Technology Mapping for FPGA's Using Boolean Matching and Spectral Techniques

by

Xiaojun Wang
B.Sc., University of Electronics Science and Technology of China, 1989

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr. D. M. Miller, Supervisor (Department of Computer Science)

[redacted]

Dr. J. C. Muzio, Department Member (Department of Computer Science)

[redacted]

Dr. N. Dimopoulos, Outside Member (Department of Electrical and Computer Engineering)

[redacted]

Dr. I. Sharf, External Examiner (Department of Mechanical Engineering)

Supervisor: Dr. D. M. Miller

ABSTRACT

Field programmable gate arrays (FPGA's) are widely used in modern logic design because of their high flexibility, fast turn-around time and increasing capacities. The complexity of FPGA architectures requires highly efficient synthesis tools in order to conduct a successful design.

Technology mapping is a critical step in FPGA synthesis. The quality of the final circuit depends heavily on this step. Many approaches to technology mapping require a quick method to determine that two Boolean functions are logically equivalent. In this thesis, we present an algorithm for technology mapping that is based on a new canonical spectrum for Boolean functions. Two functions are NPN matchable if, and only if, they can be transformed to the same canonical spectrum.

Experimental results are presented to show that the algorithm is very effective in terms of the area of the final circuit and the time taken to perform the technology mapping. The thesis concludes with a number of future research directions arising from this work.

Examiners:

[redacted]

Dr. D. M. Miller, Supervisor (Department of Computer Science)

[redacted]

Dr. J. C. Muzio, Department Member (Department of Computer Science)

[redacted]

Dr. N. Dimopoulos, Outside Member (Department of Electrical and Computer Engineering)

[redacted]

Dr. I. Sharf, External Examiner (Department of Mechanical Engineering)

Contents

List of Tables

List of Figures

Acknowledgements

I would like to express my sincerest thanks to my supervisor Dr. D. M. Miller, for his academic supervision and financial support throughout my M.Sc. program. I also want to thank him for allowing me to use his canonical form program which is a basis for this thesis.

Special thanks to Dr. J. C. Muzio for his help during the preparation of this thesis.

Finally, I would like to thank my wife and family members for their constant encouragement and support.

To My Parents

Chapter 1

Introduction

Field programmable gate arrays (FPGA's) are powerful devices for implementing digital circuits. An FPGA consists of an array of logic blocks and general routing resources. A logic block, which may be user-programmable, can typically implement a large number of combinational and sequential logic functions. Interconnections between logic blocks are provided by interconnections of routing resources via electrically programmable switches. Moreover, an FPGA can be simply re-programmed to implement different designs.

Because of their high flexibility, fast turn-around time and increasing capacities, FPGA's are ideal for system prototyping as well as for actual designs. Over the past few years, the market for field programmable gate arrays (FPGA's) has expanded dramatically. It is predicted that, in the next few years, it will continue to grow at a compound annual rate of nearly 30 percent! [1]

However, the complexity of FPGA architectures makes manual mapping of designs time-consuming and error-prone, thus offsetting the turn-around advantage. This makes it necessary to develop automatic synthesis tools for FPGA's to reduce design time and enhance circuit reliability.

One of the key steps in digital circuit design is *logic synthesis*. This is the process that maps the high-level description of a design into a circuit composed of circuit elements from a given library. The final circuit is optimal with respect to a cost function that typically incorporates measures of both area and delay. Usually, the

logic synthesis process is divided into two phases: a *technology independent optimization* phase and a *technology mapping* phase. As its name suggests, the technology independent optimization phase (also called *logic optimization*) is independent of the technology of the target device. In this stage, algorithms for manipulating general Boolean functions are applied to the original logic network and an optimized network is generated that is functionally equivalent to the original network.

Following the technology independent optimization, the technology mapping phase transforms the optimized network into a feasible circuit that consists of a restricted set of circuit elements. This transformation implies a *matching* operation which involves recognizing logic equivalence between two Boolean functions and assigning function inputs.

Technology mapping is a very critical step in the synthesis of logic circuits. The quality of the synthesized circuits, in terms of both area and performance, depends heavily on this step.

For this reason, many approaches to technology mapping have been pursued and implemented in research and commercial tools. Both rule-based methods [2], [3] and heuristic algorithms [4]–[11] have been proposed.

Generally, the algorithmic approaches to technology mapping divide the problem into subproblems. The original Boolean network is first partitioned into an interconnection of single-output subnetworks. Each subnetwork is then decomposed into an interconnection of two-input functions (e.g., AND, OR, or NAND). Finally, each decomposed subnetwork is covered by an interconnection of library cells to produce the final circuit.

The most successful approach to synthesis for ASIC technologies prior to FPGA's has used library-based technology mapping [4]. However, this approach encounters difficulties when targeted to FPGA's because the complex logic blocks of an FPGA can each implement a large number of different logic functions.

Recently, research interest has focused on technology mapping methods based on *Boolean matching* [10]–[16]. This is an approach that tries to detect logic equivalence

between two Boolean functions by the functionality rather than the structure or representation of the functions. Boolean matching involves determining whether a given Boolean function can be transformed to another given Boolean function using operations such as negation of inputs, permutation of inputs, and negation of outputs.

Spectral techniques have been used for logic design for many years. There are many problems for which spectral techniques offer solutions that are difficult to obtain by other means [17]– [19].

In this thesis, we present an algorithmic approach to technology mapping of FPGA's. A new matching algorithm is presented which is based on a novel canonical form for Boolean functions: the *spectral NPN canonical form.* The algorithm presented selects appropriate permutations and negations to transform a given Boolean function to its canonical form. Two functions are considered matchable if, and only if, they can be transformed to the same canonical form.

We also present a technology mapper which is based on the new canonical form. Test results on a set of benchmark circuits are given to show that the matching algorithm is quite effective compared with the existing methods. Although we choose the Actel Act-1 FPGA as the target technology of our mapper, the method is actually independent of target technologies and thus can be easily extended to other technologies.

The objective of the thesis is to show that, using the new canonical form provides an effective solution to the problem of matching two Boolean functions, which is the key to the technology mapping procedure considered. In particular, we wish to show that this new approach makes mapping very fast so that computational effort can be applied to technology independent optimization.

In chapter 2, we discuss technology mapping in detail. The library-based technology mapping approach is described, followed by a review of some of the existing technology mapping methods for multiplexer-based FPGA's. These methods are *structural* methods since, in performing the mapping, they exploit the specific structure of the logic blocks of the target FPGA.

Chapter 3 provides the necessary background on spectral techniques. Two applications of spectral techniques are presented: Boolean function classification and detection of symmetry conditions in Boolean functions.

In chapter 4, we first review some Boolean matching methods for technology mapping of multiplexer-based FPGA's. We then present the new spectral NPN canonical form for Boolean functions and how to transform the spectra of a given function to the canonical form.

Based on this new canonical form, a technology mapper has been implemented. Details of the mapper are given in chapter 5.

Chapter 6 presents test results of the mapper on a set of MCNC (Microelectronics Center of North Carolina) and ISCAS (International Symposium on Circuits And Systems) benchmark circuits.

Finally, chapter 7 concludes the thesis by summarizing the algorithms and listing some interesting problems for future research.

Chapter 2

Technology Mapping for FPGA's

Technology mapping is the task of transforming an arbitrary multi-level logic representation into an interconnection of circuit elements from a given set of elements, while optimizing the final circuit with respect to area, delay, or both. When targeted to FPGA's, the given set of elements consists of all the logic functions that can be implemented by an FPGA logic block. Since an FPGA logic block can implement a large number of different logic functions, FPGA's present new challenges to conventional technology mapping approaches.

In this chapter, we examine the task of technology mapping for FPGA's. Since most developed techniques deal with combinational synthesis and optimize circuit area, we thus restrict our discussion to combinational synthesis methods with area as the optimization goal. Furthermore, for the development of the following chapters, the discussion is focused on techniques for multiplexer-based FPGA's.

Section 2.1 gives basic definitions. Section 2.2 and 2.3 present a brief review of FPGA architectures and logic synthesis, respectively. Section 2.4 describes a library-based technology mapping method which is the most successful method for technology mapping of technologies prior to FPGA's. Some of the techniques used in this approach have been followed by many other algorithms. In section 2.5, we describe some existing approaches for technology mapping of multiplexer-based FPGA's. Finally, we conclude with a summary in section 2.6.

2.1 Basic Definitions

A *logic* or *Boolean* variable x can take just two values, 0 or 1. Denote by $\bar{x}$ the complement of x. Both x and $\bar{x}$ are referred to as *literals.*

A *logic function* $f : \{0,1\}^n \rightarrow \{0,1\}$ is a function of logic variables and has value in $\{0,1\}$, where n is the number of logic variables.

A *cube* is a conjunction C of literals such that $x \in C$ implies $\bar{x} \notin C$, and each $x \in C$ can appear at most once. It can also be thought of as a subset of all points (also called vertices) in the input space $\{0,1\}^n$ that make the product of all its literals equal to 1.

In general, a logic function f can be thought of as the set of all input points, which satisfy $f(v) = 1$. This set is referred to as the *on-set* of f. Similarly, the set of all input points which satisfy $f(v) = 0$ is referred to as the *off-set* of f. In a more general situation, a logic function may be *incompletely specified* in that there is a set of vertices for which we do not care if the function has a value of 1 or 0. This set of vertices is called *don't-care-set* of f.

The *Shannon cofactor* of a logic function f with respect to a variable x, denoted by f_x, is the logic function obtained from f by setting the variable x to the constant value 1. Similarly, the cofactor of f with respect to $\bar{x}$, denoted by $f_{\bar{x}}$, is the logic function obtained by setting the variable x in f to value 0. For example, if

$$f = abx + cd\bar{x}$$

then

$$f_x = ab$$
$$f_{\bar{x}} = cd$$

A Boolean function can be represented in different forms.

A *sum-of-products form* for f is a disjunction of cubes that covers all the vertices of the on-set of f and none of the off-set. Any completely specified logic function can be represented as a sum of products.

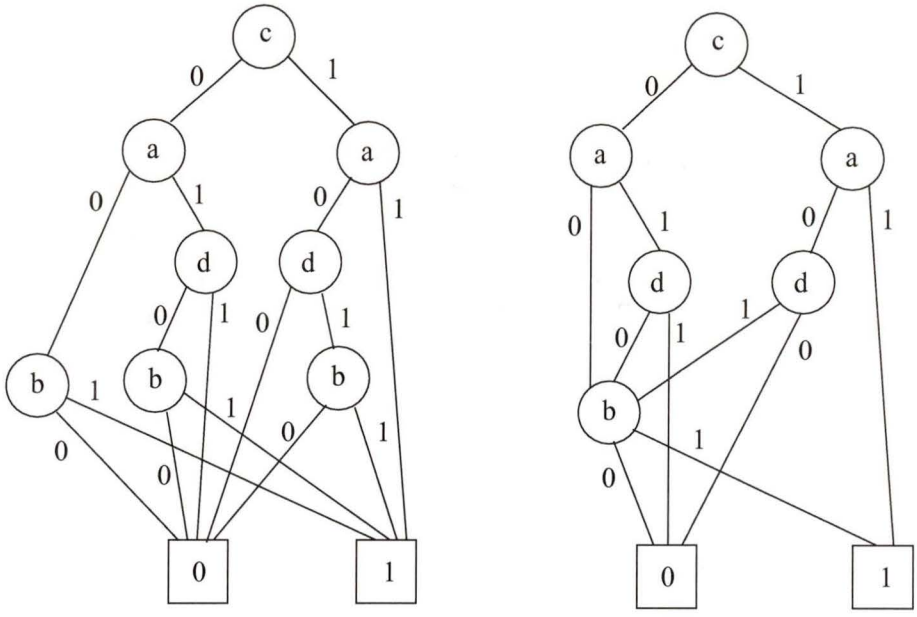


Figure 2.1: BDD and ROBDD.

A *binary decision diagram* (BDD) [20], [21] is a directed acyclic graph (DAG) representation of a logic function. It has a root node, two leaf nodes, and a number of intermediate nodes. Each non-leaf node has a variable and a function associated with it. The function associated with the root node is the function represented by the entire BDD. The two leaf nodes represent the constants 0 and 1 respectively. Each non-leaf node has two children representing the functions obtained by cofactoring the function represented at the node with respect to the variable associated with the node and its complement.

If the sequence of variables along any path in the BDD is restricted to a single given order and if no isomorphic subgraphs exist within the BDD, then the result is a canonical form known as a *reduced ordered BDD* (ROBDD). The number of nodes in an ROBDD can be dramatically lower than for the unreduced BDD, which makes it quite appealing for a number of applications. Figure 2.1 shows the ordered BDD and ROBDD for the function $f = ac + \bar{a}bd + b\bar{c}\bar{d}$ with the order c, a, d, b.

The problem with ROBDD's is that the complexity depends heavily on the vari-

able ordering and, unfortunately, finding an optimum ordering is NP-complete [21].

The *if-then-else* DAG (ITE DAG) [22] is another DAG representation of a logic function. In the ITE DAG representation, each node has three children and corresponds to a two-to-one multiplexer. The *if* child corresponds to the select line of the multiplexer. The *then* and *else* children correspond to the branches taken when the *if* child evaluates to 1 and 0 respectively and are mapped to the inputs of the multiplexer. It has been shown [22] that the ITE DAG is canonical, *i.e.*, two ITE DAG's are equal if, and only if, their Boolean expressions are also equal. Figure 2.2 shows an example ITE DAG. Details can be found in [22].

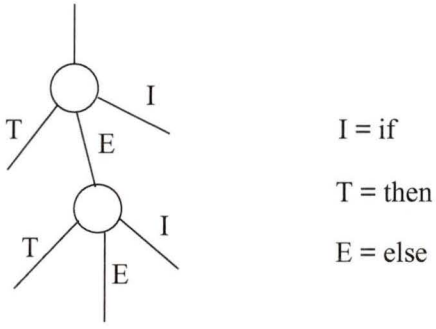


Figure 2.2: An example ITE DAG.

2.2 FPGA Architectures

Like a mask programmable gate array (MPGA), an FPGA consists of an array of possibly programmable logic blocks that can be programmably interconnected to implement different designs. Figure 2.3 shows a conceptual diagram of a typical FPGA. As depicted, it consists of a two-dimensional array of logic blocks that can be connected by general interconnection resources.

Generally, the architecture of an FPGA can be divided into two constituents: *logic block architecture* and *routing architecture* [24]. The routing architecture incorporates wire segments of varying lengths which can be interconnected via electrically programmable switches. Several different programming technologies are used to implement the programmable switches. Among these are:

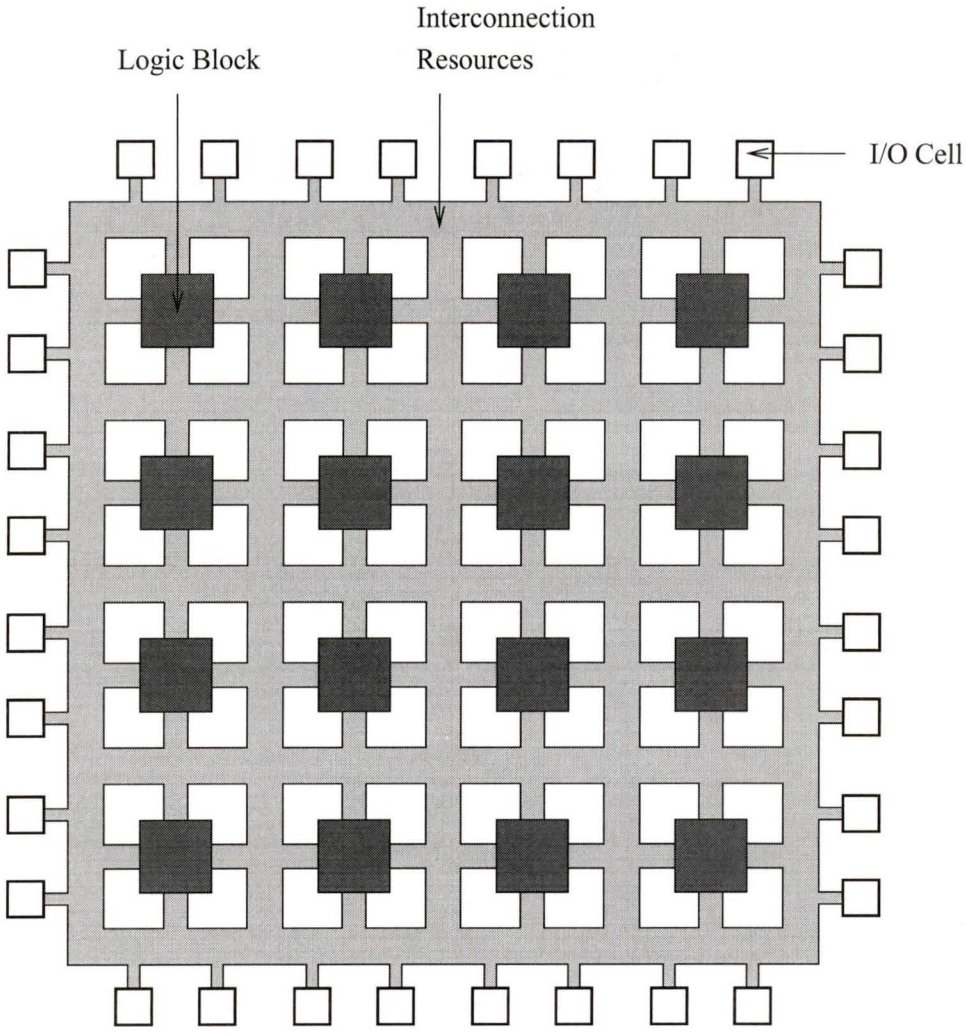


Figure 2.3: A conceptual FPGA.

- SRAM, where the switch is a pass transistor controlled by the state of a SRAM bit.
- Antifuse, which, when electrically programmed, forms a low resistance path.
- EPROM, where the switch is a floating-gate transistor that can be turned off by injecting charge onto their floating gate.

An FPGA logic block can be as simple as a transistor or as complex as a microprocessor. Generally, a logic block is capable of implementing many different combinational and sequential logic functions. The logic blocks of commercially available

FPGA's are based on one or more of the following:

- Transistor pairs.
- Simple logic gates such as two-input NAND's or exclusive-OR's.
- Multiplexers.
- Look-up tables (LUT's).
- Wide-fanin AND-OR structures.

We can see that FPGA logic blocks differ greatly in their size and implementation capability. For example, the two transistor logic block used in the Crosspoint FPGA can only implement an inverter but is very small in size, while the look-up table logic block used in the Xilinx 3000 series FPGA can implement any five-input logic function but is significantly larger. In order to capture these differences logic blocks are classified by their granularity into two categories: *fine-grain* and *coarse-grain.*

Coarse-grain FPGA's present new challenge to conventional technology mapping approaches because the logic blocks used in these FPGA's can each implement a large number of logic functions.

2.3 Logic Synthesis

The logic synthesis process is depicted in Figure 2.4. We assume that the original network is composed of a number of combinational functions. A network which contains flip-flops can be broken up into a set of combinational functions at flip-flop boundaries and then treated as a combinational network.

Logic synthesis begins with the technology independent phase. The original network is re-expressed in a certain internal form and a series of technology independent optimization operations are performed on the network to produce an optimal network in the same internal form. The technology mapping phase then maps the optimized

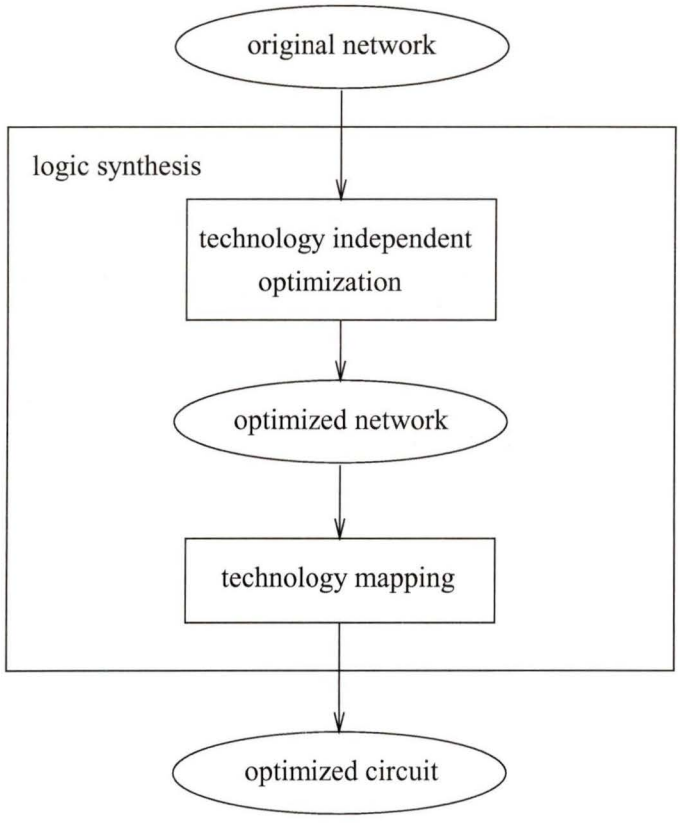


Figure 2.4: Logic synthesis.

network into a circuit composed of a set of cells that can be implemented in the target technology, while satisfying the area, delay, and testability constraints.

