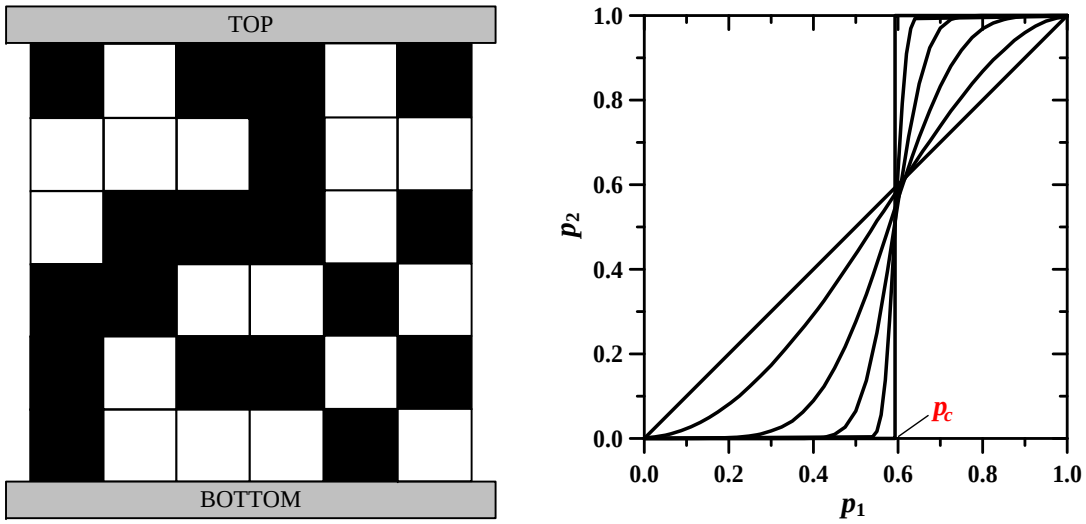


Figure 4.1: Switching network with defects.

Broadbent and Hammersley described percolation with the following metaphorical model [30]. Suppose that water is poured on top of a large porous rock. Will the water find its way through holes in the rock to reach the bottom? We can model the rock as a collection of small regions each of which is either a hole or not a hole. Suppose that each region is a hole with independent probability p_1 and not a hole with probability $1 - p_1$. The theory tells us that if p_1 is above a critical value p_c, the water will always reach the bottom; if p_1 is below p_c, the water will never reach the bottom. The transition in the probability of water reaching bottom as a function of increasing p_1 is extremely abrupt. For an infinite size rock, it is a step function from 0 to 1 at p_c.

In two dimensions, percolation theory can be studied with a lattice, as shown in Figure 4.2(a). Here each site is black with probability p_1 and white with probability $1 - p_1$. Let p_2 be the probability that a connected path of black sites exists between the top and bottom plates. Figure 4.2(b) shows the relationship between p_1 and p_2 for different square lattice sizes. Percolation theory tells us that with increasing lattice size,

the steepness of the curve increases. (In the limit, an infinite lattice produces a perfect step function.) Below the critical probability p_c, p_2 is approximately 0 and above it p_2 is approximately 1.



(a) (b)

Figure 4.2: (a): Percolation lattice with random connections; there is a path of black sites between the top and bottom plates. (b) p_2 versus p_1 for 1×1, 2×2, 6×6, 24×24, 120×120, and infinite-size lattices.

Suppose that each site of a percolation lattice is a four-terminal switch controlled by the same literal x_1. Also suppose that each switch is independently defective with the same probability. Defective switches are represented by white and black sites while the switch is supposed to be ON and OFF, respectively. Let's analyze the cases $x_1 = 0$ and $x_1 = 1$. If $x_1 = 0$ then each site is black with the defect probability, and the defective black sites might cause an error by forming a path between the top and bottom plates. In this case, p_1 and p_2 described in the percolation model correspond to the defect probability and the probability of an error in top-to-bottom connectivity, respectively. If $x_1 = 1$ then each site is white with the defect probability and the defective white sites might cause an error by destroying the connection between the top and bottom plates.

In this case, p_1 and p_2 in the percolation model correspond to $1-$(defect probability) and $1-$(probability of an error in top-to-bottom connectivity), respectively. The relationship between p_1 and p_2 is shown in Figure 4.3.

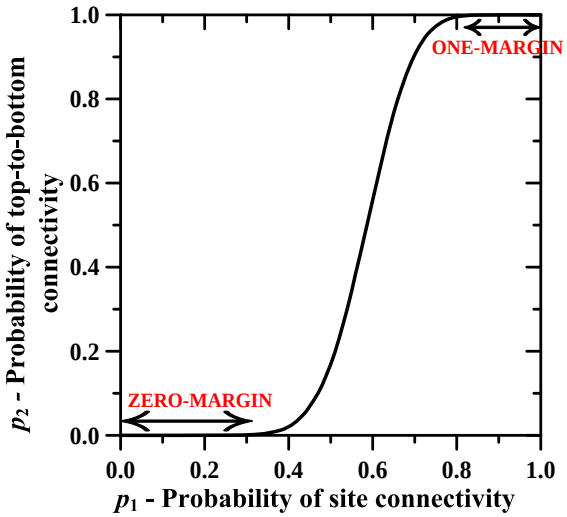


Figure 4.3: Non-linearity through percolation in random media.

Throughout this chapter, we use the concept of defect *probability* and defect *rate* interchangeably. We assume that the lattice is large enough for this to hold true.

Definition 10

We define the ***one margin*** *and* ***zero margin*** *to be the ranges of p_1 for which we interpret p_2 as unequivocally 1 and 0, respectively.* □

The percolation curve shown in Figure 4.3 tells us that unless the defect probability exceeds a zero margin (one margin), we achieve **robust connectivity**: the top and bottom plates remain disconnected (connected) with high probability. Therefore the one margin and zero margin are the indicators of defect tolerance while the lattice's top and bottom plates are connected and disconnected, respectively. In other words, the margins are the maximum defect probabilities (rates) that can be tolerated. For

example, suppose that a network has 5% zero and one margins. This means that the network will successfully tolerate defects unless the defect probability (rate) exceed 5%. If it exceeds 5%, it is obvious that we need a bigger network, one with more switches, in order to tolerate defects.

4.2 Logic Synthesis Through Percolation

We implement Boolean functions with a single lattice of four-terminal switches, as illustrated in Figure 4.4. There are $R \times C$ regions $r_{11}, \ldots, r_{RC}$ in the lattice. Each region has $N \times M$ four-terminal switches. We assign Boolean literals $x_1, \bar{x}_1, x_2, \bar{x}_2, \ldots, x_k, \bar{x}_k$ to regions as controlling inputs. If an input literal is logic 1 then all switches in the corresponding region are ideally ON; if the literal is logic 0 then all switches in the corresponding region are ideally OFF. This is illustrated in Figure 4.5.

In our synthesis method, a Boolean function is implemented by a lattice according to the connectivity between the top and bottom plates. For the purpose of elucidating our method, we will also discuss connectivity between the left and right plates. Call the Boolean functions corresponding to the top-to-bottom and left-to-right plate connectivities f_L and g_L, respectively. (Note, however, that our design method does not aim to implement separate top-to-bottom and left-to-right functions. As we explain below, f_L and g_L are related.)

As shown in Figure 4.5, each Boolean function evaluates to 1 if there exists a path between corresponding plates and evaluates to 0 otherwise. Thus, the Boolean functions f_L and g_L can be computed as the OR of all top-to-bottom and left-to-right paths, respectively. Since each path corresponds to the AND of inputs, the paths taken together correspond to the OR of these AND terms, so implement a sum-of-products expression.

Note that the values of N and M do not affect the Boolean functionality between plates; they determine the defect tolerance capability of the lattice. Therefore, for simplicity, let's set $N = 1$ and $M = 1$ while computing the Boolean functions f_L and g_L.

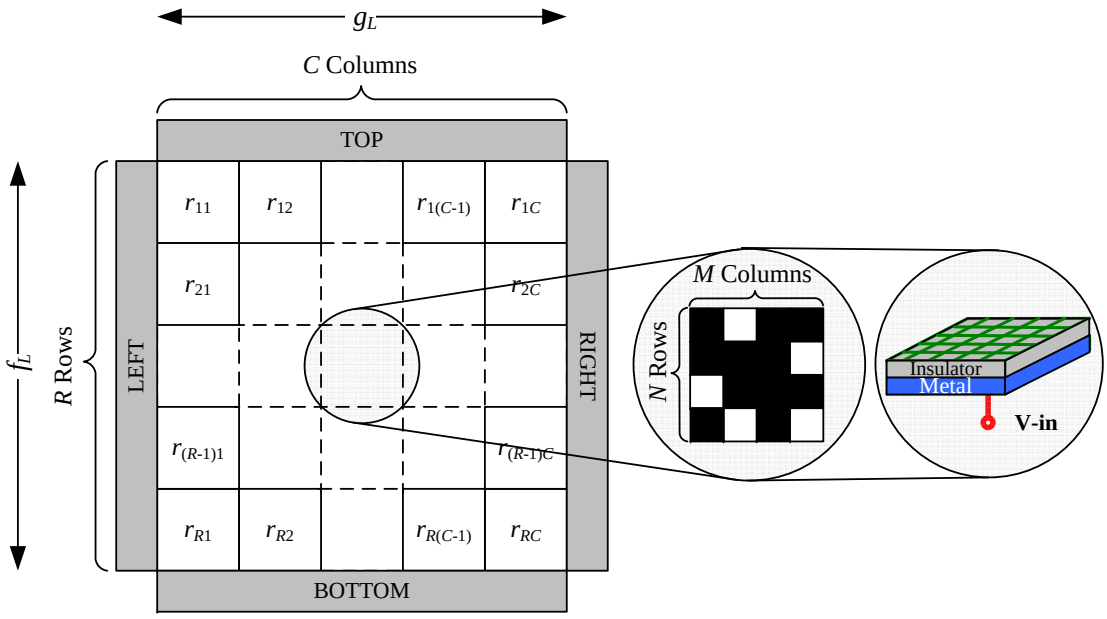


Figure 4.4: Boolean computation in a lattice, i.e., each region has $N \times M$ four-terminal switches. Each region can be realized by an $N \times M$ nanowire crossbar array with a controlling voltage V-in.

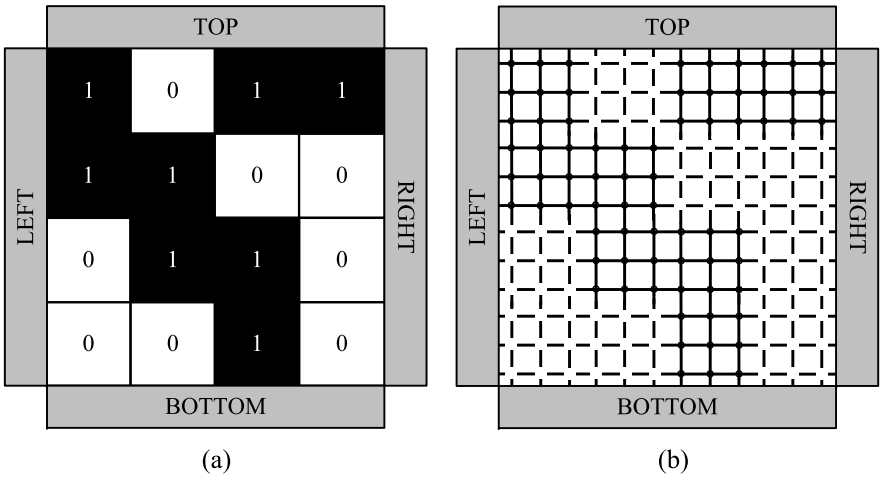


Figure 4.5: Relation between Boolean functionality and paths; $f_L = 1$ and $g_L = 0$. (a) Each of the 16 regions is assigned logic 0 or 1; $R = 4$ and $C = 4$. (b) Each region has 9 switches; $N = 3$ and $M = 3$.

