FIELD-PROGRAMMABLE GATE ARRAY LOGIC SYNTHESIS
USING BOOLEAN SATISFIABILITY

by

Andrew C. Ling

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Field-Programmable Gate Array Logic Synthesis

Using Boolean Satisfiability

Andrew C. Ling

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2005

Field-Programmable gate arrays (FPGAs) are reprogrammable logic chips that can be config-
ured to implement various digital circuits. FPGAs are fast replacing custom ASICs in many
areas due to their flexibility and fast turn around times for product development. However,
these benefits come at a heavy cost of area, speed, and power.

The FPGA architecture and technology mapping phase are fundamental in determining
the performance of the FPGA. This thesis presents novel tools using Boolean satisfiability
(SAT) to aid in both these areas. First, an architecture efficiency evaluation tool is developed.
The tool works by reading in a description of the FPGA architecture and rates how flexible
that architecture can be in implementing various circuits. Next, a novel technology mapping
approach is developed and compared to current methods. This work contrasts with current
approaches since it can be applied to almost any FPGA architecture. Finally, a resynthesis
algorithm is described which rates the utility of current FPGA technology mappers where it
can also be used to discover optimal configurations of common subcircuits to digital design.

Acknowledgements

"Research is what I'm doing when I don't know what I'm doing."

- Wernher von Braun (1912-77), *German-born American rocket engineer*

Contents

List of Tables

List of Figures

List of Algorithms

List of Terminology

Term	Description	Reference
FPGA	Field Programmable Gate Array	Sec. 1.1
LUT	Lookup Table: a programmable circuit used to implement various logic functions.	Sec. 1.1
PLB	Programmable Logic Block: a programmable circuit used to implement various logic functions, more rigid that a LUT, but has other benefits.	Sec. 1.1
Technology Mapping	The process of converting generic netlists to specific logic libraries.	Sec. 1.1
SAT	Boolean satisfiability: determining if a Boolean formula can evaluate to true.	Sec. 2.1
QSAT	Quantified Boolean satisfiability: determining if a quantified Boolean formula can evaluate to true.	Sec. 2.1
CNF	Conjunctive Normal Form: a Boolean expression consisting of a conjunction of clauses.	Sec. 2.1
clause	A Boolean expression consisting of a disjunction of variables or their complements.	Sec. 2.1
QBF	Quantified Boolean formula: a quantified Boolean formula.	Sec. 2.1
PI	Primary input: node in a circuit graph that has no input edges.	Sec. 3.2
PO	Primary output: node in a circuit graph with no output edges.	Sec. 3.2
K-Bounded Graph	A directed acyclic graph where all nodes have no more than K input edges.	Sec. 3.2
cone	A subgraph with a root node and a subset of its predecessors that have a path to the root node entirely contained in the subgraph.	Sec. 3.2
FFC	A fanout free cone that has output edges leaving the cone only from the root node.	Sec. 3.2
MFFC	A maximum fanout free cone that maximizes the number of nodes contained in the cone.	Sec. 3.2

Term	Description	Reference
depth	The sum of the edge weights in a DAG along the longest forward path starting from a primary input and ending at a node. The depth of the DAG is the longest node depth found in the graph.	Sec. 4.2.2
height	The sum of the edge weights in a DAG along the longest backward path starting from a primary output and ending at a node. The height of the DAG is the longest node height found in the graph.	Sec. 4.2.2

Chapter 1

Introduction

1.1 Background and Motivation

Since their introduction in 1984, Field-Programmable Gate Arrays (FPGAs) have revolutionized access to VLSI. FPGAs are programmable chips that can be used to quickly implement any digital circuit, as opposed to custom ASICs which have a large start up cost in terms of money and development time. However, even with this flexibility, dollar for dollar FPGAs still only comprise 5% of the digital silicon market [38]. This is because the benefits of FPGAs come at a high cost in terms of area, speed, and power. These characteristics are determined by three areas: the CAD flow responsible for implementing digital circuits on FPGAs; the FPGA architecture; and the FPGA transistor-level design. Research in all these areas is necessary if FPGAs are to overtake more sectors in the digital market. This dissertation hopes to aid this goal by introducing several tools that can be used to improve FPGA EDA tools and architecture. Also, this work introduces a novel application of Boolean Satisfiability to EDA.

The basic CAD flow to implement a circuit in an FPGA is shown in Figure 1.1. The CAD flow starts with a description of the circuit, usually in the form of a hardware description language (HDL) such as VHDL. Once the description is verified through simulation, synthesis occurs where the description is synthesized into a gate-level network consisting of primitive gates. Next, technology mapping occurs where the gate-level network produced in synthesis is converted into a network of programmable logic blocks (PLBs). Place and route is then
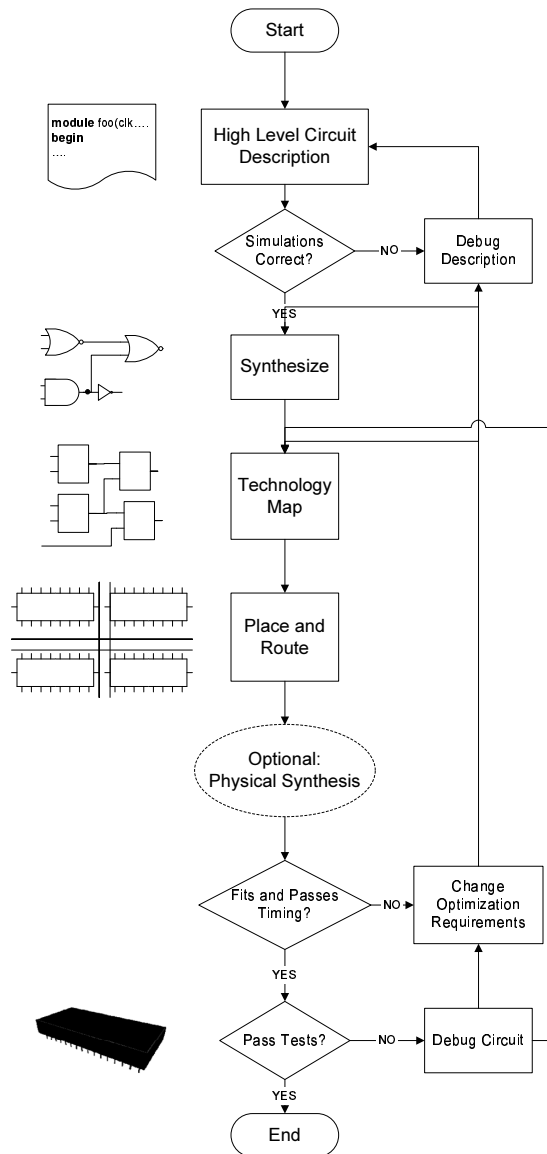
Figure 1.1: A CAD flow to realize a circuit in an FPGA.

responsible to physically arrange the PLB network such that it fits into an FPGA. As an optional step, most advanced FPGA CAD tools have a physical synthesis phase which takes advantage of information only available after place and route, such as routing wire delays, to further optimize the final circuit. Throughout the CAD flow, attempts to reach a set of speed, area, and power goals are made and a final circuit analysis is done to see if the goals were achieved. If not, designers can either try different options provided by the tools, relax their goals, or modify the design. This simplistic CAD flow omits many details, but shows the

significant role of the CAD tools on FPGA circuit implementation. Furthermore, the CAD tools must be tightly integrated with the FPGA architecture so that they can make judicious use of the available resources. The work presented in this dissertation looks specifically at the technology mapping phase.
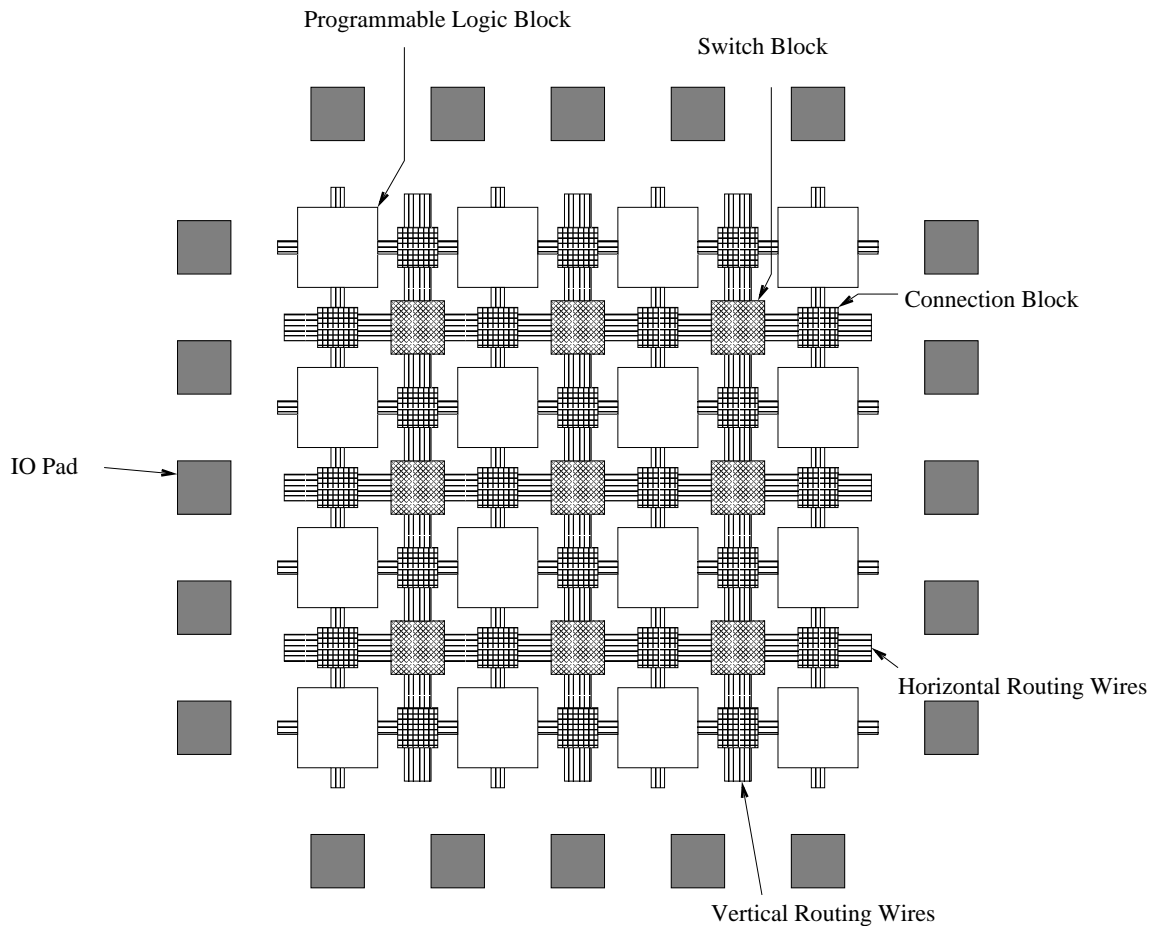
Figure 1.2: A generic island-style FPGA architecture.

The flexibility of FPGAs can be attributed to their reprogrammable routing structures and logic blocks. This is illustrated in Figure 1.2, where a generic island-style FPGA architecture is shown. The programmable logic blocks (PLBs, also known as configurable logic blocks, logic blocks, or logic elements) implement the circuit's logic functions, and the connection and switch blocks provide connections between routing wires. There exists a large body of work exploring various PLB architectures [2, 7, 14, 15, 26, 27, 39, 40], all of which are based upon the K-input lookup table (K-LUT).
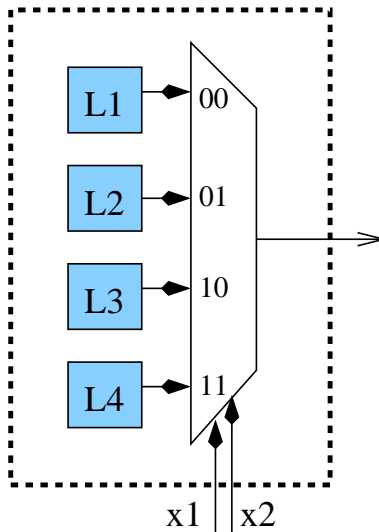
Figure 1.3: A 2-input lookup table.

A K-LUT consists of K inputs, one output, and 2^K configuration bits that serve as truth table entries. For example, Figure 1.3 shows a simple 2-LUT. By programming the 2^K configuration bits, the K-LUT can implement any K-input function. There exists a large body of research investigating an ideal K value. It has been shown that setting K to five or greater provides a significant speed advantage over smaller numbers [44]. However, when taking area into consideration, previous work has shown the benefits of setting K to four [39, 40].

Although the K-LUT is very flexible, it is usually beneficial to add non-programmable logic to the PLB [38]. For example, Figure 1.4 shows a commercial PLB architecture [6] where a large portion of the PLB is not programmable. Dedicated circuitry allows PLBs to implement a wider range of functions without the area, delay, and power costs associated with programmable logic. Furthermore, dedicated circuitry often work in conjunction with special interconnect structures to directly connect two or more PLBs in a cascade fashion [4, 49] to produce extremely fast subcircuits.

Unlike a K-LUT, a K-input PLB cannot implement all logic functions of K inputs. This property makes technology mapping to complex PLB architectures such as that in Figure 1.4 difficult. Heuristic approaches that perform this task are of two categories: a customized mapping algorithm [26, 27], or special Boolean techniques that decompose logic functions into

Figure 1.4: A PLB architecture: the Altera Stratix II ALM [6]

a form that maps directly into the PLB [14, 15]. A shortcoming of both these approaches is they lack generality: for every possible new PLB architecture that is developed, considerable time and effort is needed to create technology mapping heuristics. This dissertation addresses this problem in a general way using Boolean satisfiability through a tool called **SATMAP**. Boolean satisfiability is the problem of finding an assignment to the Boolean function $F(x_1, x_2, ..., x_n)$, such that F evaluates to 1. This is discussed in detail in Chapter 2.

1.1.1   Technology Mapping

When technology mapping a gate-level network to PLBs, the goal is to minimize area, delay, or a combination of the two. Most of the existing literature [12] on technology mapping uses the term *depth* to refer to delay. This work adopts this terminology when describing technology

mapping.

(a) Gate-level network   (b) Possible covering   (c) Final LUT network

Figure 1.5: Technology mapping as a covering problem.

The process of technology mapping is often treated as a covering problem. For example, consider the process of mapping a circuit into 4-LUTs as illustrated in Figure 1.5. Part (a) illustrates the initial gate-level network, part (b) gives a possible covering of the initial network using 4-LUTs, and part (c) shows the LUT network produced by the covering. In the mapping shown, the gate labeled x is covered by both LUTs and is said to be *duplicated*. Techniques that map for depth use large amounts of duplication to obtain solutions of reduced depth while techniques that map for area limit the use of duplication because it often increases the area of the mapped solutions. A large body of work covers this problem when applied to K-LUTs. Latter chapters demonstrate how to adapt this work to generic PLBs.

One of the earliest works to study the depth minimization problem in LUT mapping showed that the depth-optimal mapping solution can be obtained in polynomial time using a dynamic programming procedure [11]. Although this initial work assumed that the delay between

two primitive gates was constant, the result was subsequently generalized for a variable delay model [51].

In contrast to the depth minimization problem, the *area minimization* problem was shown to be NP-hard for LUTs of size four and greater [22, 31]. Thus, heuristics are necessary to solve the area minimization problem. Early work considered the decomposition of circuits into a set of trees which were then mapped for area [23, 28, 37]. The area minimization problem for trees is much simpler and can be solved optimally using dynamic programming. However, real circuits are rarely structured as trees and tree decomposition prevents much of the optimization that can take place across tree boundaries. In a *duplication-free* mapping, each gate in the initial circuit is covered by a single LUT in the mapped circuit. The area minimization problem in duplication-free mapping can be solved optimally by decomposing the circuit into a set of maximum fanout free cones (MFFCs) which are then mapped for area [10]. Although the area minimal duplication-free mapping is significantly smaller than the area minimal tree mapping, the controlled use of duplication can lead to further area savings. In [17], heuristics are used to mark a set of gates as *duplicable*. Then area optimization is considered within an extended fanout free cone (EFFC) where an EFFC is an MFFC that has been extended to include duplicable gates.

Area minimization heuristics are typically used in concert with techniques that produce depth-optimal mapping solutions. In FlowMAP-r [10], following a depth-optimal mapping procedure, noncritical parts of the circuit are remapped with a duplication-free mapper. In CutMAP [13], two strategies are used for selecting the gates covered by a LUT. Critical parts of the circuit are mapped using a depth-minimizing strategy and noncritical parts are mapped using a cost-minimizing strategy that encourages LUT sharing.

Although the previously mentioned heuristics often produce satisfactory solutions, it is unknown how close their final solutions are to the optimal. Knowing this would give a theoretical bound that area minimization heuristics could compare against. This work tackles this issue by introducing a new tool that measures how far a technology mapped circuit is from the area-optimal solution.

1.2   Purpose and Scope

This thesis shows a new application for Boolean satisfiability and demonstrates how it can be successfully used in the technology mapping phase of the FPGA CAD flow. The purpose of this work is to find an automated and robust approach to evaluate the efficiency of new FPGA architectures. Also, an assessment of state-of-the-art FPGA technology mapping algorithms in terms of *area*-optimality is done and new technology mapping techniques that are more general than the existing approaches are developed.

Chapter 2 formalizes the previously mentioned problems and demonstrates how Boolean satisfiability is applied to them. Chapter 3 describes the PLB evaluation tool and illustrates how it can be effectively used to evaluate various PLB architectures. Chapter 4 explains how to extend this tool to create a robust technology mapper, and compares it to other current LUT based technology mappers. Chapter 5 presents a study on area-optimality and describes the resynthesis technique used to perform this study. It demonstrates the power of the technique by resynthesizing various technology mapped circuits and showing the area gains achieved using this technique. Finally, Chapter 6 summarizes the results and suggests some future work.