The internal representation of a combinational network chosen by many university and industrial tools is the *Boolean network*. A Boolean network is a directed acyclic graph (DAG). Each node i of the graph is associated with a Boolean variable x_i, and a representation of a logic function f_i. There is a directed arc from node j to node i if node i uses the variable x_j explicitly in the representation f_i. There are two sets of special nodes: input nodes with no incoming arcs which represent the *primary inputs* of the network, and output nodes with no outgoing arcs which represent the *primary outputs* of the network. Figure 2.5 shows an example of a Boolean network. Note that the function is specified with each node and there can be more than one instance of a given function.

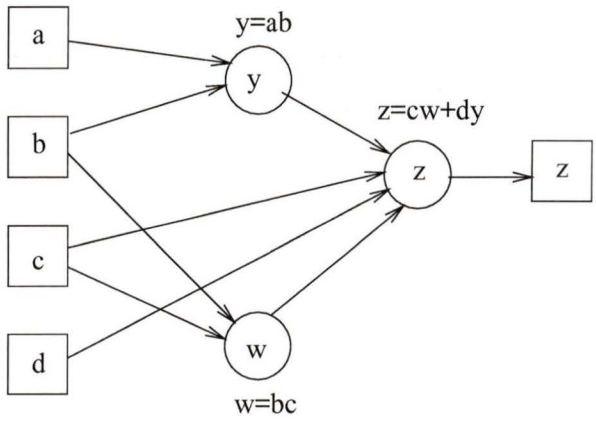


Figure 2.5: Example of a Boolean network.

2.3.1 Technology Independent Optimization

The objective of the technology independent optimization is to find the best multi-level structure for a given Boolean network. Most logic synthesis systems divide the technology independent optimization into two subproblems [25]:

1. modify the overall 'architecture' of the given logic network to produce a near-optimal 'structure' where common sublogic is identified.

2. optimize the logic with respect to the structure obtained in step 1.

The first step, called *network restructuring*, includes operations that modify the structure of the initial Boolean network by introducing new nodes, eliminating others, and by adding and removing arcs. The most important operation in this step is to identify common subexpressions among functions. In the misII synthesis system [28] the complexity of a Boolean network is measured in terms of the total number of literals in the local function for each node. Each local function is expressed in the sum-of-products form and each instance of a variable counts as one literal. For example, the following 4-input, 2-output network has 11 literals.

$$f = ac + ad + bc + bd$$
$$g = a + b + c$$

The complexity of the network can be reduced by the following modifications. The subexpression $(a + b)$ is factored out of the equations for f and g, and a new

node e implementing $(a + b)$ is created. The variable e is then substituted back into the equations for nodes f and g, and the original network is restructured as the following 5-input, 3-output network which has a total of 7 literals.

$$e = a + b$$
$$f = e(c + d)$$
$$g = e + c$$

The second step in technology independent optimization, called *node minimization*, simplifies the logic equations associated with nodes by exploiting the implicit *don't care* conditions that exist within the Boolean network. The result may lead to additional possibilities for restructuring which in turn may allow further optimization.

2.3.2 Technology Mapping

After the technology independent optimization of the Boolean network, technology mapping transforms the result into the final circuit which is optimum with respect to area, critical-path delay, or a combination of both. Typically, this is done by partitioning the network into a number of subnetworks and mapping each subnetwork to one of the available circuit elements, and specifying how these elements are to be interconnected to form the final circuit.

Existing approaches to the technology mapping problem fall into two main categories: rule-based techniques [2], [3] and algorithmic techniques [4]–[11], of which we shall only discuss the latter.

In the algorithmic approaches, a set of base functions is chosen such as a two-input NAND gate and an inverter. The Boolean network is converted into a graph where each node is restricted to one of the base functions. This graph is called the *subject graph*. The logic function for each of the available circuit elements is also converted into a graph, called a *pattern graph*, where the nodes are restricted to the same set of base functions. The technology mapping problem is then transformed into an optimization problem of finding a minimum cost covering of the subject graph by

choosing from the collection of pattern graphs for all available circuit elements. Since both the subject graph and the pattern graph are directed acyclic graphs (DAG's), the problem is referred to as *DAG covering by DAG's*. Unfortunately the problem is NP-hard [27], thus many heuristics have been proposed.

One heuristic [4] reduces the DAG-covering-by-DAG's problem to a set of tree-covering-by-trees problems. Another heuristic [10], [11] uses Boolean operations to find if a subnetwork is logically equivalent to one of the circuit elements, and is thus termed the *Boolean matching* method.

Conventionally, the set of available circuit elements is represented as a library of functions. In the case of FPGA's, the 'library' consists of the set of combinational functions that can be implemented by an FPGA logic block. However, the complex logic blocks of FPGA's can each implement a large number of functions, making the library-based approaches computationally expensive. The next section discusses the library-based tree covering heuristic, and the following sections describe technology mapping programs that deal specifically with multiplexer-based FPGA's. Technology mapping by Boolean matching is presented in Chapter 4.

2.4 Technology Mapping by Tree Covering

An important advance in technology mapping for conventional technologies was the library-based tree covering approach introduced by Keutzer in DAGON [4] and also used is misII [5]. This approach was inspired by the work on code generation for programming language compilers. It is shown that matching a graph-like description of a technology independent circuit against a technology library of patterns is similar to matching a graph-like intermediate representation of a computer program against the patterns of an instruction set of a given machine.

The DAGON approach, illustrated in Figure 2.6, consists of three steps: decomposition, partitioning, and matching and covering.

The optimized technology independent network is first decomposed into a canonical representation where each node represents either a two-input NAND gate or

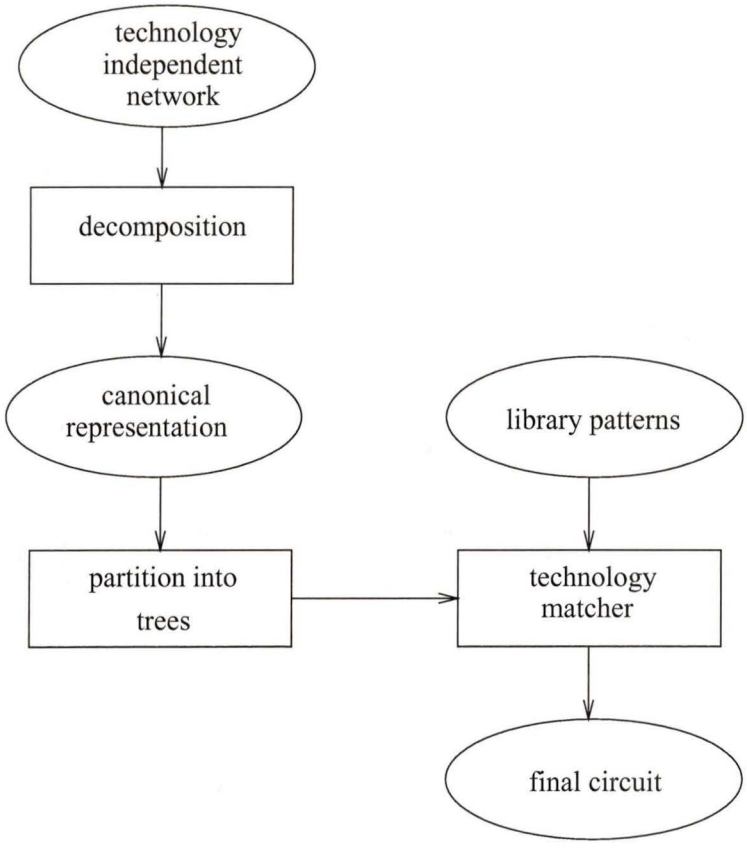


Figure 2.6: The DAGON approach to technology mapping.

an inverter. This decomposition guarantees that, as long as the library includes a two-input NAND gate, any node in the network can be implemented by the library cells. However, there is no guarantee that the decomposition is optimal since there are many possible two-input NAND decompositions.

After decomposition, the network is partitioned into a forest of trees by making each node with fanout greater than one the root of a new tree. Figure 2.7 shows the partitioning of a circuit into trees. X's mark the partition points.

The next step is to construct an optimal subcircuit covering each of the trees which is composed of the library cells. Finally, all the subcircuits are assembled to form the overall realization.

The optimal subcircuit covering each tree is constructed in two steps. First, a set of candidate matches for each of the nodes is identified by searching through the library and using tree matching to determine if each library cell matches the

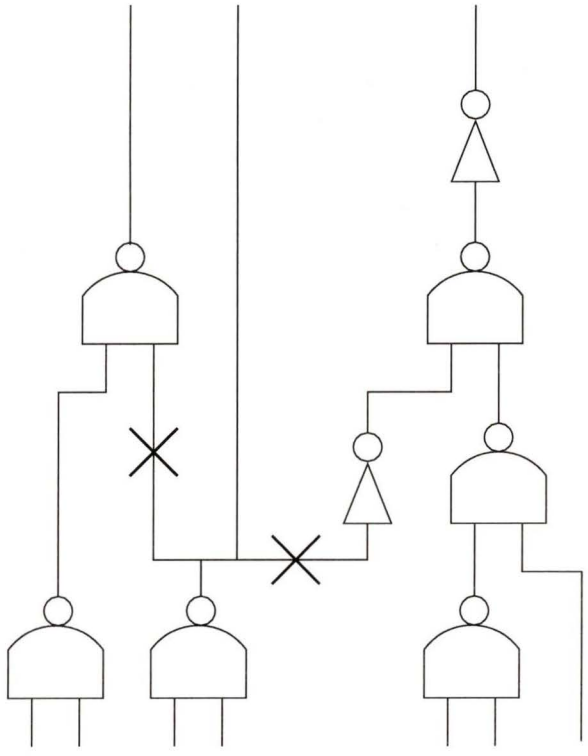


Figure 2.7: Partitioning a circuit in DAGON.

subfunction represented by the node. Then the optimal cover for the whole tree is determined using a dynamic programming traversal that proceeds from the leaf nodes to the root node. For each node r in the tree an optimal circuit implementing the subtree rooted at r is constructed by simply trying each candidate match p_i at r. The leaves of p_i will extend into the subtrees of r. The optimal circuit for each of the subtree has already been constructed in the bottom-up procedure, so the cost of matching p_i at r can be computed in terms of the cost of p_i together with the cost of each of the subtrees with roots at the leaves of p_i. After trying all the candidate matches at r, the match that yields optimal result is retained and the procedure continues upwards until the optimal match for the root node is determined.

As an example of the covering procedure, consider the technology library shown in Figure 2.8 and the Boolean network in Figure 2.9. The library cells in Figure 2.8 are standard cells and the costs are given in terms of the area of the cells. In Figure 2.9, the root of the tree is node A. The covering process starts with the leaf nodes.

The only candidate match for node E is the NAND-2, so the cost of the optimal circuit covering node E is 3. Similarly, the cost of the optimal circuit covering node C is 2. At node D, the only candidate match is the NAND-2. Therefore the optimal circuit covering node D is the cost of the NAND-2 matching node D plus the cost of the optimal circuit covering the inputs of this NAND-2: one primary input (of cost 0), and node E (of cost 3). So, the total cost of the optimal circuit covering node D is 6.

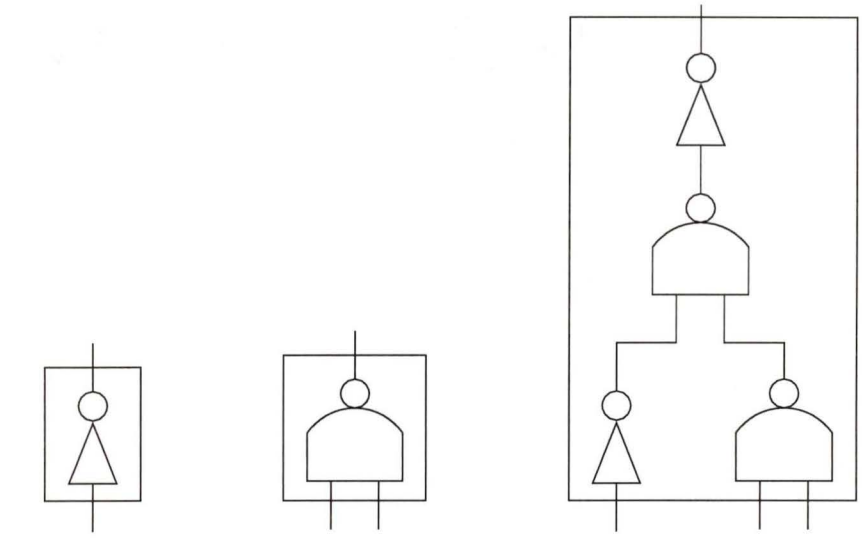


Figure 2.8: An example technology library.

Eventually we will reach the root node A. There are two candidate matches for node A: the INV with a cost of 2, and the AOI-21 with a cost of 4. The cost of matching node A with an INV would be the cost of the INV (3) plus the cost of the subtree rooted at node B, which in turn is the cost of matching node B with a NAND-2, matching node C with an INV, matching node D with a NAND-2, and matching node E with a NAND-2. The total cost of covering the whole tree would be 13. On the other hand, the cost of matching node A with an AOI-21 is the cost of the AOI-21 (4) plus the cost of the subtree rooted at the leaves of the AOI-21: two inputs (of cost 0) and node E (of cost 3). The total cost the covering the whole tree is 7. Therefore, the circuit using the AOI-21 is the optimal circuit for covering node

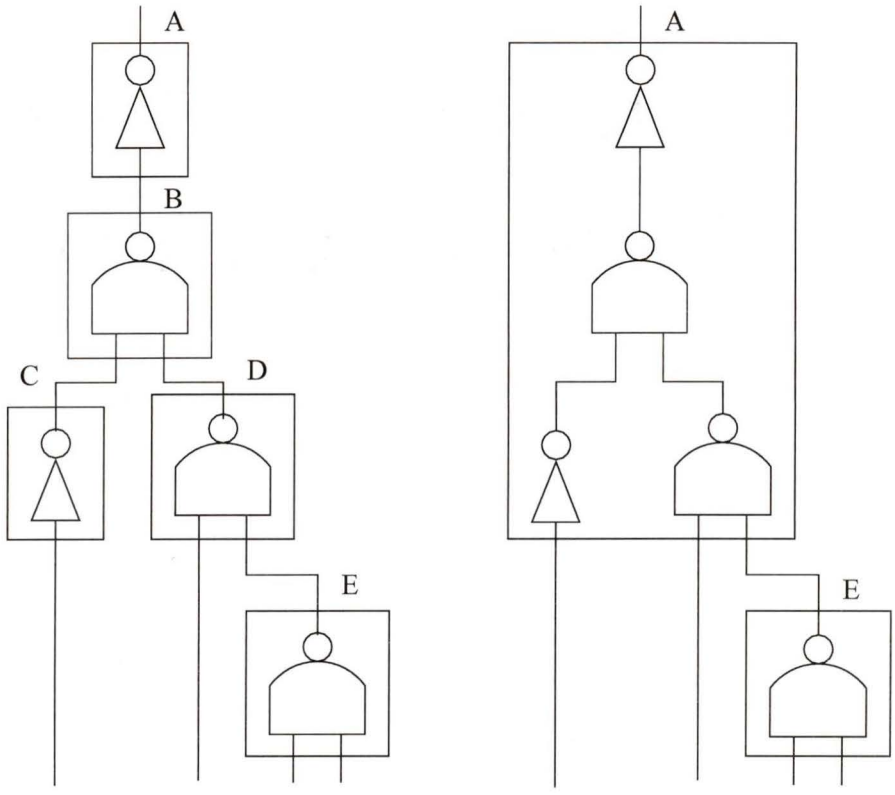


Figure 2.9: Tree covering using dynamic programming.

A.

The DAGON approach is very efficient since it is based on a firm algorithmic foundation: the optimal tree matching algorithm and dynamic programming traversal. The running time of the approach is linear in the size of the trees. However, partitioning the original network into trees precludes the possibility of covering across tree boundaries, and thus loses the guarantee of a globally optimal solution. Moreover, like other library-based techniques, the efficiency of the DAGON approach depends on the size of the library. This presents a major obstacle to its application to FPGA's.

2.5 Technology Mapping for Multiplexer-Based FPGA's

Multiplexer-based FPGA's use logic blocks that are combinations of a number of multiplexers and possibly a few additional logic gates such as AND and/or OR gates. For example, the Actel Act-1 logic block, illustrated in Figure 2.10, consists of three two-to-one multiplexers and a two-input OR gate. It has a total of eight inputs and one output, and implements the Boolean function

$$F = (g + h)(ac + b\bar{c}) + (\overline{g + h})(df + e\bar{f})$$

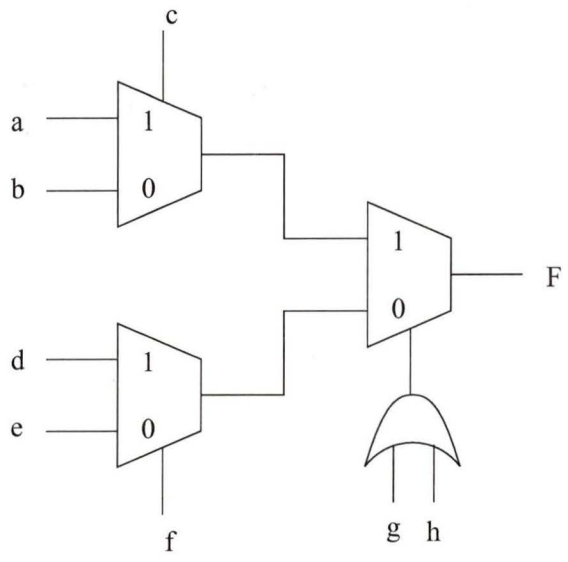


Figure 2.10: The Actel Act-1 logic block.

By connecting each variable to an input signal or to constants 0 and 1, or bridging together some of the inputs, an uncommitted logic block is *personalized* to implement different functions. For example, the Act-1 logic block can be personalized to implement the function $x_1\bar{x}_2 + \bar{x}_1x_2$ by making the input connections $a = 0$, $b = x_1$, $c = x_2$, $d = 0$, $e = x_2$, $f = 0$, $g = 0$, $h = x_1$.

Multiplexer-based logic blocks can each implement a large number of functions. A technology library composed of all these functions is certainly too large for conventional library-based technology mapping algorithms to efficiently manipulate. Al-

gorithms based on dynamic programming are quite effective for libraries with one or two hundred cells, but are too slow for significantly larger libraries. Therefore, library reduction is used to reduce the number of library cells. However, such reduction may impede significant optimizations [27].

In the following sections we describe some direct approaches to technology mapping for multiplexer-based FPGA's. These approaches make use of the structure of the logic block and do not require a library to be created.

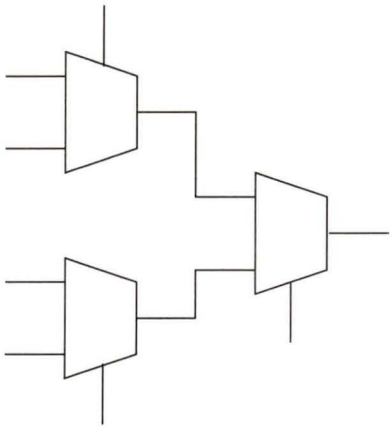
2.5.1 The *mis-pga* Approach

The *mis-pga* [6] technology mapper attempts to minimize the number of Actel Act-1 logic blocks required to implement a Boolean network. The basic idea was described earlier. The Boolean network is converted to a subject graph in terms of the base functions. Each of the logic functions that a logic block can implement is converted to a pattern graph. The subject graph is then covered by a set of pattern graphs such that the cost of this set is minimized.