In this way, there are fewer paths to count between the corresponding plates. Consider the lattice shown in Figure 4.6(a): here there are 6 regions each of which is controlled by a Boolean literal. With $N = 1$ and $M = 1$, there are 3 top-to-bottom paths and 4 left-to-right paths, as shown in Figure 4.6(b). Here f_L is the OR of the 3 products $x_1x_3, \bar{x}_1x_2, x_3x_4$ and g_L is the OR of the 4 products $x_1x_2x_3, x_1\bar{x}_1x_2x_4, \bar{x}_1x_2x_3x_3, \bar{x}_1x_3x_4$. As a result, $f_L = x_1x_3 + \bar{x}_1x_2 + x_3x_4$ and $g_L = \bar{x}_1x_3x_4 + x_2x_3$.

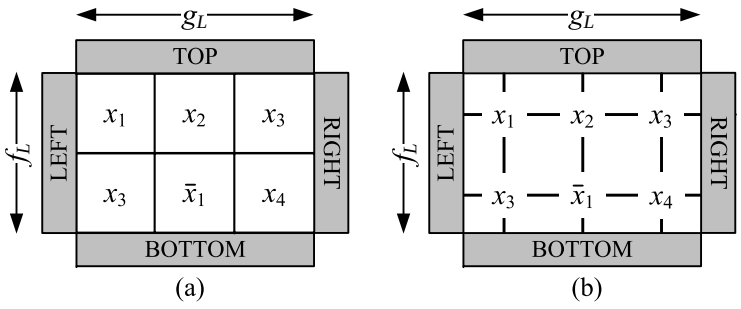


Figure 4.6: (a) A lattice with assigned inputs to 6 regions. (b) Switch-based representation of the lattice; $N = 1$ and $M = 1$.

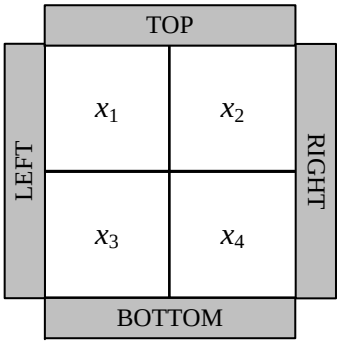
In the following section, we study the robustness of the lattice computation. We investigate the computation, implemented in terms of connectivity across the lattice, in the presence of defects.

4.2.1 Robustness

An important consideration in synthesis is the quality of the margins, defined in Definition 10. Suppose that the one and zero margins are the ranges of values for p_1 for which p_2 is always above $(1 - \epsilon)$ and below ϵ, respectively, where ϵ is a very small number. For what follows, we will use a value $\epsilon = 0.001$. The margins correlate with the degree of defect tolerance. For instance a 10% one margin means that a defect rate of up to 10% can be tolerated while the corresponding Boolean function evaluates to 1. In other words, although each switch is defective with probability 0.1, the circuit still

evaluates to 1 with high probability ($p_2 > 0.999$). The higher the margins, the higher the defect tolerance that we achieve.

Different assignments of input variables to the regions of the lattice affect the margins. Consider a 4-input 2×2 lattice shown in Figure 4.7(a). Suppose that $N = 8$ and $M = 8$ for this lattice. Figure 4.7(b) shows Boolean functionalities and margins for different input assignments. Since the lattice has 4 input variables x_1, x_2, x_3, x_4 there should be 16 different input assignments. However, there are only 7 rows in the table. Some input assignments produce the same result due to symmetries in the lattice: flipping the lattice vertically or horizontally gives us two different input assignments that are identical in terms of margins as well as the Boolean functionality. Note that each margin value in the table corresponds to either a one margin (if the corresponding Boolean function is 1) or a zero margin (if the corresponding Boolean function is 0). We define the **worst-case** one and zero margins to be the minimum one and zero margins of all input assignments. For example, the table shown in Figure 4.7(b) states that f_L has a 14% worst-case one margin and a 0% worst-case zero margin.



(a)

x_1	x_2	x_3	x_4	f_L	Margin	g_L	Margin
0	0	0	0	0	40%	0	40%
0	0	0	1	0	25%	0	25%
0	1	0	1	1	14%	0	23%
0	0	1	1	0	23%	1	14%
0	1	1	0	0	0%	0	0%
0	1	1	1	1	14%	1	14%
1	1	1	1	1	25%	1	25%

(b)

Figure 4.7: (a): A lattice with assigned inputs; $R = 2$ and $C = 2$. (b): Possible 0/1 assignments to the inputs (up to symmetries) and corresponding margins for the lattice $(N = 8, M = 8)$.

The row highlighted in grey has very low margins – indeed, these are nearly zero – so the circuit is likely to produce erroneous values for this input combination. Let's examine why. Assignments that evaluate to 0 but have diagonally adjacent assignments of blocks of 1's could be problematic because there is a chance that a weak connection will form through stray, random connections across the diagonal. This is illustrated in Figure 4.8. In this example, f_L and g_L both evaluate to 0; however the top-to-bottom and left-to-right connectivities evaluate to 1 if a defect occurs around the diagonal 1's. In effect, such defective switches are "shorting" the connection. So in this case f_L and g_L both evaluate to 1, incorrectly.

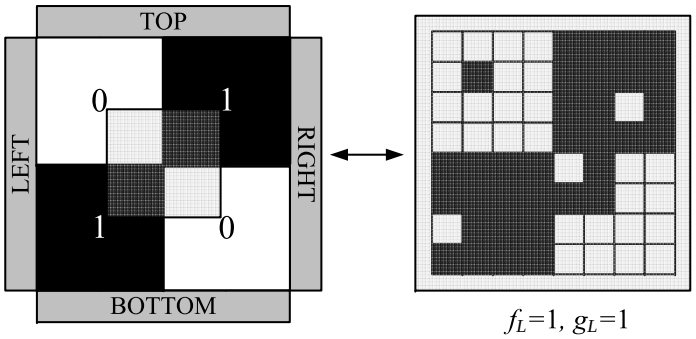


Figure 4.8: An input assignment with a low zero margin. Ideally, both f_L and g_L evaluate to 0.

Note that diagonal paths are only problematic when the corresponding Boolean function evaluates to 0 because the diagonal paths can only cause $0 \rightarrow 1$ errors. If the Boolean function evaluates to 1, these diagonal paths do not cause such an error; at best they strengthen the connection between plates. This is illustrated in Figure 4.9. In the figure, there are both top-to-bottom and left-to-right diagonal paths shown with red lines. However, only the top-to-bottom diagonal path is destructive because only f_L evaluates to 0 ($g_L = 1$).

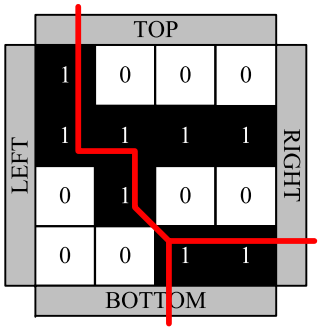


Figure 4.9: An input assignment with top-to-bottom and left-to-right diagonal paths shown by the red line.

Definition of Robustness: We call a lattice robust if there is no input assignment for which the top-to-bottom function evaluates to 0 that contains diagonally adjacent 1's.

The following theorem tells us the necessary and sufficient condition for robustness.

Theorem 5

A lattice is robust iff the top-to-bottom and left-to-right functions f_L and g_L are dual functions: $f_L(x_1, x_2, \ldots, x_k) = \bar{g}_L(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_k)$. □

(See Definition 5 for the meaning of *dual*.)

Proof of Theorem 5: In the proof, we consider two cases, namely $f_L = 1$ and $f_L = 0$.

Case 1: If $f_L(x_1, x_2, \ldots, x_k) = 1$, there must be a path of 1's between top and bottom. If we complement all the inputs $(1 \rightarrow 0, 0 \rightarrow 1)$, these connected 1's become 0's and vertically separate the lattice into two parts. Therefore no path of 1's exists between the left and right plates, i.e., $g_L(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_k) = 0$. As a result, $\bar{g}_L(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_k) = f_L(x_1, x_2, \ldots, x_k) = 1$

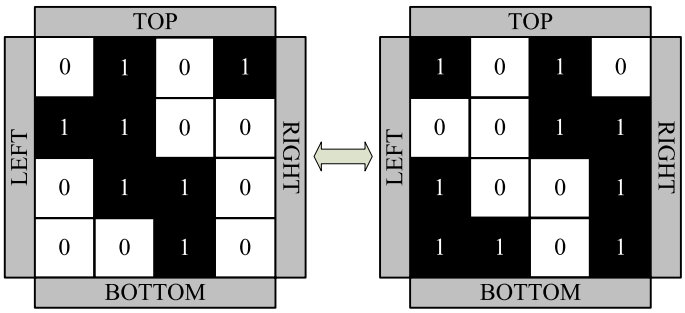


Figure 4.10: Illustration of Theorem 5.

Case 2: If $f_L(x_1, x_2, \ldots, x_k) = 0$ and there are no diagonally connected top-to-bottom paths, there must be a path of 0's between left and right. If we complement all the inputs, these connected 0's become 1's, i.e., $g_L(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_k) = 1$. As a result, $\bar{g}_L(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_k) = f_L(x_1, x_2, \ldots, x_k) = 0$

Figure 4.10 illustrates the two cases. Taken together the two cases prove that for robust computation, f_L and g_L must be dual functions. For both cases it is trivial that we can do the same reasoning in an inverse way: if f_L and g_L are dual functions then every input assignment is robust. □

We elucidate the theorem by the following example.

Example 10

Consider the lattices shown in Figure 4.11. For both lattices, $R = 2$ and $C = 2$. Let's analyze the robustness of these two lattices using Theorem 5.

Example (a): *The Boolean functions implemented by the lattice are $f_L = x_1x_3 + x_2x_4$ and $g_L = x_1x_2 + x_3x_4$. Since $f_L^D = (x_1 + x_3)(x_2 + x_4) = x_1x_2 + x_1x_4 + x_2x_3 + x_3x_4 \neq g_L$, so f_L and g_L are not dual functions. Theorem 5 tells us that if f_L and g_L are not dual then there exists an non-robust input assignment. We can easily identify it: $x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 1$.*

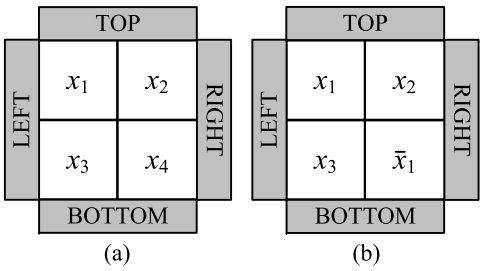


Figure 4.11: (a) An example of non-robust computation; (b) An example of robust computation.

Example (b): *The Boolean functions implemented by the lattice are* $f_L = x_1x_3 + \bar{x}_1x_2$ *and* $g_L = x_1x_2 + \bar{x}_1x_3$. *Since* $f_L^D = (x_1 + x_3)(\bar{x}_1 + x_2) = x_1x_2 + \bar{x}_1x_3 = g_L$, *so* f_L *and* g_L *are dual functions. Theorem 5 tells us that if* f_L *and* g_L *are dual then every assignment is robust. One can easily see that none of the input assignments cause diagonal 1's while the corresponding function evaluates to 0.* □

We conclude that, in order to achieve robust computation, we must design lattices that have dual top-to-bottom and left-to-right Boolean functions.