Chapter 2

Boolean Satisfiability Applied to FPGA Synthesis

2.1 Introduction to Boolean Satisfiability and Quantified Boolean Formulae

In 1971, Steve Cook introduced Boolean satisfiability as the first problem classified as NP-Complete [18: ch.34]. Given that P class problems have a polynomial run time, it is generally thought that NP-Complete class problems are harder than P class problems. This statement, although not formally proven, is often described by $NP - C \neq P$ [18: ch.34]. As a result, efficient heuristics must be applied to reduce the problem space and time requirements for solving NP-Complete problems. Research in the area of Boolean satisfiability is an example of this where heuristics have demonstrated performance improvements on a range of NP-complete problems in the order of $1000\times$ in comparison to brute force methods.

Given a Boolean formula $F(x_1, x_2, ..., x_n)$, Boolean satisfiability (SAT) asks if there is an assignment to the variables, $x_1, x_2, ..., x_n$, such that F evaluates to 1. If such an assignment exists, F is said to be *satisfiable*, otherwise, it is *unsatisfiable*. A SAT solver serves to answer the Boolean satisfiability problem.

For practical purposes, modern day SAT solvers work on Boolean formulae in Conjunctive Normal Form (CNF). Boolean formulae in CNF consist of a conjunction of clauses. A clause

$$\underbrace{(\overline{A} + B + C)}_{clause} \cdot (A + B + \underbrace{\overline{C}}_{literal})$$

(2.1)

Figure 2.1: A Boolean formula in Conjunctive Normal Form.

Logic Operation	Symbolic Representation	CNF
De Morgan's Law	$\overline{(A + B)}$	$(\overline{A} \cdot \overline{B})$
	$\overline{(A \cdot B)}$	$(\overline{A} + \overline{B})$
Implication	$(A \longrightarrow B)$	$(\overline{A} + B)$
Equivalence	$(A \longleftrightarrow B)$	$(\overline{A} + B) \cdot (A + \overline{B})$

Table 2.1: Conversion rules for CNF Construction.

is a disjunction (logical OR) of literals and a literal is any Boolean variable, $x \in \{0, 1\}$, or its complement. An example Boolean expression in CNF is shown in Figure 2.1. Any Boolean formula can be converted to CNF using basic Boolean algebraic techniques such as those shown in Table 2.1.

In CNF, the problem of SAT can be rephrased to the following: Given a Boolean formula, $F(x_1, x_2, ..., x_n)$, in Conjunctive Normal Form (CNF), seek an assignment to the variables, $x_1, x_2, .., x_n$, such that each clause has at least one literal evaluating to 1. Interestingly, when dealing with CNF Boolean formulae whose clauses all have less than three literals, SAT is a polynomial problem [18: ch.34]. The moment any clause has three or more literals, the problem becomes NP-Complete [18: ch.34].

2.2 Solving the Boolean Satisfiability Problem

One of the earliest works on SAT was done in 1960 by Davis and Putnam in [21]. This was refined a few years later by Davis, Logemann, and Loveland in [20] to create the DPLL algorithm. The DPLL algorithm was originally applied to *theorem provers* which attempt to verify a set of propositional statements. The DPLL algorithm is based upon the *splitting rule* which can be understood in terms of Shannon's Decomposition [43], as shown in Definitions 2.2.1 and 2.2.2.

```
 1  sat_solve(F, V, A)
 2  begin
 4      if (F(A) ≡ 1)
 5          return satisfiable
 6      if (F(A) ≡ 0)
 7          return unsatisfiable
 9      // select a variable and assign it to 0
10      // to check if F(0,...) is satisfiable
11      select a variable p ∈ V
12      assign V ← (V − p)
13      assign p ← 0
14      assign A ← (A ⋃ p)
15      if (sat_solve(F, V, A) ≡ satisfiable)
16          return satisfiable
17      // 0 assignment failed, assign it to 1
18      // to check if F(1,...) is satisfiable
19      assign A ← (A − p)
20      assign p ← 1
21      assign A ← (A ⋃ p)
22      if (sat_solve(F, V, A) ≡ satisfiable)
23          return satisfiable
24      // Formula is not satisfiable under the current assignment.
25      unassign p
26      assign A ← (A − p)
27      assign V ← (V ⋃ p)
28      return unsatisfiable
29  end
```
Algorithm 2.1: A recursive algorithm to solve SAT.

Definition 2.2.1 *Shannon's Decomposition:*

$$F(x_0, x_1, ..., x_n) = \overline{x_0} \cdot F(0, x_1, ..., x_n) + x_0 \cdot F(1, x_1, ..., x_n) \qquad (2.2)$$

Definition 2.2.2 *Splitting Rule:* $F(x_0, x_1, ..., x_n)$ *is satisfiable if and only if* $F(0, x_1, ..., x_n)$ *is satisfiable or* $F(1, x_1, ..., x_n)$ *is satisfiable.*

The splitting rule naturally defines a simple recursive algorithm for SAT shown in Algorithm 2.1. Algorithm sat_solve accepts three parameters: the function F, a set of variables V that define F, and a set of assignments A to the variables in V. First sat_solve checks if the current assignment leads to a satisfiable or unsatisfiable solution. If F is in an indeterminate state with respect to the current set of assignments, sat_solve selects an unassigned variable (line 11 referred to as a *free* variable) and assigns it the value 0 (line 13). sat_solve then recursively

checks if the remainder of the formula is satisfiable; if so, it returns satisfiable. If not, it toggles the last variable assignment to 1 (line 20) and repeats the recursive check. If both assignments lead to an unsatisfiable solution, sat_solve returns unsatisfiable.

Algorithm 2.1 is horribly inefficient; however, it is the core of all modern day SAT solvers, which have come a long way since the first DPLL algorithm. Some popular ones include Chaff, Grasp, and SATO [33, 34, 53]. The reason for their success stems from heuristics which can drastically reduce the search space of the SAT solver. These heuristics include, but are not limited to, Boolean Constant Propagation, conflict-driven learning, and non-chronological backtracking.

2.3 Heuristics To Solve the Boolean Satisfiability Problem

2.3.1 Boolean Constant Propagation

Boolean Constant Propagation (BCP) reduces the search space by ignoring decisions that will create an obvious unsatisfiable state. BCP works by taking advantage of *unit clauses* to force variable assignments in an *implication* process. A unit clause is a clause with only one free literal where all other literals in the clause evaluate to 0. Thus to satisfy a unit clause, the implication process forces the free literal to evaluate to 1. For example, given the assignment shown in Figure 2.2, x_4 must be assigned to 0 to satisfy the expression.

$$(\overline{x_1} + x_2) \cdot \underbrace{(\overline{x_2} + x_3 + \overline{x_4})}_{\text{unit clause}} \tag{2.3}$$

Figure 2.2: An example of a unit clause, given that $x_1 x_2 x_3 = 110$ and x_4 is free.

The BCP process can be illustrated through an *implication graph*, which is a directed acyclic graph where nodes represent variable assignments and directed edges represent implications due to BCP. At each node, the notation $x_i = b \ @ \ l$ implies x_i is assigned at time l to value b. For example, in Figure 2.3b variables x_1, x_4, and x_8 are assigned at time 0, 2, and 4 respectively. The variables assigned at time 1, 3, and 5 are not shown for simplicity. Each implication edge in Figure 2.3b is labeled to reference the clause subscripts shown in Figure 2.3a. The

referenced clause indicates the cause of the implication. For example, the edges labeled 1 imply

an implication occurred due to clause 1 ($(\overline{x_1} + x_4 + x_2)_1$) and variable assignments $x_1 = 1$ @ 0

and $x_4 = 0$ @ 2, with the resulting assignment, $x_2 = 1$ @ 6. An interesting feature of BCP is

that they may form a chain of implications. For example, referring back to Figure 2.3b, when

clause 1 becomes a unit clause, x_2 is forced to 1, which in turn causes two more implications

due to clause 2 and 4. This chain of assignments are directly related through implication and it

is impossible to satisfy the expression without following the implications to completion where

no more unit clauses remain. Thus, variables involved in a chained implication process are

grouped and all share the same time label. This is shown in Figure 2.3b where several nodes

are given a time value of 6. It is possible for BCP to create inconsistent implications. The

node labeled $x_7 = X$ @ 6 is an example of this and shows the two inconsistent implications

leading to that node. Inconsistent implications are known as conflicts. When a conflict occurs,

previous assignments must be undone until the conflict is removed. This process, referred as

backtracking, is very costly and heuristics are used such as conflict-driven analysis to minimize

the time spent backtracking.

2.3.2 Conflict Analysis

Conflict-driven learning is a process where *conflict clauses* are added to the Boolean formula to

remove solution space regions that always lead to an unsatisfiable solution. A detailed algorithm

of this can be found in [33] and is not discussed here, but referring back to Figure 2.3, a quick

explanation can be presented through an example. As stated previously, node $x_7 = X$ @ 6 is in

conflict due to the conflicting implication edges 5 and 6, thus backtracking must occur to undo

this conflict. However, without learning it is possible that this conflict arrangement may occur

again; thus, it is beneficial to add information to the Boolean formula to prevent this. Closer

inspection of the implication graph in Figure 2.3 shows that the conflict originates at edges 1

and 4 due to assignments $x_1 = 1$, $x_4 = 0$, and $x_8 = 1$. Further analysis of Equation 2.4 shows

that assignment $x_1 = 1$, $x_4 = 0$, and $x_8 = 1$ will always cause a conflict. Thus, to prevent this

in the future, a learned clause is added which is shown in Equation 2.5. The learned clause is

a redundant clause and does not change the meaning of the original Boolean formulae, but it

$$F = ...\cdot (x_1 + x_2)_0 \cdot (\overline{x_1} + x_4 + x_2)_1 \cdot (\overline{x_2} + x_6)_2 \cdot (\overline{x_4} + x_2 + \overline{x_1})_3 \cdot$$
$$(\overline{x_2} + x_5 + \overline{x_8})_4 \cdot (x_4 + \overline{x_6} + \overline{x_7})_5 \cdot (x_4 + \overline{x_5} + x_7)_6 \cdot ...$$

(2.4)

(a) Boolean formula to illustrate BCP and conflicts.
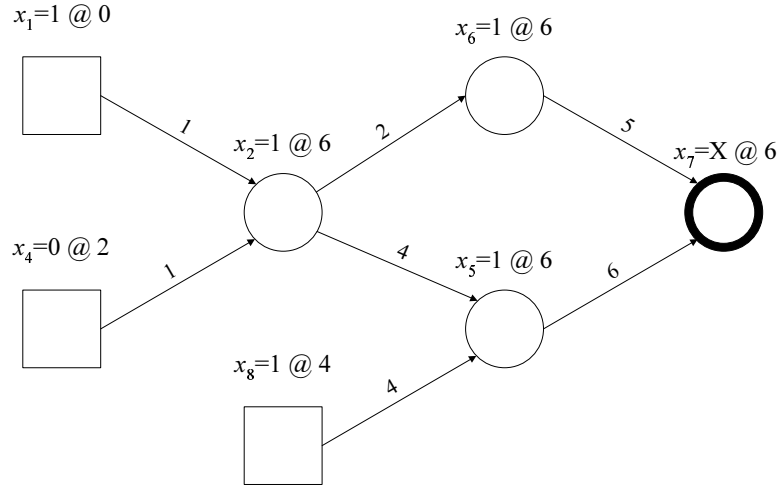
(b) Resulting implication graph from (a).

$$F = ... \cdot (\overline{x_1} + x_4 + \overline{x_8})$$

(2.5)

(c) Conflict clause added due to conflict in (b).

Figure 2.3: A conflict-driven analysis implication graph.

makes it impossible for the assignment $x_1 = 1$, $x_4 = 0$, and $x_8 = 1$ to occur again.

2.3.3   Non-Chronological Backtracking

The process of reversing the most recent assignments due to a conflict is called chronological backtracking (i.e. in-order backtracking). However, often several assignments can be reversed out of order if the source of the conflict can be identified. This is known as non-chronological backtracking. Like conflict analysis, non-chronological backtracking uses conflict information to determine the original source of the conflict and reverse all the decisions to that source. Continuing with the example in Figure 2.3, the original source of the conflict was due to node $x_8 = 1$ @ 4. Thus, the backtracking process should jump to decision level 4 to fix the conflict. This is depicted in the *decision tree* shown in Figure 2.4b which contrasts with chronological backtracking shown in Figure 2.4a. A decision tree represents the variable assignment process

where each branch represents an assignment decision and each leaf represents a satisfiable or unsatisfiable solution. In the case of unsatisfiable solutions, the decision tree backtracks up the tree to remove the conflict.

(a) Chronological backtracking.   (b) Non-chronological backtracking.
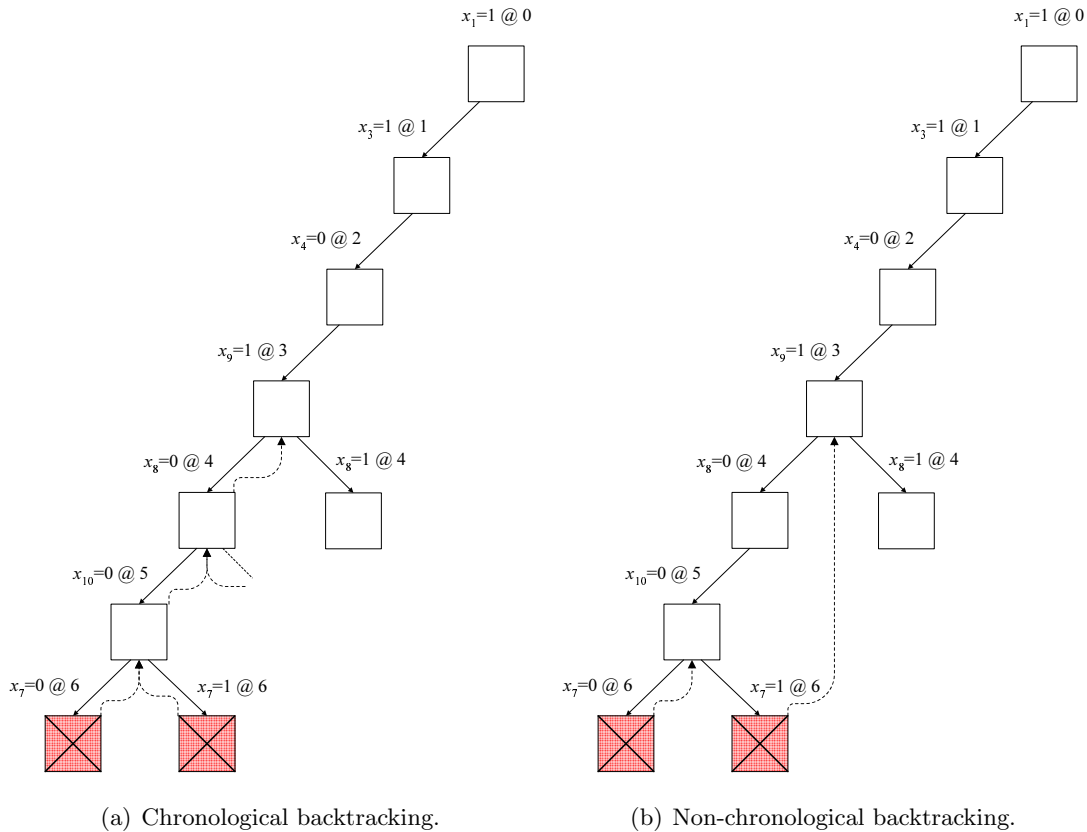
Figure 2.4: Backtracking due to a conflict in Figure 2.3.

2.4   Quantified Boolean Satisfiability (QSAT)

Often, there is the need to explore multiple solutions to a SAT expression. For example, one may be interested in the "best" solution of a set of satisfiable assignments to a single SAT expression, where "best" is defined by the given problem. Finding multiple SAT solutions can formally expressed as a quantified Boolean formula and solved using Quantified Boolean Satisfiability. Quantified Boolean Satisfiability is similar to SAT: seek an assignment to a Boolean formula such that it evaluates to 1. However, unlike SAT, it works on quantified Boolean formulae where there may exist existential or universal quantifiers. This work will refer to both Quantified Boolean Satisfiability and quantified Boolean formula both as "QBF" when its

$$(\overline{x_1} + \overline{x_2}) \cdot (x_2 + \overline{x_3} + \overline{x_4}) \cdot (x_3 + x_4 + x_5) \cdot (x_3 + \overline{x_4} + x_5) \tag{2.6}$$

$$x_1 = 0, x_2 = 1, x_3 = 0, x_4 = 0, x_5 = 1$$

(a) Simple Boolean expression and satisfying assignment.

$$\exists x_1 x_2 x_3 x_4 x_5 (\overline{x_1} + \overline{x_2}) \cdot (x_2 + \overline{x_3} + \overline{x_4}) \cdot (x_3 + x_4 + x_5) \cdot (x_3 + \overline{x_4} + x_5)) \tag{2.7}$$

(b) Same Boolean expression as in (a), but with existential quantifier shown.

$$\exists x_1 x_2 \forall x_3 x_4 \exists x_5 (\overline{x_1} + \overline{x_2}) \cdot (x_2 + \overline{x_3} + \overline{x_4}) \cdot (x_3 + x_4 + x_5) \cdot (x_3 + \overline{x_4} + x_5) \tag{2.8}$$

$$x_1 x_2 = 01, x_3 x_4 = \{00\ x_5 = 1\}, x_3 x_4 = \{01\ x_5 = 1\}, x_3 x_4 = \{10\ x_5 = 0\}, x_3 x_4 = \{00\ x_5 = 0\}$$

(c) Quantified Boolean expression with universal quantifiers added.