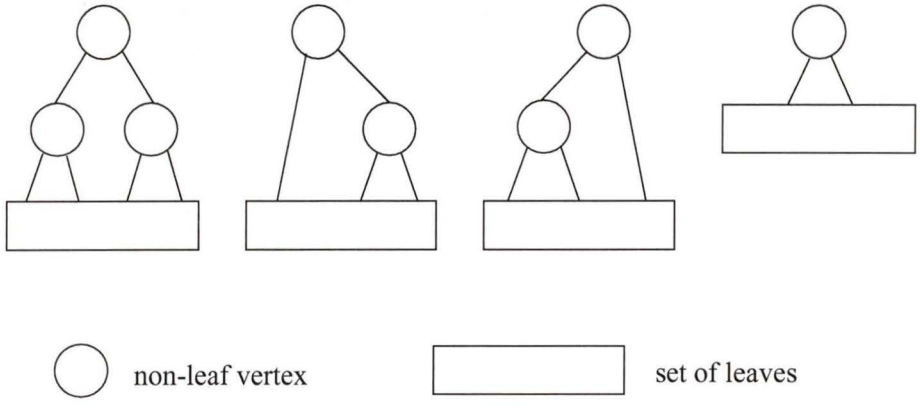
Since the number of possible functions that a logic block can implement is very large, the set of pattern graphs is also very large. However, if the structure of the logic block contains only two-to-one multiplexers, just a few pattern graphs would be able to characterize the whole library.

For example, only four patterns suffice for describing the structure of the simplified Act-1 logic block where the OR gate is ignored. Figure 2.11 shows the simplified Act-1 logic block and the four pattern graphs. If a function can be implemented by a simplified Act-1 block, then it either uses all the three multiplexers, or two of them, or just one. The four pattern graphs in Figure 2.11 are in one-to-one correspondence with these possibilities. Taking into account the OR gate, *mis-pga* uses a set of eight pattern graphs to represent the Act-1 logic block.

It is shown in [6] that constructing a subject graph for each node in the Boolean network leads to better results than constructing a single subject graph for the entire network. Thus, *mis-pga* constructs a subject graph for each of the nodes. Two



a) simplified Act-1



b) pattern graphs for the simplified Act-1

Figure 2.11: The simplified Act-1 and its pattern graphs.

heuristics are used.

In one heuristic [6], an ROBDD is used as a representation for the subject graph because it is compact and has no duplication of logic. As mentioned in section 2.1, the size of the ROBDD of a function depends heavily on the ordering of the input variables, and the problem of finding the optimum ordering is NP-complete. However, if the function has a small number of inputs, an optimum ordering can be determined quickly by trying out all possible input permutations. Therefore, in this heuristic, a decomposition step is performed first on the network to force all nodes to have at most K inputs. Then, for each node, the ROBDD's corresponding to all the input

orderings are constructed. The cost of each ROBDD is evaluated and the one with minimum cost is used.

In another heuristic [6], a BDD is constructed for each node in the network such that:

- the number of vertices in the BDD is small;
- the number of vertices with multiple parents is small.

Since it is not possible to predict which representation yields the best result, *mis-pga* uses both ROBDD and BDD and the best result is selected.

As we can see from Figure 2.1, a BDD/ROBDD vertex corresponds to a two-to-one multiplexer. Thus, both the subject graph and the pattern graphs are represented in terms of two-to-one multiplexers. *mis-pga* then uses the tree-covering heuristic proposed in DAGON to cover the subject graph with pattern graphs. Since the set of pattern graphs is small, the covering procedure is fast. Finally, an iterative improvement phase performs local transformations on the network to improve the final result.

2.5.2 The *Amap* Approach

The objective of the *Amap* [9] technology mapper is also to minimize the number of Actel Act-1 logic blocks required to implement a Boolean network. *Amap* tries to match the multiplexer-based structure of Act-1 with a multiplexer-based representation of the Boolean network. The representation chosen is the if-then-else DAG (ITE DAG). As shown in section 2.1, each node in the ITE DAG corresponds to a two-to-one multiplexer. Moreover, the main advantage of ITE DAG's over BDD's is that duplication of cubes can be avoided.

Amap creates the ITE DAG for the Boolean network and then performs two passes of operations to do the mapping. The first pass is a preprocesses pass which performs some local operations on the ITE DAG to make the second pass work better. Specifically, the mapper traverses the ITE DAG to search for two-input commutative

functions. Single literal inputs are interchanged, if necessary, to bring them to the *if* part. Interchanging can affect whether or not the OR gate is usable, and can affect the polarity of the inputs. For example, $a + \bar{b}$ can be represented as "if a then TRUE else $\bar{b}$" or "if b then a else TRUE". The first pass also assigns an appropriate polarity for each node.

After the interchanging and polarity assignments, the ITE DAG is mapped against the Act-1 cells using greedy heuristics. Starting from the root nodes of the ITE DAG, *i.e.*, the primary outputs of the Boolean network, the *Amap* algorithm processes the nodes in a recursive manner. It maps each root node against an Act-1 logic block, and then recursively maps the inputs of this Act-1 block, and so on. The recursion stops when the node to be mapped is either a primary input or it has already been implemented in some Act-1 block.

For the nodes other than the primary inputs, the algorithm has to make a tradeoff as to how much of the Act-1 block to use. Placing as much of the ITE DAG as possible into an Act-1 block results in considerable duplication of logic since nodes that could have been shared get hidden inside the block. On the other hand, placing too little of the ITE DAG into a block leaves much of the logic capacity of the block unused, so more blocks than necessary are required to implement the circuit.

The heuristic used by *Amap* is as follows. At each node, the *if* branch variable maps to the OR gate. If it is already implemented or if it has high fanout, the OR gate is used as a simple buffer by connecting one of its inputs to constant 0. Otherwise, the variable at the *if* branch is expressed as an OR or NOR, and the two inputs are connected to the OR gate. After the OR gate has been mapped, the *then* and *else* branches are mapped to the input multiplexers. If the variables corresponding to the branches are already implemented or if they have high fanout, the input multiplexers are used merely as buffers. Otherwise the next lower level node is included in the block.

2.6 Summary

In this chapter, we presented an overview of FPGA architectures and the logic synthesis process. A library-based technology mapping approach was described. This approach was very successful for technologies prior to FPGA's. However, it is inappropriate for FPGA's because the logic blocks used in FPGA's can each implement a large number of different functions. This chapter described some technology mapping algorithms that deal specifically with multiplexer-based FPGA's. Rather than enumerating the functions a logic block can implement, these algorithms use the structure of the logic block itself to represent the entire set of functions. Mapping is performed directly onto the logic blocks.

Chapter 3

Spectral Techniques

As is well known, frequency domain techniques, such as the Laplace transform and the Fourier transform, are very powerful techniques in the design and analysis of analog systems. The spectral techniques presented in this chapter play a similar role in the digital logic domain. Many problems that are difficult to solve in the original Boolean domain have efficient and elegant solutions in the spectral domain [19].

In this chapter, we present an overview of spectral techniques. The main advantage of the spectral domain representation of a Boolean function over the Boolean domain representation is that it provides information about the function that is much more global in nature. This advantage makes spectral techniques very useful in many digital design and test applications. Two applications that are relevant to the development of this thesis are presented: Boolean function classification and detection of symmetry conditions in Boolean functions.

3.1 Spectral Domain

A Boolean function $f(X)$, $X = \{x_1, x_2, \ldots, x_n\}$, is a mapping, $f : \{0,1\}^n \rightarrow \{0,1\}$, which associates either a logic 0 or 1 with each of the 2^n combinations of the input variables. $f(X)$ can be described by its truth-table which lists all the 2^n input combinations and the corresponding output values. Conventionally, the input combinations are tabulated in a standardized way, *i.e.*, in ascending order with x_n being

the highest order variable. Thus, function $f(X)$ may also be uniquely defined by a column vector $\mathbf{Y}$ which consists of the 2^n output values.

Alternatively, we can define another column vector $\mathbf{Z}$ by using the coding $\{1, -1\}$ instead of $\{0, 1\}$ for the output values, *i.e.*, a logic 0 is coded as 1 and a logic 1 is coded as -1. Clearly $f(X)$ is also uniquely defined by $\mathbf{Z}$. As an example, both the coding schemes $\{0, 1\}$ and $\{1, -1\}$ for the function $f(X) = x_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3 + x_1x_2x_3$ are shown in Table 3.1.

Table 3.1: Truthtable of function $f(X) = x_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3 + x_1x_2x_3$ in $\{0, 1\}$ and $\{1, -1\}$ coding schemes.

x_3	x_2	x_1	**Y**	**Z**
0	0	0	0	1
0	0	1	1	-1
0	1	0	1	-1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	-1

A major disadvantage of these representations of Boolean functions is that each entry in the column vectors only provides the information about the function behavior under one input combination. No information about the output value of $f(X)$ under any input combination can be obtained from any other input condition. This prevents us from identifying the global characteristics of the function from just a few entries in the column vectors.

With spectral techniques, the Boolean domain representation can be transformed to spectral domain representation, or *spectrum*, of $f(X)$. Like the truth-table, the spectrum for a function of n variables also contains 2^n values called *spectral coefficients*. Unlike the entries in the column vectors discussed above, each of the spectral

coefficients is the result of some operation performed on some of the Boolean domain function values, so each spectral coefficient can represent some global characteristics of the function. In this sense the spectral coefficients provide us global information about the function, whereas the Boolean domain consists of local information.

The general framework of spectral techniques is illustrated in Figure 3.1.

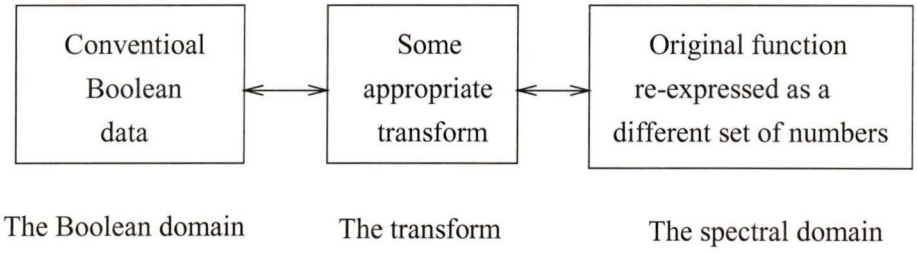


Figure 3.1: The spectral transform.

The Boolean domain representation is converted to the spectral domain representation through some kind of transform. Often the Boolean domain representation is in the form of truth-tables, and the transform can be described by a matrix multiplication. To avoid information loss due to the transform, we must ensure that the transform is invertible.

Once an appropriate spectrum is obtained, the logic design procedures involve examination of certain spectral coefficients or recognition of certain patterns in the spectrum.

Many useful transforms have been developed. However, for the development of this thesis, we only present a review of the Walsh transform.

3.2 The Walsh Transform

Given a Boolean function $f(X)$, let X denote both the binary vector $X = \{x_n, \ldots, x_1\} \in \{0,1\}^n$ and the corresponding decimal number $X = \sum x_i 2^i$. Let $\mathbf{Z}$ denote the output column vector of $f(X)$ in the $\{1,-1\}$ coding scheme.

The Walsh transform of $f(X)$ is defined as follows:

$$\hat{f}(\omega) = \sum_X W_\omega(X) f(X), \qquad 0 \le \omega \le 2^n - 1 \tag{3.1}$$

$$f(X) = 2^{-n} \sum_\omega W_X(\omega) \hat{f}(\omega), \qquad 0 \le X \le 2^n - 1 \tag{3.2}$$

where $\omega = \sum_i 2^i \omega_i$, $0 \le i \le n-1$, $\omega_i \in \{0, 1\}$.

The Walsh functions $W_\omega(x)$ are defined as

$$W_\omega(x) = (-1)^{\sum_i \omega_i x_i}, \quad 0 \le i \le n-1 \tag{3.3}$$

Equation (3.1) is the Walsh transform; (3.2) is the inverse Walsh transform.

The *Rademacher-Walsh spectrum* **S** of $f(X)$ can be computed by a matrix multiplication

$$\mathbf{S} = \mathbf{W}^n \mathbf{Z} \tag{3.4}$$

where the $2^n \times 2^n$ matrix $\mathbf{W}^n$ is called the *Walsh transform matrix*, each element of which is a Walsh function as defined above.

For example, the $n = 3$ Walsh transform matrix and the spectrum of the 3-variable function $f(x_3, x_2, x_1) = x_3 \bar{x}_2 x_1 + \bar{x}_3 x_2 \bar{x}_1$ are shown in (3.5).

$$\begin{matrix} \mathbf{S} \end{matrix} \qquad \begin{matrix} \qquad\qquad\qquad \mathbf{W}^3 \qquad\qquad\qquad \end{matrix} \qquad \begin{matrix} \mathbf{Z} \end{matrix}$$

$$\begin{bmatrix} 4 \\ 0 \\ 4 \\ 0 \\ 4 \\ 0 \\ -4 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \\ 1 \\ -1 \\ 1 \\ 1 \end{bmatrix} \tag{3.5}$$

The *Rademacher-Walsh spectrum* **S** of $f(X)$ in **Hadamard order** [18] can be computed by a matrix multiplication

$$\mathbf{S} = \mathbf{T}^n \mathbf{Z} \tag{3.6}$$

where the $2^n \times 2^n$ matrix $\mathbf{T}^n$ is called the *Hadamard transform matrix* and is defined recursively as

$$\mathbf{T}^0 = [1] \tag{3.7}$$

$$\mathbf{T}^n = \begin{bmatrix} \mathbf{T}^{n-1} & \mathbf{T}^{n-1} \\ \mathbf{T}^{n-1} & -\mathbf{T}^{n-1} \end{bmatrix} \tag{3.8}$$

Note that the dimension of the transform is $2^n \times 2^n$ for any n. For increasing n the transforms have the following structure:

$$\mathbf{T}^1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$\mathbf{T}^2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}, \quad etc.$$

For example, using the $n = 3$ Hadamard transform matrix, we can compute the spectrum of the 3-variable function in equation (3.5) as follows.

$$\begin{matrix} \mathbf{S} \end{matrix} \qquad \begin{matrix} \qquad\qquad\qquad\qquad \mathbf{T}^3 \qquad\qquad\qquad\qquad \end{matrix} \qquad \begin{matrix} \mathbf{Z} \end{matrix}$$

$$\begin{bmatrix} 4 \\ 0 \\ 0 \\ 4 \\ 0 \\ -4 \\ 4 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \\ 1 \\ -1 \\ 1 \\ 1 \end{bmatrix} \tag{3.9}$$

Comparing equation (3.9) with (3.5), we can see that the spectral coefficients are the same, but they are in a different order.

Rather than the matrix multiplication of equations (3.4) and (3.6), a more efficient butterfly algorithm [18] is usually used to compute the Rademacher-Walsh spectrum. The time complexity of the algorithm is $O(n2^n)$ and the space complexity is $O(2^n)$, which is quite reasonable for technology mapping applications where the functions typically have up to 10 inputs. For larger functions, efficient techniques have also been developed using decision diagram techniques [16].

As can be seen from equations (3.1) and (3.2), the Walsh transform is reversible. This ensures that no information is lost in the transform. Moreover, It can be readily shown that

$$(\mathbf{T}^n)^{-1} = \frac{1}{2^n}\mathbf{T}^n \tag{3.10}$$

Hence, the inverse transform back from the spectral domain to the Boolean domain is given by

$$\mathbf{Z} = \frac{1}{2^n}\mathbf{T}^n\mathbf{S} \tag{3.11}$$

Therefore, each function has a unique spectrum. The spectrum can be used to uniquely define the function.

3.3 The Meaning and Order of the Spectral Coefficients

We now examine the meaning of the Rademacher-Walsh spectral coefficients.

First of all let us take a look at the row vectors of the Hadamard transform matrix $\mathbf{T}^n$. Reading 1 as logic 0 and -1 as logic 1, we can see that the row vectors are related to the input variables as follows:

Dimension	Transform	Row vector meaning
$n = 1$	$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$	Constant 0 Input variable x_1
$n = 2$	$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$	Constant 0 Input variable x_1 Input variable x_2 Function $x_1 \oplus x_2$
$n = 3$	$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix}$	Constant 0 Input variable x_1 Input variable x_2 Function $x_1 \oplus x_2$ Input variable x_3 Function $x_1 \oplus x_3$ Function $x_2 \oplus x_3$ Function $x_1 \oplus x_2 \oplus x_3$

Each of the row vectors represents an exclusive-OR function of some subset of the input variables of $f(X)$. The row vector corresponding to $x_i \oplus x_j$, $1 \leq i, j \leq n$, $i \neq j$, can be obtained by an exclusive-OR of the vectors corresponding to x_i and x_j. Referring back to the example transform in (3.9), it will be apparent that the value of each resulting coefficient in **S** is equal to the number of occasions the minterm values in **Z** agree with the corresponding Hadamard row values minus the number of times the two values disagree. Therefore, each spectral coefficient in **S** represents a correlation between the **Z** vector of $f(X)$ and the corresponding Hadamard row function. Thus each spectral coefficient represents a correlation between $f(X)$ and a particular exclusive-OR function of the input variables.

We identify the spectral coefficients by subscripts indicating which variables are involved in the corresponding exclusive-OR function, as follows:

s_0 is the correlation of $f(X)$ and the constant 0 function;

s_i, $1 \leq i \leq n$, is the correlation between $f(X)$ and the input variable x_i;

s_{ij}, $1 \leq i, j \leq n$, $i \neq j$, is the correlation between $f(X)$ and the exclusive-OR function $x_i \oplus x_j$;

and so on for higher order coefficients. All the spectral coefficients are numerically equal to $\{\sum$ agreements between output $f(X)$ and the appropriate exclusive-OR function $- \sum$ disagreements between $f(X)$ and the exclusive-OR function$\}$.

For example, for $n = 3$ we have

$$\mathbf{S} = \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_{12} \\ s_3 \\ s_{13} \\ s_{23} \\ s_{123} \end{bmatrix} \tag{3.12}$$

The spectral coefficients are often re-grouped in terms of the number of subscripts they have, as follows.

s_0 :	the zero-order coefficient
s_i, $i = 1$ to n :	the primary or first-order coefficients
s_{ij}, $ij = 12, 13, \ldots$:	second-order coefficients
s_{ijk}, $ijk = 123, 124, 134, \ldots$:	third-order coefficients
$\vdots$	

Collectively, all second- and higher-order coefficients are termed the secondary coefficients.

The following are some of the properties of the spectral coefficients. Formal proofs of these properties can be found elsewhere [18].

1. The sum of all spectral coefficients for any fully defined function $f(X)$ is $\pm 2^n$.

2. The maximum value of any coefficient is $\pm 2^n$. The range for each coefficient is $\{-2^n, -2^n + 2, \ldots, 0, \ldots, 2^n - 2, 2^n\}$.

3. When any individual coefficient is maximum-valued, all remaining $2^n - 1$ coefficients will be zero-valued.

4. When any input variable x_i is redundant in a given function $f(X)$, the 2^{n-1} spectral coefficients that contain i in their subscripts will all be zero-valued.

3.4 Classification of Boolean Functions

In this section we shall see how spectral information can be used to classify Boolean functions.

The number of possible different functions of n independent variables is 2^{2^n}, including degenerate functions[1]. For large n, this number becomes very large and thus very difficult to manipulate. Therefore, it is desirable to classify the functions of similar structure or complexity into a set of equivalence classes, and to establish a small set of 'representative functions', one from each class. The number of representative functions can be dramatically smaller than 2^{2^n}. Any function in a particular class can be realized by the implementation of the representative function for the class and some appropriate operations that transform the function into the representative function.

Various classification methods have been developed. In this section, we discuss the PN classification, the NPN classification, and the spectral classification.

[1] A degenerate function of n input variables is a function which does not depend on all the n inputs to determine its output.

3.4.1 The PN and NPN Classification

The PN and NPN classification methods are both algebraic classification methods, in contrast to the spectral method. The algebraic classification methods involve the consideration of the following operations, taken individually or collectively:

1. Negation (N) of one or more of the input variables of the function.
2. Permutation (P) of two or more of the input variables of the function.
3. Negation (N) of the output of the function.

Functions that are equivalent under operation (1) or (2) are said to be N-equivalent and P-equivalent functions, respectively. Functions that are equivalent under both operations (1) and (2) are said to be PN-equivalent functions, or to belong to the the same PN-equivalence class. Finally, functions that are equivalent under all the three operations (1), (2), and (3) are said to be NPN-equivalent, or to belong to the same NPN-equivalent class. As an example, consider the following four functions:

$$f_1(X) = x_1\bar{x}_2 + x_2x_3$$

$$f_2(X) = x_1x_2 + \bar{x}_2x_3$$

$$f_3(X) = x_1x_3 + x_2\bar{x}_3$$

$$f_4(X) = \bar{x}_1x_3 + \bar{x}_2\bar{x}_3$$

Function $f_2(X)$ can be obtained from $f_1(X)$ by the input permutation $x_3x_2x_1x_4$, *i.e.*, exchanging x_1 and x_3, so $f_1(X)$ and $f_2(X)$ are in the same P-equivalent class. Similarly, functions $f_1(X)$ and $f_3(X)$ are in the same NP-equivalence class, since they are related by the negation $x_2 \rightarrow \bar{x}_2$ and exchanging x_2 and x_3. And finally, $f_1(X)$ and $f_4(X)$ are in the same NPN-equivalence class because $f_4(X)$ is the overall negation of $f_3(X)$. Therefore, the four functions are in the same NPN-equivalence class.