4.2.2 Logic Optimization Problem

To achieve robust computation gives rise to an interesting problem in logic optimization: given a target function f_T in SOP form, how should we assign the input literals such that $f_L = f_T$ and $g_L = f_T^D$? In other words, how should we assign literals so that the lattice implements the target function between the top and bottom plates, and implements the dual of the function between the left and right plates? As described in the previous section, having dual functions ensures robustness.

While maximizing the margins, we also need to consider the area of the lattice; this can be measured by the total number of switches $R \times C \times N \times M$ in the lattice. Here $R \times C$ and $N \times M$ represent the number of regions and the number of switches for each region, respectively.

We suggest a four-step algorithm for optimizing the lattice area while meeting prescribed worst-case margins for a given target function f_T.

Algorithm:

1. Begin with the target function f_T and its dual f_T^D both in MSOP form.
2. Find a lattice with the smallest number of regions that satisfies the conditions: $f_L = f_T$ and $g_L = f_T^D$. This determines $R \times C$.
3. Dependent on the defect rates of the technology, determine the required worst-case one and zero margin values.
4. Determine the number of switches required in each region in order to meet the prescribed margins. This determines $N \times M$.

The first step is straightforward. The dual of the target function can be computed from Definition 5. Exact methods such as Quine-McCluskey or heuristic methods such as Espresso can be used to obtain functions in MSOP form [31, 22].

For the second step of the algorithm, we point the reader to Chapter 3. In this chapter, we address the problem of assigning literals to switches in a lattice in order to implement a given target Boolean function. The goal was to minimize the number of regions. We present an efficient algorithm that produces lattices with a size that grows linearly with the number of products of the target Boolean function. Suppose that f_T and f_T^D in MSOP form have A and B product terms, respectively. Our algorithm produces lattices with $B \times A$ regions ($R = B$ and $C = A$) for which $f_L = f_T$ and $g_L = f_T^D$.

For the third step, we assume that the defect rates of the switches are known or can be estimated. Recall that we consider two types of defects: those that result in switches being OFF while they are supposed to be ON (call these "ON-to-OFF" defects), and defects that result in switches being ON while they are supposed to be OFF (call these

"OFF-to-ON" defects). We allow for different rates for both types of defects. Based upon the ON-to-OFF and OFF-to-ON defect rates, we establish the worst-case one and zero margins, respectively.

For the fourth step, we need to determine N and M such that the lattice meets the prescribed margins. Table 4.1 shows the general relationship between margins and N and M. It suggests how we should select values of N and M. For instance, suppose that we require a 20% one margin and a 5% zero margin. Table 4.1 tells us that we need to select a larger value of M than that of N. Also, from the figure, we observe that regardless of whether we increase N or M, the sum of the margins always increases. This is due to the percolation phenomenon: the larger the lattice, the steeper the non-linearity curve. Based upon these considerations, we use a simple greedy technique to set the required values of N and M. The method tries worst-case margins for different values of N and M until the prescribed margins are met.

We elucidate our algorithm with the following examples. For all of the examples, we use 10% worst-case one and zero margins.

Table 4.1: Relationship between margins, and N and M.

N	M	One Margin	Zero Margin	Sum of Margins
↑	●	↓	↑	↑
●	↑	↑	↓	↑

Example 11

Suppose that we are given the following target function f_T in MSOP form:

$$f_T = x_1 x_2.$$

First, we compute its dual f_T^D in MSOP form:

$$f_T^D = x_1 + x_2.$$

The number of products in f_T and f_T^D are 1 and 2, respectively, i.e., $A = 1$ and $B = 2$.
Then, we construct a lattice such that $f_L = f_T = x_1x_2$ and $g_L = f_T^D = x_1 + x_2$. The lattice is illustrated in Figure 4.12. Note that $R = B = 2$ and $C = A = 1$.
Finally, we find that $N = 4$ and $M = 6$ in order to satisfy 10% worst-case one and zero margins.
As a result, the lattice area $= R \times C \times N \times M = 2 \times 1 \times 4 \times 6 = 48$.

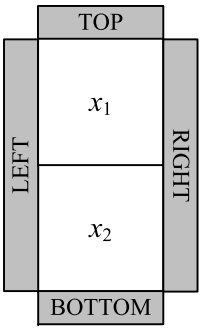


Figure 4.12: A lattice that implements $f_L = x_1x_2$ and $g_L = x_1 + x_2$.

□

Example 12

Suppose that we are given the following target function f_T in MSOP form:

$$f_T = x_1\bar{x}_2 + \bar{x}_1x_2.$$

First, we compute its dual f_T^D in MSOP form:

$$f_T^D = x_1x_2 + \bar{x}_1\bar{x}_2.$$

We have that $A = 2$ and $B = 2$.
Then, we construct a lattice such that $f_L = f_T$ and $g_L = f_T^D$. The lattice is illustrated in Figure 4.13. Note that $R = B = 2$ and $C = A = 2$.
Finally, we find that $N = 4$ and $M = 6$ in order to satisfy 10% worst-case one and zero margins.

As a result, the lattice area $= R \times C \times N \times M = 2 \times 2 \times 4 \times 6 = 96$.

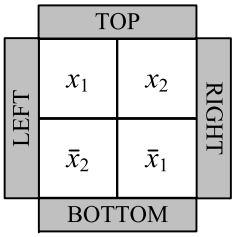


Figure 4.13: A lattice that implements $f_L = x_1\bar{x}_2 + \bar{x}_1x_2$ *and* $g_L = x_1x_2 + \bar{x}_1\bar{x}_2$.

□

Example 13

Suppose that we are given the following target function f_T *in MSOP form:*

$$f_T = x_1\bar{x}_2x_3 + x_1\bar{x}_4 + x_2x_2\bar{x}_4 + x_2x_4x_5 + x_3x_5.$$

First, we compute its dual f_T^D *in MSOP form:*

$$f_T^D = x_1x_2x_5 + x_1x_3x_4 + x_2x_3\bar{x}_4 + \bar{x}_2\bar{x}_4x_5.$$

We have that $A = 5$ *and* $B = 4$.

Then, we construct a lattice such that $f_L = f_T$ *and* $g_L = f_T^D$. *The lattice is illustrated in Figure 4.14. Note that* $R = B = 4$ *and* $C = A = 5$.

Finally, we find that $N = 4$ *and* $M = 5$ *in order to satisfy 10% worst-case one and zero margins.*

As a result, the lattice area $= R \times C \times N \times M = 4 \times 5 \times 4 \times 5 = 400$.

□

We implement the target functions with specified margins. Note that because of the lattice duality, the one and zero margins of target functions become the zero and one margins of their duals, respectively. Therefore our algorithm also gives us solutions for

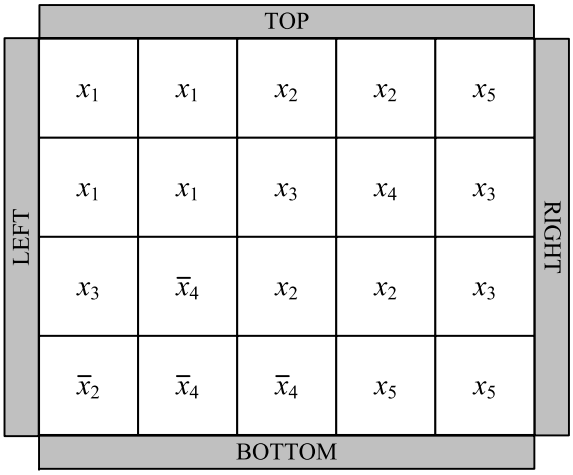


Figure 4.14: A lattice that implements $f_L = x_1\bar{x}_2x_3 + x_1\bar{x}_4 + x_2x_2\bar{x}_4 + x_2x_4x_5 + x_3x_5$ *and* $g_L = x_1x_2x_5 + x_1x_3x_4 + x_2x_3\bar{x}_4 + \bar{x}_2\bar{x}_4x_5$.

the target functions' duals with inverse margin specifications. For the examples above, the lattice areas are also valid for the target functions' duals since the zero and one margin specifications are both 10%.

In the following section, we test the effectiveness of the algorithm on benchmark circuits.

4.3 Experimental Results

We report synthesis results for some common benchmark circuits [21]. We consider each output of a benchmark circuit as a separate target Boolean function. Table 4.2 lists the required lattice areas for the target functions meeting 10% worst-case one and zero margins. Recall that the lattice area is defined as the number of switches in the lattice. It can be calculated as $R \times C \times N \times M$ where $R \times C$ and $N \times M$ represent the number of regions and the number of switches for each region, respectively.

In order to obtain the lattice areas, we follow the steps of the proposed algorithm in Section 4.2.2. We first obtain values for A and B, the number of products in the target functions and their duals, respectively. Our algorithm sets $R = A$ and $C = B$,

so produces lattices with $B \times A$ regions. We calculate values of N and M that satisfy the prescribed 10% worst-case margins.

Table 4.2 reports the lattice areas, calculated as $A \times B \times N \times M$. Examining the numbers in the table, we see that number of switches needed per region, $N \times M$, is negatively correlated with the number of regions, $A \times B$. That is to say, Boolean functions with more products (larger $A \times B$ values) need smaller regions (smaller $N \times M$ values) to meet prescribed margins. This indicates a positive scaling trend: the lattice size grows more slowly than the function size. This key behavior is due to the percolation phenomena.

The lattice areas in Table 4.2 gurantee that each circuit evaluates to a correct value with high probability (0.999) in spite of a considerably high defect rate (10%). Note that we consider random defects; there is no restriction on the number or the place of the defects.

4.4 Discussion

In this chapter, we develop a defect tolerance methodology for nanoscale lattices of four-terminal switches, each controlled by a Boolean literal. This model is conceptually general and applicable to a range of emerging technologies, including nanowire crossbar arrays [16] and magnetic switch-based structures [4]. We are investigating its applicability to DNA nanofabrics [25, 26].

Particularly with self-assembly, nanoscale lattices are often characterized by high defect rates. A variety of techniques have been proposed for mitigating against defects [32, 33, 34, 35, 36]. Significantly, unlike these techniques for defect tolerance, our method does not require defect identification followed by reconfiguration. Our method provides *a priori* tolerance to defects of any kind, both permanent and transient, provided that such defects occur probabilistically and independently. Indeed, percolation depends on a random distribution of defects. If the defect probabilities are correlated

Table 4.2: Lattice areas for the output functions of benchmark circuits in order to meet 10% worst-case one and zero margins.