Figure 2.5: Adding quantifiers to a Boolean expression to form a QBF.

meaning is obvious within the context. QBF fall under the class of problems known as P-Space Complete [46] which are thought to be harder than NP-Complete problems. To understand the complexity of QBF, consider Figure 2.5. Equation 2.6 shows a simple Boolean expression and a possible satisfying assignment. Equation 2.6 is actually equivalent to Equation 2.7, but in Equation 2.7 the existential quantifiers are shown explicitly. SAT implicitly asks if there exists a single assignment to all of its variables that satisfies every clause. This implies that SAT is a subset of QBF where SAT expressions can only include existential quantifiers. Following Equation 2.7 is Equation 2.8 which is the same expression, but with universal quantifiers added. Universal quantifiers changes the meaning of the Boolean expressions since it asks if all possible assignments to the universally quantified variables can lead to a satisfiable solution. The associated satisfying assignment for Equation 2.8 is much more elaborate due to the universally quantified variables. Furthermore, the innermost existential variable, x_5, final assignment has a new instance for each universal variable assignment. Finding such an assignment is much harder than finding a single assignment as in Equation 2.6.

Each quantifier forms a *quantification set* of variables and each variable has an associated *quantification level*. The quantification levels are ordered such that variables in the outermost quantification set is given a level of 1 and so on. For example, Equation 2.8 has three quantification levels.

Quantified Boolean Satisfiability is a fairly new area and there has been marginal success in developing QBF solvers [24, 25, 29, 35, 36, 54]. The heuristics used in SAT solvers still apply to QBF solvers; however, a variable cannot be assigned until all variables with a quantification level lower than it are assigned. For example, in Equation 2.8, x_5 cannot be assigned until x_1, x_2, x_3, and x_4 have been assigned. Furthermore, unlike SAT, QBF must ensure that all possible assignments of universally quantified variables are satisfiable. This is most easily understood through an example. Figure 2.6a illustrates a satisfiable decision tree for Equation 2.9. Note that all the decision levels for the variables obey their quantification level. Although Figure 2.6a shows one possible satisfying assignment, the problem is not yet satisfied due to the universally quantified variables. Thus, the process must backtrack to lowest universally quantified decision and check if its complemented assignment may also lead to a satisfiable solution shown in 2.6b. This process must occur for all universally quantified variables.

The previous example can be formalized into the following high level QBF algorithm shown in Algorithm 2.2. This looks very similar to the recursive algorithm for SAT; however, when selecting free variables for assignment, quantification levels must be obeyed (line 12). Also, the algorithm does not return satisfiable until all assignments of universally quantified variables have been tested (lines 23 to 33).

Although QBF solvers have shown initial promising results, it is often still faster to solve a QBF by removing the universal quantifiers and converting it to a SAT problem [47]. Removing the universal quantifiers eliminates the need to backtrack on universally quantified variables to find multiple SAT instances as in Figure 2.6, thus saving time; however, in doing so, the number of clauses and variables in the Boolean formula increases substantially. To describe this, some notation is required. Given a QBF $F = Q_1 x_1 Q_2 x_2 ... Q_n x_n f(x_1, x_2, ..., x_n)$ where $Q_i \in \{\exists, \forall\}$, $f(x_1, x_2, ..., x_n)[x_i/f_i], i \in [1..n]$, represents the Boolean formula f where all instances of the variable x_i has been replaced by $f_i \in \{0, 1, variable, expression\}$. To remove the universal quantifiers in a QBF F, its proposition, f, is replicated and each replicated proposition replaces the universally quantified variables with one possible assignment assignment to the universal variables. The replication ends when all possible assignments to the universal variables have been assigned. These replicated formulae are then conjoined with the logical `AND` operator

$$\exists x_1 x_2 x_3 \forall x_4 x_5 \exists x_6 x_7 \ (\overline{x_1} + x_2 + x_4)(x_3 + \overline{x_2} + x_6)(x_7 + x_5)(\overline{x_6} + x_3)(\overline{x_7} + \overline{x_5}) \qquad (2.9)$$

(a) QBF used to illustrate backtracking on satisfying assignments.
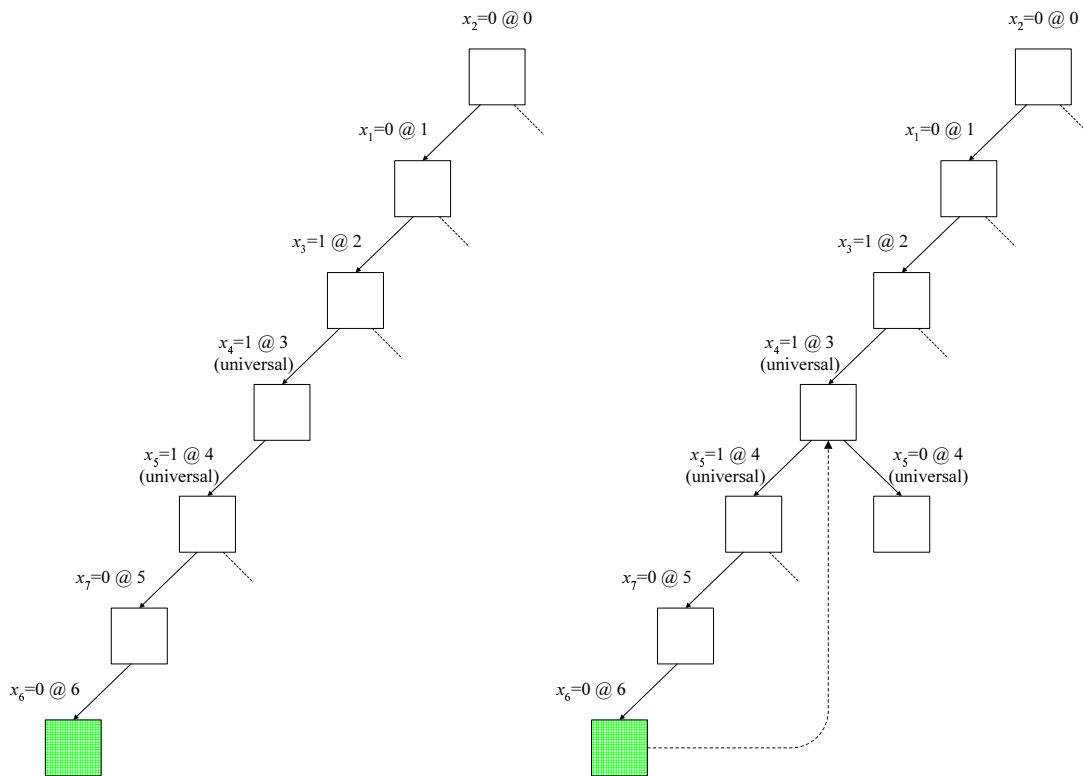
(b) A possible assignment that satisfies all clauses.   (c) Backtracking due to universal quantifiers.

Figure 2.6: A quantified Boolean satisfiability example.

```
 1  qbf_solve(F, V, A)
 2  begin
 3     if (no unassigned variables in V)
 4        return satisfiable
 6     if (F(A) ≡ 1)
 7        return satisfiable
 8     if (F(A) ≡ 0)
 9        return unsatisfiable
11     Q_set = Set of Variables in Current Quantification Level
12     select an unassigned variable p ∈ Q_set
13     assign V ← (V − p)
14     assign p ← 0
15     assign A ← (A ⋃ p)
16     // keep track of the initial result
17     assign result ← UNSAT
18     if (qbf_solve(F) ≡ satisfiable)
19        assign result ← SAT
22     // if p is a universally quantified variable
23     if ((Q_set ∈ {∀})  AND  (result ≡ UNSAT))
24        return unsatisfiable
27     // if p is an existentially quantified variable
28     if ((Q_set ∈ {∃})  AND  (result ≡ SAT))
29        return satisfiable
31     // Try the opposite value.
32     assign A ← (A − p)
33     assign p ← 1
34     assign A ← (A ⋃ p)
35     if (qbf_solve(F) ≡ satisfiable)
36        return satisfiable
37     // Formula is not satisfiable under the current assignment.
38     unassign p
39     assign A ← (A − p)
40     assign V ← (V ⋃ p)
41     return unsatisfiable
42  end
```

Algorithm 2.2: A high-level algorithm to solve Quantified Boolean Satisfiability.

to form a Boolean function that can be solved with SAT. For example, consider the QBF in

Equation 2.10. With the technique described previously, it can be expressed as a SAT problem

with no explicit quantifiers as shown in Equation 2.11.

$$F = \exists x_1 \forall x_2 x_3 \exists x_4 x_5 \ f$$

where

$$f = (x_1 + x_2) \cdot (x_2 + x_3 + x_4) \cdot (\overline{x_2} + x_5 + \overline{x_4}) \tag{2.10}$$

$$
\begin{aligned}
F &= f[x_2/0, x_3/0, x_4/x_6, x_5/x_7] \cdot f[x_2/0, x_3/1, x_4/x_8, x_5/x_9] \cdot \\
&\quad f[x_2/1, x_3/0, x_4/x_{10}, x_5/x_{11}] \cdot f[x_2/1, x_3/1, x_4/x_{12}, x_5/x_{13}] \\
&= (x_1 + 0) \cdot (0 + 0 + x_6) \cdot (\overline{0} + x_7 + \overline{x_6}) \cdot \\
&\quad (x_1 + 0) \cdot (0 + 1 + x_8) \cdot (\overline{0} + x_9 + \overline{x_8}) \cdot \\
&\quad (x_1 + 1) \cdot (1 + 0 + x_{10}) \cdot (\overline{1} + x_{11} + \overline{x_{10}}) \cdot \\
&\quad (x_1 + 1) \cdot (1 + 1 + x_{12}) \cdot (\overline{1} + x_{13} + \overline{x_{12}}) \\
&= (x_1) \cdot (x_6) \cdot \\
&\quad (x_1) \cdot \\
&\quad (x_{11} + \overline{x_{10}}) \cdot \\
&\quad (x_{13} + \overline{x_{12}}) \\
&= (x_1) \cdot (x_6) \cdot (x_{11} + \overline{x_{10}}) \cdot (x_{13} + \overline{x_{12}})
\end{aligned}
\tag{2.11}
$$

Note that in the previous example, the innermost existential variables x_4, x_5 are replaced with unique variables in each replicated formula f to preserve the levels of the existential quantifiers. Although the final expression is fairly small due to simplification, this example illustrates how removing quantifiers can potentially increase the size of the Boolean formula; thus it is generally only used for QBFs with three or less quantification levels.

2.5   Converting Problem to Boolean Satisfiability

There are three different FPGA areas discussed in this dissertation: architecture evaluation, technology mapping, and resynthesis; however, they all share a common core related to Boolean function legality checking phrased as the following problem:

Problem 2.5.1 *Given an n-variable Boolean function, $F(x_1, x_2, ..., x_n)$, does there exists a programmable configuration to a circuit such that the output of the circuit will equal $F(x_1, x_2, ..., x_n)$ for all inputs?*

Assuming that the programmable circuit can be represented as a Boolean function $G = G(x_1..x_n, L_1..L_m, z_1..z_o)$ where x_i, L_j, z_k, G represent the input signals, configuration bits, in-

termediate circuit signals, and output function of the circuit respectively, Problem 2.5.1 can be represented formally as a QBF as follows.

$$\exists L_1...L_m \forall x_1...x_n \exists z_1...z_o (G \equiv F) \tag{2.12}$$

Given the process to remove universal quantifiers described in Section 2.1, Equation 2.12 can be solved with SAT where a satisfying assignment implies the function can be realized in the configurable circuit.

$$F_{AND} = (A + \overline{Z}) \cdot (B + \overline{Z}) \cdot (\overline{A} + \overline{B} + Z)$$
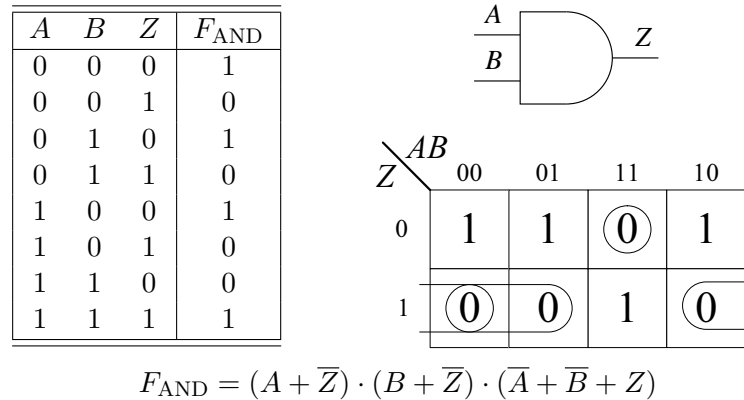
Figure 2.7: A characteristic equation derivation for 2-input `AND` gate.

To translate Problem 2.5.1 into a SAT problem, first the proposition $(G \equiv F)$ in Equation 2.12 must be represented as a Boolean formula. This can be done using a well known derivation technique that converts logic circuits into a *characteristic function* in CNF [30]. This CNF representation contain variables representing the primary input, primary output, and intermediate circuit signals and evaluates to 1 if these signals are consistent. For example, consider the `AND` gate shown in Figure 2.7. The table to the left gives the truth table for the `AND` gate characteristic function where the onset contains all valid input-output relations of an `AND` gate. This can be converted to CNF using any standard minimization technique, such as a Karnaugh map as shown. Table 2.5 lists several other common gates and digital functions with their associated characteristic function [45: ch.2]. Notice `FULL-ADD` has two outputs to the circuit (i.e. s, c_{out}). The technique shown in Figure 2.7 is directly applicable to multiple output circuits. One simply adds all output variables to the characteristic function truth table where its onset still defines valid input-output vectors.

Deriving CNF functions directly from the circuit input-output relation is only practical for primitive gates and logic blocks where the number of inputs and outputs is small. Fortunately,

Gate	Function	CNF formula
AND	$y = x_1 \cdot x_2 \cdots x_n$	$\left(\prod_{i=1}^{n}(x_i + \overline{y})\right) \cdot \left(\sum_{i=1}^{n}\overline{x_i} + y\right)$
NAND	$y = \overline{x_1 \cdot x_2 \cdots x_n}$	$\left(\prod_{i=1}^{n}(x_i + y)\right) \cdot \left(\sum_{i=1}^{n}\overline{x_i} + \overline{y}\right)$
OR	$y = x_1 + x_2 + \cdots + x_n$	$\left(\prod_{i=1}^{n}(\overline{x_i} + y)\right) \cdot \left(\sum_{i=1}^{n}x_i + \overline{y}\right)$
NOR	$y = \overline{x_1 + x_2 + \cdots + x_n}$	$\left(\prod_{i=1}^{n}(\overline{x_i} + \overline{y})\right) \cdot \left(\sum_{i=1}^{n}x_i + y\right)$
XOR	$y = x_1 \oplus x_2$	$(\overline{x_1} + \overline{x_2} + \overline{y}) \cdot (x_1 + x_2 + \overline{y}) \cdot$ $(\overline{x_1} + x_2 + y) \cdot (x_1 + \overline{x_2} + y)$
NXOR	$y = \overline{x_1 \oplus x_2}$	$(\overline{x_1} + \overline{x_2} + y) \cdot (x_1 + x_2 + y) \cdot$ $(\overline{x_1} + x_2 + \overline{y}) \cdot (x_1 + \overline{x_2} + \overline{y})$
BUFFER	$y = x$	$(x + \overline{y}) \cdot (\overline{x} + y)$
NOT	$y = \overline{x}$	$(x + y) \cdot (\overline{x} + \overline{y})$
MUX 2:1	$y = (s_0 \leftrightarrow 1)\,?\,x_1 : x_0$	$(s_0 + \overline{x_0} + y) \cdot (s_0 + x_0 + \overline{y}) \cdot$ $(\overline{s_0} + x_1 + \overline{y}) \cdot (\overline{s_0} + \overline{x_1} + y)$
MUX 4:1	$y = (s_1 \leftrightarrow 1)\,?$ $[(s_0 \leftrightarrow 1)\,?\,x_3 : x_2]:$ $[(s_0 \leftrightarrow 1)\,?\,x_1 : x_0]$	$(s_0 + s_1 + \overline{x_0} + y) \cdot (s_0 + s_1 + x_0 + \overline{y}) \cdot$ $(s_0 + \overline{s_1} + \overline{x_1} + y) \cdot (s_0 + \overline{s_1} + x_1 + \overline{y}) \cdot$ $(\overline{s_0} + s_1 + \overline{x_2} + y) \cdot (\overline{s_0} + s_1 + x_2 + \overline{y}) \cdot$ $(\overline{s_0} + \overline{s_1} + \overline{x_3} + y) \cdot (\overline{s_0} + \overline{s_1} + x_3 + \overline{y})$
FULL-ADD	$s = a \oplus b \oplus c_{in}$ $c_{out} = (a \oplus b) \cdot c_{in} + ab$	$(a + b + \overline{c_{out}}) \cdot (\overline{a} + \overline{b} + c_{out}) \cdot$ $(a + c_{in} + \overline{c_{out}}) \cdot (\overline{a} + \overline{c_{in}} + c_{out}) \cdot$ $(b + c_{in} + \overline{c_{out}}) \cdot (\overline{b} + \overline{c_{in}} + c_{out}) \cdot$ $(s + \overline{a} + c_{out}) \cdot (\overline{s} + a + \overline{c_{out}}) \cdot$ $(s + \overline{b} + c_{out}) \cdot (\overline{s} + b + \overline{c_{out}}) \cdot$ $(s + \overline{c_{in}} + c_{out}) \cdot (\overline{s} + c_{in} + \overline{c_{out}}) \cdot$ $(s + \overline{a} + \overline{b} + \overline{c_{in}}) \cdot (\overline{s} + a + b + c_{in})$

Table 2.2: Characteristic functions for basic logic elements [45: ch.2].
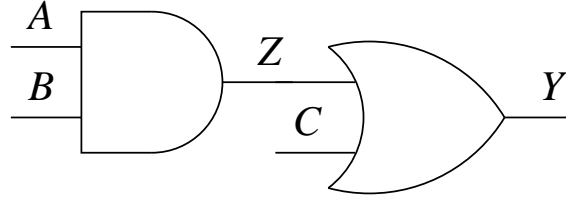
(a) Two cascaded gates.

$$F = \underbrace{(A + \overline{Z}) \cdot (B + \overline{Z}) \cdot (\overline{A} + \overline{B} + Z)}_{\texttt{AND}} \cdot \underbrace{(\overline{Z} + Y) \cdot (\overline{C} + Y) \cdot (Z + C + \overline{Y})}_{\texttt{OR}}$$

(b) Characteristic equation for (a).