Table 3.2: PN and NPN classification statistics.

Number of inputs variables	2	3	4	5	6
Total number of functions	16	256	65,536	$\approx 4.3 \times 10^9$	$\approx 1.8 \times 10^{19}$
PN Classification	6	22	402	1,228,158	$\approx 4 \times 10^{14}$
NPN Classification	4	14	222	616,126	$\approx 2 \times 10^{14}$

Table 3.2 [18] shows the number of PN and NPN equivalence classes for functions of up to six variables.

As mentioned above, each equivalence class can be represented by a representative function from the class. For instance, for the PN and NPN classification, we have the following representative functions for all possible $n \leq 2$ functions.

Representative functions in PN-classification:

$f(X) = 0$ (one function in the class)
$f(X) = 1$ (one function in the class)
$f(X) = x_1$ (four functions in the class)
$f(X) = x_1 x_2$ (four functions in the class)
$f(X) = x_1 + x_2$ (four functions in the class)
$f(X) = x_1 \oplus x_2$ (two functions in the class)

Representative functions in NPN-classification:

$f(X) = 1$ (two functions in the class)
$f(X) = x_1$ (four functions in the class)
$f(X) = x_1 + x_2$ (eight functions in the class)
$f(X) = x_1 \oplus x_2$ (two functions in the class)

3.4.2 Spectral Classification

Spectral techniques were discussed in the previous sections. It was shown that the Boolean domain representation of a function $f(X)$ can be transformed into the spectral domain representation, and vice versa, without any loss of information. The Rademacher-Walsh spectrum **S** of $f(X)$ is uniquely related to $f(X)$. We shall now consider how the spectral coefficients can be applied to function classification.

Comparing the spectra of a number of functions of any given n, we will find that identical sets of magnitudes occur in many spectra, although with position and sign changes. This suggests that it is possible to classify the functions by the set of magnitudes of the spectral coefficients.

Consider the following four 3-variable functions.

$$f_1(X) = x_1x_2 + \bar{x}_2x_3 + x_2\bar{x}_3$$
$$f_2(X) = x_1\bar{x}_2 + x_2x_3 + \bar{x}_2\bar{x}_3 = \text{negation of } x_2 \text{ in } f_1(X)$$
$$f_3(X) = x_1\bar{x}_3 + x_2x_3 + \bar{x}_2\bar{x}_3 = \text{permutation of } x_2 \leftrightarrow x_3 \text{ in } f_2(X)$$
$$f_4(X) = \bar{x}_2x_3 + \bar{x}_1x_2\bar{x}_3 = \text{negation of } f_3(X)$$

Clearly the four functions belong to the same NPN-equivalence class. Their spectra are

	s_0	s_1	s_2	s_3	s_{12}	s_{13}	s_{23}	s_{123}
$f_1(X)$:	−2	2	2	2	−2	−2	6	2
$f_2(X)$:	−2	2	−2	2	2	−2	−6	−2
$f_3(X)$:	−2	2	2	−2	−2	2	−6	−2
$f_4(X)$:	2	−2	−2	2	2	−2	6	2

We can see that the magnitudes

$$|2|, |2|, |2|, |2|, |2|, |2|, |6|, |2|$$

characterize these four NPN-equivalent functions.

It has been shown [18] that there are a total of five functional operations that are easily implemented in the spectral domain. These operations are referred to

as *invariance operations* because the set of magnitudes of the spectral coefficients remain invariant under any of these operations.

The invariance operations are listed below. We shall use the Greek letter α to denote a set of variable indices. α is a subset, possibly empty, of $\{1, 2, \ldots, n\}$.

1. Permutation of any pair of input variables x_i and x_j, $i \neq j$, $i, j \in \{1, 2, \ldots, n\}$: this requires the interchange of 2^{n-2} pairs of spectral coefficients:

$$s_{i\alpha} \leftrightarrow s_{j\alpha}, \quad i, j \notin \alpha$$

2. Negation of any input variable x_i, $i \in \{1, 2, \ldots, n\}$: this requires the negation of 2^{n-1} spectral coefficients:

$$s_{i\alpha} \rightarrow -s_{i\alpha}, \quad i \notin \alpha$$

3. Negation of the function output: this requires the negation of all 2^n spectral coefficients:

$$s_0 \rightarrow -s_0$$
$$s_\alpha \rightarrow -s_\alpha, \quad \alpha \neq \emptyset$$

4. Replacement of any input variable x_i with $x_i \oplus x_j, i \neq j, i, j \in \{1, 2, \ldots, n\}$: this requires the interchange of 2^{n-2} pairs of spectral coefficients:

$$s_{i\alpha} \leftrightarrow s_{ij\alpha}, \quad i, j \notin \alpha$$

   Note that coefficients $s_0, s_{j\alpha}$, $i, j \notin \alpha$ remain unchanged.

5. Replacement of the function output $f(X)$ with $f(X) \oplus x_i, i \in \{1, 2, \ldots, n\}$: this requires the interchange of 2^{n-1} pairs of spectral coefficients:

$$s_i \leftrightarrow s_0$$
$$s_{i\alpha} \leftrightarrow s_\alpha, \quad \alpha \neq \emptyset, \quad i \notin \alpha$$

   Note that all 2^n coefficients are involved in this operation.

The first three operations are the NPN operations that we have seen before and they do not alter the order of any spectral coefficient. The fourth operation interchanges certain spectral coefficients from all orders except s_0, and the fifth operation interchanges the order of all coefficients including s_0.

We now can see that the spectral coefficients provide an alternative way to classify Boolean functions, *i.e.*, we can classify functions in terms of the set of magnitudes of the spectral coefficients. Furthermore, since the five invariance operations fully cover the NPN operations and have two extra operations, we can expect that the spectral classification is more compact than the NPN classification. For comparison purpose, Table 3.3 [18], [16] shows the classification statistics for all the classification methods we have discussed so far.

Table 3.3: Function classification statistics.

Number of input variables	2	3	4	5	6
Total number of functions	16	256	65,536	$\approx 4.3 \times 10^9$	$\approx 1.8 \times 10^{19}$
Number of non-degenerate functions	10	218	64,594	$\approx 4.3 \times 10^9$	$\approx 1.8 \times 10^{19}$
Number of degenerate functions	6	38	942	325,262	2.58×10^{10}
PN Classification	6	22	402	1,228,158	$\approx 4 \times 10^{14}$
NPN Classification	4	14	222	616,126	$\approx 2 \times 10^{14}$
Spectral Classification	2	3	8	48	$-^*$

* For n=6, the number of spectral classes has not been computed due to the computational complexity.

In the spectral domain, instead of the representative functions in the case of the PN and NPN classifications, canonical spectra are used to represent the equivalence classes in the spectral classification. The five invariance operations can be used to transform a spectrum into its canonical form.

3.5 Symmetry Conditions in Boolean Functions

In general, every Boolean function possesses some form of symmetry. Exploiting the symmetry conditions may lead to a more efficient synthesis of the function. In this section, we discuss how spectral information can be used to detect symmetry conditions in Boolean functions. For the purpose of this thesis, we restrict our discussion to *equivalence symmetry* and *non-equivalence symmetry* of degree 2. Interested readers should refer to [18] for more details.

A Boolean function $f(X)$ is symmetric in some non-empty subset of k input variables if any interchanges within this subset leave $f(X)$ unchanged. If the subset of variables consists of two variables x_i and x_j, then $f(X)$ is said to have a symmetry of degree 2 in $\{x_i, x_j\}$. For example, $f(X) = x_1\bar{x}_3 + x_2\bar{x}_3 + x_1x_2x_4$ is symmetric in $\{x_1, x_2\}$ since interchanging the variables x_1 and x_2 in the function leaves it unchanged.

Classical methods for the detection of symmetry conditions involve identifying certain patterns in the Karnaugh map of the function. For example, since the above function is symmetric in $\{x_1, x_2\}$, we can interchange x_1 and x_2 without changing the function, that is, $f(0, 1, x_3, x_4) = f(1, 0, x_3, x_4)$. Figure 3.2 shows the Karnaugh map of the function. The symmetry condition is indicated by the two identical columns as shown.

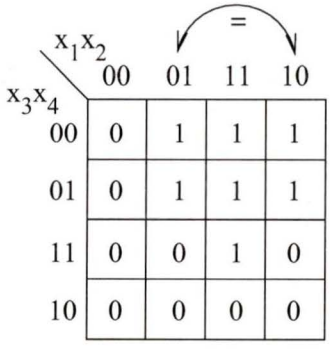


Figure 3.2: Karnaugh map of function $f(X) = x_1\bar{x}_3 + x_2\bar{x}_3 + x_1x_2x_4$, symmetric in $\{x_1, x_2\}$.

A function $f(x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n)$ possesses an equivalence symmetry in $\{x_i, x_j\}$

if $f(x_1,\ldots,0,\ldots,0,\ldots,x_n) = f(x_1,\ldots,1,\ldots,1,\ldots,x_n)$. This means interchanging of x_i and $\bar{x}_j$, or equivalently, $\bar{x}_i$ and x_j, leaves the function unchanged.

A function $f(x_1,\ldots,x_i,\ldots,x_j,\ldots,x_n)$ possesses a non-equivalence symmetry in $\{x_i, x_j\}$ if $f(x_1,\ldots,0,\ldots,1,\ldots,x_n) = f(x_1,\ldots,1,\ldots,0,\ldots,x_n)$. This means interchanging of x_i and x_j leaves the function unchanged.

Both equivalence symmetry and non-equivalence symmetry imply equal rows, columns, or other groupings in the Karnaugh map of the function. However, if the function has large number (e.g., more than five) of input variables, using a Karnaugh map to detect symmetry conditions would be very difficult.

We now consider how spectral information can be used to determine the symmetry conditions.

As shown in section 3.2, the spectrum of a n-variable function $f(x_1,\ldots,x_n)$ is given by $\mathbf{S} = \mathbf{T}^n\mathbf{Z}$, where $\mathbf{Z}$ is the truth-table column vector of $f(X)$, using the $\{1,-1\}$ coding scheme.

Let

$$\mathbf{S}_u = \mathbf{T}^{n-2}\mathbf{Z}_u, \quad 0 \le u \le 3$$

be the spectrum of function $f_u(x_1,\ldots,x_{i-1},x_{i+1},\ldots,x_{j-1},x_{j+1},\ldots,x_n)$, $0 \le u \le 3$, where

$$\begin{aligned}
f_0 &= f(x_1,\ldots,x_{i-1},0,x_{i+1},\ldots,x_{j-1},0,x_{j+1},\ldots,x_n)\\
f_1 &= f(x_1,\ldots,x_{i-1},0,x_{i+1},\ldots,x_{j-1},1,x_{j+1},\ldots,x_n)\\
f_2 &= f(x_1,\ldots,x_{i-1},1,x_{i+1},\ldots,x_{j-1},0,x_{j+1},\ldots,x_n)\\
f_3 &= f(x_1,\ldots,x_{i-1},1,x_{i+1},\ldots,x_{j-1},1,x_{j+1},\ldots,x_n)
\end{aligned}$$

It is shown in [18] that

$$[\mathbf{S}_0\mathbf{S}_1\mathbf{S}_2\mathbf{S}_3] = \frac{1}{4}[\mathbf{S}^0\mathbf{S}^1\mathbf{S}^2\mathbf{S}^3]\mathbf{T}^2$$

where

$$\mathbf{S} = \begin{bmatrix} \mathbf{S}^0 \\ \mathbf{S}^1 \\ \mathbf{S}^2 \\ \mathbf{S}^3 \end{bmatrix}.$$

We have

$$\begin{aligned} 4\mathbf{S}_0 &= \mathbf{S}^0 + \mathbf{S}^1 + \mathbf{S}^2 + \mathbf{S}^3 \\ 4\mathbf{S}_1 &= \mathbf{S}^0 - \mathbf{S}^1 + \mathbf{S}^2 - \mathbf{S}^3 \\ 4\mathbf{S}_2 &= \mathbf{S}^0 + \mathbf{S}^1 - \mathbf{S}^2 - \mathbf{S}^3 \\ 4\mathbf{S}_3 &= \mathbf{S}^0 - \mathbf{S}^1 - \mathbf{S}^2 + \mathbf{S}^3 \end{aligned} \tag{3.13}$$

$\mathbf{S}^u$, $0 \leq u \leq 3$ has the following meaning:

$\mathbf{S}^0$ includes all coefficients in $\mathbf{S}$ that involve neither of x_i or x_j;

$\mathbf{S}^1$ includes all coefficients in $\mathbf{S}$ that involve x_i but not x_j;

$\mathbf{S}^2$ includes all coefficients in $\mathbf{S}$ that involve x_j but not x_i;

$\mathbf{S}^3$ includes all coefficients in $\mathbf{S}$ that involve both of x_i and x_j;

From (3.13), it is easy to write down conditions to determine the equivalence and non-equivalence symmetries in $\{x_i, x_j\}$, as shown in Table 3.4.

Table 3.4: Spectral conditions for equivalence and non-equivalence symmetries in $\{x_i, x_j\}$.

Symmetry	Conditions	Tests
equivalence	$\mathbf{S}_0 = \mathbf{S}_3$	$\mathbf{S}^1 + \mathbf{S}^2 = \mathbf{0}$
non-equivalence	$\mathbf{S}_1 = \mathbf{S}_2$	$\mathbf{S}^1 - \mathbf{S}^2 = \mathbf{0}$

For example, function $f(X) = \bar{x}_1\bar{x}_2(x_3 + x_4) + x_3x_4(\bar{x}_1 + \bar{x}_2) + \bar{x}_3\bar{x}_4(\bar{x}_1x_2 + x_1\bar{x}_2)$ $+x_1x_2x_3\bar{x}_4$ has the following set of spectral coefficients:

s_0	s_1	s_2	s_{12}	s_3	s_{13}	s_{23}	s_{123}	s_4	s_{14}	s_{24}	s_{124}	s_{34}	s_{134}	s_{234}	s_{1234}
0	−4	−4	0	4	0	0	4	0	4	4	0	−4	0	0	12

For $\{x_1, x_2\}$, we have

$$\mathbf{S}^0 = \begin{bmatrix} s_0 \\ s_3 \\ s_4 \\ s_{34} \end{bmatrix} = \begin{bmatrix} 0 \\ 4 \\ 0 \\ -4 \end{bmatrix}, \quad \mathbf{S}^1 = \begin{bmatrix} s_1 \\ s_{13} \\ s_{14} \\ s_{134} \end{bmatrix} = \begin{bmatrix} -4 \\ 0 \\ 4 \\ 0 \end{bmatrix},$$

$$\mathbf{S}^2 = \begin{bmatrix} s_2 \\ s_{23} \\ s_{24} \\ s_{234} \end{bmatrix} = \begin{bmatrix} -4 \\ 0 \\ 4 \\ 0 \end{bmatrix}, \quad \mathbf{S}^3 = \begin{bmatrix} s_{12} \\ s_{123} \\ s_{124} \\ s_{1234} \end{bmatrix} = \begin{bmatrix} 0 \\ 4 \\ 0 \\ 12 \end{bmatrix}.$$

We can see that $\mathbf{S}^1 - \mathbf{S}^2 = \mathbf{0}$, so $f(X)$ possesses non-equivalence symmetry in $\{x_1, x_2\}$.

3.6 Summary

This chapter presented an overview of spectral techniques. Basically, spectral techniques involve a transform of the Boolean domain representation of a function to an alternative representation: the spectral representation. It was shown the spectral coefficients resulted from the transform can each provide some global information about the function under consideration, while the Boolean domain representation is local in nature. This is the main advantage of spectral techniques. This chapter also presented two applications of spectral techniques, namely, Boolean function classification and detection of symmetry conditions in Boolean functions. In both cases, the problems can be solved in the Boolean domain, but spectral techniques provide much more efficient and elegant solutions. In the next chapter, we will see how spectral techniques can be applied to the Boolean matching problem.

Chapter 4

Boolean Matching

Technology mapping was reviewed in chapter 2. As was shown, most algorithmic approaches to technology mapping divide the problem into subproblems. The optimized Boolean network is first decomposed into a subject graph which is an interconnection of some selected base functions. The cells of the technology library are represented as pattern graphs which consist of the same set of base functions. Then the subject graph is covered by the interconnection of a set of pattern graphs to produce the optimal final circuit.

The covering process implies an operation of graph matching, *i.e.*, identifying the equivalence of a pattern graph and a portion of the subject graph. Equivalently, the graph matching problem can be described as: given a Boolean function $F(x_1, x_2, \ldots, x_n)$, determine if it can be implemented by a library-function $G(y_1, y_2, \ldots, y_m)$, $m \geq n$. This is known as the *Boolean matching* problem.

In this chapter, we discuss technology mapping methods that are based on Boolean matching. Unlike the approaches described in chapter 2, the Boolean matching approach depends solely on the functionality of the subcircuit and the library cell and not on their structure. The advantage of the Boolean matching approach is that logic equivalence between two functions can be detected regardless of how they are represented. As is well known, a Boolean function can be represented in different forms, such as the factored form, the sum-of-products form, the ROBDD, *etc.*. When two logically equivalent functions are represented in non-canonical forms, it is very

hard, for a purely structural matching algorithm to detect the logic equivalence. For example, functions $f_1 = ab + ac + bc$ and $f_2 = a(b\bar{c} + \bar{b}c) + bc$ are logically equivalent, but structurally entirely different. A matching algorithm that can recognize logic equivalence independent of the representations can yield better results than structural methods.

4.1 Technology Mapping by Boolean Matching

4.1.1 The *Ceres* Approach

Similar to DAGON, *Ceres* [10] divides the technology mapping process into three major tasks: partitioning, decomposition, and covering. The original Boolean network is first partitioned into a set of single-output subnetworks. Each subnetwork is then decomposed into a network of two-input functions to increase the network granularity. This fine-grain structure enables the function of nodes in the original network to be split across more than one library cell. After partitioning and decomposition, a dynamic programming approach is used to construct an optimal circuit of library cells that covers each single-output subnetwork. The covering algorithm uses the notion of *cluster* and *cluster function*. A cluster is a connected subgraph which has only one node with zero out-degree. The associated cluster function is the Boolean function obtained by collapsing the Boolean expressions associated with the nodes into a single Boolean function.

As an example, consider the Boolean subnetwork of Figure 4.1 that, after an AND/OR decomposition, is described by

$$
\begin{aligned}
f &= j + t \\
j &= xy \\
x &= e + z \\
y &= a + c \\
z &= \bar{c} + d.
\end{aligned}
$$

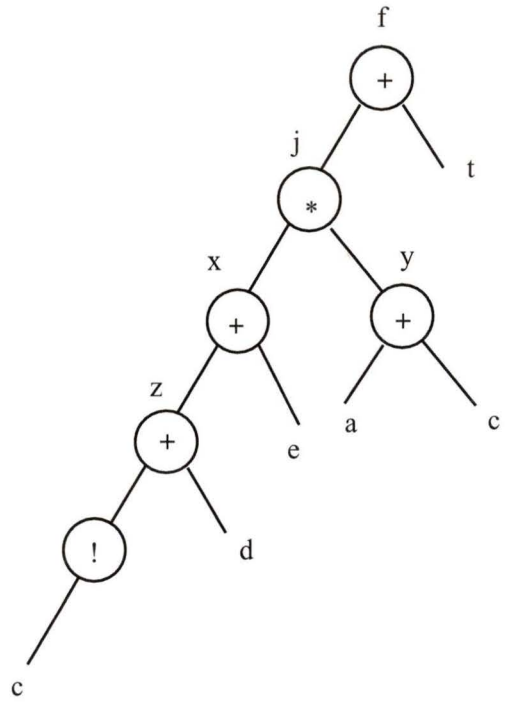


Figure 4.1: Clusters and cluster functions.

There are six possible cluster functions associated with variable j, corresponding to the increased depth in the collapsing of the function associated with j:

$$k_{j,1} = xy$$
$$k_{j,2} = x(a + c)$$
$$k_{j,3} = (e + z)y$$
$$k_{j,4} = (e + z)(a + c)$$
$$k_{j,5} = (e + \bar{c} + d)y$$
$$k_{j,6} = (e + \bar{c} + d)(a + c)$$

The covering algorithm selects clusters repeatedly and attempts to match them to a library cell. A cover is a set of clusters matched to library cells that cover the sub-network. When matches exist for multiple clusters, then for any decomposition with tree-structure, the choice of the match of minimal cost guarantees the minimality of the total cost of the matched sub-network.