Circuit	A	B	Number of regions	N	M	Lattice area
alu1	3	2	6	5	6	180
alu1	2	3	6	4	6	144
alu1	1	3	3	4	6	72
clpl	4	4	16	4	5	320
clpl	3	3	9	4	5	180
clpl	2	2	4	4	6	96
clpl	6	6	36	4	5	720
clpl	5	5	25	4	5	500
newtag	8	4	32	5	5	800
dc1	4	4	16	4	5	320
dc1	2	3	6	4	6	144
dc1	4	4	16	4	5	320
dc1	4	5	20	4	6	480
dc1	3	3	9	4	5	180
misex1	2	5	10	4	7	280
misex1	5	7	35	4	6	840
misex1	5	8	40	4	6	960
misex1	4	7	28	4	6	672
misex1	5	5	25	4	5	500
misex1	6	7	42	4	5	840
misex1	5	7	35	4	6	840
b12	4	6	24	4	6	576
b12	7	5	35	5	5	875
b12	7	6	42	5	5	1050
b12	4	2	8	5	6	240
b12	4	2	8	5	6	240
b12	5	1	5	6	5	150
b12	9	6	54	5	5	1350
b12	6	4	24	5	5	600
b12	7	2	14	6	5	420
newbyte	1	5	5	4	7	140
newapla2	1	6	6	4	7	168
c17	3	3	9	4	5	180
c17	4	2	8	5	6	240
rd53	5	10	50	4	6	1200
rd53	10	10	100	4	5	2000
rd53	16	16	256	3	5	3840

across regions, then the steepness of the percolation curve decreases; as a result, the defect tolerance diminishes. In future work, we will study this tradeoff mathematically and develop synthesis strategies to cope with correlated probabilities in defects.

Chapter 5

Dualization Problem

The problem of testing whether a monotone Boolean function in irredundant disjuntive normal form (IDNF) is self-dual is one of few problems in circuit complexity whose precise tractability status is unknown. This famous problem is called the *monotone self-duality problem* [8]. It impinges upon many areas of computer science, such as artificial intelligence, distributed systems, database theory, and hypergraph theory [37, 38].

Consider a monotone Boolean function f in IDNF. Suppose that f has k variables and n disjuncts:

$$f(x_1, x_2, \ldots, x_k) = D_1 \vee D_2 \vee \cdots \vee D_n$$

where each disjunct D_i is a prime implicant of f, $i = 1, \ldots n$. The relationship between k and n is a key aspect of the monotone self-duality problem. Prior work has shown that if f is self-dual then $k \leq n^2$ [6, 9]. We improve on this result. In Section 5.2, by Corollary 1, we show that if f is self-dual then $k \leq n$. Our result can also be applied to dual Boolean functions by using the statement in Lemma 9: Boolean functions f and g are dual pairs iff a Boolean function $af \vee bg \vee ab$ is self-dual where a and b are Boolean variables. In Section 5.2, by Corollary 2, we show that if f and g are dual pairs then $k \leq n + m - 1$ where k is the number of variables, and n and m are the numbers of

disjuncts of f and g, respectively. Prior work has shown that if f and g are dual pairs then $k \leq n \times m$ [6, 8].

In Section 5.3, we consider the monotone self-duality problem for Boolean functions with the same number of variables and disjuncts (i.e., $n = k$). For such functions, we propose an algorithm that runs in $\mathcal{O}(n^4)$ time.

In the following section, we present definitions that are used throughout this chapter.

5.1 Definitions

Definition 11

Consider k independent **Boolean variables**, $x_1, x_2, \ldots, x_k$. **Boolean literals** *are Boolean variables and their complements, i.e., $x_1, \bar{x}_1, x_2, \bar{x}_2, \ldots, x_k, \bar{x}_k$.* □

Definition 12

A **disjunct (D)** *of a Boolean function f is an AND of literals, e.g., $D = x_1\bar{x}_3x_4$, that implies f.* **A disjunct set (SD)** *is a set containing all the disjunct's literals, e.g., if $D = x_1\bar{x}_3x_4$ then $SD = \{x_1, \bar{x}_3, x_4\}$. A* **disjunctive normal form (DNF)** *is an OR of disjuncts.* □

Definition 13

A **prime implicant (PI)** *of a Boolean function f is a disjunct that implies f such that removing any literal from the disjunct results in a new disjunct that does not imply f.* □

Definition 14

An **irredundant disjunctive normal form (IDNF)** *is a DNF where each disjunct is a PI of a Boolean function f and no PI can be deleted without changing f.* □

Definition 15

Boolean functions f and g are **dual pairs** *iff $f(x_1, x_2, \ldots, x_k) = g^D = \bar{g}(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_k)$. A Boolean function f is* **self-dual** *iff $f(x_1, x_2, \ldots, x_k) = f^D = \bar{f}(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_k)$.*

Given an expression for a Boolean function in terms of AND, OR, NOT, 0, and 1, its dual can also be obtained by interchanging the AND and OR operations as well as interchanging the constants 0 and 1. For example, if $f(x_1, x_2, x_3) = x_1x_2 \vee \bar{x}_1x_3$ *then* $f^D(x_1, x_2, x_3) = (x_1 \vee x_2)(\bar{x}_1 \vee x_3)$. *A trivial example is that for* $f = 1$, *the dual is* $f^D = 0$. □

Definition 16

A Boolean function f *is* **monotone** *if it can be constructed using only the AND and OR operations (specifically, if it can constructed without the NOT operation).* □

Definition 17

The Fano plane *is the smallest finite projective plane with seven points and seven lines such that and every pair of its lines intersect in one point.* **A Boolean function that represents the Fano plane** *is a monotone self-dual Boolean function with seven variables and seven disjuncts such that every pair of its disjuncts intersect in one variable. An example is* $f = x_1x_2x_3 \vee x_1x_4x_5 \vee x_1x_6x_7 \vee x_2x_4x_6 \vee x_2x_5x_7 \vee x_3x_4x_7 \vee x_3x_5x_6$. □

5.2 Number of disjuncts versus number of variables

Our main contribution in this section is Theorem 6. It defines a necessary condition for monotone self-dual Boolean functions. For such functions, there exists a *matching* between its variables and disjuncts, i.e., every variable can be paired to a distinct disjunct that contains the variable. From this theorem we derive our two main results, presented as Corollary 1 and Corollary 2.

5.2.1 Preliminaries

We define the **intersection property** as follows. A Boolean function f satisfies the *intersection property* if every pair of its disjuncts has a non-empty intersection.

The following lemma is from from [6].

Lemma 6

Consider a monotone Boolean function f in IDNF. If f is self-dual then f satisfies the intersection property. □

Proof of Lemma 6: The proof is by contradiction. Consider a disjunct D of f. We assign 1's to the all variables of D and 0's to the other variables of f. This makes $f = 1$. If f does not satisfy the intersection property then there must be a disjunct of f having all assigned 0's. This makes $f^D = 0$, so $f \neq f^D$. This is a contradiction. □

Lemma 7

Consider a monotone Boolean function f in IDNF satisfying the intersection property. Suppose that we obtain a new Boolean function g by removing one or more disjuncts from f. There is an assignment of 0's and 1's to the variables of g such that every disjunct of g has both a 0 and a 1. □

Proof of Lemma 7: Consider one of the disjuncts that was removed from f. We focus on the variables of this disjunct that are also variables of g. Suppose that we assign 1's to all of these variables of g and 0's to all of the other variables of g. Since f is in IDNF, the assigned 1's do not make $g = 1$. Therefore $g = 0$; every disjunct of g has at least one assigned 0. Since f satisfies the intersection property, every disjunct of g has at least one assigned 1. As a result, every disjunct of g has both a 0 and a 1. □

We define a **matching** between a variable x and a disjunct D as follows. There is a matching between x and D iff x is a variable of D. For example, if $D = x_1x_2$ then there is a matching between x_1 and D as well as x_2 and D.

Lemma 8

Consider a monotone Boolean function f in IDNF satisfying the intersection property. Suppose that f has k variables and n disjuncts. If each of the b variables of f can be

matched with a distinct disjunct of f where $b < k$ and $b < n$, and all other unmatched disjuncts of f do not have any of the matched variables, then f is not self-dual. □

Proof of Lemma 8: Lemma 8 is illustrated in Figure 5.1. Note that a variable x_i is matched with a disjunct D_i for every $i = 1, \ldots, b$. To prove that f is not self-dual, we assign 0's and 1's to the variables of f such that every disjunct of f has both 0 and 1. This results in $f = 0$ and $f^D = 1$; $f \neq f^D$. We first assign 0's and 1's to the variables of $D_{b+1} \vee \ldots \vee D_n$ to make each disjunct of $D_{b+1} \vee \ldots \vee D_n$ have both a 0 and a 1. Lemma 7 allows us to do so. Note that none of the variables $x_1, \ldots, x_b$ has an assignment yet. Since f satisfies the intersection property, each disjunct of $D_1 \vee \ldots \vee D_b$ should have at least one previously assigned 0 or 1. If a disjunct of $D_1 \vee \ldots \vee D_b$ has a previously assigned 1 then we assign 0 to its matched (circled) variable; if a disjunct of $D_1 \vee \ldots \vee D_b$ has a previously assigned 0 then we assign 1 to its matched (circled) variable. As a result, every disjunct of f has both a 0 and a 1; therefore f is not self-dual.

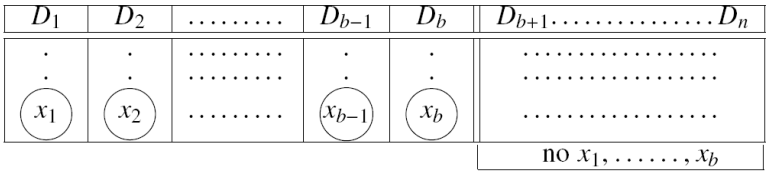


Figure 5.1: An illustration of Lemma 8.

□

The following lemma is from [6].

Lemma 9

Boolean functions f and g are dual pairs iff a Boolean function $af \vee bg \vee ab$ is self-dual where a and b are Boolean variables. □

Proof of Lemma 9: From the definition of duality, if $af \vee bg \vee ab$ is self-dual then $(af \vee bg \vee ab)_{a=1,\ b=0} = f$ and $(af \vee bg \vee ab)_{a=0,\ b=1} = g$ are dual pairs. From the definition of duality, if f and g are dual pairs then $(af \vee bg \vee ab)^D = (a^D \vee f^D)(b^D \vee g^D)(a^D \vee b^D) = (a \vee g)(b \vee f)(a \vee b) = (af \vee bg \vee ab)$. □

5.2.2 The Theorem

Theorem 6

Consider a monotone Boolean function f in IDNF. If f is self-dual then each variable of f can be matched with a distinct disjunct. □

Before proving the theorem we elucidate it with examples.

Example 14

Consider a monotone self-dual Boolean function in IDNF

$$f = x_1x_2 \vee x_1x_3 \vee x_2x_3.$$

The function has three variables x_1, x_2, and x_3, and three disjuncts $D_1 = x_1x_2$, $D_2 = x_1x_3$, and $D_3 = x_2x_3$. As shown in Figure 5.2, every variable is matched with a distinct disjunct; the circled x_1, x_2, and x_3 are matched with D_1, D_3, and D_2, respectively. We see that the theorem holds for this example. Note that the required matching – each variable to a distinct disjunct – might not be unique. For this example, another possibility is having x_1, x_2, and x_3 matched with D_2, D_1, and D_3, respectively.

D_1	D_3	D_2
x_2	x_3	x_1
$\textcircled{x_1}$	$\textcircled{x_2}$	$\textcircled{x_3}$

Figure 5.2: An example to illustrate Theorem 6: x_1, x_2, and x_3 matched with D_2, D_1, and D_3, respectively.