Figure 2.8: A cascaded gate characteristic function.

characteristic equations for larger circuits can be derived iteratively from the conjunction of its subcircuit characteristic functions. For example, Figure 2.8a shows two cascaded gates. Notice the wire connecting the `AND` gate to the `OR` gate is labeled with variable Z for CNF construction. The characteristic function of the cascaded circuit is simply the conjunction of the `AND` and `OR` gate characteristic functions with variable Z as the logical link between the two functions. The characteristic equation in Figure 2.8b evaluates to 1 if all wire signals are consistent. This includes the primary inputs and outputs as in Figure 2.7, plus any intermediate wire signals (i.e. Z). This concept can be extended to much larger circuits such as a programmable logic circuit to create a CNF function, Ψ, dependent on variables $x_1, ..., x_n,\ L_1, ..., L_m,\ z_1, ..., z_o$, and G which represent the inputs, programmable bits, intermediate wires, and output of the circuit respectively. Thus, the proposition $(G \equiv F)$ can be presented as Equation 2.13, or equivalently as Equation 2.14 when its quantifiers are removed.

$$\Psi = \exists L_1...L_m \forall x_1...x_n \exists z_1...z_o \psi(x_1, ..., x_n, L_1, ..., L_m, z_1, ..., z_o, G)\,[G/F(x_1, ..., x_n)]$$
where
$F(\boldsymbol{X})$ is the function being fit and $\boldsymbol{X} = x_1, ..., x_n$.
$$(2.13)$$

$$\Psi = \psi_0[x_1/0, x_2/0, ..., x_n/0, z_1/z_{o+1}, ..., z_o/z_{2o}, G/F_0]\cdot$$
$$\psi_1[x_1/0, x_2/0, ..., x_n/1, z_1/z_{2o+1}, ..., z_o/z_{3o}, G/F_1]\cdot$$
$$...$$
$$\psi_{2^n-1}[x_1/1, x_2/1, ..., x_n/1, z_1/z_{(2^n-1)o+1}, ..., z_o/z_{2^n o}, G/F_{2^n-1}]$$
where
$F_i = F(\boldsymbol{X}_i),\ \boldsymbol{X}_i = x_1, ..., x_n = i$ and $\psi_j = \psi(x_1, ..., x_n, L_1, ..., L_m, z_1, ..., z_o, G)$
$$(2.14)$$

In order to give better understanding to the previously described procedure, an example is given. Assume that the function listed in Figure 2.9 needs to be implemented in the adjacent programmable circuit. The circuit in Figure 2.9 consists of a 2-LUT which feeds into a 2-input `AND` gate. In order to test if Figure 2.9 can implement the given function, the following steps are taken.

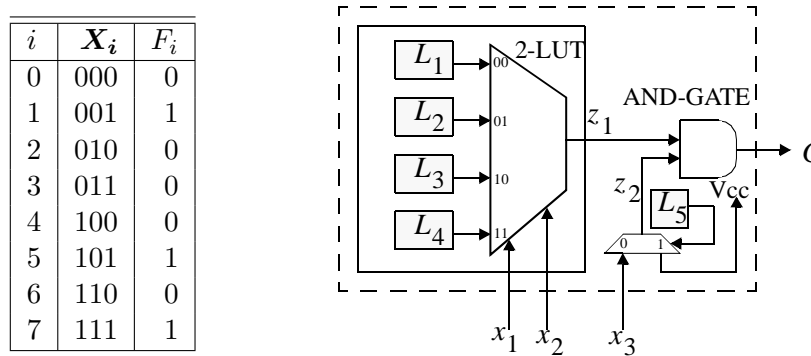| i	X_i	F_i
0	000	0
1	001	1
2	010	0
3	011	0
4	100	0
5	101	1
6	110	0
7	111	1

Figure 2.9: An example programmable circuit.

Step 1: Find the characteristic functions for individual elements in programmable circuit.

$$\begin{aligned}
G_{LUT} = & (x_1 + x_2 + \overline{L_1} + z_1) \cdot (x_1 + x_2 + L_1 + \overline{z_1}) \cdot \\
& (x_1 + \overline{x_2} + \overline{L_2} + z_1) \cdot (x_1 + \overline{x_2} + L_2 + \overline{z_1}) \cdot \\
& (\overline{x_1} + x_2 + \overline{L_3} + z_1) \cdot (\overline{x_1} + x_2 + L_3 + \overline{z_1}) \cdot \\
& (\overline{x_1} + \overline{x_2} + \overline{L_4} + z_1) \cdot (\overline{x_1} + \overline{x_2} + L_4 + \overline{z_1})
\end{aligned} \tag{2.15}$$

$$G_{MUX} = (L_5 + \overline{x_3} + z_2) \cdot (L_5 + x_3 + \overline{z_2}) \cdot (\overline{L_5} + z_2) \tag{2.16}$$

$$G_{AND} = (z_1 + \overline{G}) \cdot (z_2 + \overline{G}) \cdot (\overline{z_1} + \overline{z_2} + G) \tag{2.17}$$

Step 2: Derive the programmable circuit characteristic function from the basic characteristic functions (Equations 2.15, 2.16, and 2.17).

$$G = G_{LUT} \cdot G_{MUX} \cdot G_{AND} \tag{2.18}$$

Step 3: Replicate Equation 2.18 to remove the universally quantifiers on the input variables in $\mathbf{X}$. This formulates G_{Total} where a satisfiable assignment to G_{Total} implies F can be realized in the programmable circuit.

$$\begin{aligned}
G_{Total} = & G[\mathbf{X}/\mathbf{X_0}, G/F_0, z_1/z_3, z_2/z_4] \cdot G[\mathbf{X}/\mathbf{X_1}, G/F_1, z_1/z_5, z_2/z_6] \cdot \\
& G[\mathbf{X}/\mathbf{X_2}, G/F_2, z_1/z_7, z_2/z_8] \cdot G[\mathbf{X}/\mathbf{X_3}, G/F_3, z_1/z_9, z_2/z_{10}] \cdot \\
& G[\mathbf{X}/\mathbf{X_4}, G/F_4, z_1/z_{11}, z_2/z_{12}] \cdot G[\mathbf{X}/\mathbf{X_5}, G/F_5, z_1/z_{13}, z_2/z_{14}] \cdot \\
& G[\mathbf{X}/\mathbf{X_6}, G/F_6, z_1/z_{15}, z_2/z_{16}] \cdot G[\mathbf{X}/\mathbf{X_7}, G/F_7, z_1/z_{17}, z_2/z_{18}]
\end{aligned} \tag{2.19}$$

In the previous example, the pins on the programmable circuit in Figure 2.9 are not permutable. Given the labeling convention in Figure 2.9, the function $F = (x_1 + x_2) \cdot x_3$ can be implemented; however, the function $F = (x_1 + x_3) \cdot x_2$ cannot. There is no need for restricting the labeling of the input pins in this manner because most programmable circuits are able to route signals to any input pins, such as FPGAs. In order to model this flexibility, virtual multiplexers controlled by virtual configuration bits, V_p, are added at each input pin of the programmable circuit. Going back to the circuit shown in the last example, Figure 2.10 illustrates the previous circuit with virtual multiplexers added at the input pins. Thus, if $F = (x_1 + x_3) \cdot x_2$ is to be mapped into this network then the virtual multiplexers would force x_1 and x_3 onto the first two pins of the circuit and x_2 to the third pin feeding the AND gate to generate a satisfiable solution. Adding virtual multiplexers to the previous example is straightforward and is shown in the following steps.
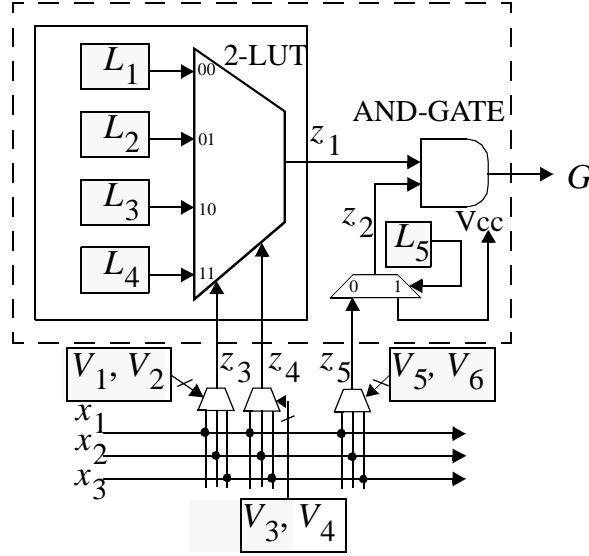
Figure 2.10: An example programmable circuit with virtual multiplexers added.

Step 1': Find the characteristic function for individual elements in programmable circuit *with virtual multiplexers added.*

$$
\begin{aligned}
G_{LUT} = \ &(z_3 + z_4 + \overline{L_1} + z_1) \cdot (z_3 + z_4 + L_1 + \overline{z_1}) \cdot \\
&(z_3 + \overline{z_4} + \overline{L_2} + z_1) \cdot (z_3 + \overline{z_4} + L_2 + \overline{z_1}) \cdot \\
&(\overline{z_3} + z_4 + \overline{L_3} + z_1) \cdot (\overline{z_3} + z_4 + L_3 + \overline{z_1}) \cdot \\
&(\overline{z_3} + \overline{z_4} + \overline{L_4} + z_1) \cdot (\overline{z_3} + \overline{z_4} + L_4 + \overline{z_1})
\end{aligned} \tag{2.20}
$$

$$
G_{MUX} = (L_5 + \overline{z_5} + z_2) \cdot (L_5 + z_5 + \overline{z_2}) \cdot (\overline{L_5} + z_2) \tag{2.21}
$$

$$
G_{AND} = (z_1 + \overline{G}) \cdot (z_2 + \overline{G}) \cdot (\overline{z_1} + \overline{z_2} + G) \tag{2.22}
$$

$$
\begin{aligned}
G_{VMUX1} = \ &(V_1 + V_2 + \overline{x_1} + z_3) \cdot (V_1 + V_2 + x_1 + \overline{z_3}) \cdot \\
&(V_1 + \overline{V_2} + \overline{x_2} + z_3) \cdot (V_1 + \overline{V_2} + x_2 + \overline{z_3}) \cdot \\
&(\overline{V_1} + V_2 + \overline{x_3} + z_3) \cdot (\overline{V_1} + V_2 + x_3 + \overline{z_3}) \cdot \\
&(\overline{V_1} + \overline{V_2})
\end{aligned} \tag{2.23}
$$

$$
\begin{aligned}
G_{VMUX2} = \ &(V_3 + V_4 + \overline{x_1} + z_4) \cdot (V_3 + V_4 + x_1 + \overline{z_4}) \cdot \\
&(V_3 + \overline{V_4} + \overline{x_2} + z_4) \cdot (V_3 + \overline{V_4} + x_2 + \overline{z_4}) \cdot \\
&(\overline{V_3} + V_4 + \overline{x_3} + z_4) \cdot (\overline{V_3} + V_4 + x_3 + \overline{z_4}) \cdot \\
&(\overline{V_3} + \overline{V_4})
\end{aligned} \tag{2.24}
$$

$$
\begin{aligned}
G_{VMUX3} = \ &(V_5 + V_6 + \overline{x_1} + z_5) \cdot (V_5 + V_6 + x_1 + \overline{z_5}) \cdot \\
&(V_5 + \overline{V_6} + \overline{x_2} + z_5) \cdot (V_5 + \overline{V_6} + x_2 + \overline{z_5}) \cdot \\
&(\overline{V_5} + V_6 + \overline{x_3} + z_5) \cdot (\overline{V_5} + V_6 + x_3 + \overline{z_5}) \cdot \\
&(\overline{V_5} + \overline{V_6})
\end{aligned} \tag{2.25}
$$

Step 2': Derive the programmable circuit characteristic function from the basic characteristic functions (Equations 2.20 to 2.25).

$$
G = G_{LUT} \cdot G_{MUX} \cdot G_{AND} \cdot G_{VMUX1} \cdot G_{VMUX2} \cdot G_{VMUX3} \tag{2.26}
$$

Step 3': Replicate equation 2.26 to remove the universally quantifiers on the input variables in $\boldsymbol{X}$. This formulates G_{Total} where a satisfiable assignment to G_{Total} implies F can be realized in the programmable circuit. *Clauses that contain only programmable configuration bits variables do not need to be replicated since this would create redundant clauses.*[1]

$$
\begin{aligned}
G_{Total} = {} & G[\boldsymbol{X}/\boldsymbol{X_0}, G/F_0, z_1/z_6, z_2/z_7, z_3/z_8, z_4/z_9, z_5/z_{10}] \cdot \\
& G[\boldsymbol{X}/\boldsymbol{X_1}, G/F_1, z_1/z_{11}, z_2/z_{12}, z_3/z_{13}, z_4/z_{14}, z_5/z_{15}] \cdot \\
& G[\boldsymbol{X}/\boldsymbol{X_2}, G/F_2, z_1/z_{16}, z_2/z_{17}, z_3/z_{18}, z_4/z_{19}, z_5/z_{20}] \cdot \\
& G[\boldsymbol{X}/\boldsymbol{X_3}, G/F_3, z_1/z_{21}, z_2/z_{22}, z_3/z_{23}, z_4/z_{24}, z_5/z_{25}] \cdot \\
& G[\boldsymbol{X}/\boldsymbol{X_4}, G/F_4, z_1/z_{26}, z_2/z_{27}, z_3/z_{28}, z_4/z_{29}, z_5/z_{30}] \cdot \\
& G[\boldsymbol{X}/\boldsymbol{X_5}, G/F_5, z_1/z_{31}, z_2/z_{32}, z_3/z_{33}, z_4/z_{34}, z_5/z_{35}] \cdot \\
& G[\boldsymbol{X}/\boldsymbol{X_6}, G/F_6, z_1/z_{36}, z_2/z_{37}, z_3/z_{38}, z_4/z_{39}, z_5/z_{40}] \cdot \\
& G[\boldsymbol{X}/\boldsymbol{X_7}, G/F_7, z_1/z_{41}, z_2/z_{42}, z_3/z_{43}, z_4/z_{44}, z_5/z_{45}]
\end{aligned}
\tag{2.27}
$$

2.6   Limitations

As the previous examples show, the number of replications of the CNF expression is exponential in relation to input size of the configurable circuit. Furthermore, the runtime is strongly dependent on the number of configuration bits in the circuit. Table 2.3 illustrates this relation. *Input Size* shows the number of inputs to the configurable circuit; *Configuration Bits* shows the number of programmable bits (virtual and real) in the circuit; *Variable Count* and *Clause Count* are the number of variables and clauses in the CNF expression respectively; and *SAT* and *UNSAT* are the runtimes of a satisfying and unsatisfying expressions respectively. The runtimes are the averages of 10 runs in seconds using the Chaff [34] SAT solver run on a Sunblade 150 with 1.5 GB of RAM. Table 2.3 shows that this technique becomes unmanageable as the circuit input size grows larger than 10. In addition to this, depending on its application, the input size restriction will vary. For example, if applied in a short compile process where the SAT solver is run millions of times, the configurable circuit may have to be restricted to as small as 5 inputs. Work in speeding up the SAT process and reducing the size of the CNF expression is necessary if this technique is to be applied to fairly large configurable circuits or if a fast runtime is required.

---

[1]In **Step 1'**, some clauses only had virtual configuration bits. This was to prevent invalid bit configurations since there are only three inputs (x_1, x_2, x_3) into the four input virtual multiplexer, thus configuration $\{V_i, V_{i+1}\} = 11$ is invalid.

Input Size	Configuration Bits	Variable Count	Clause Count	SAT	UNSAT
4	12	512	872	≈0	≈0
5	31	1376	2144	≈0	≈0
8	56	19200	33760	2	17
9	68	45568	76639	13	35
10	88	120832	217152	43	2 HOUR TIMEOUT

Table 2.3: The runtimes of the Chaff SAT solver [34] and the associated CNF expression sizes.

2.7   Summary

This chapter presents a function fitting technique using Boolean satisfiability. The power of this technique stems from its generality and is the core problem solved in this dissertation. The major limitation of this technique is the exponential relation of the solution runtime and memory with respect to the input size of the configurable circuit. However, even with this limitation, there are still a few practical areas that this technique can be applied as demonstrated in the following chapters.