The covering algorithm differs from DAGON in the matching step. Matching is formulated as checking the equivalence between a cluster function and each of the

functions representing library cells. Permutation and phase-assignment of the input variables of the cluster function are also taken into consideration. In addition, the *don't care* set of the cluster function is used to facilitate the matching.

The Matching Algorithm

The matching problem is formalized as follows. Given a cluster function $F(x_1, \ldots, x_n)$, the phase of its input variable x_i is denoted by $\phi_i \in \{0, 1\}$, such that $x_i^{\phi_i} = x_i$ for $\phi_i = 1$, $x_i^{\phi_i} = \bar{x}_i$ for $\phi_i = 0$. The *don't care* set of the cluster function is denoted by $DC(x_1, \ldots, x_n)$. And the technology library is denoted by $L : \{G_1, \ldots, G_n\}$. Each library cell is a multiple-input, single-output function. The matching problem is defined as:

> Given a cluster function $F(x_1, \ldots, x_n)$, its *don't care* set $DC(x_1, \ldots, x_n)$, and a library cell $G(y_1, \ldots, y_n)$, find an ordering $\{i, \ldots, j\}$ and a phase assignment $\{\phi_1, \ldots, \phi_n\}$, of the input variables of F, such that either of the following two equations is true:
>
> $$F(x_i^{\phi_i}, \ldots, x_j^{\phi_j}) = G(y_1, \ldots, y_n),$$
> $$\overline{F}(x_i^{\phi_i}, \ldots, x_j^{\phi_j}) = G(y_1, \ldots, y_n),$$
>
> for each value of $(y_1, \ldots, y_n)$ and each value of $(x_i^{\phi_i}, \ldots, x_j^{\phi_j}) \notin DC$.

Note that the cluster function and a library function are considered matchable if they are equivalent with respect to input permutation, input negation, and output negation of the cluster function for all care conditions.

The matching algorithm is based on Shannon decomposition. The two functions to be matched are recursively cofactored to generate two decomposition trees. The two functions are matchable if they have the same logic value for all the leaves of the recursion that are not in the *don't care* set. The process is repeated for all possible permutations and phase assignments of the input variables of the cluster function until a match is found, if one exists.

Improvement of the Matching Algorithm

The number of all possible input permutations and phase-assignments for a function with n inputs is $n!\cdot 2^n$. Therefore, the matching algorithm needs up to $n!\cdot 2^n$ Shannon decompositions for each match. Obviously, the computational cost is too high for large n.

To increase the efficiency of the Boolean matching process, the matching algorithm is improved by employing the following rules:

- Any input permutation must associate each unate (binate) variable [1] in the cluster function to a unate (binate) variable in the library function.
- Groups of symmetric variables in the cluster function must also be symmetric in the library function.

The first rule implies that if the cluster function has m binate variables, then only $m!\cdot(n-m)!$, instead of $n!$, permutations of the input variables need to be considered.

The second rule implies that symmetry classes can be used to reduce the search space. A symmetry class of a function is a set of variables that are interchangeable without changing the functionality of the function. Symmetry classes can be used at two different stages to simplify the matching algorithm. First, they can be used as a filter to quickly find a set of candidate library cells for the matching, since a necessary condition for two functions to be matchable is that they have exactly the same symmetry classes. If the symmetry classes for each library cell are identified and stored along with the cells before the technology mapping process starts, then, for a given cluster function, only those cells that have the same symmetry classes as the cluster function need to be checked for matching.

Secondly, symmetry classes can be used during the Boolean matching process. After a set of candidate library cells has been identified, the symmetry sets of the cluster function and those of a candidate cell are compared. Note that matching can

[1] A *unate* variable x of function F is a variable such that either x or $\bar{x}$ appears in the expression of F, but not both. If both x and $\bar{x}$ appear in the expression of F, then x is a *binate* variable.

only occur between two symmetry sets of the same size. Also note that, since all the variables in a given symmetry set are equivalent, the ordering of the variables within the set is irrelevant. Therefore, permutations of variables need only be performed over symmetry sets of the same size. This reduces the number of permutations required to detect a match to $\prod_{i=1}^{p}(S_i!)$, where S_i is the number of sets of cardinality i, and p is the size of the largest symmetry set.

Use of Don't Care Sets

When don't care conditions are considered, the cluster function cannot be uniquely defined by its symmetry sets. Therefore the techniques based on symmetry sets presented above no longer apply.

Ceres deals with this problem by introducing the *matching compatibility* graph which can be used to exploit *don't care* conditions. For a given number of input variables n, each node of the matching compatibility graph corresponds to a set of n-variable functions. All the functions in the same set are equivalent with respect to input permutation, input negation, and output negation. There is an edge between two nodes if, and only if, the functions represented by the two nodes differ in one minterm. Figure 4.2 shows the matching compatibility graph for $n = 3$.

After the graph is constructed, each node is annotated with the library cells that match the corresponding function. Then, for a given cluster function F, the corresponding node v_F can be determined. It is shown that, there is a match between F and a library cell G if there is a path from v_F to v_G. The edges of the path correspond to the minterm in the *don't care* set of F.

4.1.2 The *proserpine* Approach

The general framework of *proserpine* [11] is based on *Ceres*. It uses the same partitioning, decomposition, and covering techniques as those used in *Ceres*, with a different matching algorithm. Moreover, it does not require a technology library to be generated, and is thus more suitable for technology mapping targeted to FPGA's.

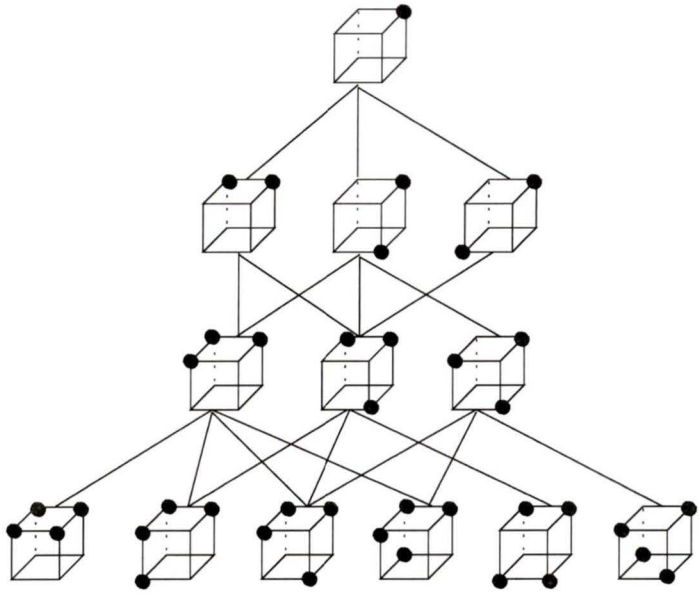


Figure 4.2: Matching compatibility graph for 3-variable Boolean space.

As in *Ceres*, the original Boolean network is first partitioned and decomposed into a set of subnetworks of two-input functions. Then, a dynamic programming approach is used to construct an optimal cover for each subnetwork, which is composed of the logic blocks of the target FPGA.

The matching algorithm of *proserpine* checks whether a cluster function can be implemented by a logic block by a run-time personalization of the logic block. It is shown that personalization of a logic block can be modeled by a combination of *stuck-at-0* faults, *stuck-at-1* faults, and *bridging* faults. The matching algorithm also takes into account input ordering to perform the matching. The matching process is divided into two steps. In the first step, the algorithm attempts to find a match using only the stuck-at faults. If it fails, both stuck-at faults and bridging faults are then considered in the second step. ROBDDs are used to represent the logic block and the cluster function, and subgraph isomorphism is used to detect a match. Figure 4.3 illustrates the multiplexer-based logic block $a(bc + \bar{b}d) + \bar{a}(ef + \bar{e}g)$ and the ROBDD representing this logic block with the input variable ordering (a, b, c, d, e, f, g).

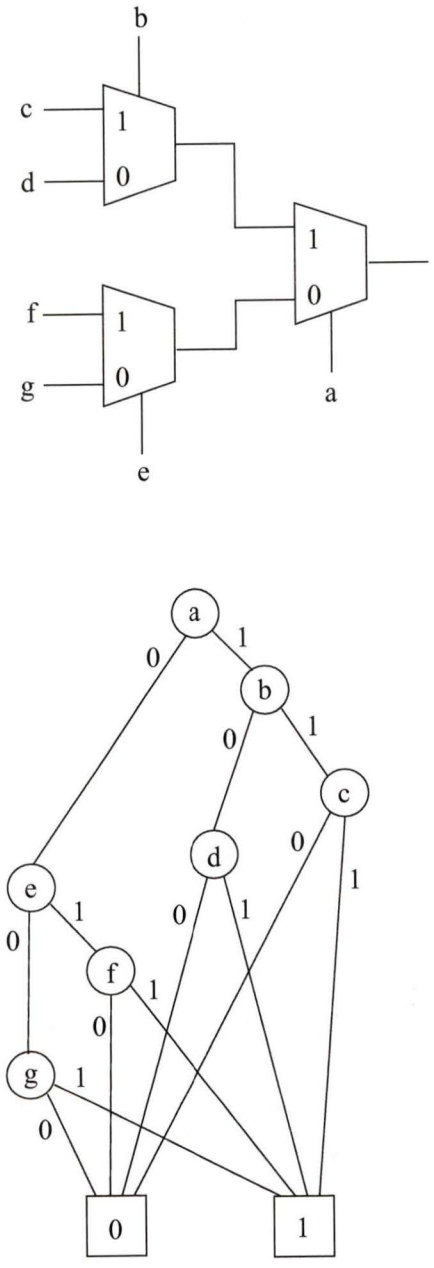


Figure 4.3: A multiplexer-based logic block and its ROBDD.

Matching with Stuck-At Faults

In the first step of matching, the algorithm considers a simplified matching problem in which only stuck-at-0 and stuck-at-1 faults are used to personalize the logic block. The ROBDD representing the cluster function is checked against subgraphs in the ROBDD representing the logic block. If a subgraph is isomorphic to the cluster function ROBDD, then it represents the same function as the cluster function. In this case, the logic block can be personalized to implement the cluster function by an appropriate set of stuck-at faults. The required set of stuck-at faults is determined by the path leading from the root of the logic block ROBDD to this subgraph. The input assignment is specified by the correspondence between the nodes in the cluster function ROBDD and the nodes in the subgraph. Consider for example the logic block $a(bc+\bar{b}d)+\bar{a}(ef+\bar{e}g)$ and the cluster function $xy+\bar{x}z$. Figure 4.4 shows their ROBDD's with the input ordering (a,b,c,d,e,f,g) and (x,y,z), respectively. The cluster function ROBDD is isomorphic to the subgraph in the logic block ROBDD with b as the root. Therefore, the logic block can be personalized to implement the cluster function by the stuck-at fault $a=1$ and the input assignment $b=x$, $c=y$, and $d=z$.

However, since the structure of a ROBDD depends on the input ordering used to construct the ROBDD, the existence of a subgraph in the logic block ROBDD which is isomorphic to the cluster function ROBDD also depends on the input ordering. In Figure 4.4 the logic block ROBDD is constructed using the input ordering (a,b,c,d,e,f,g) and there exists a subgraph that is isomorphic to the cluster function ROBDD. However, if the logic block ROBDD is constructed using a different input ordering (a,d,b,c,e,f,g) as shown in Figure 4.5, there is no subgraph that is isomorphic to the cluster function ROBDD.

Therefore, the *proserpine* matching algorithm constructs the logic block ROBDD's corresponding to all possible input orderings, and searches each ROBDD for a subgraph that is isomorphic to the cluster function ROBDD. Actually, it is not necessary to search all the ROBDD's, but only those ROBDD's that are different from one an-

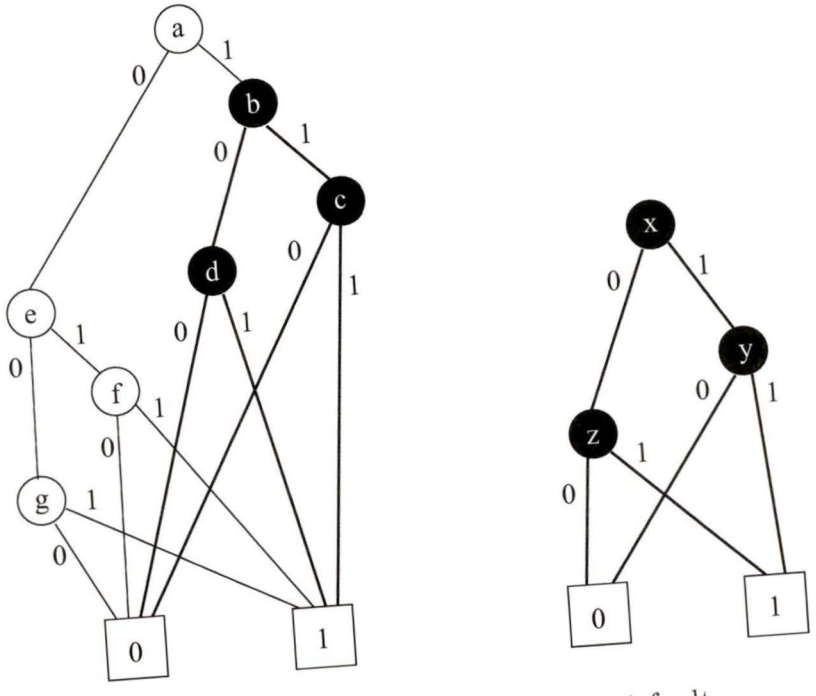


Figure 4.4: Matching with a stuck-at-1 fault.

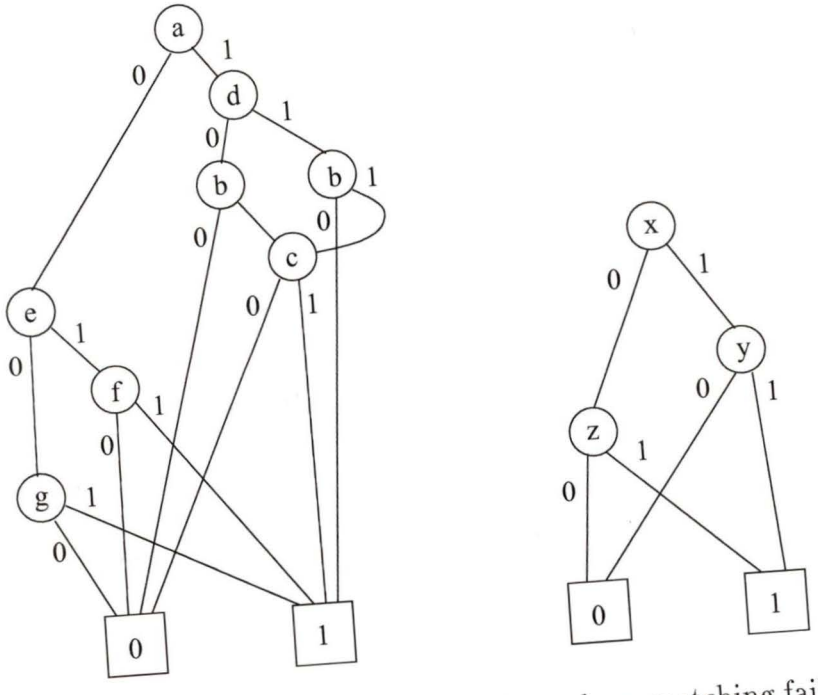


Figure 4.5: Logic block ROBDD ordering where matching fails.

other. For a logic block of n inputs, there are $n!$ input permutations, but far fewer unique ROBDD's.

To further reduce the search space, *proserpine* employs a new structure called *global BDD* (GBDD). This is based on the fact that many of the subgraphs within the logic block ROBDD's corresponding to different input orderings are isomorphic to one another, and only one of these subgraphs needs to be considered in the search for isomorphism between a subgraph and the cluster function ROBDD. A GBDD is constructed by first assembling the logic block ROBDD's for all possible input permutations into a single DAG, and then eliminating all redundant subgraphs such that there are no two subgraphs isomorphic to each other.

Matching with Bridging Faults

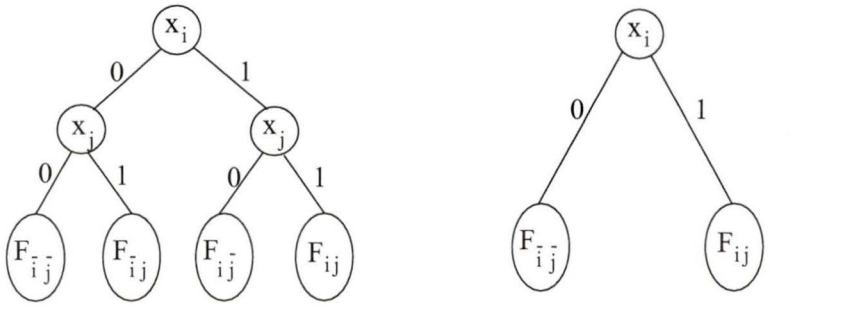
If the first step of the matching algorithm fails to find a match for the cluster function by using stuck-at faults to personalize the logic block, then the second step considers using bridging faults in order to find a match. The idea is that, by bridging some of the logic block inputs together, the structure of the logic block ROBDD would be modified, which leads to the possibility of finding a match that can not be detected if only stuck-at faults are used. For example, given a function $F(x_1, \ldots, x_n)$, its Shannon decomposition with respect to variables x_i, x_j is given by

$$F(x_1, \ldots, x_n) = x_i(x_j F_{ij} + \bar{x}_j F_{i\bar{j}}) + \bar{x}_i(x_j F_{\bar{i}j} + \bar{x}_j F_{\bar{i}\bar{j}}), \tag{4.1}$$

where the functions F_{ij}, $F_{i\bar{j}}$, $F_{\bar{i}j}$, and $F_{\bar{i}\bar{j}}$ are the cofactors of F with respect to $x_i x_j$, $x_i \bar{x}_j$, $\bar{x}_i x_j$, and $\bar{x}_i \bar{x}_j$ respectively. If there is a bridging fault between the variables x_i and x_j, then equation 4.1 becomes

$$F_B(x_1, \ldots, x_i, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n) = x_i F_{ij} + \bar{x}_i F_{\bar{i}\bar{j}} \tag{4.2}$$

Figure 4.6(a) shows the ROBDD for F with an input ordering where x_i and x_j are adjacent. Figure 4.6(b) shows the ROBDD for F_B. As we can see, bridging the two variables x_i and x_j, corresponds to merging the nodes representing x_j into the nodes representing x_i, and removing the branches leading to $F_{i\bar{j}}$ and $F_{\bar{i}j}$.



a) Original BDD b) BDD with bridging fault $x_i=x_j$

Figure 4.6: Modifying a ROBDD with a bridging fault.

This observation can be extended to bridging any number of adjacent input variables. An arbitrary bridging fault can be described by finding an input ordering where the bridged variables are adjacent and then modifying the corresponding ROBDD. Actually, since the GBDD contains ROBDD's corresponding to all input permutations, by using GBDD, all possible subsets of input variables are considered whose bridging could be used to personalize the logic block to implement the given cluster function. Moreover, since all possible bridgings are considered, it is guaranteed that a match, if it exists, can be found by bridging inputs of the logic block in addition to using the stuck-at faults.

Since the bridge-fault matching has much higher run-time cost than the stuck-at matching, the overall algorithm, as described before, first attempts to find a match using only stuck-at faults; if no match is found, it then tries all possible combinations of stuck-at and bridging faults until a match is found.

The *proserpine* approach has the following advantages. First, using only one logic function that describes the uncommitted logic block, the matching algorithm can capture the entire family of functions that can be implemented by a logic block. Thus, the entire technology library is replaced by one function, which makes it appealing in FPGA designs. Secondly, the algorithm is independent of the structure of the logic block, so, it can be extended to any type of logic block that can be represented by a single-output logic function.

4.2 Spectral Method for Boolean Matching

The spectral method for Boolean matching is based on the NPN equivalence classes discussed in section 3.4. Two functions are considered to be matchable if they are equivalent with respect to the operations of input permutation, input negation, and output negation. Unlike most Boolean matching methods which directly match two functions, the spectral method presented here uses the spectral domain to compute a canonical form for the functions to be matched. Two functions are NPN matchable if, and only if, they have the same canonical form. The combination of the permutations and negations required to transform each function to the canonical form can be used to transform one function to the other. When applied to technology mapping, the spectral method can be used to preprocess the technology library, so that the only significant computation needed to perform the matching is to transform the target function into the canonical form. Furthermore, the spectral method may also be used to reduce the size of the library without any loss of functionality, making it faster to find a match.