□

Example 15

Consider a monotone self-dual Boolean function in IDNF

$$f = x_1x_2x_3 \vee x_1x_3x_4 \vee x_1x_5x_6 \vee x_2x_3x_6 \vee x_2x_4x_5 \vee x_3x_4x_6 \vee x_3x_5.$$

The function has six variables x_1, x_2, x_3, x_4, x_5, and x_6, and seven disjuncts $D_1 = x_1x_2x_3$, $D_2 = x_1x_3x_4$, $D_3 = x_1x_5x_6$, $D_4 = x_2x_3x_6$, $D_5 = x_2x_4x_5$, $D_6 = x_3x_4x_6$, and $D_7 = x_3x_5$. As shown in Figure 5.3, every variable is matched with a distinct disjunct; the circled x_1, x_2, x_3, x_4, x_5, and x_6 are matched with D_1, D_4, D_2, D_5, D_3, and D_6, respectively. We see that the theorem holds for this example.

D_1	D_4	D_2	D_5	D_3	D_6	D_7
x_2	x_3	x_1	x_2	x_1	x_3	
x_3	x_6	x_4	x_5	x_6	x_4	x_3
$\textcircled{x_1}$	$\textcircled{x_2}$	$\textcircled{x_3}$	$\textcircled{x_4}$	$\textcircled{x_5}$	$\textcircled{x_6}$	x_5

Figure 5.3: An example to illustrate Theorem 6: x_1, x_2, x_3, x_4, x_5, and x_6 are matched with D_1, D_4, D_2, D_5, D_3, and D_6, respectively.

□

Proof of Theorem 6: The proof is by contradiction. We suppose that at most a variables of f can be matched with distinct disjuncts, where $a < k$. We consider two cases, $n = a$ and $n > a$ where n is the number of disjuncts of f. For both cases, we find an assignment of 0's and 1's to the variables of f such that every disjunct of f has both a 0 and a 1. This results in a contradiction since such an assignment makes $f = 0$ and $f^D = 1$; $f \neq f^D$.

Case 1: $n = a$.

This case is illustrated in Figure 5.4. To make every disjunct of f have both a 0 and a 1, we first assign 0 to x_1 and 1 to x_{a+1}. Then we assign a 0 or a 1 to each of the variables $x_2, \ldots, x_a$ step by step. In each step, if a disjunct has a previously assigned 1 then we assign 0 to its matched (circled) variable; if a disjunct has a previously assigned 0 then we assign 1 to its matched (circled) variable. After these steps, if every disjunct

of f has both a 0 and a 1 then we have proved that f is not self-dual. If there remain disjuncts, these disjuncts should not have any previously assigned variables. Lemma 8 identifies this condition and it tells us that f is not self-dual. This is a contradiction.

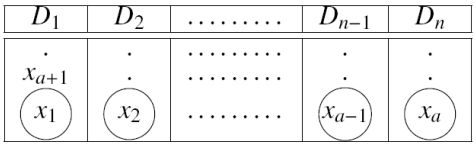


Figure 5.4: An illustration of Case 1.

Case 2: $n > a$

This case is illustrated in Figure 5.5. We show that f always satisfies the condition in Lemma 8; accordingly f is not self-dual.

As shown in Figure 5.5, the expression $D_{a+1} \vee \ldots \vee D_n$ does not have the variable x_1 or the variable x_{a+1}. If it had then at least $a+1$ variables would be matched; this would go against our assumption. For example, if $D_{a+1} \vee \ldots \vee D_n$ has x_1 then x_1 would be matched with a disjunct from $D_{a+1} \vee \ldots \vee D_n$ and x_{a+1} would be matched with D_1. So $a+1$ variables would be matched with distinct disjuncts.

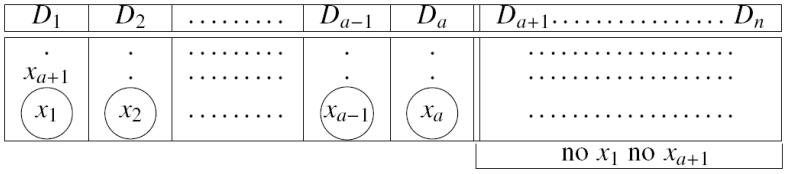


Figure 5.5: An illustration of Case 2.

If $D_{a+1} \vee \ldots \vee D_n$ does not have any of the variables $x_2, \ldots, x_a$ then f satisfies the condition in Lemma 8; f is not self-dual. If it does then the number of disjuncts not having x_1 or x_{a+1} increases. This is illustrated in Figure 5.6. Suppose that $D_{a+1} \vee \ldots \vee D_n$ has variables $x_j, \ldots, x_{a-1}$ where $j \geq 2$. As shown in the table, $D_j \vee \ldots \vee D_n$ does not have x_1 or x_{a+1}. If it had then at least $a+1$ variables would be matched;

this would go against our assumption. For example, if D_j had x_{a+1} then x_{a+1} would be matched with D_j and x_j would be matched with a disjunct from $D_{a+1} \vee \ldots \vee D_n$. So $a+1$ variables would be matched with distinct disjuncts.

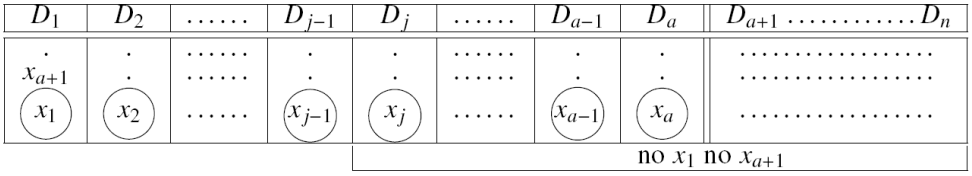


Figure 5.6: An illustration of Case 2.

If $D_j \vee \ldots \vee D_n$ does not have any of the variables $x_2, \ldots, x_{j-1}$ then f satisfies Lemma 8; f is not self-dual. If it does have any of these variables then the number of disjuncts not having x_1 or x_{a+1} increases.

As a result the number of disjuncts not having x_1 or x_{a+1} increases unless the condition in Lemma 8 is satisfied. Since there must be disjuncts having x_1 or x_{a+1}, this increase should eventually stop. When it stops, the condition in Lemma 8 will be satisfied. As a result, f is not self-dual. This is a contradiction. □

Corollary 1

Consider a monotone Boolean function f in IDNF. Suppose that f has k variables and n disjuncts. If f is self-dual then $k \leq n$. □

Proof of Corollary 1: We know that if f is self-dual then f should satisfy the matching defined in Theorem 6. This matching requires that f does not have more variables than disjuncts, so $k \leq n$. □

Corollary 2

Consider monotone Boolean functions f and g in IDNF. Suppose that f has k variables and n disjuncts and g has k variables and m disjuncts. If f and g are dual pairs then $k \leq n + m - 1$. □

Proof of Corollary 2:

From Lemma 9 we know that the Boolean functions f and g are dual pairs iff a Boolean function $af \vee bg \vee ab$ is self-dual where a and b are Boolean variables. If neither a nor b is a variable of f (or of g) then $af \vee bg \vee ab$ has $n+m+1$ disjuncts and $k+2$ variables. From Corollary 1, we know that $k+2 \leq n+m+1$, so $k \leq n+m-1$. □

5.3 The self-duality problem

In this section we propose an algorithm to test whether a monotone Boolean function in IDNF with n variables and n disjuncts is self-dual. The runtime of the algorithm is $\mathcal{O}(n^4)$.

5.3.1 Preliminaries

The following theorem is from [39, 40]

Theorem 7

Consider a disjunct D_i of a monotone self-dual Boolean function f in IDNF. For any variable x of D_i there exists at least one disjunct D_j of f such that $SD_i \cap SD_j = \{x\}$. □

Before proving the theorem we elucidate it with an example.

Example 16

Consider a monotone self-dual Boolean function function in IDNF

$$f = x_1x_2x_3 \vee x_1x_3x_4 \vee x_1x_5x_6 \vee x_2x_3x_6 \vee x_2x_4x_5 \vee x_3x_4x_6 \vee x_3x_5.$$

The function has seven disjuncts $D_1 = x_1x_2x_3$, $D_2 = x_1x_3x_4$, $D_3 = x_1x_5x_6$, $D_4 = x_2x_3x_6$, $D_5 = x_2x_4x_5$, $D_6 = x_3x_4x_6$, and $D_7 = x_3x_5$. Consider the disjunct $D_1 = x_1x_2x_3$. Since $SD_1 \cap SD_3 = \{x_1\}$, $SD_1 \cap SD_5 = \{x_2\}$, and $SD_1 \cap SD_6 = \{x_3\}$, the theorem holds for any variable of D_1. Consider the disjunct $D_2 = x_1x_3x_4$. Since

$SD_2 \cap SD_3 = \{x_1\}$, $SD_2 \cap SD_4 = \{x_3\}$, and $SD_2 \cap SD_5 = \{x_4\}$, the theorem holds for any variable of D_2. □

Proof of Theorem 7: The proof is by contradiction. Suppose that there is no disjunct D_j of f such that $SD_i \cap SD_j = \{x\}$. From Lemma 6, we know that D_i has a non-empty intersection with every disjunct of f. If we extract x from D_i then a new disjunct D_i' should also have a non-empty intersection with every disjunct of f. This means that if we assign 1's to the all variables of D_i' then these assigned 1's make $f = f^D = (1+\ldots)(1+\ldots)\ldots(1+\ldots) = 1$. So D_i' implies f; D_i' is a disjunct of f. This disjunct covers D_i. However, in IDNF, all disjuncts including D_i are irredundant, not covered by another disjunct of f. So we have a contradiction □

Lemma 10

Consider a disjunct D of a monotone self-dual Boolean function f in IDNF. Consider all disjuncts $D_1, \ldots, D_y$ of f such that $SD \cap SD_i = \{x\}$ for every $i = 1, \ldots, y$. A Boolean function $g = (D_{x=1})((D_1 \vee \ldots \vee D_y)_{x=1})^D$ implies (i.e., is covered by) f. □

Before proving the lemma we elucidate it with an example.

Example 17

Consider a monotone self-dual Boolean function function in IDNF

$$f = x_1x_2x_3 \vee x_1x_3x_4 \vee x_1x_5x_6 \vee x_2x_3x_6 \vee x_2x_4x_5 \vee x_3x_4x_6 \vee x_3x_5.$$

The function has seven disjuncts $D_1 = x_1x_2x_3$, $D_2 = x_1x_3x_4$, $D_3 = x_1x_5x_6$, $D_4 = x_2x_3x_6$, $D_5 = x_2x_4x_5$, $D_6 = x_3x_4x_6$, and $D_7 = x_3x_5$. Consider the disjunct $D_1 = x_1x_2x_3$. The disjunct $D_3 = x_1x_5x_6$ is the only disjunct that intersects D_1 in x_1. Since $g = ((D_1)_{x_1=1})\,((D_3)_{x_1=1})^D = x_2x_3x_5 \vee x_2x_3x_6$ implies f, the lemma holds for this case. The disjuncts $D_6 = x_3x_4x_6$ and $D_7 = x_3x_5$ are the only disjuncts that intersect D_1 in x_3. Since $g = ((D_1)_{x_3=1})\,((D_6 \vee D_7)_{x_1=1})^D = x_1x_2x_4x_5 \vee x_1x_2x_5x_6$ implies f, the lemma holds for this case. □

Proof of Lemma 10: To prove the statement we check if $g = 1$ always makes $f = f^D = 1$ (by assigning 1's to the variables of g). Suppose that f has n disjuncts $D_1, \ldots, D_y, D, D_{y+2}, \ldots, D_n$. If $g = 1$ then both $(D_{x=1}) = 1$ and $((D_1 \vee \ldots \vee D_y)_{x=1})^D = 1$. From Lemma 6, we know that if $(D_{x=1}) = 1$ then every disjunct of $D_{y+2}, \ldots \vee, D_n$ has at least one assigned 1. From the definition of duality, we know that if $((D_1 \vee \ldots \vee D_y)_{x=1})^D = 1$ then every disjunct of $D_1, \ldots, D_y$ has at least one assigned 1. As a result, every disjunct of f has at least one assigned 1 making $f = f^D = (1+\ldots)\ldots(1+\ldots) = 1$.