Chapter 3

Evaluation of FPGA Programmable Logic Blocks

3.1   The Programmable Logic Blocks

Programmable Logic Blocks (PLB) form the basic building block of most commercial FPGAs [5, 49]. An example of a simplistic PLB is shown in Figure 3.1. It consists of a 4-input lookup table (4-LUT) that is capable of implementing and arbitrary 4 variable Boolean function. The LUT consists of a multiplexer fed by a set of static RAM bits whose select lines are controlled by the input variables of the function. Thus, to implement a 4 variable function, the 16 (2^4) SRAM bits are programmed to match the truth-table values of the function being implemented. Furthermore, the LUT output can either be sent directly to the PLB output or be registered.

Although the k-LUT is extremely flexible, adding custom logic to the PLB is beneficial such as adders or basic gates. This allows a PLB to implement a wider range of functions without the speed, area, and power costs of their programmable counterparts. However, mapping logic to PLBs is much more harder than mapping logic to k-LUTs since a k-input PLB can only implement a subset of all k-input functions. For example, consider the PLB shown on the left side of Figure 3.2. It consists of a 2-LUT feeding an OR gate. One obvious function that this PLB cannot implement is a 3-input AND function as shown on the right side of Figure 3.2.

Fortunately, realistic circuits usually contain a small subset of all k-input functions. Thus,
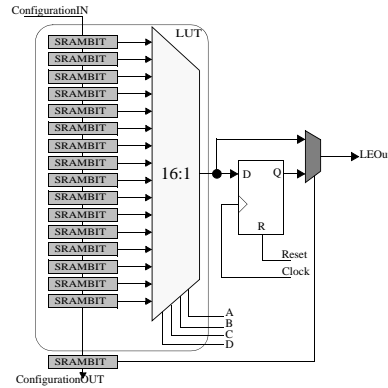
Figure 3.1: An example of a PLB.
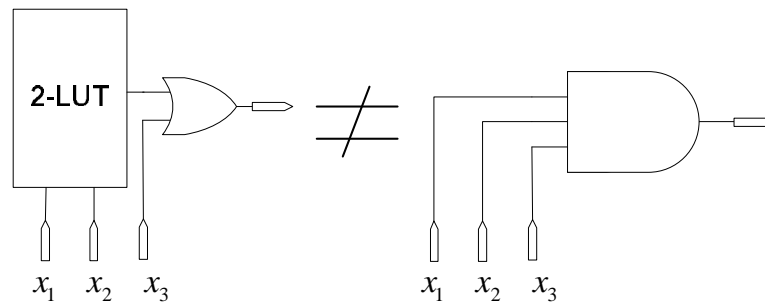
Figure 3.2: A limitation of a 3-input PLB.

so long as a PLB can capture most functions that occur in real circuits, it can efficiently implement logic circuits. The difficulty is designing PLBs with this characteristic. As far as we know, there has been no previously published work that helps address this design problem. This chapter demonstrates how to use the QBF based technique described in Chapter 2 to help in the PLB design process.

3.2 Evaluation Method

When designing new PLB architectures, the custom circuitry must add very little overhead in terms of power consumption and area. Furthermore, the custom components must complement the programmable portion of the PLB such that they will be utilized often. Utilization can be measured by extracting a set of functions and attempting to fit them into the PLB. A high fit percentage implies the custom components are useful. The set of functions should be extracted from various unrelated benchmark circuits to give a realistic sampling set. Of course, if the

PLB is only applied to a specific set of applications, the sampling set can be more restrictive to represent the application in question.

Since a circuit implements functions through cones of logic, the problem of extracting functions from a circuit is the same as extracting cones from it. In order to understand the extraction process, some terminology is needed.

The combinational portion of a Boolean circuit can be represented as a directed acyclic graph (DAG) $G = (V_G, E_G)$. A node in the graph $v \in V_G$ represents a logic gate, primary input or primary output, and a directed edge in the graph $e \in E_G$ with head, $u = head(e)$, and tail, $v = tail(e)$, represents a signal in the logic circuit that is an output of gate u and an input of gate v. The set of *input edges* for a node v, $iedge(v)$, is defined to be the set of edges with v as a tail. Similarly, the set of *output edges* for v, $oedge(v)$, is defined to be the set of edges with v as a head. A *primary input* (PI) node has no input edges and a *primary output* (PO) node has no output edges. An *internal* node has both input edges and output edges. A node v is *K-feasible* if $|inode(v)| \leq K$. If every node in a graph is K-feasible then the graph is *K-bounded*.

A *cone* of node v, C_v, is a subgraph, $g \in G$ consisting of node v and some of its predecessors such that any node $u \in C_v$ has a path to v that lies entirely in C_v. Node v is referred to as the *root* of the cone. At a cone C_v, the set of input edges, $iedge(C_v)$, is the set of edges with a tail in C_v and the set of output edges, $oedge(C_v)$, is the set of edges with v as a head. With input edges and output edges so defined, a cone can be viewed as a node, and notions that were previously defined for nodes can be extended to handle cones. If the cone only has output edges leaving the cone from its root node, v, the cone is said to be a Fanout Free Cone (FFC). If the FFC maximizes the number of nodes in the cone, it is said to be a Maximum Fanout Free Cone (MFFC). Figure 3.3 shows an example of a cone and a MFFC.

In order to generate a set of K-feasible cones from a circuit, this work adapts the cone generation method discussed in [17, 41]. The K-feasible cones are generated as the graph is traversed in topological order from primary inputs to primary outputs. At every internal node v, new cones are generated by combining the cones at the input nodes. In contrast to the original algorithm in [17, 41] which combined the cones in every possible way, in this work, the cone generation algorithm combines cones if they have no more than $(K + e)$ inputs in total.

Cone

Maximum Fanout Free Cone
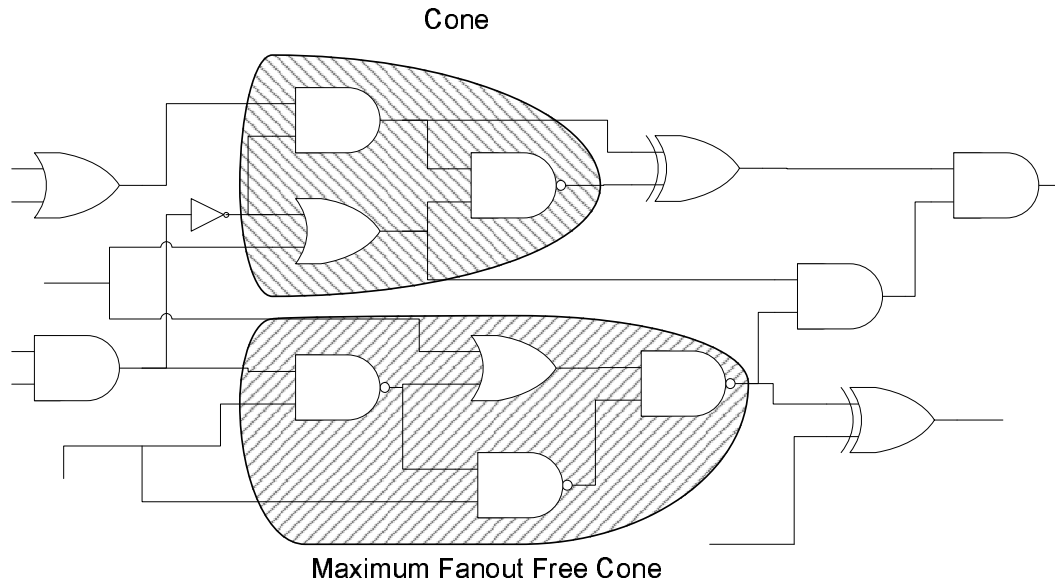
Figure 3.3: A cone of logic and a maximum fanout free cone example.

As long as e was set to a sufficiently high number, this heuristic speeds up the cone generation process without significantly impacting the quality of the mapping solution. The reason for the value e is due to reconvergent cones. For example, Figure 3.4a shows a possible 3-input cone; however, by being able to expand to 4-inputs, as shown in Figure 3.4b, a larger 3-input cone can be found, as shown in Figure 3.4c. Since technology mapping to K-LUTs is thought as a covering problem of cones, capturing reconvergent cones often leads to better quality solutions in terms of both area and depth.

(a) Simple Cone   (b) Expanded Cone   (c) Reconvergent Cone
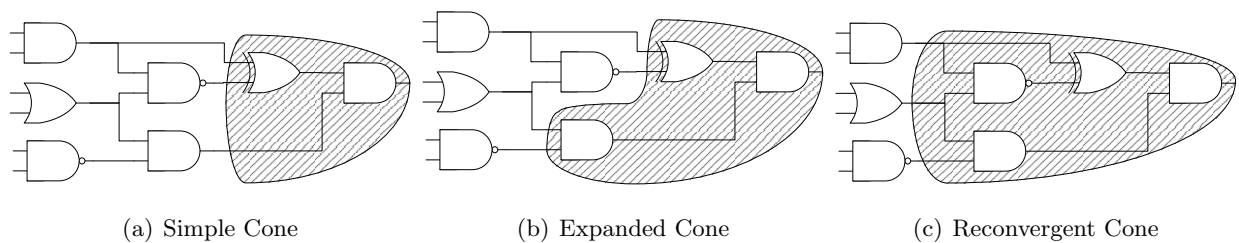
Figure 3.4: A reconvergent cone example.

Once a set of cones is generated, the functions they implement can be extracted and tested for fit in the PLB using the QBF based method described in Chapter 2. A fit percentage can be gathered from this where a high fit percentage is desired.

3.3 Experiments

To show the power of this PLB evaluation method, several unrelated PLB architectures were evaluated. Figure 3.5 shows the five different PLB architectures used for evaluation. Four are simplified models of commerically available PLBs ([50], [48], [3], [4]), and one (MUX PLB) was generated for research purposes only.

(a) Simplified Apex20k variant #1   (b) Simplified Virtex-II

(c) Simplified Apex20k variant #2   (d) Simplified Spartan-IIE   (e) MUX PLB
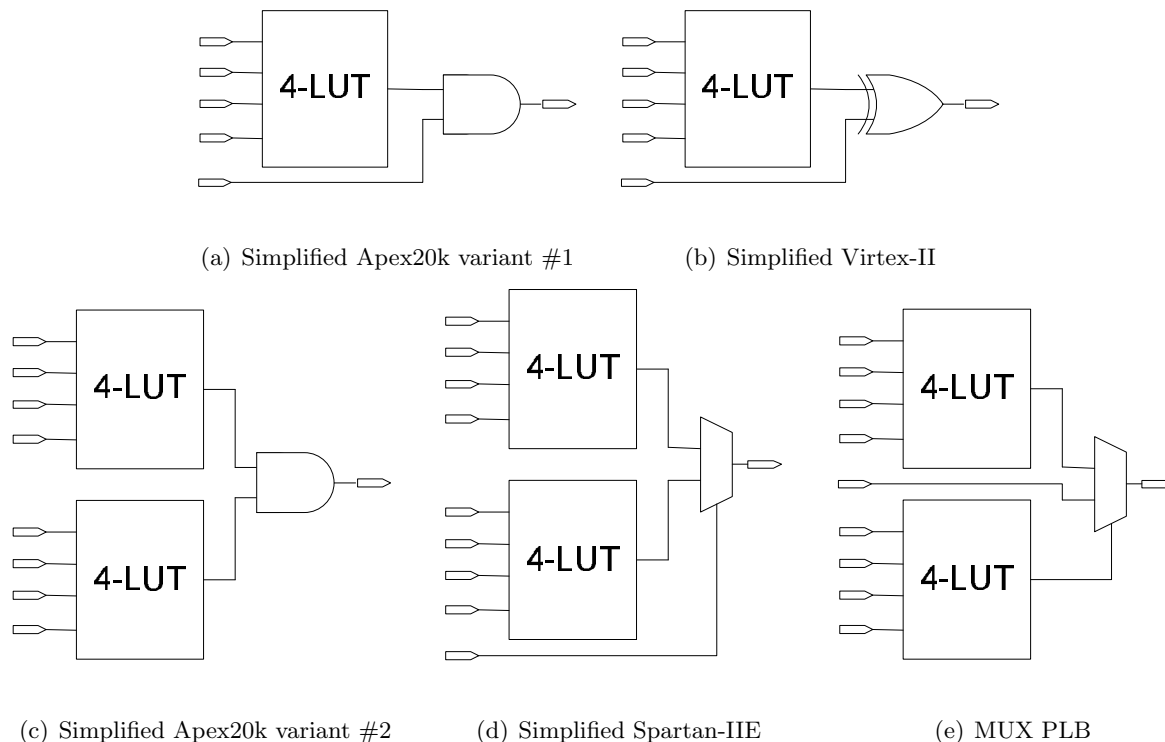
Figure 3.5: PLB architectures used in experiments.

The algorithm is built on top of Berkeley's MVSIS project [9] and the SAT evaluation was done using the Chaff SAT solver [34].[1] To evaluate the application of each n-input PLB, approximately 1000 n-input cones were extracted from a subset of circuits from the MCNC benchmarks [52] and tested for PLB fit. The circuits varied in functionality to create a realistic representation of cones found in digital designs. Table 3.1 shows the percentage of cones that were able to fit into the given PLB per circuit where the last row shows the total cone fit percentage.

---

[1]Experiments run on a Sunblade 150 with 1GB of RAM.

Circuit	Xilinx Virtex II	Xilinx Spartan IIE	Altera Apex20k #1	Altera Apex20k #2	MUX PLB
C2670	1.59	0	27.9	41.8	0
ex5p	0	0	91.4	49.7	0
clma	0	1.29	61.5	40.5	1.29
dalu	0	0	78.2	38.5	0
des	0	0	12.2	72.6	0
i9	0	0	87.4	18.8	0
x3	0	20.1	21.5	38.9	20.2
f51m	0	0	21.7	18.0	0
misex3	0	12.9	70.2	45.4	11.8
mm30a	0	0	20.8	0.20	0
mult16b	0	0	2.91	0^2	0
Total % Fit	0.151	3.44	46.0	36.4	3.34

Table 3.1: The percentage of cones that fit into a given PLB.

Note that the cone fit percentage varies wildly for all PLBs depending on the circuit. This shows that PLB usefulness is dependent on the application of the circuit. Interestingly, the simplified Virtex-II PLB failed for all circuits except the ALU circuit (C2670). Since the Virtex-II PLB has a cascaded XOR gate, it would fair poorly in most control circuits where XOR gates are rare. Also, the simplified Spartan-IIE PLB was only able to fit cones for a few circuits. This was expected since the Spartan-IIE PLB is primarily used to implement 5-input functions or a 4:1 MUX, and is rarely used as a general 9-input function generator [48]. Thus, the efficiency of a PLB cannot be evaluated by a simple n-input function fit percentage.

To understand this, in addition to generating 9-input functions for the Spartan-IIE PLB, 6, 7, and 8-input functions were evaluated as shown in Table 3.2. As the numbers show, this PLB looks much more useful when adding a wider range of functions.

As a final note, this tool cannot evaluate PLB efficiency on its own. Performance issues such as power, speed, and transistor area should be used in conjunction with this tool. The tool presented here gives a quick picture of the expected are performance of a PLB, but deeper analysis in other aspects is necessary to get a full evaluation of a given PLB architecture.

---

[2]No 8 or 9-input functions could be found in circuit mult16b

Circuit	6-input	7-input	8-input
C2670	1.00	0.00	0.00
ex5p	7.07	1.02	2.10
clma	8.62	5.13	2.24
dalu	6.77	1.45	0.00
des	0.00	0.00	0.00
i9	2.20	0.34	0.00
x3	28.3	25.2	17.7
f51m	38.7	11.5	0.10
misex3	16.7	15.7	13.8
mm30a	13.1	3.35	2.84
mult16b	0.00	0.00	0.00^3
Total % Fit	14.7	6.41	3.50

Table 3.2: The percentage of cones that fit into the simplified Spartan-IIE PLB.

3.4   Summary

This chapter has presented a method for evaluating the efficiency of an arbitrary FPGA PLB. It was applied to several existing FPGA logic structures where results were shown. The power of this technique comes from its automation and generality. Thus, it may become useful in the FPGA design cycle by evaluating new architectures.

---

[3]No 8-input functions could be found in circuit mult16b

Chapter 4

Technology Mapping to Programmable Logic Blocks

4.1 Extending Programmable Logic Block Mapper

The previous chapter discussed a robust PLB mapping tool to evaluate various PLB architectures. This tool can be extended for other applications, one of which is a robust PLB technology mapper called **SATMAP** discussed in this chapter.

The problem of technology mapping to K-LUTs was discussed in Section 1.1.1 and for convenience is briefly covered here. Technology mapping is thought of as a covering problem: Given a Boolean circuit represented as a graph $G = (V_G, E_G)$, attempt to find a set of K-feasible cones to cover the graph. A cone, C_v, is a subgraph of G that consists of a root node v and some of its predecessors, $u \in C_v$, such that all paths between v and u are contained in the cone. A cone is referred as K-feasible if it has K-inputs or less. The covering of cones should be optimized such that area, depth, or a combination of both is minimized. Once a cover is found, each K-feasible cone can be directly translated into a K-LUT. The translation process is direct since a K-LUT can implement any K-feasible cone.

K-LUT technology mapping can be extended to a general PLB; however, unlike a K-LUT, a K-input PLB cannot implement any arbitrary K-feasible cone. Thus, a legality check must be done during the translation process from cones to PLBs to verify if a cone can be realized in

```
1  IMap(int MaxI)
2  begin
3     GENERATECONES()
4     for (int i = 1 to MaxI)
5         TRAVERSEFWD()
6         TRAVERSEBWD()
7     CONESTOLUTS()
8  end
```

Algorithm 4.1: A high-level overview of the original IMap algorithm.

the given PLB. Our SAT based approach presented in Chapter 2 can be used for this legality check to build a general PLB technology mapper called **SATMAP**.

4.2 Iterative Technology Mapper - IMap

SATMAP was built on top of IMap [32]: one of the best publicly available K-LUT technology mappers to date. The general IMap algorithm is described here, but a more detailed description can be found in [32].