4.2.1 The Canonical Form for Boolean Matching

As discussed in chapter 2, any given Boolean function is uniquely related to a spectrum. Thus, the notion of functional NPN-equivalence classes can be extended directly to spectral NPN-equivalence classes. It was also mentioned that the spectral coefficients of the functions belonging to the same NPN-equivalence class have the same set of magnitudes. And, for each equivalence class, a canonical spectrum can be selected as the representative of the class.

One canonical form for spectral classification of Boolean functions is the so-called *positive canonical form* [18]. In this canonical form, the spectral coefficients are in the order s_0; $s_1, s_2, \ldots, s_n$; $s_{12,\ldots}$; $s_{12\ldots n}$. Note that this order is different from the Hadamard order. All the zero- and first-order coefficients of the canonical spectrum are positive and arranged in decreasing magnitude order. Thus s_0 is the highest

magnitude coefficient. Table 4.1 shows the positive canonical spectra for functions of $n \leq 4$.

Table 4.1: The positive canonical spectra for all functions of $n \leq 4$ under the full spectral coefficient invariance operations.

	Spectral coefficients															
n	s_0	s_1	s_2	s_3	s_4	s_{12}	s_{13}	s_{14}	s_{23}	s_{24}	s_{34}	s_{123}	s_{124}	s_{134}	s_{234}	s_{1234}
$\leq$ 2:(1)	4	0	0			0										
(2)	2	2	2			-2										
$\leq$ 3:(1)	8	0	0	0		0	0		0			0				
(2)	6	2	2	2		-2	-2		-2			2				
(3)	4	4	4	0		-4	0		0			0				
$\leq$ 4:(1)	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
(2)	14	2	2	2	2	-2	-2	-2	-2	-2	-2	2	2	2	2	-2
(3)	12	4	4	4	0	-4	-4	0	-4	0	0	4	0	0	0	0
(4)	10	6	6	2	2	-6	-2	-2	-2	-2	2	2	2	-2	-2	2
(5)	8	8	8	0	0	-8	0	0	0	0	0	0	0	0	0	0
(6)	8	8	4	4	4	-4	-4	-4	0	0	0	0	0	0	-4	4
(7)	6	6	6	6	6	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	6
(8)	4	4	4	4	4	4	4	-4	-4	4	-4	-4	-4	4	4	-4

However, transformation to the positive canonical form involves all the five invariance operations discussed in section 3.4.

We now present a new canonical form, the *NPN canonical form*, for Boolean functions, which is based on the NPN-equivalence classes and involves only the NPN operations. The canonical form and the transformation method presented in the next section were introduced and developed in [16].

In section 3.5, we discussed equivalence and non-equivalence symmetries in Boolean functions. As was shown, a function $f(X)$ has a non-equivalence symmetry in $\{x_i, x_j\}$ if, and only if, $s_{i\alpha} = s_{j\alpha}$ for all $\alpha \subseteq \{1, 2, \ldots, n\}$, $i, j \notin \alpha$; $f(X)$ has an equivalence-symmetry if, and only if, $s_{i\alpha} = -s_{j\alpha}$ for all $\alpha \subseteq \{1, 2, \ldots, n\}$, $i, j \notin \alpha$. We employ

the notion of *negative symmetry* as a synonym of equivalence symmetry, and non-equivalence symmetry is simply called symmetry.

Let

$$v_p = \sum_{\forall \alpha \subseteq \beta} s_{p\alpha}^2, \quad p \in \{1, 2, \ldots, n\}, \quad \beta = \{1, 2, \ldots, p-1, p+1, \ldots, n\} \tag{4.3}$$

v_p is the sum of the squares of all the spectral coefficients with p as a subscript.

It can be easily seen that $v_i = v_j$ if $f(X)$ is symmetric or negative-symmetric in $\{x_i, x_j\}$, but the converse does not necessarily hold. A set of input variables which are in pair symmetric or negative-symmetric is called a *symmetry set* for simplicity.

NPN Canonical Form We define the NPN canonical spectrum in the spectral NPN equivalence class **E** as the spectrum **S** which satisfies the following conditions:

1. $s_0 \geq 0$;
2. $s_i \geq 0, \quad 1 \leq i \leq n$;
3. the input variables are ordered according to the following rules:
   (a) $v_i \geq v_{i+1}, \quad 1 \leq i \leq n$; so $\{x_1, x_2, \ldots, x_n\}$ may be partitioned into a number of sets $P_1, P_2, \ldots$ in terms of the v value of the input variables. All the variables in the same partition have the same v value.
   (b) Each P_i is in turn partitioned into an appropriate number of symmetry sets $Q_1^i, Q_2^i, \ldots$ which are ordered by decreasing cardinality, *i.e.* $|Q_j^i| \geq |Q_{j+1}^i|$ for all j.
   (c) For every $|Q_j^i| = |Q_{j+1}^i|$, $|s_\alpha| \geq |s_{\hat{\alpha}}|$ for the first α such that $|s_\alpha| \neq |s_{\hat{\alpha}}|$, where $\hat{\alpha}$ is the rearrangement of α obtained by the pair-wise interchange of the variables of Q_j^i and Q_{j+1}^i.
4. Let **C** (**C** $\subseteq$ **E**) denote the set (including **S**) of all spectra satisfying 1–3, **S** is such that for the first β (in Hadamard coefficient order) where $s_\beta < 0$, there is no $\hat{\mathbf{S}} \in \mathbf{C}$ such that $\hat{s}_\gamma \geq 0$, for all $\hat{s}_\gamma$ up to and including $\hat{s}_\beta$ (in Hadamard coefficient order).

5. Let $\mathbf{D}$ ($\mathbf{D} \subseteq \mathbf{E}$) denote the set (including $\mathbf{S}$) of all spectra that satisfy 4, $\mathbf{S}$ is such that for all other $\hat{\mathbf{S}} \in \mathbf{D}$, $s_\delta > 0$ for the first $\hat{s}_\delta$ after $\hat{s}_\beta$ (in Hadamard order) such that $s_\delta \neq \hat{s}_\delta$.

The uniqueness of the NPN canonical spectrum is readily shown. First of all, the zero order coefficients of any two functions belonging to the same NPN-equivalence class are of the same absolute value. Thus, a spectrum either satisfies criterion 1 or can be mapped to a spectrum in the same NPN-equivalence class that does by negation of the function output. Similarly, a spectrum either satisfies criteria 2 and 3 or can be mapped to one that does by appropriate input negations and permutations. Let $\mathbf{C}$ be the set of all spectra satisfying criteria 1-3. Clearly $\mathbf{C}$ is not empty, but the cardinality of $\mathbf{C}$ may be greater than one because not all the positions of the inputs are fixed.

Criterion 3 divides the symmetry sets into two categories: those whose positions are fixed and those which are interchangeable with a neighbouring set. We term the later *interchangeable sets*. Note that while the corresponding coefficients of two interchangeable sets are of equal magnitude, the two sets can not have strictly equal coefficients or strictly negated coefficients since that would indicate a symmetry of negative symmetry and the two sets would have been combined. Therefore, criterion 3 determines a unique ordering of all the symmetry sets with the exception of the interchangeable symmetry sets. So the ordering prescribed by criterion 3 is unique up to absolute value of the coefficients. Suppose $\mathbf{S}^1$, $\mathbf{S}^2$ both satisfy criteria 1-3, we have $|s^1_\alpha| = |s^2_\alpha|$ for all α.

If $\mathbf{S}^1$ and $\mathbf{S}^2$ both satisfy criterion 4, clearly they must do so at the same coefficient position β. However, they can not then both satisfy criterion 5 since there must be a δ later than β such that s^1_δ and s^2_δ have different signs, otherwise they would be the same spectrum. Therefore, there is exactly one spectrum in $\mathbf{E}$ which satisfies all five criteria.

4.2.2 Transformation of Spectrum to Canonical Form

We have presented a new canonical form for the spectral NPN-equivalence classes and have shown the uniqueness of such a canonical form. We now consider the problem of how to transform a given spectrum to its canonical form, which is more of interest to the Boolean matching problem.

The transform method presented in [16] consists of two phases. In the first phase, the given spectrum $\mathbf{S}$ is transformed into $\hat{\mathbf{S}}$ which satisfies criteria 1-3 of the canonical form. First of all, function negation is applied if $s_0 < 0$, and input negation is applied for each input variable x_i if $s_i < 0$. All the negations are applied in a single pass through the coefficients. For each input x_i, v_i is also computed during the same pass. The v_i values are then used to identify symmetry or negative symmetry conditions between input variable pairs. Note that $v_i = v_j$ is a necessary condition for variables x_i and x_j to be symmetric or negative symmetric.

Next the input variables are permuted to satisfy criterion 3 of the canonical form. Criteria 3(a) and 3(b) only require appropriate sorting. For criterion 3(c), since reordering a pair of variables can affect the coefficient magnitude relationships for other variable pairs, a simple interchange sort is repeatedly applied until no changes are required.

In the second phase, a set of candidate spectra for the canonical form is established by applying a particular set of negation to $\hat{\mathbf{S}}$. $\hat{\mathbf{S}}$ itself is also included in the candidate set. If $\hat{s_0} = 0$, the function may be negated. If $\hat{s_i} = 0$, the input variable x_i may be negated. Therefore, in the worst case, the number of candidates is 2^{n+1}. However, it is shown that if the zero order coefficient is an odd multiple of 2, all the first order coefficients are non-zero, and thus no negation searching is necessary. This can be used to avoid unnecessary computations.

After it is constructed, the set of candidate spectra is searched for the canonical form. For each candidate spectrum, its interchangeable symmetry sets are ordered such that within each P_i, each pair of adjacent interchangeable sets are such that for x_k the first variable in Q_j^i and x_t the first variable in Q_{j+1}^i, $s_{k\alpha}$ is positive and

$s_{t\alpha}$ is negative for the first $s_{k\alpha} \neq s_{t\alpha}$. Such an α must exist or the two variables would be symmetric. Once again, since reordering of a pair can affect other pairs, the interchange process is repeated until no changes are required.

During the searching process, criteria 4 and 5 of the canonical form are used to keep track of the best candidate considered so far. At the end of the search, the best candidate found is the canonical form.

To avoid unnecessary computations, a check is performed while a candidate is being formed. If the first negative valued coefficient of the candidate is at a position earlier than the first negative valued coefficient of the best candidate, and it does not involve any variables in interchangeable symmetry sets, the candidate can be rejected without completing its construction or reordering its interchangeable sets. Experimental results in [16] show this rejection process is reasonably effective.

We now consider the following two examples to show the above procedure.

Example 1: Consider function $f(x_1, x_2, x_3, x_4) = x_1x_2\bar{x}_3 + \bar{x}_1\bar{x}_2\bar{x}_4 + \bar{x}_1\bar{x}_2x_3 + \bar{x}_2x_3\bar{x}_4 + \bar{x}_1x_3\bar{x}_4$. The spectral coefficients of $f(X)$ are as follows.

s_0	s_1	s_2	s_{12}	s_3	s_{13}	s_{23}	s_{123}	s_4	s_{14}	s_{24}	s_{124}	s_{34}	s_{134}	s_{234}	s_{1234}
2	2	−2	−2	2	2	6	−10	−6	−6	−2	−2	2	2	−2	−2

And the v_i values are: $v_1 = 160$, $v_2 = 160$, $v_3 = 160$, $v_4 = 96$.

Since $s_0 > 0$, and both s_2 and s_4 are negative, the first step in the transform is to negate x_2 and x_4. We have the normalized spectrum satisfying criteria 1 and 2 of the canonical form as follows.

s_0	s_1	s_2	s_{12}	s_3	s_{13}	s_{23}	s_{123}	s_4	s_{14}	s_{24}	s_{124}	s_{34}	s_{134}	s_{234}	s_{1234}
2	2	2	2	2	2	−6	10	6	6	−2	−2	−2	−2	−2	−2

Next we sort the variables according to criterion 3 of the canonical form and obtain the following spectrum.

s_0	s_1	s_2	s_3	s_4	s_{12}	s_{13}	s_{14}	s_{23}	s_{24}	s_{34}	s_{123}	s_{124}	s_{134}	s_{234}	s_{1234}
2	2	2	2	2	−6	2	10	6	−2	6	−2	−2	−2	−2	−2

Since the zero-order and all the first-order coefficients are non-zero, the above spectrum is the only candidate in the second phase of the transform. Therefore it is the final canonical form.

Example 2: Consider function $f(x_1, x_2, x_3, x_4) = x_3\bar{x}_2\bar{x}_1 + x_4x_3\bar{x}_2 + x_4x_3x_1 + \bar{x}_3\bar{x}_2x_1 + \bar{x}_4x_2\bar{x}_1$. We have the spectral coefficients as follows.

s_0	s_1	s_2	s_{12}	s_3	s_{13}	s_{23}	s_{123}	s_4	s_{14}	s_{24}	s_{124}	s_{34}	s_{134}	s_{234}	s_{1234}
0	0	−4	4	4	4	0	8	0	−8	4	4	−4	4	0	0

The v_i values are: $v_1 = 192$, $v_2 = 128$, $v_3 = 128$, $v_4 = 128$.

The normalized spectrum satisfying criteria 1 and 2 is:

s_0	s_1	s_2	s_{12}	s_3	s_{13}	s_{23}	s_{123}	s_4	s_{14}	s_{24}	s_{124}	s_{34}	s_{134}	s_{234}	s_{1234}
0	0	4	−4	4	4	0	−8	0	−8	−4	−4	−4	4	0	0

And we have the category labels [16] 0, 1, 1, 0 which indicate that the symmetry sets $\{x_2\}$ and $\{x_3\}$ are interchangeable.

When applying criterion 3(c), the only possible pairwise interchange of variables is $x_2 \leftrightarrow x_3$. And the spectrum satisfying criteria 1–3 is:

s_0	s_1	s_2	s_{12}	s_3	s_{13}	s_{23}	s_{123}	s_4	s_{14}	s_{24}	s_{124}	s_{34}	s_{134}	s_{234}	s_{1234}
0	0	4	4	4	−4	0	−8	0	−8	−4	4	−4	−4	0	0

Note that s_0, s_1 and s_4 are zero-valued. Therefore, in the second phase of the transform, we have a total of $2^3 = 8$ candidates, as shown below.

	s_0	s_1	s_2	s_{12}	s_3	s_{13}	s_{23}	s_{123}	s_4	s_{14}	s_{24}	s_{124}	s_{34}	s_{134}	s_{234}	s_{1234}
(1)	0	0	4	4	4	−4	0	−8	0	−8	−4	4	−4	−4	0	0
(2)	0	0	4	4	4	−4	0	−8	0	8	4	−4	4	4	0	0
(3)	0	0	4	−4	4	4	0	8	0	8	−4	−4	−4	4	0	0
(4)	0	0	4	−4	4	4	0	8	0	−8	4	4	4	−4	0	0
(5)	0	0	−4	−4	−4	4	0	8	0	8	4	−4	4	4	0	0
(6)	0	0	−4	−4	−4	4	0	8	0	−8	−4	4	−4	−4	0	0
(7)	0	0	−4	4	−4	−4	0	−8	0	−8	4	4	4	−4	0	0
(8)	0	0	−4	4	−4	−4	0	−8	0	8	−4	−4	−4	4	0	0

Candidates (5)–(8) can be rejected because they have a negative-valued s_2. After ordering the interchangeable symmetry sets (in this case, the only possible interchangeable sets are $\{x_2\}$ and $\{x_3\}$), we finally obtain the canonical spectrum as follows:

s_0	s_1	s_2	s_{12}	s_3	s_{13}	s_{23}	s_{123}	s_4	s_{14}	s_{24}	s_{124}	s_{34}	s_{134}	s_{234}	s_{1234}
0	0	4	4	4	−4	0	8	0	8	−4	4	−4	−4	0	0

4.3 Summary

In this chapter, we first reviewed some Boolean matching approaches to technology mapping for FPGA's. We pointed out that the advantage of the Boolean matching methods is that logic equivalence can be detected independent of the representation or structure of the functions under consideration. We presented a new canonical form for Boolean functions which is defined in the spectral domain. We also discussed the method for transforming a given Boolean function to its canonical form.

Chapter 5

Technology Mapping Algorithm Based on Spectral Boolean Matching

Technology mapping methods for multiplexer-based FPGA's are described in chapter 2 and section 4.1. As we have seen, most methods divide the technology mapping problem into three subproblems: partitioning, decomposition, and covering. A Boolean network is first partitioned into an interconnection of single-output subnetworks. Each subnetwork is then decomposed into an interconnection of two-input functions such as AND, and OR. Finally, each decomposed subnetwork is covered by an interconnection of library cells to produce the final circuit. Covering is the most important step and it involves a matching operation, *i.e.*, detecting logic equivalence between two logic functions.

A successful covering algorithm was proposed by Keutzer [4] in which both the library cells and the target Boolean network are represented by trees and a dynamic programming approach is used to cover the target network. While the general framework is followed by many others, different matching algorithms have been proposed. As was shown in section 4.1, the Boolean matching approach has the advantage of recognizing logic equivalence regardless of how the functions are represented.

In this chapter, we propose an approach to technology mapping that uses network

partitioning, decomposition, and covering techniques similar to those used in [4], but with a different matching method. The matching method uses spectral techniques and is based on the NPN canonical form presented in section 4.2. Two logic functions are considered to be matchable if, and only if, they can be transformed to the same canonical form.

5.1 Processing of Technology Library

Library-based technology mapping method was reviewed in chapter 2. The major problem with this kind of algorithms is that there is no effective way in the Boolean domain to determine which library cells could possibly be a match for a given target function. Thus, in general, exhaustive search is required, which makes the algorithm inefficient in the case of large libraries.

Our spectral algorithm also requires a technology library. The difference from the conventional library-based approach is that we are concerned about the functionality, not the structure, of the library cells. Using the spectral information, we can preprocess the library such that, in the matching step, a set of candidate library cells can be quickly identified for a given target function.

Specifically, given a technology library, we classify the library functions in terms of their spectral NPN canonical forms (see section 4.2). The technology library is re-organized as a number of NPN-equivalence classes, and a library called *NPN-LIB* is generated such that each function in NPN-LIB represents an NPN-equivalence class in the original library and there are no two functions in NPN-LIB that are NPN-equivalent.

Generally, the NPN-LIB is much smaller in size than the original library. As we will see later, in the matching step, with the NPN-LIB, we can quickly identify a set of candidate library cells for a given target function.

Consider as an example the Actel Act-1 FPGA. We generated a technology library which consists of all the logic functions that an Act-1 logic block can implement. Since the Act-1 logic block can have up to eight inputs, the library functions are put

Table 5.1: Statistics for the Act-1 technology library.

Number of input variables	1	2	3	4	5	6	7	8	Total
Number of functions in P-LIB	2	8	47	210	285	128	21	1	702
Number of functions in NPN-LIB	1	2	8	43	75	37	9	1	176
Number of functions in the largest NPN-equivalent class in P-LIB	2	6	11	19	16	8	4	1	-

into eight different files according to the number of their inputs. The functions are generated by enumeration of different ways of connecting the eight inputs of the Act-1 logic block to input variables, constants 0 or 1, or bridging them together. Each library cell contains the truth-table of the function it implements and the information about how the Act-1 logic block is personalized to realize the function. Moreover, input permutations are also taken into consideration. If one function can be obtained by a certain permutation of the inputs of another function, then the two functions are considered the same function. For example, functions $ab+c$ and $ac+b$ are considered the same. Therefore, each library function actually represents a class of P-equivalent functions. The generated library is thus called *P-LIB*. Figure 5.1 shows the algorithm for generating the Act-1 technology library.

Next, we classify the functions in P-LIB in terms of their spectral NPN canonical forms and generate an NPN-LIB in which each function represents an NPN-equivalence class in P-LIB.

Table 5.1 shows the statistics for P-LIB and NPN-LIB for Act-1. A total of 702 different functions can be implemented by an Act-1 logic block. These 702 functions belong to 176 different NPN-equivalent classes.

```
FOR i=1 TO 8 {
   WHILE there are more ways of connecting
            input variables to Act-1 block {
      connect input variables to Act-1 block ;
      WHILE there are more ways of connecting
               the rest of the Act-1 inputs {
         connect the rest of the Act-1 inputs ;
         generate a function ;
         IF none of the functions corresponding to all the input
            permutations of the function is already in the library {
            add the functions to the library ;
         }
      }
   }
}
```

Figure 5.1: Algorithm for generating Act-1 library.