□

Lemma 11

Consider a monotone self-dual Boolean function f in IDNF with k variables. A set of b variables of f has a non-empty intersection with at least $b+1$ disjunct sets of f where $b < k$. □

Before proving the lemma we elucidate it with an example.

Example 18

Consider a monotone self-dual Boolean function function in IDNF

$$f = x_1x_2x_3x_4 \vee x_1x_5 \vee x_1x_6 \vee x_2x_5x_6 \vee x_3x_5x_6 \vee x_4x_5x_6.$$

The function has six disjuncts $D_1 = x_1x_2x_3x_4$, $D_2 = x_1x_5$, $D_3 = x_1x_6$, $D_4 = x_2x_5x_6$, $D_5 = x_3x_5x_6$, and $D_6 = x_4x_5x_6$. Consider a set of two variables $\{x_2, x_3\}$; $b = 2$. Since it has a non-empty intersection with three disjunct sets SD_1, SD_4, and SD_5, the lemma holds for this case. Consider a set of one variable $\{x_1\}$; $b = 1$. Since has a non-empty intersection with three disjunct sets SD_1, SD_2, and SD_3, the lemma holds for this case.

□

Proof of Lemma 11: The proof is by contradiction. From Theorem 6, we know that each of the k variables should be matched with a distinct disjunct, so a set of b variables of f should have a non-empty intersection with at least b disjunct sets of f. Suppose

that a set of b variables of f has a non-empty intersection with exactly b disjunct sets of f. Lemma 8 identifies this condition and it tells us that f is not self-dual. This is a contradiction. □

Theorem 8

Consider a monotone self-dual Boolean function f in IDNF with k variables. If every variable of f occurs at least three times then a set of b variables of f has a non-empty intersection with at least $b+2$ disjunct sets of f where $b < k-1$. □

Proof of Theorem 8: The proof is by induction on b.

The base case: $b = 1$.

Since a variable of f occurs three times, a set of one variable should have a non-empty intersection with at least three disjunct sets of f.

The inductive step: Assume that the theorem holds for $b \leq m$ where $m \geq 2$. We show that it also holds for $b = m+1$.

Consider a set of $m+1$ variables $S = \{x_1, \ldots, x_{m+1}\}$. Consider a disjunct D of f such that $SD \cap S = \{x_1, \ldots, x_c\}$. From Theorem 7, we know that there is at least one disjunct that intersects D in x_i for every $i = 1, \ldots, c$. We consider two cases.

For the cases we suppose that f does not have a disjunct set intersecting S in one variable; if it does then the theorem holds for S (by using the inductive assumption). Also we suppose that f does not have a disjunct set that is a subset of S; if it does then it is obvious that the theorem holds for S.

Case 1: There is only one disjunct that intersects D in x_i for every $i = 1, \ldots, c$.

Suppose that D_i is the only disjunct that intersects D in x_i for every $i = 1, \ldots, c$. Consider a variable set $SD_{x_1 - x_c}$ of $((D_1)_{x_1=1} \vee \ldots \vee (D_c)_{x_c=1})$; $SD_{x_1-x_c}$ includes all variables of $((D_1)_{x_1=1} \vee \ldots \vee (D_c)_{x_c=1})$. From Lemma 10, we know that $((D)_{x_i=1})\,((D_i)_{x_i=1})^D$ implies f for every $i = 1, \ldots, c$. This means that f should have at least $|SD_{x_1-x_c} \cap S|$ disjuncts such that each of them has one distinct variable from $SD_{x_1-x_c} \cap S = \{x_{c+1}, x_{c+2}, \ldots, x_{m+1}\}$ and none of them is covered by $(D \vee D_1 \vee \ldots \vee D_c)$.

If $SD_{x_1-x_c} \cap S = \{x_{c+1}, x_{c+2}, \ldots, x_{m+1}\}$ then f has at least $|SD_{x_1-x_c} \cap S| = m - c + 1$ disjunct sets such that each of them intersects $\{x_{c+1}, x_{c+2}, \ldots, x_{m+1}\}$ in one variable. Therefore, including SD, SD_1, SD_2, ..., and SD_c, f has at least $m + 2$ disjunct sets such that each of them has a non-empty intersection with S. If f has exactly $m + 2$ disjunct sets then each disjunct of f has a non-empty intersection with $(x_{c+1}x_{c+2}\ldots x_{m+1})(D_{x_1=1,\ldots,x_c=1})$. This means that f should have a disjunct that covers $(x_{c+1}x_{c+2}\ldots x_{m+1})(D_{x_1=1,\ldots,x_c=1})$. Since none of the $m + 2$ disjuncts covers $(x_{c+1}x_{c+2}\ldots x_{m+1})(D_{x_1=1,\ldots,x_c=1})$, f needs one more disjunct to cover $(x_{c+1}x_{c+2}\ldots x_{m+1})(D_{x_1=1,\ldots,x_c=1})$ that has a non-empty intersection with S. This is a contradiction. As a result, f has at least $m+3$ disjunct sets such that each of them has a non-empty intersection with S; the theorem holds for S.

If $SD_{x_1-x_c} \cap S = \{x_{c+1}, x_{c+2}, \ldots, x_n\}$ where $n < m + 1$ then from our inductive assumption we know that the variable set $\{x_{n+1}, x_{n+2}, \ldots, x_{m+1}\}$ intersects at least $m - n + 3$ disjunct sets. As a result, f has at least $(c + 1) + |SD_{x_1-x_c} \cap S| = (n - c) + (m - n + 3) = m + 4$ disjunct sets such that each of them has a non-empty intersection with S. So the theorem holds for S.

Case 2: For at least one of the variables of $x_1, \ldots, x_c$, say x_c, there are at least two disjuncts such that each of them intersects D in x_c.

The proof has c steps. In each step, we consider all disjuncts of f such that each of them intersects D in x_i where $1 \leq i \leq c$. We first consider disjuncts $D_1, \ldots, D_y$ such that each of them intersects D in x_1. Consider a variable set SD_{x_1} of $(D_1 \vee \ldots \vee D_y)_{x_1=1}$; SD_{x_1} includes all variables of $(D_1 \vee \ldots \vee D_y)_{x_1=1}$. From Lemma 10, we know that $(D_{x_1=1})((D_1 \vee \ldots \vee D_y)_{x_1=1})^D$ implies f. Therefore along with $D_1 \vee \ldots \vee D_y$, f should have disjuncts that cover $(D_{x_1=1})((D_1 \vee \ldots \vee D_y)_{x_1=1})^D$. This means that f includes a dual-pair of $(D_1 \vee \ldots \vee D_y)_{x_1=1}$ and $((D_1 \vee \ldots \vee D_y)_{x_1=1})^D$. From Lemma 9 and Lemma 11, we know that $SD_{x_1} \cap S$ requires at least $|SD_{x_1} \cap S| + 1$ disjunct sets of f such that each of them has a non-empty intersection with S.

We apply the same method for x_2, x_3, and x_{c-1}, respectively. Consider a variable set SD_{x_i} for every $i = 2, \ldots, c-1$; SD_{x_i} is obtained in the same way as SD_{x_1} was obtained in the first step. In each step if $SD_{x_i} \cap S$ has new variables that are the variables not included in $(SD_{x_1} \cup \ldots \cup SD_{x_{i-1}}) \cap S$, then these new variables result in new disjuncts. From Lemma 9 and Lemma 11, we know that the number of new disjuncts is at least one more than the number of the new variables. Therefore before the last step, including SD, f has at least $|(SD_{x_1} \cup \ldots \cup SD_{x_{c-1}}) \cap S| + (c-1) + 1$ disjunct sets ($+1$ is for SD) such that each of them has a non-empty intersection with S.

The last step corresponds to x_c. If $|(SD_{x_1} \cup \ldots \cup SD_{x_{c-1}}) \cap S| = ((m+1) - c)$ then SD_{x_c} does not have any new variables. Since there are at least two disjuncts such that each of them intersects D in x_c, f has at least $(m+1-c)+(c)+(2) = m+3$ disjunct sets such that each of them has a non-empty intersection with S. So the theorem holds for S. If $|(SD_{x_1} \cup \ldots \cup SD_{x_{c-1}}) \cap S| = n$ where $n < (m+1)-c$ then S has $(m-n-c+1)$ variables that are not included in $((SD_{x_1} \cup \ldots \cup SD_{x_{c-1}}) \cup SD)$. From our inductive assumption, we know that these $(m-n-c+1)$ variables results in at least $(m-n-c+1+2)$ new disjunct sets. As a result, f has at least $(m+1-c)+(c)+(2) = m+3$ disjunct sets such that each of them has a non-empty intersection with S. So the theorem holds for S. □

Lemma 12

Consider a monotone self-dual Boolean function f in IDNF with the same number of variables and disjuncts. If f has a variable occurring two times then f has at least two disjuncts of size two. □

Proof of Lemma 12: If a variable of f, say x_1, occurs two times then from Theorem 7, we know that two disjuncts that have x_1 should intersect in x_1. Consider the disjuncts $x_1 x_{a1} \ldots x_{an}$ and $x_1 x_{b1} \ldots x_{bm}$ of f. From Lemma 10, we know that both $g = (x_{a1} \ldots x_{an})(x_{b1} \vee \ldots \vee x_{bm})$ and $h = (x_{b1} \ldots x_{bm})(x_{a1} \vee \ldots \vee x_{an})$ should be covered by f. Note that g and h have total of $n+m$ disjuncts. These $n+m$ disjuncts

should be covered by at most $n+m-2$ disjuncts of f; otherwise Lemma 11 is violated. For example, if $n+m$ disjuncts are covered by $n+m-1$ disjuncts of f then along with the disjuncts $x_1x_{a1}\ldots x_{an}$ and $x_1x_{b1}\ldots x_{bm}$ there are $n+m+1$ disjuncts having $n+m+1$ variables. This means that a set of the remaining variables, say b variables, has a non-empty intersection with at most b disjuncts of f, so Lemma 11 is violated.

Any disjunct of f with more than two variables can only cover one of the $m+n$ disjuncts of $g \vee h$. Therefore to cover $m+n$ disjuncts of $g \vee h$ with $m+n-2$ disjuncts, f needs disjuncts of size two. Since a disjunct of size two can cover at most two of the $m+n$ disjuncts of $g \vee h$, f should have at least two disjuncts of size two. □

Lemma 13

Consider a monotone self-dual Boolean function f in IDNF with the same number of variables and disjuncts. If each variable of f occurs at least three times then f is a unique Boolean function that represents the Fano plane. □

Proof of Lemma 13: We consider two cases.

Case 1: A pair of disjuncts of f intersect in multiple variables.

We show that if a pair of disjuncts of f intersect in multiple variables then f is not self-dual. Consider two disjuncts D_1 and D_2 of f such that they intersect in multiple variables. Suppose that both D_1 and D_2 have variables x_1 and x_2. This case is illustrated in Figure 5.7. Note that x_3, x_4, …, x_k are matched with D_3, D_4, …, D_k, respectively. This is called *perfect matching*. Hall's theorem describes a necessary and sufficient condition for this matching: a subset of b variables of $\{x_3,\ldots,x_k\}$ has a non-empty intersection with at least b disjunct sets from $SD_3,\ldots,SD_k$. From Theorem 8, we know that a set of b variables of f has a non-empty intersection with at least $b+2$ disjunct sets of f. This satisfies the necessary and sufficient condition for the perfect matching between $x_3,\ldots,x_k$ and $D_3,\ldots,D_k$.

