Multi-level Logic Synthesis Based on Function Decomposition

Kuo-Rueih Ricky Pan

CENG-96-12

Department of Electrical Engineering - Systems
University of Southern
Los Angeles, California 90089-2562
(213)740-4458

May 1996

Multi-level Logic Synthesis Based on Function Decomposition

by

Kuo-Rueih Ricky Pan

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(Electrical Engineering)

May 1996

Copyright 1996 Kuo-Rueih Ricky Pan

To my parents

Chao-Ming Charles Pan and Pi-Lee Alice Chen

Acknowledgments

I am indebted to my advisor Dr. Massoud Pedram for his continuous support, constant encouragement, and guidance. Without his help, this research would have not been possible. I would also like to thank Profs. Viktor K. Prasanna and Doug Ierardi for being on my dissertation and guidance committees, and Profs. Peter A. Beerel and Sandeep Gupta for being on my guidance committee.

I would like to express my thanks to my friend Yung-Te Lai for many fruitful discussions. I have also had the good fortune of meeting many wonderful people at USC. In particular, I would like to thank Chen-Huan Chiang, Chihshun Ding, Cheng-Ta Hsieh, Sasan Iman, Diana Marculescu, Radu Marculescu, Chun-Li Pu and Wei-Li Wang.

The funding for my research was provided by National Science Foundation under contract No. MIP-9111206 and MIP-9211668. I gratefully acknowledge that.

I am grateful to my father and mother who encouraged me to pursue academics.

Contents

List Of Tables vii

List Of Figures viii

1 Introduction 1

1.1 CAD Systems 1

1.2 Logic Synthesis 2

1.2.1 Logic Restructuring 2

1.2.2 Technology Mapping 4

1.3 Overview 5

2 Background 6

2.1 Function Decomposition 6

2.2 Ordered Binary Decision Diagrams (OBDDs) 9

2.3 OBDD-based Function Decomposition 11

2.3.1 Disjunctive Decomposition 14

2.3.2 Nondisjunctive Decomposition 15

3 Common Subfunction Extraction 19

3.1 Column Encoding 19

3.1.1 Output Grouping 26

3.2 Uni-code Shared Subfunction Encoding 28

3.2.1 Permissible G-functions 30

3.2.2 Minimum Subfunction Covering Problem 33
3.2.3 Minimum Support for G-functions 35
3.3 Multi-code Shared Subfunction Encoding 38
3.4 Common Subfunction Extraction for Large Bound Sets 42
3.4.1 Encoding Complexity 42
3.4.2 An Encoding Scheme for Large Compatible Class Size 44
3.4.3 Graph-based Encoding Using Multi-code Assignments 52
3.4.4 Extending Graph-Based Shared Subfunction Encoding . . . 55
3.4.5 Output Partitioning for Graph-based Encoding 56
3.5 Summary 59

4 Other Objective Functions 60

4.1 Decomposition for Minimum Delay 60
4.1.1 Minimum Delay Decomposition Using Unit Delay Model . . 60
4.1.2 Minimum Delay Decomposition Using Unit Fanout Delay Model 67
4.2 Decomposition for Minimum Energy Dissipation 73
4.2.1 Energy Consumption Model 73
4.2.2 Energy Consumption and Function Decomposition 74
4.2.3 Minimum Energy Decomposition 77
4.3 Energy-Delay Optimum Decomposition 81
4.4 Decomposition into Special Classes of Functions 84
4.4.1 Decomposition and Symmetric Functions 84
4.4.2 Decomposition and Unate Functions 88
4.5 Summary 89

5 Application to LUT-Based FPGA Synthesis 90

5.1 Field Programmable Gate Arrays 90
5.2 Overview of FGSyn 92

5.2.1 Node Clustering 93
5.2.2 Generating the Subject Network 95
5.3 Mapping for XC3000 Device 97
5.3.1 Look-up Table Merge 98
5.4 Mapping for XC4000 Device 98
5.4.1 Direct Decomposition 99
5.4.2 Two-Layer Decomposition 102
5.5 Mapping for XC5000 Device 107
5.5.1 Mapping Using Fixed Input Size LUTs 107
5.5.2 Mapping Using Variable Input Size LUTs 108
5.6 Summary 109