5.2 Partitioning and Decomposition

As in the case with other technology mappers, the input to our technology mapper is a technology-independently optimized Boolean network. Specifically, we use the SIS system [29] to perform the technology independent optimization. At the end of the optimization, the network is decomposed so that the Boolean function associated with each node in the optimized network has at most two products and each product has at most two literals. There are two reasons for this decomposition. First, it will make the decomposition in the technology mapping phase much easier. Secondly, a network with such fine granularity will make the dynamic programming traversal work better and thus lead to optimal result. However, there is no limitation to the number of parents a node can have.

The first step in the technology mapping phase is to partition the Boolean network into a set of single output subnetworks while preserving the information about how these subnetworks are interconnected. As in DAGON, we choose to partition the network into a forest of trees.

The network is read and all the primary outputs and the intermediate nodes are put into a single linked list. A key is associated with each node which indicates whether the node is a primary input, a primary output, or an intermediate node. If an intermediate node has more than one parent, it is made the root of a new tree. This is done by simply changing the key associated with it. From its parents' point of view, this node acts as a primary input; from its children's point of view, this node acts as a primary output. However, if a node has only two parents and all the children of the node are primary inputs, we choose to duplicate this node and its children rather than to make it the root of a new tree. Experiments showed that this leads to better results in most cases (see chapter 6). The partitioning algorithm is shown in Figure 5.2.

```
WHILE there are more non-leaf nodes {
    pick the next node ;
    IF the node has more than one parent node {
        IF the node has two parents and
            all its children are primary inputs {
            duplicate the node and all its children ;
        }
        ELSE {
            make the node the root of a new tree ;
        }
    }
}
```

Figure 5.2: Partitioning algorithm.

After the network is partitioned into a forest of trees, an elimination operation is performed to remove all the redundant nodes. If two nodes are logically equivalent, then one of the nodes is eliminated.

Next, each of the trees is decomposed such that each node in the tree has at most two children, and the Boolean function associated with each node is either a two-input AND or a two-input OR. The rationale for this decomposition is that the technology library contains two-input AND and OR functions. Therefore, it is guaranteed that

the Boolean network can be realized by an interconnection of the library cells. Since the network was already decomposed during the technology independent phase, the decomposition into two-input AND/OR function is straightforward. If a node has only one child, it must be the logic complement of its child, since all redundant nodes have been removed. In that case, we create another child for the node that is the same as the child it already has. This makes all the nodes uniformly have two children without any functionality change. The decomposition algorithm is shown in Figure 5.3. Each tree is decomposed in a recursive fashion until all the trees are decomposed.

```
decompose(node) {
   IF the node is a leaf node {
      RETURN ;
   }
   ELSE {
      WHILE there are more child nodes to be decomposed {
         decompose(the next child node) ;    /* recursively decompose child nodes */
      }
      IF the node function has two products {
         FOR each product {
            IF there are two literals {
               create a new node for the product ;
            }
         }
      }
      ELSE {   /* node function has only one product */
         IF the product has only one literal {   /* the node is the complement of its child */
            duplicate its child;
         }
      }
   }
}
```

Figure 5.3: Decomposition algorithm.

5.3 Covering

After partitioning and decomposition, the forest of trees are to be covered by an interconnection of the library cells, while optimizing some cost function. Here we consider logic block count as our optimization goal.

For this purpose, we use the notions of cluster and cluster function [11]. As discussed in section 4.1, a cluster is a connected subgraph which has only one node with zero out-degree. It is characterized by its depth and number of inputs. For a certain node v, the smallest cluster consists of node v only; the largest cluster contains all the non-leaf nodes of the tree rooted at v. In the technology mapping procedure, we only consider clusters with input number up to the number of input a target logic block can possibly have. For example, for the Act-1 FPGA, we only consider the clusters with eight or fewer inputs, since an Act-1 logic block can have at most eight inputs.

In order to find the optimal cover for the tree under consideration, the covering algorithm uses a dynamic programming traversal that proceeds from the leaf nodes of the tree to the root node. At each non-leaf node v, the covering algorithm attempts to match each cluster function to a library cell. As mentioned above, since the library contains two-input AND and OR functions, at least one cluster (the smallest one) has a match in the library. If a cluster function $f_{v,i}$ has a match in the library, the cost of implementing the subtree rooted at v is evaluated in terms of the cost of implementing $f_{v,i}$ and all the inputs of $f_{v,i}$. Since the algorithm runs in a bottom-up fashion, the minimal cost of implementing the inputs of $f_{v,i}$ has already been determined. Therefore, the cost of implementing the subtree rooted at v using cluster function $f_{v,i}$ can be easily obtained. The area cost is computed by adding one to the cost of the inputs of $f_{v,i}$.

During the procedure of trying all the clusters, we keep track of the *best* cluster considered so far—the cluster with the least cost. When considering a new cluster, we first estimate the cost by adding one to the cost of the inputs of the cluster. If the cost is less than that of the best cluster, we then proceed to match the cluster

function. Otherwise, the cluster can be rejected and the next cluster is considered. After all the clusters rooted at v have been tried, the best cluster is retained and the procedure continues until the optimal cover for the whole tree is determined. The covering algorithm is shown in Figure 5.4.

```
cover(node) {
   IF the node is a leaf node {
      RETURN;
   }
   ELSE {
      WHILE there are more child nodes to be covered {
         cover(the next child node);    /* recursively cover the child nodes */
      }
      FOR each of the cluster rooted at the node {
         IF the estimated cost is less than
                  that of the best cluster so far {
            match the cluster;
            IF the cluster has a match {
               replace the best cluster;
            }
         }
      }
   }
}
```

Figure 5.4: Covering algorithm.

5.4 Boolean Matching

We now consider the matching step. Matching is used in the covering stage to verify if a particular cluster function is logically equivalent to a library function. For this purpose, we use the spectral NPN canonical form presented in section 4.2. As we have mentioned, two functions are NPN-equivalent if, and only if, they have the same canonical form.

The matching algorithm first transforms the cluster function under consideration

into its spectral NPN canonical form. Then, there are various ways to do the matching. One way is to use only the NPN-LIB mentioned in section 5.1. If the NPN canonical form of the cluster function is found in the NPN-LIB, then the library function and the cluster function are NPN-equivalent. The transformations (input negation, input permutation, and output negation) used to transform the two functions to their common canonical form can be used to transform one function to the other. For example, to transform function $f(a,b,c) = ac + b\bar{c}$ to its canonical form, we only need the permutation *bac*. To transform function $g(x,y,z) = x\bar{y} + \bar{x}z$ to its canonical form, we need the negation of y, and the permutation zyx. Therefore, function $g(x,y,z)$ can be transformed from $f(a,b,c)$ by connecting $\bar{y}$ to a, z to b, and x to c, *i.e.*, $g(x,y,z) = f(\bar{y},z,x)$.

The advantage of this approach is that the size of the library can be greatly reduced, and searching for a match is therefore much faster. The disadvantage of this approach is that the two matching functions are only NPN-equivalent, and transforming the cluster function to its matching library function may require input negations and/or output negation as well as input permutations. If the complement of a variable is required but neither provided nor implemented by another logic block, then, a logic block implementing an inverter would be necessary. In the worst case, an inverter for each of the primary inputs and the intermediate nodes is required. This would be an inefficient way to use the logic blocks and more of them are needed.

For this reason, we choose to do the matching in another way. First of all, the functions in the technology library are organized according to their spectral NPN canonical form. The NPN-LIB, which contains all the different spectral NPN canonical form, is also used as an index file. We first transform the cluster function into its spectral NPN canonical form, and search the NPN-LIB for this form. If it is not found, the cluster function is not matchable. Otherwise, we start searching for a match. For each of the intermediate nodes to be matched, we try to match both its positive and negative phases. That is, for a certain cluster function $f_{v,i}$ of node v, we try to match both $f_{v,i}$ and $\bar{f}_{v,i}$, and keep a record of whether both can be matched,

or which one of the two can be matched if only one of them can be matched. If in the best cluster of v, only the negative phase of v can be matched, we invert the phase of v, so node v is now node $\bar{v}$, and only the positive of this new node can be matched.

Searching for a match is done in two stages. In the first stage, we only consider the positive phase of the cluster inputs. We use the index information in the NPN-LIB to search the P-LIB. This can greatly reduce the searching space. For example, when targeted to the Actel Act-1, for a four-variable function, in the worst case, we only need to search 43 cells in NPN-LIB and 19 cells in P-LIB, a total of 62 cells instead of 210 (see Table 5.1).

In most cases, there is more than one function in the technology library that has the same canonical form as the cluster function. Since we do not consider input negation, we are looking for the one that is P-equivalent to the cluster function, *i.e.*, the cluster function can be transformed to the library function by only permuting its inputs.

Once again, we use spectral information to speed up the searching.

Let

$$w_p = \sum_{\forall \alpha \subseteq \beta} s_{p\alpha}, \quad 1 \leq p \leq n, \quad \beta = \{1, \ldots, p-1, p+1, \ldots, n\}$$

w_p is the sum of the spectral coefficients with p as a subscript. We associate with each library function its zero-order, first-order, and the highest order spectral coefficients and all the v_i and w_i values.

It can be easily seen that two P-equivalent functions have the same zero-order and highest-order spectral coefficients. Moreover, they have the same set of first-order coefficients and v_i and w_i values respectively. We use these observations as our criterion in the searching. If a library function satisfies this criterion, we then proceed to identify the correspondence between the variables of the two functions. For each variable of the library function, we establish a set of candidate variables of the cluster function, using the first-order coefficient and the v_i and w_i values as a criterion. If a one-to-one correspondence is established between the variables of

the two functions, the input permutation is determined. Otherwise, we try all the combinations of the candidates to finally determine the correspondence.

If the first stage fails to find a match, we consider input negations in the second stage. As in the first stage, we examine the same set of candidate functions in P-LIB, using the index information in NPN-LIB. For each candidate function, we can easily determine the necessary input permutations and negations for a match. Since the phase information for each cluster input has been obtained, as described above, in the bottom-up procedure, it can be easily determined whether or not a match can be established.

5.5 Summary

In this chapter, we presented an approach to technology mapping which uses the similar partitioning, decomposition, and covering techniques to those used in previous algorithms. A matching method was proposed which is based on the new spectral canonical form presented in chapter 4. We have shown that spectral information can be used in two stages of the technology mapping procedure to improve its efficiency. First, it can be used in a preprocessing stage to process the technology library. Secondly, it can be used in the matching step to quickly identify a match.

Chapter 6

Experimental Results

We have presented a new matching algorithm for technology mapping of multiplexer-based FPGA's. A technology mapper has been implemented to test the effectiveness of the new matching algorithm. The mapper has been implemented in the C language and has been tested on a set of MCNC and ISCAS benchmark circuits.

In this chapter, we present the test results of our mapper. The results of other well known mappers are also presented for comparison purpose.

Section 6.1 provides a brief description of the mapper. Section 6.2 presents the test results. The chapter concludes with a brief summary in section 6.3.

SIS Script

As mentioned before, technology independent optimization is performed prior to technology mapping. We use the SIS system [29] to perform technology independent optimization of the original Boolean network. SIS is an interactive tool for synthesis and optimization of sequential circuits. It is build on top of MISII [28] and includes all the combinational optimization techniques in MISII as well as many enhancements. Like MISII, SIS can be run in batch mode in which a script file is used which contains all the SIS command to be executed for the optimization. Figure 6.1 shows the script file we used.

After SIS is invoked, we first use the command "*read_blif*" or "*read_pla*", depending on the file format, to read the file of the circuit description. The script file is

```
sweep
simplify -m nocomp
resub
gkx
gcx
resub
eliminate 0
gkx
gcx
resub
eliminate 0
decomp
resub
eliminate 0
simplify -m nocomp
gkx
gcx
decomp
resub
tech_decomp -a 2 -o 2
```

Figure 6.1: Script file for SIS.

executed by using the command "*source -x* **script**", where **script** is the filename of the script.

At the end of the script file, we use the command "*tech_decomp -a 2 -o 2*". This command decomposes the Boolean network such that the Boolean function associated with each node has no more than two products, and each product has no more than two literals. This is a particular requirement of our technology mapper.

After the script has been executed, the optimized network is written to a file in BLIF (Berkeley Logic Interchange Format) format [30] using the command "*write_blif*". This file is the input of our mapper.

Benchmark Circuits

For test purposes, a set of MCNC and ISCAS benchmark circuits are used. All the benchmarks are two-level or multi-level combinational circuits. Table 6.1 provides a brief description of all the benchmarks used. For each circuit, it shows the name of the circuit, number of inputs, number of outputs, and file format of the original circuit description, in columns "Benchmark", "Inputs", "Outputs", and "Format", respectively. Table 6.1 also gives the number of product terms for two-level circuits, and number of gates for multi-level circuits, in columns "Products" and "Gates" respectively.

6.1 The Technology Mapper

The technology mapper we developed is fairly standard except for the use of the new matching algorithm. The mapper reads the Boolean network to be mapped, which is in BLIF format, and partitions it into a forest of trees. Each of the tree is then decomposed such that the Boolean function associated with each node in the tree is a two-input AND/OR function. Finally, a bottom-up dynamic programming approach is used to cover each tree with an interconnection of library cells. We use the Actel Act-1 FPGA as our example technology. The output of the mapper consists of three files. A text file—.map file, shows how many Act-1 logic blocks are required to implement a design and how these logic blocks are personalized. A .v file describes the implementation in Verilog HDL format, and a testbench file, which is also in Verilog format, provides test vectors for the .v file. The test vectors are generated from the minterms in the "pla" files (for "blif" files, we first convert them to "pla" files). For each minterm, we assign a logic 1 to all the inputs in the minterm, and randomly assign either a logic 1 or a logic 0 to the inputs that are not in the minterm. With the .v file and the testbench, verification can be easily performed using Verilog.

The program [16] used to transform a given Boolean function to its canonical form

Table 6.1: A set of MCNC and ISCAS benchmarks.

Benchmark	Inputs	Outputs	Products	Gates	Format
z4ml	7	4	-	20	blif
9symml	9	1	-	43	blif
f51m	8	8	-	43	blif
b9	41	21	-	125	blif
count	35	16	-	143	blif
apex7	49	37	-	176	blif
C499	41	32	-	202	blif
alu2	10	8	-	335	blif
C880	60	26	-	383	blif
apex6	135	99	-	452	blif
alu4	14	8	-	681	blif
rot	135	107	-	691	blif
C1908	33	25	-	880	blif
C5315	178	123	-	2307	blif
des	256	245	-	4000	blif
misex2	25	18	29	-	pla
misex1	8	7	32	-	pla
sao2	10	4	58	-	pla
e64	65	65	65	-	pla
5xp1	7	10	75	-	pla
9sym	9	1	87	-	pla
duke2	22	29	87	-	pla
bw	5	28	87	-	pla
vg2	25	8	110	-	pla
rd73	7	3	141	-	pla
clip	9	5	167	-	pla
rd84	8	4	256	-	pla
apex2	39	3	1035	-	pla

takes as input the truth-table and the number of variables of the given function. The output of the program is the permutations and negations needed to transform the function to its canonical form.

In developing the mapper, we employed some heuristics. In order to test the effect of these heuristics, the mapper provides some options corresponding to the heuristics. In the partitioning step, we have the option of either creating a new tree for every intermediate node that has more than one parent, or duplicating an intermediate node if it has only two parents and none of its children is intermediate node. In the covering step, we have the option of matching only the positive phase of a node for every node, or matching both the positive and negative phases of a node if it is an intermediate node. Finally, we would like to see how the primary inputs affect the final results.

We shall represent the combination of options as a three digit code *icp* where:

- *i* represents the option of the primary input conditions and takes either of the following two values:

  0: indicating that only the positive phase of the primary inputs are provided.

  1: indicating that both positive and negative phases of each primary inputs are provided.

- *c* represents the option in the covering step and takes either of the following two values:

  0: indicating that for each non-leaf node, only the positive phase is to be matched.

  1: indicating that for each non-leaf node, the positive phase is to be matched, and for each intermediate node, the negative phase is also to be matched.

- p represents the option in the partitioning step and takes either of the following two values:

  0: indicating that every intermediate node with more than one parent is made the root of a new tree.

  1: indicating that an intermediate node is duplicated if it has only two parents and none of its children is intermediate node, and each of the other intermediate nodes which have more than one parent is made the root of a new tree.

However, not all of the eight possible option combinations are valid. We restrict the choices such that i can not take the value 1 when c takes the value 0. Therefore, only six of the eight possible option combinations are valid: "000", "001", "010", "011", "110", "111".

Table 6.2 shows the results on the benchmark circuits using all the valid option combinations. The results given are the number of Act-1 logic blocks needed to implement the corresponding circuit. No results are reported for the benchmark **bw** since the circuit contains some external *don't care* information which the mapper does not consider at present.

To see the effects of the heuristic, however, instead of using all the valid combinations, it would be sufficient and clearer to use options "010" and "011" to show the effect of the partitioning heuristic, "001" and "011" to show the effect of the covering heuristic, and "011" and "111" to show the effect of the primary input conditions, as shown in Table 6.3.

In Table 6.3, column "Partitioning" shows the effect of using the partitioning heuristic. Column "Covering" shows the results of using the covering heuristic. And column "Primary Inputs" gives the result under different primary input conditions: only the positive phase is available, or both positive and negative phases are available.

Table 6.4 summarizes Table 6.3 by indicating the better results under the options used in Table 6.3. In columns "Partitioning" and "Covering", "No" indicates the

Table 6.2: Test results for all valid options of the mapper.

Benchmark	Valid options					
	000	001	010	011	110	111
z4ml	17	15	17	15	16	15
9symml	94	92	94	92	84	82
f51m	42	41	41	40	38	38
b9	50	51	48	49	45	46
count	47	47	47	47	47	47
apex7	108	109	107	108	97	96
C499	168	166	168	166	168	158
alu2	186	183	185	182	180	177
C880	157	156	156	155	139	138
apex6	354	352	352	350	325	321
alu4	330	325	330	325	322	317
rot	252	248	252	247	238	233
C1908	179	172	178	171	178	169
C5315	597	593	592	590	586	584
des	1669	1666	1668	1665	1667	1664
misex2	38	39	38	39	37	37
misex1	18	17	18	17	18	17
sao2	58	60	56	56	51	49
e64	95	95	95	95	95	95
5xp1	44	46	42	43	42	43
9sym	95	93	95	93	92	91
duke2	207	207	205	205	193	193
bw	-	-	-	-	-	-
vg2	36	36	35	35	29	30
rd73	29	27	29	27	27	25
clip	62	63	62	62	59	56
rd84	72	70	70	68	69	67
apex2	136	131	130	127	125	120

Table 6.3: Test results for various mapper options.

Benchmark	Partitioning		Covering		Primary Inputs	
	010	011	001	011	011	111
z4ml	17	15	15	15	15	15
9symml	94	92	92	92	92	82
f51m	41	40	41	40	40	38
b9	48	49	51	49	49	46
count	47	47	47	47	47	47
apex7	107	108	109	108	108	96
C499	168	166	166	166	166	158
alu2	185	182	183	182	182	177
C880	156	155	156	155	155	138
apex6	352	350	352	350	350	321
alu4	330	325	325	325	325	317
rot	252	247	248	247	247	233
C1908	178	171	172	171	171	169
C5315	592	590	593	590	590	584
des	1668	1665	1666	1665	1665	1664
misex2	38	39	39	39	39	37
misex1	18	17	17	17	17	17
sao2	56	56	60	56	56	49
e64	95	95	95	95	95	95
5xp1	42	43	46	43	43	43
9sym	95	93	93	93	93	91
duke2	205	205	207	205	205	193
bw	-	-	-	-	-	-
vg2	35	35	36	35	35	30
rd73	29	27	27	27	27	25
clip	62	62	63	62	62	56
rd84	70	68	70	68	68	67
apex2	130	127	131	127	127	120

heuristic is not used, while "Yes" indicates the heuristic is used. In column "Primary Input", "Positive" indicates only the positive phase of the primary inputs are provided, while "Both" indicates both positive and negative phases are available. "*" indicates that a better result is achieved under the corresponding condition. In the case of a tie, both columns have a "*".

We can see from Table 6.4 that in most cases (17 out of 27), using the partitioning heuristic produces better results than not using the heuristic. Only on four benchmarks, not using the heuristic yields better results.

As is expected, in all the cases, using the covering heuristic results in fewer, or at least equal, number of logic blocks needed, compared with not using the heuristic.

Finally, it is not surprising to see that better results are achieved if both positive and negative phases of the primary inputs are provided.

6.2 Comparing Results

In this section, we present the results obtained using our mapper and other well known mappers, including MIS-pga1, MIS, Ceres, proserpine,and Amap. The results for the other mappers are taken from [27].