We find an assignment of 0's and 1's to the variables of f such that every disjunct of f has both a 0 and a 1. To make every disjunct of f have both 0 and 1, we first

assign 0 to x_1 and 1 to x_2. Then we assign a 0 or a 1 to each of the variables $x_3, \ldots, x_k$ step by step. In each step, if a disjunct has a previously assigned 1 then we assign 0 to its matched (circled) variable; if a disjunct has a previously assigned 0 then we assign 1 to its matched (circled) variable. After these steps, if every disjunct of f has both a 0 and a 1 then we have proved that f is not self-dual. If there remain disjuncts, these disjuncts should not have any previously assigned variables. Lemma 8 identifies this condition and it tells us that f is not self-dual.

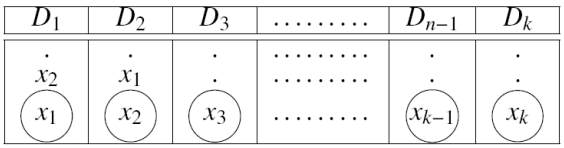


Figure 5.7: An illustration of Case 1.

Case 2: Every pair of disjuncts of f intersect in one variable.

Suppose that a variable of f, say x_1, occurs three times. Consider disjuncts $D_1 = x_1x_{a1}\ldots x_{an}$, $D_2 = x_1x_{b1}\ldots x_{bm}$, and $D_3 = x_1x_{c1}\ldots x_{cl}$ of f where $n \leq m \leq l$. From Lemma 10, we know that f should cover $(x_{a1}\ldots x_{an})(x_{b1}\vee\ldots\vee x_{bm})(x_{c1}\vee\ldots\vee x_{cl})$ where $n \leq m \leq l$. This means that f should cover $m \cdot l$ disjuncts. These disjuncts are covered by at least $m \cdot l$ disjuncts of f; otherwise the intersection property does not hold for f. Along with D_1, D_2, and D_3, f has $m \cdot l + 3$ disjuncts having $m+n+l+1$ variables. From Lemma 6, we know that $m \cdot l + 3 \leq m + n + l + 1$. The only solution of this inequality is that $n = 2$, $m = 2$, and $l = 2$. This results in a self-dual Boolean function representing the Fano plane, e.g., $f = x_1x_2x_3 \vee x_1x_4x_5 \vee x_1x_6x_7 \vee x_2x_4x_6 \vee x_2x_5x_7 \vee x_3x_4x_7 \vee x_3x_5x_6$.

If a variable of f occurs more than three times then the value on left hand side of the inequality $m \cdot l + 3 \leq m + n + l$ increases more than that on the right hand side does, so there is no solution. □

Lemma 14

A Boolean function f is self-dual iff $f_{x_a=x_b}$, $f_{x_a=x_c}$, and $f_{x_b=x_c}$ are all self-dual Boolean functions where x_a, x_b, and x_c are any three variables of f. □

Proof of Lemma 14: From the definition of duality, f is self-dual iff each assignment of 0's and 1's to the variables of f, corresponding to a row of the truth table, satisfies $f(x_1, x_2, \ldots, x_k) = \bar{f}(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_k)$. Any dependency between variables of f only eliminates some rows of f's truth table. Therefore, if f is self-dual then $f_{x_a=x_b}$, $f_{x_a=x_c}$, and $f_{x_b=x_c}$ are all self-dual. For each row of f's truth table either $x_a = x_b$ or $x_a = x_c$, or $x_b = x_c$. Therefore, if $f_{x_a=x_b}$, $f_{x_a=x_c}$, and $f_{x_b=x_c}$ are all self-dual then f is self-dual. □

5.3.2 The Algorithm

We present a four-step algorithm:

Input: A monotone Boolean function f in IDNF with n variables n disjuncts.

Output: "YES" if f is self-dual; "NO" otherwise.

1. Check if f is a single variable Boolean function. If it is then return "YES".
2. Check if f represents the Fano plane. If it does then return "YES".
3. Check if the intersection property holds for f. If it does not then return "NO".
4. Check if f has two disjuncts of size two, $x_a x_b$ and $x_a x_c$ where x_a, x_b, and x_c are variables of f. If it does not then return "NO"; otherwise obtain a new function $f = f_{x_b=x_c}$ in IDNF. Repeat this step until f consists of a single variable; in this case, return "YES".

If f is self-dual then f should be in one of the following three categories: (1) f is a single variable Boolean function; (2) at least one variable of f occurs two times; (3) each variable of f occurs at least three times. From Theorem 7, we know that if f is self-dual and not in (1) then every variable of f should occur at least two times, so f should be

in either (2) or (3). Therefore these three categories cover all possible self-dual Boolean functions.

The first step of our algorithm checks if f is self-dual and in (1). The second step of our algorithm checks if f is self-dual and in (3). From Lemma 13, we know that if f is self-dual and in (3) then f is a unique Boolean function that represents the Fano plane. The third and fourth steps of our algorithm check if f is self-dual and in (2). From Lemma 6, we know that if f is self-dual then f should satisfy the intersection property. From Lemma 12, we know that if f is self-dual and in (2) then f should have at least two disjuncts of size two, $x_a x_b$ and $x_a x_c$. From Lemma 14, we know that f is self-dual iff $f_{x_a=x_b}$, $f_{x_a=x_c}$, and $f_{x_b=x_c}$ are all self-dual. Since f satisfies the intersection property, both $f_{x_a=x_b} = x_a$ and $f_{x_a=x_c} = x_a$ are self-dual. This means that f is self-dual iff $f_{x_b=x_c}$ is self-dual. Note that $f_{x_b=x_c}$ in IDNF has $n-1$ variables and $n-1$ disjuncts. Since $f_{x_b=x_c}$ satisfies the intersection property and does not represent the Fano plane, we just need to repeat step four to check if the function is self-dual. Note that to check if f is self-dual and in (2), we need to repeat step four at most n times.

The steps three and four of the algorithm run in $\mathcal{O}(n^4)$ and $\mathcal{O}(n^3)$ time, respectively. Therefore the run time of the algorithm is that of the step three $\mathcal{O}(n^4)$.

We present a few examples to elucidate our algorithm.

Example 19

Suppose that we are given the following function f in IDNF

$$f = x_1x_2x_3 \vee x_1x_5x_6 \vee x_2x_3x_6 \vee x_2x_4x_5 \vee x_3x_4x_6 \vee x_3x_5.$$

The function has 6 variables and 6 disjuncts, so $n = 6$. The algorithm does not return "YES" or "NO" in its first three steps, so we should apply step four. First, we need to check if f has two disjuncts of size two. Since f has only one disjunct x_3x_5 of size two, the algorithm returns "NO". This means that f is not self-dual. Indeed, f is not self-dual; $f \neq f^D$. □

Example 20

Suppose that we are given the following function f in IDNF

$$f = x_1x_2 \vee x_1x_3 \vee x_1x_4x_5x_6 \vee x_1x_4x_6x_7 \vee x_2x_3x_4 \vee x_2x_3x_5x_7 \vee x_2x_3x_6x_7.$$

The function has 7 variables and 7 disjuncts, so $n = 7$. The algorithm does not return "YES" or "NO" in its first three steps, so we should apply step four. First, we need to check if f has two disjuncts of size two. The function has two disjuncts x_1x_2 and x_1x_3 of size two; $x_a = x_1$, $x_b = x_2$, and $x_c = x_3$. We obtain a new function $f = f_{x_2=x_3}$ in IDNF as follows

$$f = x_1x_3 \vee x_1x_4x_5x_6 \vee x_1x_4x_6x_7 \vee x_3x_4 \vee x_3x_5x_7 \vee x_3x_6x_7.$$

The function has 6 variables and 6 disjuncts, so $n = 6$. Again, we apply step four. The function has two disjuncts x_1x_3 and x_3x_4 of size two; $x_a = x_3$, $x_b = x_1$, and $x_c = x_4$. We obtain a new function $f = f_{x_1=x_4}$ in IDNF as follows

$$f = x_3x_4 \vee x_3x_5x_7 \vee x_3x_6x_7 \vee x_4x_5x_6 \vee x_4x_6x_7.$$

The function has 5 variables and 5 disjuncts, so $n = 5$. Again, we apply step four. Since f only has one disjunct x_3x_5 of size two, the algorithm returns "NO". This means that f is not self-dual. Indeed, f is not self-dual; $f \neq f^D$. □

Example 21

Suppose that we are given the following function f in IDNF

$$f = x_1x_2x_3x_4x_5 \vee x_1x_2x_3x_4x_7 \vee x_1x_6 \vee x_2x_6 \vee x_3x_6 \vee x_4x_6 \vee x_5x_6x_7.$$

The function has 7 variables and 7 disjuncts, so $n = 7$. The algorithm does not return "YES" or "NO" in its first three steps, so we should apply step four. First, we need to check if f has two disjuncts of size two. The function has four disjuncts x_1x_6, x_2x_6, x_3x_6, and x_4x_6 of size two. We need to select any two of them, say x_1x_6 and x_2x_6;

$x_a = x_6$, $x_b = x_1$, and $x_c = x_2$. We obtain a new function $f = f_{x_1=x_2}$ in IDNF as follows

$$f = x_2x_3x_4x_5 \vee x_2x_3x_4x_7 \vee x_2x_6 \vee x_3x_6 \vee x_4x_6 \vee x_5x_6x_7.$$

The function has 6 variables and 6 disjuncts, so $n = 6$. Again, we apply step four. The function has three disjuncts x_2x_6, x_3x_6, and x_4x_6 of size two. We need to select any two of them, say x_2x_6 and x_3x_6; $x_a = x_6$, $x_b = x_2$, and $x_c = x_3$. We obtain a new function $f = f_{x_2=x_3}$ in IDNF as follows

$$f = x_3x_4x_5 \vee x_3x_4x_7 \vee x_3x_6 \vee x_4x_6 \vee x_5x_6x_7.$$

Again, we apply step four. The function has 5 variables and 5 disjuncts, so $n = 5$. Again, we apply step four. The function has two disjuncts x_3x_6 and x_4x_6 of size two. We obtain a new function $f = f_{x_3=x_4}$ in IDNF as follows

$$f = x_4x_5 \vee x_4x_6 \vee x_4x_7 \vee x_5x_6x_7.$$

With applying step four we obtain a new function $f = f_{x_5=x_6}$ in IDNF as follows

$$f = x_4x_6 \vee x_6x_7 \vee x_4x_7.$$

With applying step four we obtain a new function $f = f_{x_4=x_7}$ in IDNF as follows

$$f = x_7.$$

Since f consists of a single variable, the algorithm returns "YES". This means that f is self-dual. Indeed, f is self-dual; $f = f^D$. □

5.4 Discussion

In this chapter, we investigate monotone self-dual Boolean functions. We present many new properties for these Boolean functions. Most importantly, we show that

monotone self-dual Boolean functions in IDNF (with k variables and n disjuncts) do not have more variables than disjuncts; $k \leq n$. This is a significant improvement over the prior result showing that $k \leq n^2$.

We focus on the famous problem of testing whether a monotone Boolean function in IDNF is self-dual. We examine this problem for monotone Boolean functions with the same number of variables and disjuncts; $k = n$. Our algorithm runs in $\mathcal{O}(n^4)$ time. As a future work we plan to extend our results to monotone Boolean functions with variety of different k-n relations.