A high-level overview of the IMap algorithm is presented in Figure 4.1. First, a call to GENERATECONES generates the set of all K-feasible cones for every node in the graph. Then a series of forward and backward graph traversals is started. The forward traversal, TRAVERSE-FWD, selects a cone for each node, and the backward traversal, TRAVERSEBWD, selects a set of cones to cover the graph. Iteration is beneficial because every backward traversal influences the behavior of the forward traversal that follows it. Finally, a call to CONESTOLUTS converts the cones selected by the final backward traversal into LUTs.

4.2.1 Generating K-Feasible Cones

A version of the algorithm described in [17, 41] is used to generate and store all K-feasible cones in the graph. The K-feasible cones are generated as the graph is traversed in *topological order* from *primary inputs* (PI) to *primary outputs* (PO). Topological order implies that a node is visited after all of its fanins have been visited. A fanin to a node v, is any input node to v. The set of fanin edges to v can be symbolized as *iedges*(v). Similar to this is a fanout, where a fanout to a node v, is any output node to v. The fanout edges to v can be symbolized as

```
1  TraverseFwd
2  begin
3     foreach (v ∈ PI) {
4        depth(v) ← 0
5        af(v) ← 0
6        foreach (e ∈ oedges(v)) {
7           depth(e) ← 1
8           af(e) ← 0
9        }
10    }
12    foreach (v ∈ TSORT(V_G − PI − PO) {
13       C_v ← BESTCONE(v)
14       depth(v) ← depth(C_v)
15       af(v) ← 1 + ∑_{i∈iedge(C_v)} af(i)
16       foreach (e ∈ oedges(v)) {
17          depth(e) ← depth(v) + 1
18          af(e) ← af(v) / ‖oedges(v)‖_est
19       }
20    }
21 end
```

Algorithm 4.2: The algorithm used for forward traversal.

$oedges(v)$. A primary input is a node with no fanins, and a primary output is a node with no fanouts. At every internal node v, new cones are generated by combining the cones at the input nodes. In contrast to the original IMap algorithm which combined the cones in every possible way, in this work, the cone generation algorithm combines cones if they have no more $(K + e)$ inputs in total. As long as e was set to a sufficiently high number (2 in the experiments), this heuristic sped up the cone generation process without significantly impacting the quality of the mapping solution.

4.2.2 Forward Traversal

The algorithm used for forward traversal is presented in Figure 4.2. During the traversal, the algorithm updates the *depth* and the *area flow* for every node and edge encountered. Depth, $depth(v)$, is the length of the longest path from a primary input to v. The length of a path is the sum of the delays of the edges along the path. The depth for a primary input is zero. At an edge e, the depth, $depth(e)$, is the length of the longest path from a primary input to e. The depth of an edge includes the delay due to the edge itself. The depth of a graph is the

$$depth(C_v) \leq ODepth - height(v). \tag{4.1}$$

Figure 4.1: Depth bound for cones selected by BESTCONE during depth-orientated mapping.

length of the longest path in the graph. Furthermore, considering that a cone can be viewed as a node, all definitions that apply to a node can also apply to a cone. Area flow, denoted $af(\cdot)$, is a heuristic for estimating the area of the mapping solution below a node or an edge. The area flow at a node, v, is computed as the total area flow coming in on the input edges of the cone, C_v, selected to cover v (line 15), and the area flow at an output edge of v is computed as the total area flow at v divided by its estimated fanout size, $\|oedges(v)\|_{est}$ (line 18). During mapping, the minimization of the area flow was shown to lead to smaller mapping solutions [32].

At each internal node v, a call to BESTCONE(v) is used to select a cone rooted at v to be used in covering v and some of its predecessors in a mapping solution. This cone determines the depth and area flow for v and its output edges (line 15-18). The quality of the mapping solution is determined by the cones selected by BESTCONE. The cone selection strategy changes depending on the mapping objective that is being used.

During depth-oriented mapping, on the first mapping iteration, the cone with the lowest depth is selected, and if cones are equivalent in depth, then the one with the lowest area flow is selected. The first forward traversal establishes the optimal mapping depth, $ODepth$, which can then be used in subsequent iterations to bound the depth of cones selected at every node. Using the optimal depth, and the height of a node v (established during the preceding backward traversal), a bound can be defined on the depth of a cone C_v at v as follows *Height* is a property analogous to depth, but is measured as the length of a path starting at a primary output. Hence, the height of a primary output is zero and the height of a node, $height(v)$, is the length of the longest path from a primary output to v. Cones that meet the bound requirement in Equation 4.1 are preferred and among a set of cones that meet the bound requirement, cones with lower area flows are preferred. This selection strategy ensures that the mapping solutions will still achieve the optimal depth but the greater flexibility in cone selection when the depth bound has been met leads to mapping solutions that are smaller in area.

During area-oriented mapping, the cone with the lowest area flow is selected and if cones

(a) Two Fanouts   (b) One Fanout
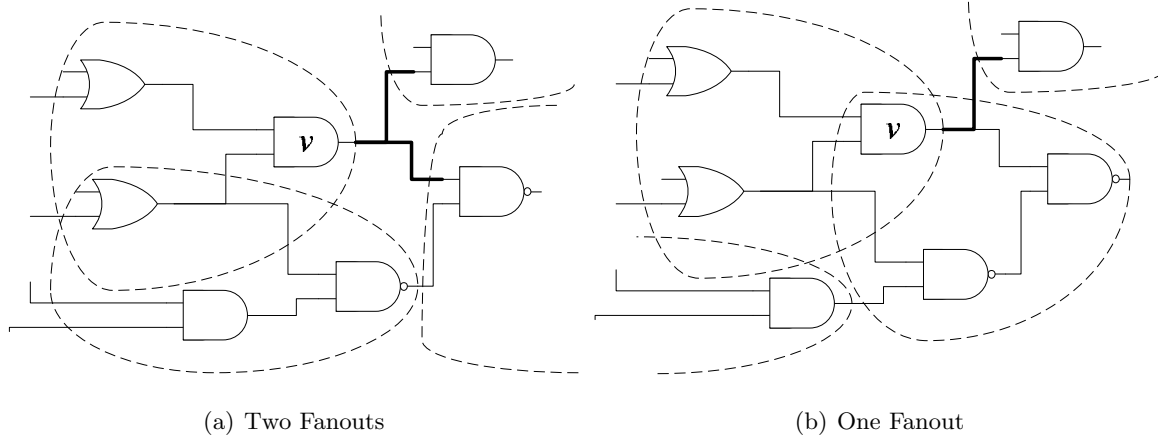
Figure 4.2: Illustration of fanout dependency on the cone covering.

$$\|oedges(v)\|_{est} = \frac{\|oedges(v)'\|_{est} + \alpha\|oedges(v)\|}{1 + \alpha} \qquad (4.2)$$

Figure 4.3: Equation for estimating a node's fanout size.

are equivalent in area flow, then the one with the lowest depth is selected.

When calculating the area flow of an edge (line 18), the number of output edges is necessary. However, this number is not known until a cone covering is selected in the following backward traversal. For example, consider Figure 4.2. If the covering chosen in the backward traversal is Figure 4.2a, node v has a fanout of 2; however, if the covering chosen is Figure 4.2b, the node v has a fanout of 1.   This problem is avoided by using an estimate for the fanout. This is done by taking a weighted average of previous fanout sizes per node as shown in Equation 4.2 where $\|oedges(v)'\|_{est}$ is the estimate found in the previous iteration and $\|oedges(v)\|$ is the actual number of edges found in the last backward traversal. In cases where v was not a cone root node in the last covering chosen, $\|oedges(v)\|$ is set to 1. On the first traversal, $\|oedges(v)'\|_{est}$ and $\|oedges(v)\|$ does not exist. Thus, $\|oedges(v)\|_{est}$ is simply set to the number of output edges of the node v in the original netlist. Experiments showed that setting α between 1.5 and 2.5 was best for iterations of 8 or less [32].

4.2.3   Backward Traversal

The algorithm used for backward traversal is presented in Figure 4.3. Backward traversal is responsible for finding a final cover for all the nodes in the circuit using the cones found in

```
1  TraverseBwd
2  begin
3     S ← ∅
4     foreach (v ∈ PO) {
5        foreach (e ∈ iedges(v)) {
6           height(e) ← 1
7        }
8        S ← S ∪ inode(v)
9     }
11    foreach (v ∈ RTSORT(V_G − PI − PO)) {
12       if (v ∈ S) {
14          h ← max{height(e) : e ∈ oedges(v)}
15          foreach (u ∈ V(C_v))
16             height(u) ← max{height(u), h}
18          foreach (e ∈ iedges(C_v))
19             height(e) ← max{height(e), h + 1}
21          S ← S ∪ inode(C_v)
22       }
23    }
25 end
```

Algorithm 4.3: The algorithm used for backward traversal.

the proceeding forward traversal. Furthermore, backward traversal updates all the heights of each node to match the height of the cone selected to cover it. The heights are necessary in the succeeding forward traversal. To find the covering, internal nodes of the graph are visited in the *reverse topological* order generated by RTSORT. Reverse topological ordering implies that a node is processed after its fanout nodes have been processed. Although all internal nodes are visited, only the nodes that are required to be *visible* in a mapping solution are processed. A node is visible if it is the root of a cone used in the final covering solution. Initially, only the primary outputs are added to the visible set S, as they are required to be (line 8). However, as each node in the visible set, S, is covered by its cones, the fanin nodes of the cone are added to the visible set (line 21). For each visible node, the cone C_v which was selected by BESTCONE in the preceding forward traversal is used to cover the visible node and some of its predecessors.

Also, the heights of all nodes are updated during the covering process. It is assumed that before the algorithm is run, the heights of all nodes and edges have been set to zero. First, the algorithm updates the heights of edges attached to primary outputs. At a visible node v, the maximum height of the node's output edges, h, is used to determine the height of all nodes

<div align="center">

(a) Detailed Model   (b) Simplified Model
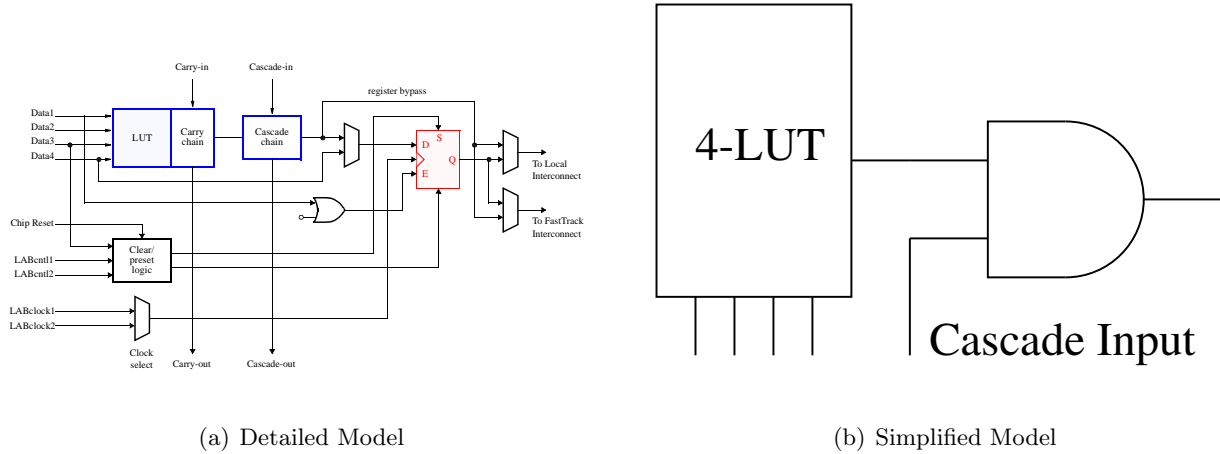
Figure 4.4: The Altera Apex20k PLB.

</div>

covered by C_v. A node may be part of several cones; thus, the new height h is only assigned to a node if it is higher than its previous height. Similarly, an edge may serve as an input to several cones; thus, the height of the path through C_v is only assigned to an edge if it is higher that its previous height. The order of traversal (reverse topological order) guarantees that the edge heights have settled into their final values before they are actually used.

4.2.4 Extending to PLBs

In order to adapt IMap to PLBs to form **SATMAP**, GENERATECONES is modified such that every generated K-feasible cone is run through a legalization step to ensure that it can be realized in the PLB. Cones that fail the fit are discarded leaving a subset of legal cones used in the forward, TRAVERSEFWD, and backward traversals, TRAVERSEBWD. Thus, the last function call of IMap, CONESTOLUTS(), can be changed to, CONESTOPLBS().

4.3 Results

The results of incorporating SAT into IMap for PLB technology mapping (the full **SATMAP** algorithm) are shown here. The PLB of choice was the 5-input Altera Apex20k PLB shown in Figure 4.4a where a simplified model shown adjacent to it is used in the experiments. This consists of an AND gate fed by a 4-LUT and cascade input. Although a simplified PLB was used, routing constraints were maintained. These constraints required the PLB cascade input

to come from an adjacent PLB as illustrated in Figure 4.5. This constraint prevents primary inputs and non-adjacent PLBs feeding the cascade input.
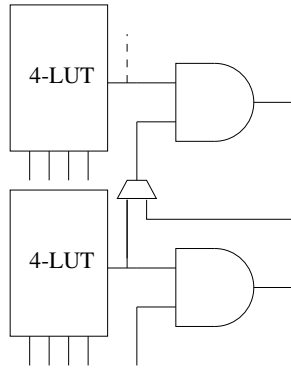
Figure 4.5: Apex20k PLB routing constraints.

Table 4.1 and Table 4.2 show the area and depth costs for the largest 20 MCNC benchmark circuits [52]. For depth the unit delay model is used where each edge is given a delay of 1. For area each 4-LUT is given a cost of 1. Area cost uses the assumption that the area of an `AND` gate is insignificant compared to the area of a 4-LUT. Table 4.1 shows results when depth was the primary optimizing goal and Table 4.2 shows results when area was the primary optimizing goal.

The results clearly show that **SATMAP** is an effective tool for technology mapping directly to PLBs as it outperforms any 4-LUT technology mapper. This is not surprising since it is assumed that industrial PLBs would perform better than LUT only FPGA architectures.

4.4 Summary

This chapter illustrates a powerful PLB technology mapper. With this tool, custom heuristics are unnecessary to map circuits to new PLB architectures. This speeds up the development time for FPGAs. An added advantage of the work is that it can be adapted to any new technology mapping algorithm. Thus, progression in technology mapping heuristics for area will directly benefit this work.

	Depth Orientated $k = 4$							
	SATMAP		*IMap*		*FlowMap-r0*		*ZMap*	
Circuit	Area	Depth	Area	Depth	Area	Depth	Area	Depth
alu4	1003	7	1045	7	1244	7	1204	7
apex2	1173	8	1236	8	1468	8	1400	8
apex4	997	7	1131	6	1131	6	1113	6
bigkey	1145	4	1586	3	1362	3	1698	3
C6288	814	24	1023	26	539	25	546	25
clma	4689	15	5032	14	5359	15	5297	15
des	1128	6	1239	6	1522	6	1359	6
diffeq	904	13	1025	12	1420	12	1013	12
dsip	1367	4	1144	4	1591	4	1144	4
elliptic	2094	16	2239	16	3560	16	2708	16
ex1010	2180	8	2639	8	2684	8	2657	8
ex5p	983	6	991	7	1055	7	1051	7
frisc	2275	21	2397	21	3396	21	2858	21
i10	811	14	914	15	953	14	867	14
misex3	1117	6	1115	7	1293	7	1244	7
pdc	1829	10	2180	10	2216	10	2147	10
s38417	4228	10	4296	10	3992	10	3720	10
s38584.1	3963	10	4124	11	4437	10	4176	10
seq	1102	6	1120	7	1385	7	1292	6
spla	1271	8	1380	8	1612	8	1549	8
Total	35073	203	37856	206	42219	204	39043	203
Ratio	1.000	1.000	1.079	1.015	1.204	1.005	1.113	1.000

Table 4.1: **SATMAP** comparisons with depth-oriented mapping.

| | Area Orientated $k = 4$ | | | | | | | |
| | *SATMAP* | | *IMap* | | *FlowMap-r3* | | *ZMap* | |
Circuit	Area	Depth	Area	Depth	Area	Depth	Area	Depth
alu4	1002	9	1020	9	1144	9	1129	11
apex2	1222	13	1173	11	1290	10	1308	13
apex4	978	9	1011	10	1099	8	1115	9
bigkey	920	5	1034	4	1254	4	1145	4
C6288	765	34	972	44	549	25	562	27
clma	4261	22	4476	20	4950	17	5014	26
des	1103	10	1161	9	1237	8	1194	9
diffeq	921	18	1049	16	931	14	941	14
dsip	1146	5	1144	4	1145	4	1367	5
elliptic	2009	22	2204	23	2133	18	2342	22
ex1010	1995	13	2138	14	2397	11	2325	14
ex5p	906	11	923	10	956	9	993	11
frisc	2152	29	2353	33	2659	24	2624	29
i10	785	23	899	22	774	17	779	21
misex3	1062	10	1078	9	1185	9	1184	10
pdc	1773	15	1755	17	1907	12	1928	17
s38417	4039	15	4084	13	3803	12	3586	14
s38584.1	3929	14	4018	16	3921	12	3762	16
seq	1109	10	1083	11	1204	9	1182	11
spla	1287	11	1284	12	1412	11	1403	12
Total	33364	298	34859	307	35950	243	35883	295
Ratio	1.000	1.000	1.045	1.030	1.078	0.815	1.076	0.990

Table 4.2: **SATMAP** comparisons with area-oriented mapping.