6 Experimental Results 110

6.1 Description of Benchmarks 110
6.2 OBDD-based Function Decomposition 112
6.3 Other Objective Functions Decomposition 118
6.4 Decomposition for Other Architectures of FPGA 124

7 Conclusion and Future Work 130

Appendix A

FGSyn 132

List Of Tables

3.1 *pg*-set size versus column_set size k 44

6.1 Description of benchmark circuits 111

6.2 Comparison between cube-based and OBDD-based decompositions . 112

6.3 Experimental results of FGSyn for XC3000 device 114

6.4 Runtime Comparison of FGSyn for XC3000 device 115

6.5 Large size benchmarks results for XC3000 device. 116

6.6 Experimental results for XC3000 device. 117

6.7 Experimental results of FGSyn_d 118

6.8 Delay minimum decomposition (unit delay vs. unit fanout delay model) 119

6.9 Energy comparison between mis-pga(new) and FGSyn_e 120

6.10 Energy-delay product comparison between mis-pga(new) and FGSyn_ed 121

6.11 Comparison of FGSyn_e and FGSyn_ed 122

6.12 Comparison of FGSyn, FGSyn_d, and FGSyn_e 123

6.13 Experimental results of FGSyn options for XC4000 device 125

6.14 Experimental results for XC4000 device 126

6.15 Experimental results of various bound sets for XC5000. 127

6.16 Energy comparison of various bound sets decomposition for XC5000. 128

List Of Figures

2.1 Function decomposition 7

2.2 The Karnaugh map and the decomposition chart 8

2.3 Function decomposition 9

2.4 A function represented in (a) OBDD and (b) decomposition chart. . 12

2.5 An example for operator *cut_vector* and *cut_set*. 14

2.6 OBDD function decompositions, B = {x1, x2, x3, x4, x5} 16

2.7 An example of non-disjunctive decomposition. 18

3.1 An example of multiple-output decomposition. 21

3.2 An example of multiple-output decomposition in OBDD representation. 25

3.3 Decomposition charts of f_1 and f_2 30

3.4 Resulting g-functions of two encodings 37

3.5 (Code) dependency graph 45

3.6 Partitioned graph 47

3.7 Conversion to bin packing problem 48

3.8 Dependency graph with large subgraph 53

3.9 Divisible item (multi-code assignment) 54

3.10 A connected acyclic bipartite graph 55

3.11 N-dimensional bin packing problem (N = 3) 57

4.1 Upper and lower bound delay estimation 61

4.2 Delay estimation 63

4.3 Delay minimization decomposition 65
4.4 Encoding effects node fanout 70
4.5 Delay minimization under unit fanout delay models 72
4.6 OBDD function decompositions with different encoding 76
4.7 Example of multiple-output function decomposition 81
4.8 Delay minimum vs. energy minimum decompositions 83
4.9 Symmetric property in OBDD 86

5.1 The FPGA structure 91
5.2 The LUT structure 92
5.3 The Xilinx XC3000 CLB 97
5.4 The Xilinx XC4000 CLB. 98
5.5 XC4000 patterns. 100
5.6 the non-disjunctive decomposition view of pattern (b). 100
5.7 Non-disjunctive mapping for the XC4000 device. 102
5.8 Graphical representation of type I two-layer decomposition. 103
5.9 The proof of type I two-layer decomposition. 104
5.10 Graphical representation of type II two-layer decomposition. 104
5.11 Conditions for type II two-layer decomposition. 106
5.12 The Xilinx XC5000 logic cell 108
5.13 The different configurations of XC5000 CLB 109

6.1 Selected benchmarks CLBs count for fix and variable bound set decomposition 129

Abstract

With the growing complexity of VLSI circuits, automatic synthesis of digital circuits has gained increasing importance. The synthesis process transforms an abstract representation of a circuit into an implementation in a target technology optimizing some objective function. One of the key steps in this process is logic synthesis, which produces an optimal gate level design from a register-transfer level description.