Chapter 6

Conclusion and Future Directions

In this dissertation, we discuss strategies of achieving digital computation with lattices of four-terminal switches. We also discuss the mathematics of lattice-based implementations.

We develop a model of regular lattices of four-terminal switches. In our model each switch is controlled by a Boolean literal. If the literal takes the value 1, the corresponding switch is connected to its four neighbours; else it is not connected. A Boolean function is implemented in terms of connectivity across the lattice: it evaluates to 1 iff there exists a connected path between two opposing edges of the lattice.

We present our model at the technology-independent level. Although conceptually general, we comment that our synthesis results are applicable to various emerging technologies, including nanowire crossbar arrays and magnetic switch-based structures with independently controllable crosspoints. As a future work, we study our model's applicability to nanoscale arrays of switches, such as optical switch-based structures and arrays of single-electron transistors.

We present our synthesis method to implement a given target Boolean function with a lattice of four-terminal switches. Our method produces a lattice size that grows linearly with the number of products of the target Boolean function. The time complexity

of our synthesis algorithm is polynomial in the number of products. we also derive a lower bound on the size of a lattice required to implement a Boolean function. With comparing our results to a lower bound, we see that the synthesis results are not optimal. However, this is hardly surprising: at their core, most logic synthesis problems are computationally intractable; the solutions that are available are based on heuristics. Furthermore, good 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.)

In particular, our method will not be effective for Boolean functions such that the functions' duals have much more products than the functions do have. The lattices for such functions will be inordinately large. The cost of implementing such functions could be mitigated by decomposing and implementing Boolean functions with separate lattices (or physically separated regions in a single lattice). For example, $f_T = x_1x_2x_3 + x_4x_5x_6$ can be implemented by two lattices each of which is for each product, so the target function is implemented by two 3×1 lattices. An apparent disadvantage of this technique is the necessity of using multiple lattices rather than a single lattice to implement a target function. We do not get into further details of the technique of decomposition and sharing; We would like to keep it as a future work.

Another future direction is to extend the results to lattices of eight-terminal switches, and then to 2^k-terminal switches, for arbitrary k.

We develop a novel probabilistic framework for digital computation with lattices of nanoscale switches based on the mathematical phenomenon of percolation. With random connectivity, percolation gives rise to a sharp non-linearity in the probability of global connectivity as a function of the probability of local connectivity. This phenomenon is exploited to compute Boolean functions robustly, in the presence of defects. It is shown that the margins, defined in terms of the steepness of the non-linearity,

translate into the degree of defect tolerance. Achieving good margins entails a mapping problem. Given a target Boolean function, the problem is how to assign literals to regions of the lattice such that there are no diagonal paths of 1s in any assignment that evaluates to 0. Assignments with such paths result in poor error margins due to stray, random connections that can form across the diagonal. A necessary and sufficient condition is formulated for a mapping strategy that preserves good margins: the top-to-bottom and left-to-right connectivity functions across the lattice must be dual functions. Based on lattice duality, an efficient algorithm to perform the mapping is proposed. The algorithm optimizes the lattice area while meeting prescribed worst-case margins.

Particularly with self-assembly, nanoscale lattices are often characterized by high defect rates. A variety of techniques have been proposed for mitigating against defects. Significantly, unlike these techniques for defect tolerance, our method does not require defect identification followed by reconfiguration. Our method provides *a priori* tolerance to defects of any kind, both permanent and transient, provided that such defects occur probabilistically and independently. Indeed, percolation depends on a random distribution of defects. If the defect probabilities are correlated across regions, then the steepness of the percolation curve decreases; as a result, the defect tolerance diminishes. In future work, we will study this tradeoff mathematically and develop synthesis strategies to cope with correlated probabilities in defects.

A significant tangent for this work is its mathematical contribution: lattice-based implementations present a novel view of the properties of Boolean functions. In Chapter 5, we investigate monotone self-dual Boolean functions. We present many new properties for these Boolean functions. Most importantly, we show that monotone self-dual Boolean functions in IDNF (with k variables and n disjuncts) do not have more variables than disjuncts; $k \leq n$. This is a significant improvement over the prior result showing that $k \leq n^2$.

We study the applicability of these properties to the famous problem of testing whether a monotone Boolean function in IDNF is self-dual. This is one of the few problems in circuit complexity whose precise tractability status is unknown. We examine this problem for monotone Boolean functions with the same number of variables and disjuncts; $k = n$. Our algorithm runs in $\mathcal{O}(n^4)$ time. As a future work we plan to extend our results to monotone Boolean functions with variety of different k-n relations.

References

[1] C. E. Shannon. A symbolic analysis of relay and switching circuits. *Transactions of the AIEE*, 57:713–723, 1938.

[2] A. DeHon. Nanowire-based programmable architectures. *ACM Journal on Emerging Technologies in Computing Systems*, 1(2):109–162, 2005.

[3] Y. Luo, C. P. Collier, J. O. Jeppesen, K. A. Nielsen, et al. Two-dimensional molecular electronics circuits. *ChemPhysChem*, 3(6):519–525, 2002.

[4] A. Khitun, M. Bao, and K. L. Wang. Spin wave magnetic nanofabric: A new approach to spin-based logic circuitry. *IEEE Transactions on Magnetics*, 44(9):2141–2152, 2008.

[5] Y. Zomaya. Molecular and nanoscale computing and technology. In *Handbook of Nature-Inspired and Innovative Computing*, chapter 14, pages 478–520. Springer, 2006.

[6] M. L. Fredman and L. Khachiyan. On the complexity of dualization of monotone disjunctive normal forms. *Journal of Algorithms*, 21(3):618–628, 1996.

[7] T. Ibaraki and T. Kameda. A theory of coteries: Mutual exclusion in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):779–794, 1993.

[8] T. Eiter, K. Makino, and G. Gottlob. Computational aspects of monotone dualization: A brief survey. *Discrete Applied Mathematics*, 156(11):1952–2005, 2008.

[9] D. R. Gaur and R. Krishnamurti. Self-duality of bounded monotone Boolean functions and related problems. *Discrete Applied Mathematics*, 156(10):1598–1605, 2008.

[10] S. B. Akers. A rectangular logic array. *IEEE Transactions on Computers*, 21(8):848–857, 1972.

[11] M. Chrzanowska-Jeske, Y. Xu, and M. Perkowski. Logic synthesis for a regular layout. *VLSI Design*, 10(1):35–55, 1999.

[12] M. Chrzanowska-Jeske and A. Mishchenko. Synthesis for regularity using decision diagrams. In *Proceedings of IEEE International Symposium on Circuits and Systems ISCAS*, pages 4721–4724, 2005.

[13] G. M. Whitesides and B. Grzybowski. Self-assembly at all scales. *Science*, 295(5564):2418–2421, 2002.

[14] H. Yan, S. H. Park, G. Finkelstein, J. H. Reif, and T. H. LaBean. DNA-templated self-assembly of protein arrays and highly conductive nanowires. *Science*, 301(5641):1882–1884, 2003.

[15] M. M. Ziegler and M. R. Stan. CMOS/nano co-design for crossbar-based molecular electronic systems. *IEEE Transactions on Nanotechnology*, 2(4):217–230, 2003.

[16] Y. Cui and C. M. Lieber. Functional nanoscale electronic devices assembled using silicon nanowire building blocks. *Science*, 291(5505):851–853, 2001.

[17] Y. Huang, X. Duan, Y. Cui, L. J. Lauhon, K. Kim, and C. M. Lieber. Logic gates and computation from assembled nanowire building blocks. *Science*, 294(5545):1313–1317, 2001.

[18] X. Duan, Y. Huang, and C. M. Lieber. Nonvolatile memory and programmable logic from molecule-gated nanowires. *Nano Letters*, 2(5):87–90, 2002.

[19] S. Kaeriyama, T. Sakamoto, H. Sunamura, M. Mizuno, H. Kawaura, T. Hasegawa, K. Terabe, T. Nakayama, and M. Aono. A nonvolatile programmable solid-electrolyte nanometer switch. *IEEE Journal of Solid-State Circuits*, 40(1):168–176, 2005.

[20] M. M. Eshaghian-Wilner, A. Khitun, S. Navab, and K. Wang. A nano-scale reconfigurable mesh with spin waves. In *International Conference on Computing Frontiers*, pages 5–9, 2006.

[21] K. McElvain. IWLS93 benchmark set: Version 4.0, distributed as part of the IWLS93 benchmark distribution, http://www.cbl.ncsu.edu:16080/benchmarks/lgsynth93/, 1993.

[22] R. K. Brayton, C. McMullen, G. D. Hachtel, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[23] A. Mishchenko et al. ABC: A system for sequential synthesis and verification, 2007.

[24] R. E. Bryant. Boolean analysis of MOS circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(4):634–649, 1987.

[25] C. Pistol, A. R. Lebeck, and C. Dwyer. Design automation for dna self-assembled nanostructures. In *Design Automation Conference*, 2006.

[26] P. W. K. Rothemund. Folding dna to create nanoscale shapes and patterns. *Nature*, 440(7082):297–302, 2006.

[27] I. Wegener. Lower bounds on circuit complexity. In *The Complexity of Boolean Functions*, chapter 5, pages 119–144. John Wiley & Sons, 1987.

[28] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78(2):264–300, 1990.

[29] M. Altun, M. D. Riedel, and C. Neuhauser. Nanoscale digital computation through percolation. In *Design Automation Conference*, pages 615–616, 2009.

[30] S. R. Broadbent and J. M. Hammersley. Percolation processes I. crystals and mazes. In *Proceedings of the Cambridge Philosophical Society*, pages 629–641, 1957.

[31] E. J. McCluskey. *Logic Design Principles with Emphasis on Testable Semicustom Circuits*. Prentice-Hall, 1986.

[32] J. Huang, M. Tahoori, and F. Lombardi. On the defect tolerance of nano-scale two-dimensional crossbars. In *International Symposium on Defect and Fault Tolerance of VLSI Systems*, pages 96–104, 2004.

[33] P. Kuekes, W. Robinett, G. Seroussi, and R. Williams. Defect-tolerant interconnect to nanoelectronic circuits: Internally redundant demultiplexers based on error-correcting codes. *Nanotechnology*, 16(6):869–882, 2005.

[34] F. Sun and T. Zhang. Two fault tolerance design approaches for hybrid CMOS/nanodevice digital memories. In *International Workshop on Defect and Fault Tolerant Nanoscale Architectures*, 2006.

[35] T. Hogg and G. Snider. Defect-tolerant logic with nanoscale crossbar circuits. *Journal of Electronic Testing*, 23:117–129, 2007.

[36] G. Snider and R. Williams. Nano/CMOS architectures using a field-programmable nanowire interconnect. *Nanotechnology*, (18):1–11, 2007.

[37] K. Makino. Efficient dualization of O(log n)-term monotone disjunctive normal forms. *Discrete Applied Mathematics*, 126:305–312, 2003.

[38] T. Eiter and G. Gottlob. Hypergraph transversal computation and related problems in logic and AI. *Lecture Notes in Computer Science*, 2424:549–564, 2002.

[39] M. Altun and M. D. Riedel. Lattice-based computation of Boolean functions. In *Design Automation Conference*, pages 609–612, 2010.

[40] M. Altun and M. D. Riedel. Logic synthesis for switching lattices. *Will appear in IEEE Transactions on Computers*, 2011.