Chapter 5

Technology Mapping Evaluation using Resynthesis

5.1 Evaluation of Area Driven Technology Mapping

Technology mapping is a vital step in producing high quality solutions in FPGA synthesis. There has been much research in this area which have been able to produce satisfactory solutions(see Section 1.1.1). However, because *area*-driven technology mapping is an NP-hard problem [22, 31], it is uncertain how close these solutions are to the optimal. The chapter seeks to explore this by assessing state-of-the-art FPGA technology mapping algorithms in terms of *area*-optimality. The distance to the area-optimal solution can be measured in terms of removable LUTs in a technology mapped circuit. These LUTs are redundant since removing them does not change the functionality of the original circuit. The more removable LUTs, the farther the original circuit is from the optimal. The fundamental question addressed in this chapter is: Given the LUT-level network created by a technology mapping algorithm, how much can its LUT count be reduced? Reducing the LUT count involves reconfiguring the surrounding network of a removed LUT. For small subcircuits, it is possible to do this in an optimal manner. Consider an arbitrary function $f(i_0, i_1, \ldots, i_n)$ implemented with four or more 2-LUTs. Suppose that we seek to determine if it can be implemented in three or fewer 2-LUTs. This problem can be solved by considering the Configurable Virtual Network (CVN) shown in Fig-

ure 5.1. The CVN consists of input lines connected to the variables $i_0 \ldots i_n$ and three 2-LUTs. A crossbar allows the LUT inputs to select any of the input lines or outputs from other LUTs. Each "switch" on the crossbar is configured by a virtual configuration bit. A **0** indicates that the crosspoint intersection is unconnected, while a **1** indicates a connection at the crosspoint. Clearly, it is possible to enumerate every possible circuit configuration involving three 2-LUTs by manipulating the virtual configuration bits for the crossbar as well as the truth table configuration bits for each of the 2-LUTs. In fact, as detailed in Section 2.5, we can express the CVN as a Boolean formula involving the variables $i_0 \ldots i_n$, the virtual configuration bits $V_1 \ldots V_m$ and the truth table configuration bits $L_1 \ldots L_o$. Using this formula, we can verify if there exists a configuration to $V_1 \ldots V_m$ and $L_1 \ldots L_o$ such that it implements $f(i_0, i_1, \ldots, i_n)$. The solu-
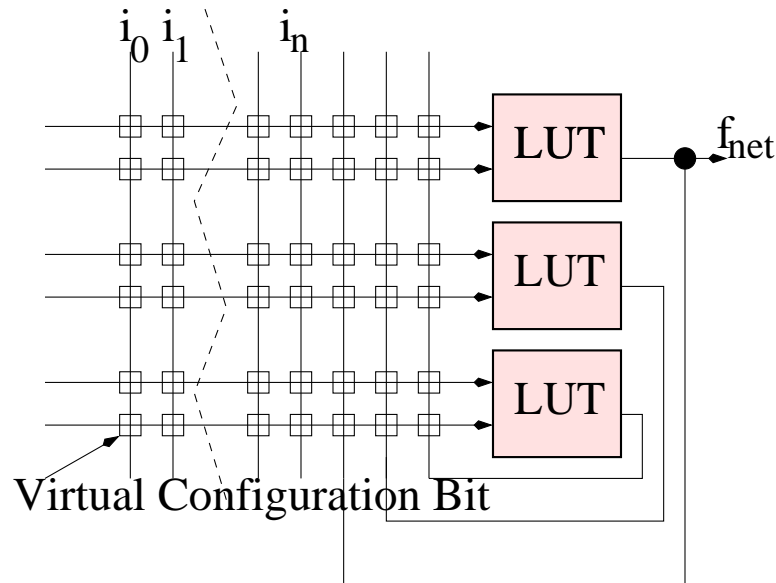
Figure 5.1: A configurable virtual network.

tion space of this problem grows exponentially with respect to the input size, n, of the virtual network. However, the SAT based technique described in Chapter 2 can be used to exactly resynthesize small circuits.

Given this optimal resynthesis method for small subcircuits, it is applied iteratively to small portions of a larger circuit in a sliding window fashion until no additional improvement can be achieved. This approach does not guarantee the mapping optimality of the large circuit, but

it does give some indication of the area "left on the table" by the original technology mapping solution.

5.2 Resynthesizing for Area

When resynthesizing for area, one must take an existing LUT circuit and attempt to reduce the number of LUTs in the circuit yet maintain the original functionality. The more LUTs that can be removed, the farther the original circuit is from the optimal mapping.

As mentioned previously, reducing the number of LUTs can be achieved by resynthesizing smaller subcircuits and applying this in a sliding window fashion over the larger circuit. These subcircuits form a cone where a cone consists of a root node and some predecessors such that all paths between the predecessors and root node are contained in the cone. Thus, resynthesizing several cones will reduce the LUT count of the overall LUT network.

5.2.1 Converting Resynthesis Problem into Boolean Satisfiability

Determining if a K-feasible cone implemented with X K-LUTs can be resynthesized into an K-feasible cone implemented with X-1 K-LUTs or less can be posed as a function mapping problem. The goal is to determine if the function extracted from the original cone containing X K-LUTs can map into a new cone of logic containing X-1 K-LUTs or less. This can be verified using the SAT based technique described in Chapter 2.

To illustrate this process, consider Figure 5.2. The original cone 5.2a consists of three 2-LUTs which implements a three input function. Since only three inputs enter the cone, it may be possible to resynthesize 5.2a into 5.2b to save one LUT.

To determine if resynthesis from 5.2a to 5.2b is possible, the function that 5.2a implements must be extracted and 5.2b must be converted into a CNF expression with the function from the original circuit incorporated into it. As stated before, if the expression is satisfiable, resynthesis can occur.
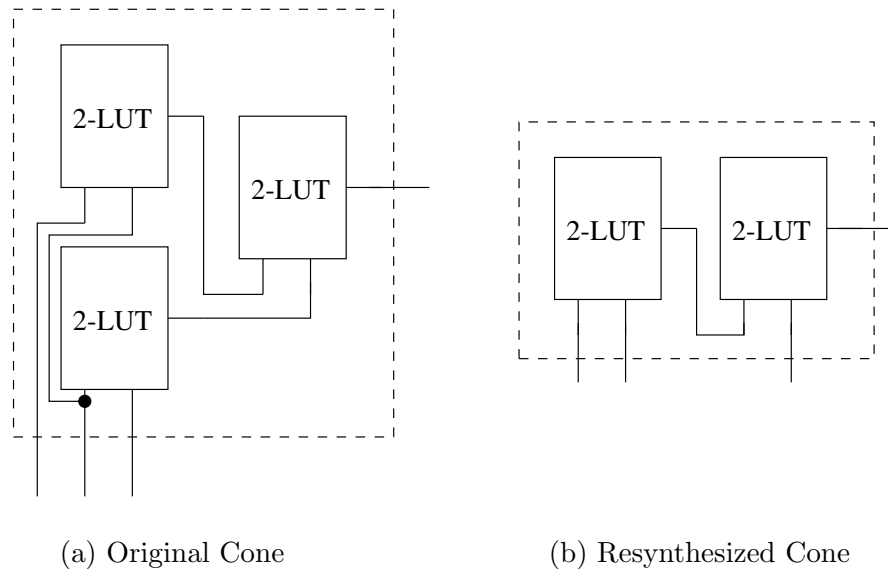
(a) Original Cone   (b) Resynthesized Cone

Figure 5.2: Resynthesis of a 3-input cone example.

5.2.2 Generation of Cones

As in Chapters 3 and 4, a version of the algorithm described in [17, 41] is used to generate and store all resynthesis cones in the graph. The resynthesis cones are generated as the graph is traversed in topological order from primary inputs (PIs) to primary outputs (POs). Topological ordering implies that a node is processed after all of its fanin nodes have been processed. At every internal LUT, new cones are generated by combining the cones at the input LUTs. In contrast with [17], which combined the cones in every possible way, in this work, the cone generation algorithm combines cones if they have no more $(K + e)$ inputs in total, where K is the fanin size of the largest resynthesis cone and e is an expansion size. As long as e was set to a sufficiently high number (8 in the experiments), this heuristic sped up the cone generation process without significantly impacting the quality of the resynthesis solution.

Special consideration must be taken for non-fanout free cones. Non-fanout free cones are cones with fanouts originating from nodes other than the root node as illustrated in Figure 5.3a. If resynthesis is to occur on these cones, all fanouts must be preserved. This can be done either by duplicating the nodes feeding the fanouts of nodes other than the root node or resynthesizing the node such that all the fanouts are preserved in the new resynthesis cone. For example,

(a) Original Cone   (b) Duplication Resynthesis   (c) Multiple Output Resynthesis
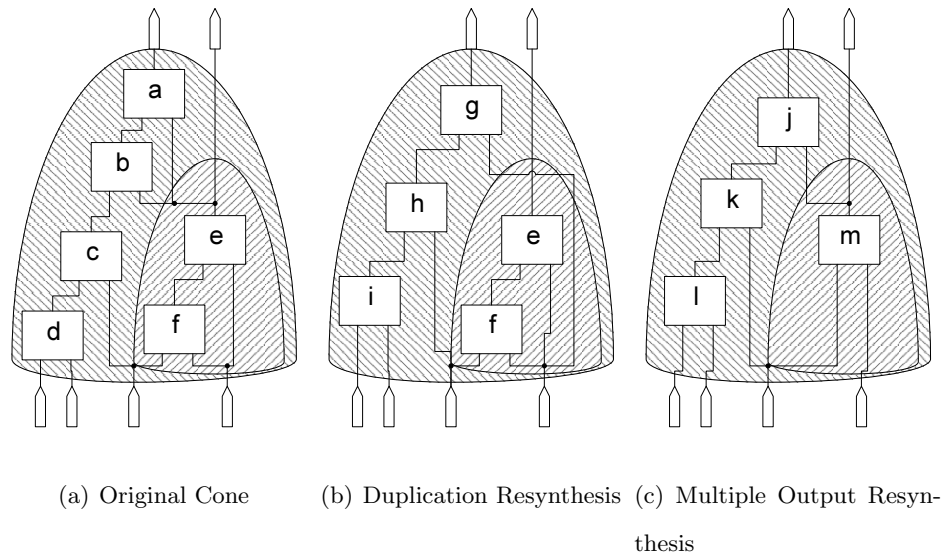
Figure 5.3: A multiple output cone used for resynthesis.

consider Figure 5.3. Figure 5.3a is the original cone, Figure 5.3b is resynthesis with duplication, and Figure 5.3c is resynthesis with multiple fanouts. In the duplicated case, if LUTs a through f are resynthesized to LUTs g, h, and i, LUTs e and f must be duplicated to keep the fanout at LUT e. However, if the resynthesis cone is able to maintain all fanouts, duplication is unnecessary and LUT m is used to feed the fanout originally stemming from LUT e.

5.3 Results

The resynthesis algorithm discussed in the previous section was applied to circuits produced by the ZMap technology mapper — one of the best publicly available FPGA area-driven technology mappers developed by J. Cong et al. at UCLA [16]. Given a set of circuits, ZMap was used to technology map these circuits to 4-LUTs. After some post processing done by RASP [16] to further improve area, the resynthesizer [1] was applied on these LUT networks.

The number of resynthesis structures is exponential in relation to the resynthesis structure input size; however, considering that the size of the CNF equation is exponential to the number of resynthesis structure inputs, for practical purposes, this work dealt with cones of fanin size

---

[1] The resynthesizer was incorporated with the Berkeley MVSIS project [9]

10 or less. This limits the number of resynthesis structures to the ones shown in Figure 5.4. Figure 5.4a is applied for cones with a fanin size of seven or less and containing more than

(a) 7-input Cone

(b) 10-input Cone #1

(c) 10-input Cone #2

(d) 10-input Cone Multiple Output
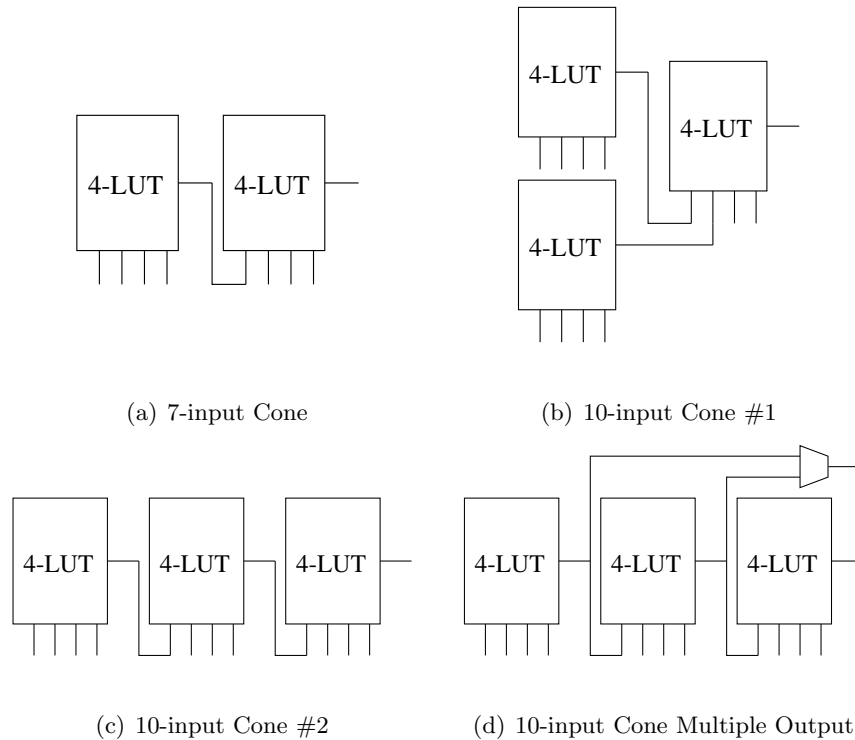
Figure 5.4: Resynthesis structures used in experiments.

two 4-LUTs; Figures 5.4b and 5.4c are applied for cones with a fanin size of 10 or less and containing more than three 4-LUTs; and Figure 5.4d is applied to cones with a fanin size of 10 or less and containing more than three 4-LUTs while having one other fanout not originating from the root node. Resynthesis checking was done using the Chaff SAT solver developed by M. W. Moskewicz et al. [34].

In order to reduce the number of candidate cones for resynthesis, at most only two LUTs were allowed to be duplicated. Experiments showed that increasing the duplication count above two increased the set of cones to the point that the resynthesis tool would take an extraordinarily long time to execute.

Circuit	ZMap	Resynth No (d)	Ratio	Resynth All	Ratio
b20	5996	5530	0.92	5415	0.90
clma	5014	4792	0.96	4791	0.96
b15_1	4291	4112	0.96	4088	0.95
b15_1_opt	3879	3772	0.97	3748	0.97
s38584.1	3771	3454	0.92	3417	0.91
s38417	3586	3444	0.96	3355	0.94
b14	3072	2902	0.94	2859	0.93
frisc	2624	2571	0.98	2566	0.98
pdc	1928	1875	0.97	1873	0.97
misex3	1184	1156	0.98	1154	0.97
seq	1182	1162	0.98	1159	0.98
alu4	1129	1103	0.98	1101	0.98
ex5p	993	968	0.97	966	0.97
i10	789	764	0.97	763	0.97
Sub-total 3000-	9829	9599	0.98	9582	0.97
Sub-total 3000+	29609	28006	0.95	27673	0.93
Total	32075	33442	0.96	31840	0.95

Table 5.1: Benchmark circuit resynthesis results.

5.3.1 Benchmark Circuits

In the first set of experiments, circuits taken from the MCNC and ITC'99 benchmark suites ([52],[19]) were resynthesized. These circuits were optimized using SIS [42] and RASP, technology mapped with ZMap, and resynthesized with this work. The optimization in SIS is particularly important since the structure of the gate-level netlist can have a significant impact on the mapped area. Table 5.1 shows the results. The *ZMap* column indicates the number of 4-LUTs the circuit was technology mapped to. The *Resynth All* column indicates the number of 4-LUTs after resynthesis when all resynthesis structures shown in Figure 5.4 were used and *Resynth No (d)* column lists the resynthesis results when Figure 5.4d was omitted. The respective ratios when compared against *ZMap* are shown in the adjacent column. The *Sub-total 3000-* and *3000+* rows list the total LUT counts for circuits with less than 3000 LUTs and more than 3000 LUTs respectively. The results clearly show that ZMap does not achieve optimal results; this implies that all FPGA technology mappers that perform worse then ZMap also have much room for improvement. Also, adding the multiple output resynthesis structure marginally reduces the LUT count. It is interesting that the sub-totals suggest that the largest

```
module BarrelShifter16Bit(SHIFT,D,Q)
 input[1:0] SHIFT;input[15:0] D;
 output[15:0] Q;reg[15:0] Q;
 always  @ (D or Q or SHIFT)
  case  (SHIFT)
   2'b00 :  Q=D;
   2'b01 :  Q={D[3:0],D[15:4]};
   2'b10 :  Q={D[7:0],D[15:8]};
   2'b11 :  Q={D[11:0],D[15:12]};
  endcase
endmodule
```
Algorithm 5.1: 16-Bit Barrel Shifter Verilog Code

Building Block	*Resynth*	*ZMap*	*Ratio*
4:1 MUX	2	3	0.67
16:1 MUX	21	29	0.72
32-Bit Priority Encoder	59	74	0.80
4-Bit Barrel Shifter	8	12	0.67
16-Bit Barrel Shifter	32	48	0.67
6-Bit Set Reset Checker	2	3	0.67
2-Bit Sum Compare Constant	2	6	0.33
2-Bit Sum Compare	2	3	0.67
6-Bit Priority Checker	3	6	0.50
8-Bit Bus Multiplexer	16	24	0.67
Total	188	253	0.74

Table 5.2: Logic block resynthesis results.

decreases in area are seen in circuits with 3000 LUTs or more, with the largest decrease of 10%
seen in circuit b20. This suggests that the deviation from the optimal solution is proportional
to the size of the circuit. The fact that area driven technology mapping is NP-hard [22, 31]
supports this claim.