In this thesis, we describe a multi-level logic synthesis approach based on function decomposition. In particular, we present Boolean methods for extracting common subfunctions from multiple-output Boolean functions under different objectives including area, delay, energy, and energy-delay product. The extraction problem is cast as an encoding problem and a number of encoding methods are proposed. These methods include column encoding, shared subfunction encoding, and a graph-based approach for extracting logic with a large number of supporting variables. We use ordered binary decision diagrams to represent Boolean functions so that this approach can be implemented more efficiently.

Application of these methods to the synthesis of look-up table (LUT)-based field programmable gate arrays (FPGAs) is presented next. In many instances, we had to adapt the proposed extraction techniques to the FPGA architecture. For example, we used a two-layer decomposition technique to map to Xilinx XC4000 device and used variable input-size decomposition to map to Xilinx XC5000 device. These techniques produce results which are much better than state-of-the-art techniques in terms of area, delay, and power.

Optimal Clock Period FPGA Technology Mapping for Sequential Circuits*

Peichen Pan
Dept. of Electrical & Computer Eng.
Clarkson University
Potsdam, NY 13699

C. L. Liu
Dept. of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract – In this paper, we study the technology mapping problem for sequential circuits for LUT-based FPGAs. Existing approaches map the combinational logic between flip-flops (FFs) while assuming the positions of the FFs are fixed. We study in this paper a new approach to the problem, in which retiming is integrated into the technology mapping process. We present a polynomial time technology mapping algorithm that can produce a mapping solution with the *minimum* clock period while assuming FFs can be arbitrarily repositioned by retiming. The algorithm has been implemented. Experimental results on benchmark circuits clearly demonstrate the advantage of our approach. For many benchmark circuits, our algorithm produced mapping solutions with clock periods not attainable by a mapping algorithm based on existing approaches, even when it employs an optimal delay mapping algorithm for combinational circuits.

1 Introduction

A look-up table (LUT) based FPGA consists of an array of programmable logic blocks together with programmable interconnections [17]. The core of a programmable logic block is a k-input LUT (k-LUT) which can implement any combinational logic with up to k inputs and a single output, where k is a small positive integer. There are also several flip-flops (FFs) in each programmable logic block which can be connected to the inputs and outputs of its LUT to realize sequential behavior.

The technology mapping problem for LUT-based FPGAs is to produce, for a given circuit, an equivalent circuit comprised of LUTs. This problem has been studied extensively. However, almost all proposed mapping algorithms are designed for combinational circuits. Mapping algorithms for combinational circuits (will be referred to as combinational mapping algorithms from now on) have been proposed for different optimization criteria: performance [2, 6, 9, 18], area [4, 5, 7, 11, 16], routability [1, 14], and combinations of these [3, 13]. In particular, Cong and Ding [2] proposed an optimal delay combinational mapping algorithm for the unit delay model and Yang and Wong [18] proposed an optimal combinational mapping algorithm for the general delay model.

Existing approaches to technology mapping for sequential circuits use combinational mapping algorithms to map the combinational logic between FFs. These approaches have two obvious shortcomings: (i) failing to consider signal dependencies across FF boundaries, and (ii) not considering the possibility of exposing the combinational logic between FFs in different ways. Note that FFs in a sequential circuit can be repositioned by a technique called retiming [8]. Two recent sequential circuit technology mapping methods [10, 15] also assume the initial positions of the FFs are fixed, though retiming is used as a post-processing step in [15].

In this paper, we study a new approach to sequential circuit technology mapping, proposed in [12]. In this approach the FF positions are assumed to be fully dynamic in the sense that they can be arbitrarily repositioned by retiming. Our main objective is to obtain mapping solutions with minimized clock period, which is the maximum number of LUTs between any two successive FFs. We will present an efficient (polynomial time) algorithm that produces a minimum clock period mapping solution for any sequential circuit[1].

2 The new approach

To further motivate the new approach, let us examine two examples. Consider the circuit in Figure 1(a). Suppose that we want to map it to an FPGA architecture in which each LUT has at most 3 inputs. One possible mapping solution, without repositioning the FFs, is shown in Figure 1(a), where the gates enclosed by a dashed circle are mapped to one LUT. Figure 1(b) shows the mapping solution in terms of LUTs. This mapping solution uses two LUTs and has a clock period equal to two. Also note that the clock period of this mapping solution cannot be further reduced by retiming. Actually, it can be shown that any mapping solution must use at least two LUTs and have a clock period two no matter what combinational mapping algorithm is used. However, if gate b is retimed by a value one (the FF f at the output of b is moved to its inputs) as shown in Figure 1(c), all the gates can be mapped to one 3-LUT as shown in Figure 1(d). Note

*The work was partially supported by the National Science Foundation under grant MIP-9222408.