The starting networks for MIS-pga1, MIS, Ceres, and Amap are the same. Our mapper has a different starting point. Proserpine may also have a different starting point. The option used for Amap was *-an3*. The options used for MIS-pga1 is "act_map -h3 -n1 -q -d4 -f3 -M4 -l -g0.001". And the option used for our mapper is "011", *i.e.*, we use both the partitioning heuristic and covering heuristic, and assume only the positive phase of the primary inputs is available.

MIS-pga1 and MIS were run on a DEC5500 (a 28 mips machine), whereas Amap was run on a SUN4/370 (a 12.5 mips machine). Our mapper was run on a SUN SPARCstation 10 (a 96.2 mips machine).

Table 6.5 shows the number of Act-1 logic blocks needed to implement the benchmarks. The results reported for our mapper have been verified using Verilog. A "-" in Table 6.5 indicates that the information is not available.

Table 6.4: Summary of better results obtained using various options of the mapper.

Benchmark	Partitioning		Covering		Primary Inputs	
	No	Yes	No	Yes	Positive	Both
z4ml		*	*	*	*	*
9symml		*	*	*		*
f51m		*		*		*
b9	*			*		*
count	*	*	*	*	*	*
apex7	*			*		*
C499		*	*	*		*
alu2		*		*		*
C880		*		*		*
apex6		*		*		*
alu4		*	*	*		*
rot		*		*		*
C1908		*		*		*
C5315		*		*		*
des		*		*		*
misex2	*		*	*		*
misex1		*	*	*	*	*
sao2	*	*		*		*
e64	*	*	*	*	*	*
5xp1	*			*	*	*
9sym		*	*	*		*
duke2	*	*		*		*
bw	-	-	-	-	-	-
vg2	*	*		*		*
rd73		*	*	*		*
clip	*	*		*		*
rd84		*		*		*
apex2		*		*		*

We can see that our mapper produces good results. On 11 (out of a total of 27) benchmarks, our mapper yields better results than any other mapper reported here. Those results are highlighted in Table 6.5. However, it produces the worst results on two of the benchmarks. This is due to the fact that heuristics are used to solve the technology mapping problem which is formulated as a DAG covering DAG's problem and is NP-hard (see chapter 2). It is not surprising to see that, in Table 6.5, no mapper consistently produces better results than any other mappers. Here, we would like to point out that the final results are greatly affected by the starting point. To show the effect of the starting point, we have run SIS on the same starting networks for our mapper, and the results are given in Table 6.6. In order to show the effect of starting point, Table 6.6 also copies the results for MIS-pga1 and MIS from Table 6.5. So, in Table 6.6, SIS and our mapper (Spectral) have the same starting point, different from that of MIS-pga1 and MIS. The options used for SIS is the same as for MIS-pga1: "act_map -h3 -n1 -q -d4 -f3 -M4 -l -g0.001".

SIS is built on top of MIS, and incorporates many different programs and algorithms, including the algorithms of MIS-pga1. Therefore, given the same starting point, SIS should produce results close to those of MIS-pga1 and MIS. Comparing the column "SIS" with the columns "MIS-pga1" and "MIS" in Table 6.6, we can see how much the starting point affects the final results.

Table 6.7 shows the time taken in seconds to perform the mapping. Three times are reported for our mapper. "SIS" is the time taken for SIS to do the technology independent optimization and is obtained by adding the command "time" at the end of the script file shown in Figure 6.1. "TM" is the time taken for our mapper to do the technology mapping as reported by the UNIX "time" command. "Total" is the sum of the two.

Since the results for different mappers are obtained on machines with different mips values, it is clearer to normalize the results with respect to a reference mips value. Table 6.8 shows the normalized results. The reference value is 12.5 mips. This comparison is not fully valid due to difference in the machines, but is indicative of

Table 6.5: Test results for various mappers in terms of the number of logic blocks.

Benchmark	Size	MIS-pga1	MIS	Ceres	Proserpine	Amap	Spectral
	gates						
z4ml	20	16	20	17	-	20	**15**
9symml	43	80	**73**	-	-	74	92
f51m	43	48	52	54	63	56	**40**
b9	125	65	64	101	-	81	**49**
count	143	46	63	62	-	**41**	47
apex7	176	**96**	113	106	121	104	108
C499	202	166	174	166	170	**136**	166
alu2	335	208	193	**173**	-	188	182
C880	383	171	175	177	-	190	**155**
apex6	452	**289**	360	441	396	392	350
alu4	681	**132**	149	326	350	160	325
rot	691	288	313	418	465	335	**247**
C1908	880	175	188	192	-	**158**	171
C5315	2307	656	704	725	-	653	**590**
des	4000	1749	**1571**	1638	-	1634	1665
	prods						
misex2	29	41	46	42	45	47	**39**
misex1	32	20	22	22	25	25	**17**
sao2	58	62	**52**	86	-	56	56
e64	65	**94**	95	95	-	105	95
5xp1	75	45	51	47	53	**42**	43
9sym	87	119	99	136	-	106	**93**
duke2	87	**166**	176	172	177	175	205
bw	87	65	81	**61**	67	83	-
vg2	110	36	47	42	46	44	**35**
rd73	141	31	32	32	-	32	**27**
clip	167	**51**	57	62	73	60	62
rd84	256	**61**	62	**61**	70	62	68
apex2	1035	124	**106**	175	-	122	127

the relative performance.

In Table 6.8, the results reported for MIS-pga1 and MIS include the time to do the technology independent optimization. For Amap, the results are only the time taken for technology mapping. Therefore, we should use the "Total" time to compare with MIS-pga1 and MIS, and use the "TM" time to compare with Amap. We can see that our mapper is quite fast. Note that MIS-pga1 takes much more time to run on the symmetric circuits than on other circuits (the symmetric circuits include 9symml, rot, 9sym, rd73, rd84), but, our mapper does not show this characteristic. Also, it is easy to see that, compared with technology independent optimization, our mapper only accounts for a very small portion in the total time. For almost all the benchmarks, the mapping can be done in a matter of seconds. Benchmark "des" takes much longer than other benchmarks because the circuit is much larger than the others (see Table 6.1).

6.3 Summary

In this chapter, we presented test results for our mapper and other well-known mappers on a set of benchmark circuits. The results have shown that the matching algorithm presented in this thesis is very effective in terms of both the final results and the time taken to perform the mapping.

Table 6.6: Test results to show the effect of starting point. Two different starting points are used: one for MIS-pga1 and MIS, the other for SIS and Spectral.

Benchmark	MIS-pga1	MIS	SIS	Spectral
z4ml	16	20	20	15
9symml	80	73	123	92
f51m	48	52	60	40
b9	65	64	60	49
count	46	63	62	47
apex7	96	113	145	108
C499	166	174	290	166
alu2	208	193	251	182
C880	171	175	215	155
apex6	289	360	454	350
alu4	132	149	472	325
rot	288	313	346	247
C1908	175	188	295	171
C5315	656	704	958	590
des	1749	1571	2451	1665
misex2	41	46	52	39
misex1	20	22	24	17
sao2	62	52	80	56
e64	94	95	108	95
5xp1	45	51	60	43
9sym	119	99	119	93
duke2	166	176	265	205
bw	65	81	75	-
vg2	36	47	42	35
rd73	31	32	37	27
clip	51	57	83	62
rd84	61	62	97	68
apex2	124	106	152	127

Table 6.7: Test results for various mappers in terms of the time taken to do the mapping.

Benchmark	Size	MIS-pga1	MIS	Amap	Spectral		
					SIS	TM	Total
	gates						
z4ml	20	19.6	2.0	0.8	1.0	0.1	1.1
9symml	43	3123.8	10.5	2.7	8.5	0.9	9.4
f51m	43	36.6	5.2	2.2	2.7	0.2	2.9
b9	125	31.6	5.9	2.7	2.9	0.1	3.0
count	143	13.2	6.7	1.5	2.5	0.1	2.6
apex7	176	42.1	10.6	3.8	4.1	0.2	4.3
C499	202	35.3	17.8	7.0	10.3	0.3	10.6
alu2	335	824.5	24.2	8.3	86.4	1.2	87.6
C880	383	77.2	18.2	7.2	10.3	0.4	10.7
apex6	452	255.5	34.4	15.0	13.3	1.2	14.5
alu4	681	145.9	31.6	6.1	345.6	2.0	347.6
rot	691	1071.8	28.9	11.9	19.6	0.8	20.4
C1908	880	646.5	19.3	7.6	16.9	0.4	17.3
C5315	2307	673.1	88.0	26.9	98.5	2.9	101.4
des	4000	762.9	756.6	67.2	402.1	22.3	424.4
	prods						
misex2	29	6.1	4.3	1.5	1.8	0.1	1.9
misex1	32	11.5	1.9	1.1	0.9	0.1	1.0
sao2	58	54.2	8.0	2.0	3.6	0.3	3.9
e64	65	3.9	7.9	3.3	18.6	0.3	18.9
5xp1	75	60.0	4.4	1.7	2.9	0.2	3.1
9sym	87	17582.8	14.1	4.1	15.2	0.9	16.1
duke2	87	403.5	17.9	6.9	121.9	1.2	123.1
bw	87	20.1	7.1	3.6	-	-	-
vg2	110	18.4	3.9	1.5	2.5	0.1	2.6
rd73	141	37.2	2.7	1.6	2.3	0.1	2.4
clip	167	91.9	4.9	2.5	5.9	0.3	6.2
rd84	256	151.4	6.5	2.6	13.8	0.4	14.2
apex2	1035	8.3	11.8	5.0	71.4	1.1	72.5

Table 6.8: Normalized test results for various mappers in terms of the time taken to do the mapping.

Benchmark	Size	MIS-pga1	MIS	Amap	Spectral		
					SIS	TM	Total
	gates						
z4ml	20	43.9	4.5	0.8	7.7	0.8	8.5
9symml	43	6997.3	23.5	2.7	65.5	6.9	72.4
f51m	43	82.0	11.7	2.2	20.8	1.5	22.3
b9	125	70.8	13.2	2.7	22.3	0.8	23.1
count	143	29.6	15.0	1.5	19.3	0.8	20.1
apex7	176	94.3	23.7	3.8	31.6	1.5	33.1
C499	202	79.1	39.9	7.0	79.3	2.3	81.6
alu2	335	1846.9	54.2	8.3	665.3	9.2	674.5
C880	383	172.9	40.8	7.2	79.3	3.1	82.4
apex6	452	572.3	77.1	15.0	102.4	9.2	111.6
alu4	681	326.8	70.8	6.1	2661.1	1.5	2662.6
rot	691	2400.8	64.7	11.9	150.9	6.2	157.1
C1908	880	1448.2	43.2	7.6	130.1	3.1	133.2
C5315	2307	1507.7	197.1	26.9	758.5	22.3	780.8
des	4000	1708.9	1694.8	67.2	3096.2	171.7	3267.9
	prods						
misex2	29	13.7	9.6	1.5	13.9	0.8	14.7
misex1	32	25.8	4.3	1.1	6.9	0.8	7.7
sao2	58	121.4	17.9	2.0	27.7	2.3	30.0
e64	65	8.7	17.7	3.3	143.2	2.3	145.5
5xp1	75	134.4	9.9	1.7	22.3	1.5	23.8
9sym	87	39385.5	31.6	4.1	117.0	6.9	123.9
duke2	87	903.8	40.1	6.9	938.6	9.2	947.8
bw	87	45.0	15.9	3.6	-	-	-
vg2	110	41.2	8.7	1.5	19.3	0.8	20.1
rd73	141	83.3	6.1	1.6	17.7	0.8	18.5
clip	167	205.9	11.0	2.5	45.4	2.3	47.7
rd84	256	339.1	14.6	2.6	106.3	3.1	109.4
apex2	1035	18.6	26.4	5.0	549.8	8.5	558.3

Chapter 7

Conclusions and Future Directions

In this thesis, we presented a new matching algorithm for the technology mapping of multiplexer-based FPGA's. The algorithm is based on a new spectral NPN canonical form for Boolean functions. As has been shown, two functions are considered matchable if, and only if, they have the same canonical form.

In chapter 2, we pointed out that a major obstacle to applying the conventional library-based technology mapping approaches to FPGA's is the large number of different functions an FPGA logic block can implement. In the case of the Actel Act-1 FPGA, the library would have 702 functions. The problem with the library-based approaches is that there is no effective way to determine which library cells can possibly be matchable with a given target function. One way to solve the problem is reducing the library. However, we believe that such reduction may impede significant optimizations.

Using the new spectral NPN canonical form, we can organize the library functions in terms of their canonical form. A given target function is transformed to its canonical form and only those library cells that have the same canonical form need to be considered for a match. This greatly reduces the searching space. For example, for a four-variable function, with Act-1 as the target technology, at most 62 library cells need to be considered instead of 210. If we consider NPN equivalence, then, only 43 library cells need to be considered. Furthermore, spectral information can be used to expedite input assignment after a match has been detected. If the library

is further organized in terms of the spectral information, such as s_0, s_i, and v_i, input assignment would be made highly efficient.

Another advantage of using the new canonical form is that, the method, like other Boolean matching methods, relies solely on the functionality of the functions to be matched. Logic equivalence can be detected regardless of the representation of the functions. Also, the method is independent of the structure of the logic block of the target FPGA. Therefore, it can be easily extended to other technologies.

Based on the new canonical form, we have implemented a technology mapper and applied it to a set of MCNC and ISCAS benchmark circuits. The test results presented in chapter 6 suggest that the algorithm is very effective in terms of both the area of the final circuit and the time it takes to do the mapping.

Future Work

As a result of our research work, we identify the following problems that could be investigated in the future to improve both the canonical form algorithm and the technology mapper.

- For the canonical form algorithm, one major concern is the interaction between negation and permutation. For example, functions $a + \bar{b}$ and $\bar{a} + b$ are NPN equivalent, but one function can be transformed to the other by either permuting the variables or inverting them. Clearly, permutation is preferable because it does not require extra hardware. Therefore, a more detailed study should be made to investigate the situations where permutation can be substituted for negation. Ideally, we will no longer need the P-LIB mentioned in chapter 5, which would certainly result in a great improvement of the performance of our mapper.

- The partitioning algorithm we used partitions the Boolean network into a forest of trees. This precludes the possibility of covering across tree boundaries. It would be interesting to use other partitioning algorithms such that the subnet-

works resulted in are not restricted to trees. Moreover, in the partitioning step, we used a heuristic that only considers two extreme cases: either making all the intermediate nodes with more than one parent the root of a new tree, or duplicate the nodes that has only two parents and children being intermediate node. We suppose the heuristic can be improved to achieve optimal results.

- In the covering step of the mapper, we try to match both phases of the intermediate nodes. However, for the root node of the new trees created in the partitioning step, we only try to match its positive phase, because the node has more than one parents and different parents may require different phases of the node. In this case, inverters would be necessary. The problem is how to decide when an inverter should be created. Actually, this problem applies to all the nodes in the network.

- For the primary inputs, our algorithm only considers two extreme situations: only the positive phase is available or both the positive and the negative phases are available for every primary input. It would be desirable if the mapper could handle all possible situations. For example, the primary input condition could be represented as a mask which indicates the phase of each primary input.

- Our technology mapper did not consider the symmetry and *don't care* conditions within a circuit. Exploiting these conditions should produce better results.

Research on the above problems would certainly enhance the techniques presented in this thesis. Also, as mentioned above, the matching algorithm is not restricted to one technology. It would be of interest to apply the method to other technologies to see the effectiveness of the algorithm. Another research direction is to optimize circuit delay or routability rather than area, or any combination of the three measures.

Bibliography

[1] *FPGA Data Book and Design Guide*, Actel Corporation, 1995.

[2] J. Darringer, D. Brand, J. Gerbi, W. Joyner, and L. Trevillyan, "LSS: A system for production logic synthesis," *IBM Journal of Research and Development*, vol. 28, no. 5, pp. 537–545, September 1984.

[3] D. Gregory, K. Bartlett, A. de Geus, and G. Hachtel, "Socrates: A system for automatically synthesizing and optimizing combinational logic,", *Proceedings of the 23rd Design Automation Conference*, pp. 79–85, June 1986.

[4] K. Keutzer, "DAGON: Technology binding and local optimization by DAG matching," *Proceedings of the 24th Design Automation Conference*, pp. 341–347, June 1987.

[5] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Technology Mapping in MIS," *International Conference on Computer-Aided Design*, IEEE, pp. 116–119, November 1987.

[6] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Logic Synthesis for Programmable Gate Arrays," *Proceedings of the 27th Design Automation Conference*, pp. 620–625, June 1990.

[7] R. Murgai, R. K. Brayton, A. Sangiovani-Vincentelli, "An Improved Synthesis Algorithm for Multiplexor-based PGA's," *Proceedings of the 29th Design Automation Conference*, pp. 380–386, 1992.

[8] K. Karplus, "Xmap: a Technology Mapper for Table-lookup Field-Programmable Gate Arrays," *Proceedings of the 28th Design Automation Conference*, pp. 240–243, 1991.

[9] K. Karplus, "Amap: a Technology Mapper for Selector-based Field-Programmable Gate Arrays," *Proceedings of the 28th Design Automation Conference*, pp. 244–247, 1991.

[10] F. Mailhot and G. De Micheli, "Technology Mapping Using Boolean Matching and Don't Care Sets," *European Design Automation Conference*, pp. 212–216, 1990.

[11] S. Ercolani and G. De Micheli, "Technology Mapping for Electrically Programmable Gate Arrays," *Proceedings of the 28th Design Automation Conference*, pp. 234–239, 1991.

[12] J. R. Burch and D. E. Long, "Efficient Boolean Function Matching," *Proceedings of the International Conference on Computer-Aided Design*, pp. 408–411, 1992.

[13] H. Savoj, M. J. Silva, R. K. Brayton, A. Sangiovanni-Vincentelli, "Boolean Matching in Logic Synthesis," *1992 European Design Automation Conference*, pp. 168–174, 1992.

[14] K. Zhu and D. F. Wong, "Fast Boolean Matching for Field-Programmable Gate Arrays," *Proceedings of the 1993 European Design Automation Conference*, pp. 352–357, 1993

[15] B. Kapoor, "Improved Technology Mapping Using A New Approach to Boolean Matching," *Proceedings of the 1995 European Design and Test Conference*, pp. 86–90, 1995.

[16] D. M. Miller, "A Spectral Method For Boolean Function Matching," *1996 European Design and Test Conference*, 1996.

[17] M. G. Karpovsky, Ed., *Spectral Techniques and Fault Detection*, Academic Press, Orlando, 1985.

[18] S. L. Hurst, D. M. Miller and J. C. Muzio, *Spectral Techniques in Digital Logic*, Academic Press, London, 1985.

[19] T. Sasao, Ed., *Logic Synthesis and Optimization*, Kluwer Academic Publishers, Boston, 1993

[20] S. B. Akers, "Binary Decision Diagrams," *IEEE Transactions on Computers*, vol. 27, pp. 509–516, June 1978.

[21] R. E. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. 35, pp. 677–691, August 1986.

[22] K. Karplus, "Using if-then-else DAGs for Multi-level Logic Minimization," *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pp. 101-118, March 1989.

[23] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, *Field-Programmable Gate Arrays*, Kluwer Academic Publishers, Boston, 1992.

[24] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli, "Architecture of Field-Programmable Gate Arrays," *Proceedings of the IEEE*, vol. 81, no. 7, pp. 1013–1029, July 1993.

[25] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel Logic Synthesis," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 264–300, February 1990.

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

[27] A. Sangiovanni-Vincentelli, A. El Gamal, and J. Rose, "Synthesis Methods for Field Programmable Gate Arrays," *Proceedings of the IEEE*, vol. 81, no. 7, pp. 1057–1083, July 1993.

[28] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A Multiple-Level Logic Optimization System," *IEEE Transactions on Computer-Aided Design*, pp. 1062–1081, 1987.

[29] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, A. Sangiovanni-Vincentelli, *SIS: A System for Sequential Circuit Synthesis*, Memorandum No. UCB/ERL M92/41, Electronics Research Laboratory, Department of Electrical Engineering and Computer Science, University of California, Berkeley, May 1992.

[30] S. Yang, *Logic Synthesis and Optimization Benchmarks User Guide*, Technical Report, Microelectronics Center of North Carolina, January 1995.

VITA

Surname: Wang Given Names: Xiaojun

Place of Birth: Huayin, Shaanxi, China

Educational Institutions Attended:

University of Victoria	1994 to 1996
University of Electronics Science and Technology of China	1985 to 1989

Degrees Awarded:

B.Sc.	University of Electronics Science and Technology of China	1989

Honours and Awards:

Publications:

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

Technology Mapping for FPGA's Using Boolean Matching and Spectral Techniques

Author: [redacted]

Xiaojun Wang

July 30, 1996