5.3.2   Building Block Circuits

In the second set of experiments, common digital circuit logic blocks were resynthesized. Start-
ing from Verilog code, the circuit description was synthesized using VIS [8], then optimized and
techmapped as in Section 5.3.1. For illustration, Module 5.1 shows an example Verilog code
block that was synthesized then resynthesized. More code is shown in the Appendix for further
reference.   Table 5.2 shows the results, where a reduction as large as 67% and an average
reduction of 26% was achieved. Since these logic blocks are common in digital circuits, their

optimal configurations found by the resynthesis tool can be stored in a cache for future use. Heuristics can be used to identify these blocks during technology mapping and replace them to their optimal configuration found in the cache.

In general, most of the functions that implement the logic blocks shown in Table 5.2 are highly non-disjoint. It appears that finding the optimal LUT covering for such functions is still very difficult. This can be explained by the seperation of synthesis and technology mapping. Synthesis of logic into 2-input gates occurs independently of technology mapping to LUTs. Thus, there can be times where beneficial steps in synthesis may not be beneficial for technology mapping. For example, in the case of non-disjoint functions, the optimal 2-input gate solution often will not correspond to a good starting point for technology mapping to LUTs.

It is interesting to note the dramatic differences in results between the benchmark circuits and the results of the individual building blocks. A speculative reason for this is that common building blocks are being collapsed with other random or glue logic in the benchmarks. Since the size of the subcircuit resynthesis procedure is limited, it is likely that some key resynthesis opportunities are missed.

5.4   Summary

In this chapter, a method to help understand the optimality of state-of-the-art FPGA technology mapping algorithms was presented. The approach involves optimally resynthesizing small portion of the circuit until no further improvement can be found. This approach is itself non-optimal. However, if this localized optimal resynthesis approach is able to improve the mapping result from an existing technology mapping algorithm, then it gives an indication of the mapper's "distance" from optimality.

Chapter 6

Conclusions and Future Work

6.1 Contributions

This dissertation presented a novel application of Boolean satisfiability (SAT) to FPGA logic synthesis. Chapter 2 demonstrated a powerful SAT based function mapping technique where it can determine if a logic function can be realized in configurable circuit. The power of this technique comes from its generality where it can be applied to any logic function on any FPGA architecture. No previous method has existed that accomplishes this.

Several applications of this function mapping technique were shown in Chapters 3 to 5. Chapter 3 applied the technique to create a PLB evaluation tool. The tool worked by determining a fit percentage of Boolean functions extracted from various benchmark circuits which gives a value assessment of new PLB architectures. The higher the fit percentage, the more valuable the PLB. Chapter 4 extended the work in Chapter 3 to create a robust technology mapper. Unlike previous work which depended on custom heuristics to map functions to PLBs, this work used a general approach to map functions which produced a technology mapper applicable to any PLB architecture. Finally, Chapter 5 presented a study of area optimality. Using the SAT based function mapper, a resynthesis technique was developed where small portion of the circuit were resynthesized until no further improvement could be found. This approach was itself non-optimal. However, if the localized optimal resynthesis approach was able to improve the mapping result from an existing technology mapping algorithm, then it gives an indication

of the mapper's "distance" from optimality.

6.2 Future Work

The function mapping technique described in Chapter 2 has potential to be applied in numerous areas in logic synthesis. However, one major limitation of this technique is runtime. As Table 2.3 in Chapter 2 indicated, the runtime increases dramatically with respect to the number of configurable circuit inputs. This is due to the exponential relationship between the Boolean expression size and the configurable circuit input size. Techniques to speed up the SAT runtime are necessary before the SAT based function mapper can be applied elsewhere.

There has be some initial success in accelerating SAT using hardware [1, 55]. These SAT solvers are built in configurable hardware such as FPGAs so they can solve any CNF expression as long as it fits in the hardware. Initial numbers suggest that these SAT solvers can speed up the SAT process dramatically and have simulated speedups of up to 80 times. Note that these hardware based SAT solvers [1, 55] are general SAT solvers, thus the hardware must be configured and programmed at runtime. Work by Zhong et al. [55] shows that this overhead has a significant effect on the overall runtime. Since the SAT based function mapper is aware of the configurable circuit structure before it begins, the CNF expression is predefined. Thus, it can cut out configuration and program time which further increases the speedup.

A problem with hardware based SAT solvers is that they require several FPGAs to implement large CNF formulae containing thousands of clauses. Unfortunately, the technique presented in Chapter 2 creates a CNF formula with tens of thousands of clauses. One way to reduce the number of clauses in the expression is to solve the function mapping problem as a QBF. As a QBF, there exists only a few hundred clauses in the expression thus a hardware QBF solver implementation could potentially fit on one FPGA.

There also exists several software QBF solvers; though they are still in their infancy and have not been too successful in terms of runtime [24, 25, 29, 35, 36, 54]. Continual research in this area may open the door for QBFs such that it can be applied commercially and reach running times similar to SAT [47].

Another variant of the QBF solver is an all-solution SAT solver. In contrast to a traditional SAT solver which returns only one satisfiable assignment if one exists, all-solution SAT solvers may return several assignments. The purpose of an all-solution SAT solver is to explore several possible assignments in cases where one assignment may be preferred over another.

Cone generation should also be improved. In the technology mapping tool presented in Chapter 4, generating cones takes approximately 90% of the running time for 5-input PLBs. To reduce the runtime of generating cones, a non-exhaustive approach must be taken. One such approach involves a top-down method where a cone is grown downwards starting from the root node. This contrasts with the current bottom up approach we use, which is described in [17]. This could effectively make cone generation constant time with respect to the number of nodes in the network. Heuristics such as area flow would be used to find cones that minimize area costs. An added benefit of reducing the number of cones is that SAT would be applied to a much smaller set of cones, further improving runtime.

The tools presented previously have several areas to build on. The PLB evaluator demonstrated in Chapter 3 can be incorporated into a fully automated search engine for new PLB architectures. Genetic algorithms could be used to create candidate PLBs from primitive elements such as lookup tables, basic gates, multiplexers and adder structures. The structures that the genetic algorithms come up with would be rated with the PLB evaluator. In addition to this, the choice of PLB cannot be explored in complete isolation from routing architectures, timing, and power. Issues such as PLB-pin permutability can have a significant impact on the overall area of the FPGA even if the total number of PLBs required to map a circuit is reduced. Furthermore, architects may often sacrifice area for a reduction in delay. All of these factors must be taken into account when evaluating new PLB architectures.

The resynthesis work presented in Chapter 5 could be incorporated into a post-processing step for K-LUT technology mapping. The results shown in Section 5.3.2 clearly indicate that conventional K-LUT technology mappers perform poorly on some very common logic blocks. In order to develop a post-processing step, the resynthesizer would be used to find several optimal logic block configurations. These configurations would be cached for future reference. The post-processing step would involve replacing entire logic blocks in technology mapped circuits

with an optimized configuration found in the cache. For circuits consisting of several logic blocks found in the cache, this would lead to a significant area reduction.

Additionally, multiple output functions could be considered for resynthesis. Considering only single output functions in Chapter 5 yielded a marginal reduction in area (about 5% for benchmark circuits). In general, it appears that single output function decomposition is quite good, and usually yields an optimal solution; however, single output decomposition ignores sub-function sharing between multiple functions. Multiple output decomposition would solve this problem, but it is generally not done in synthesis and is still in its infancy. Adding multiple output resynthesis structures to our experiments could give further insight into area reduction opportunities missed due to single output based synthesis.

The work presented in this dissertation further expands the application of Boolean satisfiability to EDA. Also, improvements to SAT will directly benefit this work. The fundamental function mapping problem solved in this work has numerous applications; of which, three of them have been demonstrated in the previous chapters. There is hope that this work will not end here and has created a path for others to build on.

Bibliography

[1] M. Abramovici and J. T. de Sousa, "A SAT solver using reconfigurable hardware and virtual logic," *Journal of Automated Reasoning*, vol. 24, no. 1/2, pp. 5–36, 2000. [Online]. Available: citeseer.ist.psu.edu/abramovici00sat.html

[2] E. Ahmed and J. Rose, "The effect of LUT and cluster size on deep-submicron FPGA performance and density," *IEEE Transactions on VLSI*, vol. 12, no. 3, pp. 288–298, Mar. 2004.

[3] Altera Corporation, "Flex 10k ii data sheet," Jan. 2003.

[4] ——, *APEX 20K Data Sheet*, Mar. 2004.

[5] ——, "Component selector guide ver 14.0," 2004.

[6] ——, *Stratix II Device Handbook*, Oct. 2004.

[7] V. Betz and J. Rose, "How much logic should go in a FPGA logic block," *IEEE Design and Test Magazine*, vol. 15, no. 1, pp. 10–15, Jan–Mar 1998.

[8] R. K. Brayton and G. D. H. et al., "VIS: a system for verification and synthesis," in *Proceedings of the Eighth International Conference on Computer Aided Veri fication CAV*, 1996, pp. 428–432. [Online]. Available: citeseer.ist.psu.edu/brayton96vis.html

[9] D. Chai, J. Jiang, Y. Jiang, Y. Li, A. Mishchenko, and R. B. on, "MVSIS 2.0 Programmer's Manual, UC Berkeley," Electrical Engineering and Computer Sciences, University of California, Berkeley, Tech. Rep., 2003.

[10] J. Cong and Y. Ding, "On area/depth trade-off in LUT-based FPGA technology mapping," in *Design Automation Conference*, 1993, pp. 213–218. [Online]. Available: citeseer.ist.psu.edu/cong94areadepth.html

[11] ——, "An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Transactions on Computer-Aided Design*, vol. 13, no. 1, pp. 1–13, Jan. 1994.

[12] ——, "Combinational logic synthesis for LUT based field programmable gate arrays," *ACM Transactions on Design Automation of Electronic Systems*, vol. 1, no. 2, pp. 145–204, Apr. 1996.

[13] J. Cong and Y.-Y. Hwang, "Simultaneous depth and area minimization in LUT-based FGPA mapping," in *FPGA*, Feb. 1995, pp. 68–74.

[14] ——, "Boolean matching for complex PLBs in LUT based FPGAs with application to architecture evaluation," in *FPGA*, Feb. 1998, pp. 27–34. [Online]. Available: citeseer.ist.psu.edu/cong98boolean.html

[15] ——, "Boolean matching for LUT-based logic blocks with applications to architecture evaluation and technology mapping," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1077–1090, Sept. 2001.

[16] J. Cong, J. Peck, and Y. Ding, "RASP: A general logic synthesis system for SRAM-based FPGAs," in *FPGA*, 1996, pp. 137–143. [Online]. Available: citeseer.ist.psu.edu/cong96rasp.html

[17] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: enabling a general and efficient fpga mapping solution," in *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field p rogrammable gate arrays*. ACM Press, 1999, pp. 29–35.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, Massachusetts: The MIT Press, 2001.

[19] F. Corno, M. Reorda, and G. Squillero, "RT-level ITC 99 benchmarks and first ATPG results," 2000. [Online]. Available: citeseer.ist.psu.edu/article/corno00rtlevel.html

[20] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, pp. 394–397, July 1962.

[21] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the ACM*, vol. 7, pp. 201–215, 1960.

[22] A. Farrahi and M. Sarrafzadeh, "Complexity of the lookup-table minimization problem for FPGA technology mapping," *IEEE Transactions on Computer-Aided Design*, vol. 13, no. 11, pp. 1319–1332, 1994.

[23] R. J. Francis, J. Rose, and K. Chung, "Chortle: a technology mapping program for lookup table-based field programmable gate arrays," in *Proceedings of the 27th ACM/IEEE conference on Design automation.* ACM Press, 1990, pp. 613–619.

[24] E. Giunchiglia, M. Narizzano, and A. Tacchella, "Qube: A system for deciding quantified boolean formulas satisfiability," in *IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning.* Springer-Verlag, 2001, pp. 364–369.

[25] ——, "Backjumping for quantified boolean logic satisfiability," *Artif. Intell.*, vol. 145, no. 1-2, pp. 99–120, 2003.

[26] A. Kaviani and S. D. Brown, "Efficient implementation of array multipliers in FPGAs," June 1998.

[27] ——, "The hybrid field programmable architecture," *IEEE Design and Test Magazine*, pp. 74–83, Apr–Jun 1999.

[28] K. Keutzer, "Dagon: Technology binding and local optimization by dag matching," in *DAC*, 1987, pp. 341–347.

[29] H. Kleine-Bning, M. Karpinski, and A. Flgel, "Resolution for quantified boolean formulas," *In Information and Computation*, vol. 117, no. 1, pp. 12–18, 1995.

[30] T. Larrabee, "Test pattern generation using Boolean satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 1, pp. 4–15, Jan. 1992.

[31] I. Levin and R. Y. Pinter, "Realizing Expression Graphs using Table-Lookup FPGAs," in *Proceedings of the European Design Automation Conference*, 1993, pp. 306–311.

[32] V. Manohararajah, S. D. Brown, and Z. G. Vranesic, "Heuristics for area minimization in lut-based fpga technology mapping," in *International Workshop on Logic and Synthesis (IWLS'04)*, 2004.

[33] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 506–521, May 1999.

[34] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient sat solver," in *Proceedings Design Automation Conference*, June 2001, pp. 530–535.

[35] J. Rintanen, "Improvements to the evaluation of quantified boolean formulae," in *IJCAI '99: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers Inc., 1999, pp. 1192–1197.

[36] ——, "Partial implicit unfolding in the davis-putnam procedure for quantified boolean formulae," in *LPAR '01: Proceedings of the Artificial Intelligence on Logic for Programming*. Springer-Verlag, 2001, pp. 362–376.

[37] J. R. Robert J. Francis and Z. G. Vranesic, "Chortle-crf: Fast technology mapping for lookup table-based FPGAs," in *IEEE Design Automation Conference*, June 1991, pp. 227–233.

[38] J. Rose, "Hard vs. soft: the central question of pre-fabricated silicon," in *Proceedings International Symposium on Multiple-Valued Logic*, May 2004, pp. 2–5.

[39] J. Rose, R. J. Francis, P. Chow, and D. Lewis, "The effect of logic block complexity

on area of programmable gate arrays," in *Proceedings IEEE Custom Integrated Circuits Conference*, May 1989, pp. 5.3.1–5.3.3.

[40] J. Rose, R. J. Francis, D. Lewis, and P. Chow, "Architecture of field-programmable gate arrays: The effect of logic block functionality on area efficiency," *IEEE Journal of Solid State Circuits*, vol. 25, no. 5, pp. 1217–1225, Oct. 1990.

[41] M. D. F. Schlag, J. Kong, and P. K. Chan, "Routability-driven techology mapping for lookup-table-based fpgas," in *Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*.   IEEE Computer Society, 1992, pp. 86–90.

[42] E. M. Sentovich, K. J. Singh, L. Lavagno, C. M. R. Murgai, A. Saldanha, H. . Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," Electrical Engineering and Computer Sciences, University of California, Berkeley, Tech. Rep., 1992. [Online]. Available: citeseer.ist.psu.edu/sentovich92sis.html

[43] C. E. Shannon, "A Symbolic Analysis of Relay and Switching Circuits," *Transactions of the American Institute of Electrical Engineers AIEE*, vol. 57, pp. 713–723, 1938.

[44] S. Singh, J. Rose, P. Chow, and D. Lewis, "The effect of logic block architecture on FPGA performance," *IEEE Journal of Solic State Circuits*, vol. 27, no. 3, pp. 281–287, Mar. 1992.

[45] A. D. Smith, "Diagnosis of combinational logic circuits using boolean satisfiability," Master's thesis, University of Toronto, 2004.

[46] L. J. Stockmeyer and A. R. Meyer, "Word problems requiring exponential time(preliminary report)," in *STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing*.   ACM Press, 1973, pp. 1–9.

[47] D. Tang, Y. Yu, D. P. Ranjan, and S. Malik, "Analysis of search based algorithms for satisfiability of quantified boolean formulas arising from circuit state space diameter problems," in *SAT '04: The Seventh International Conference on Theory and Applications of Satisfiability Testing*, May 2004, pp. 10–13.

[48] Xilinx Corporation, "Spartan-iie 1.8v fpga family: Function description," July 2003.

[49] ——, "Virtex-ii complete data sheet ver 3.3," 2004.

[50] ——, "Virtex-ii platform fpgas: Complete data sheet," June 2004.

[51] H. Yang and D. F. Wong, "Edge-map: Optimal performance driven technology mapping for iterative LUT based FPGA designs," in *IEEE International Conference on Computer-Aided Design*, Nov. 1994, pp. 150–155.

[52] S. Yang, "Logic synthesis and optimization benchmarks user guide version," 1991. [Online]. Available: citeseer.ist.psu.edu/yang91logic.html

[53] H. Zhang, "SATO: an efficient propositional prover," in *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, 1997, pp. 272–275. [Online]. Available: citeseer.ist.psu.edu/zhang97sato.html

[54] L. Zhang and S. Malik, "Conflict driven learning in a quantified boolean satisfiability solver," in *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design.* ACM Press, 2002, pp. 442–449.

[55] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Accelerating boolean satisfiability with configurable hardware," in *IEEE Symposium on FPGAs for Custom Computing Machines*, K. L. Pocek and J. Arnold, Eds. Los Alamitos, CA: IEEE Computer Society Press, 1998, pp. 186–195. [Online]. Available: citeseer.ist.psu.edu/zhong98accelerating.html