[1] The algorithm has been extended to the general delay model in which case, it produces a mapping solution with a clock period provably close to minimum.

33rd Design Automation Conference®
Permission to make digital/hard copy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 96 - 06/96 Las Vegas, NV, USA
©1996 ACM 0-89791-779-0/96/0006..$3.50

that this mapping solution has a clock period of one.

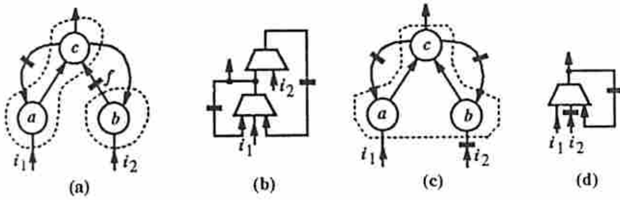


Figure 1: Advantage of retiming.

To fully exploit the potential of retiming, logic replication is necessary since replication can help produce mapping solutions which are otherwise impossible to obtain. Consider the circuit in Figure 2(a). Assume $k = 4$. It can be shown that any mapping solution must use at least six 4-LUTs and have a clock period at least two, even if retiming is used. However, if we duplicate a (to become a and a'), b (to become b and b'), and c (to become c and c'), then retime the FFs across gates a', b', and c' as shown in Figure 2(b), we can map all the gates (including the duplicated ones) to a single 4-LUT to obtain the mapping solution in Figure 2(c), which has a clock period of one.

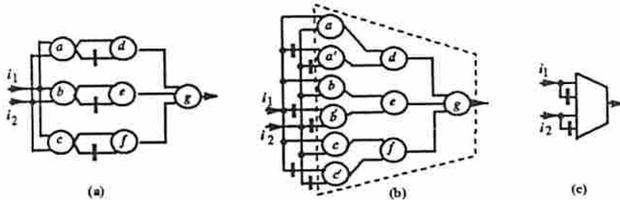


Figure 2: Advantage of logic replication.

Based on the above observations, we study the technology mapping problem in the most general setting in which the techniques of retiming and replication are exploited. Conceptually, the solution space that will be explored can be described by the diagram in Figure 3. Namely, the mapping solution space consists of all the circuits that can be obtained by retiming and replicating the given initial circuit, then mapping the combinational logic between FFs, followed by another retiming and replication step[2]. It is obvious that the solution space explored in this new approach is enormous since there are too many ways to retime and replicate a circuit.

3 Preliminaries and problem definition

A (synchronous) sequential circuit can be modeled as an edge-weighted directed (multi-)graph. The nodes are the primary inputs (PIs), the primary outputs (POs), and the

[2]A technology mapping algorithm based on existing approaches may try to alleviate its drawbacks by carrying out these conceptual steps in sequence. However, as long as it actually employs a combinational mapping algorithm, the same drawbacks are still there.

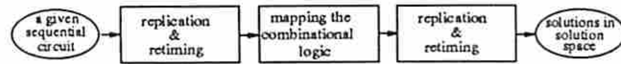


Figure 3: Solution space explored in the new approach.

combinational processing elements (PEs) in the circuit. (A PE is either a gate or a LUT depending on whether the circuit is the initial one or a mapping solution.) The edges are the interconnections. There is an edge e from u to v (denoted $u \xrightarrow{e} v$) with weight t if the output of u, after passing through t FFs, is an input to v. The *clock period* of a circuit is the maximum number of PEs on the combinational paths (paths without FFs) in the circuit.

Retiming is a technique of repositioning the FFs in a circuit without changing its functionality or the structure [8]. Retiming a node by a value i means removing i FFs from each fanout edge and adding i FFs to each fanin edge of the node. Figure 4 shows the case in which $i = 1$ or -1. In general, the nodes in a circuit can be retimed collectively (referred to as retiming the circuit). It can be shown that the retimed circuit and the original one have the same functionality if no retiming is performed at the PIs and POs (i.e., the retiming values for the PIs and POs are all zero).

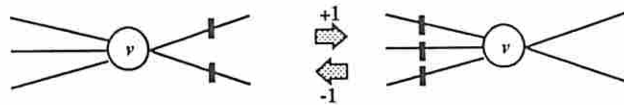


Figure 4: Retiming a node.

Refer again to Figure 3. We use N to denote the circuit to be mapped. We assume that N is *k-bounded*, namely, each node in N has at most k fanins. We will use $w(e)$ to denote the weight of an edge e in N. Let N' be a circuit obtained from N by replication and retiming and N'' be a mapping solution of the combinational logic of N'. Let S be the circuit obtained from N'' by putting the FFs back and followed by another retiming and replication. (Note that the PEs in N'' are LUTs.) S is then a *mapping solution* of N. The technology mapping problem addressed in this paper is as follows:

Problem 1 *Find a mapping solution with the minimum clock period.*

Finally, we list several graph-theoretic concepts. In a directed acyclic graph with one sink but possibly several sources, a *cut* $(X, \overline{X})$ is a partition of the nodes such that the sink is in $\overline{X}$ and all the sources are in X. The *edge-set* $E(X, \overline{X})$ of the cut is the set of edges from X to $\overline{X}$, the *node-set* $V(X, \overline{X})$ is the set of nodes in X that are connected to one or more nodes in $\overline{X}$. If $|V(X, \overline{X})| \leq k$, $(X, \overline{X})$ is called a *k-feasible* cut, or k-cut for short.

4 Formation of LUTs

In this section, we will present a method for forming LUTs for nodes in a sequential circuit.

Although the formation of LUTs is rather straightforward for combinational circuits, it is complicated for sequential circuits because a circuit may be arbitrarily retimed and replicated in the new approach. In other words, we are working with a family of circuits. To overcome this difficulty, we introduce the concept of *expanded circuits*. Our LUT formation procedure will be carried out on the expanded circuits.

An expanded circuit is constructed by replication and it has the property that all paths from any given node to the only output node have the same number of FFs.

The expanded circuits for a node v are defined recursively as follows: As the base case, the circuit with one node v^0 but no edge is an expanded circuit. Suppose $\mathcal{E}$ is an expanded circuit. Let I be the set of sources (nodes with indegree 0) in $\mathcal{E}$. We pick a node in I, say u^d. Then, for each edge $x \xrightarrow{e} u$ in N, add a node x^{d_1} where $d_1 = d + w(e)$ to $\mathcal{E}$ if it is not there, and add an edge $x^{d_1} \xrightarrow{e'} u^d$ with weight $w(e)$ to $\mathcal{E}$. The resulting circuit is also an expanded circuit.

An important class of expanded circuits consists of: $\mathcal{E}_v^i$, for $i \geq 0$. $\mathcal{E}_v^i$ denotes the expanded circuit in which the shortest distance (in terms of the number of edges) from each source that is not a replicate of a PI, to v^0 is i.

For the circuit in Figure 1(a), Figure 5 shows five expanded circuits for node c. From (a) to (e) each expanded circuit is constructed from the preceding one by expanding the shaded node. Actually, the circuit in (a) is $\mathcal{E}_v^0$, in (b) is $\mathcal{E}_v^1$, in (d) is $\mathcal{E}_v^2$, and in (e) is $\mathcal{E}_v^3$.

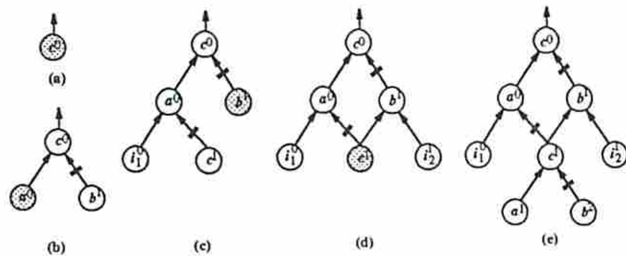


Figure 5: Expanded circuits.

We now show that a LUT for a node can be derived from a cut in the expanded circuits for the node. Let $(X, \overline{X})$ be a k-cut in an expanded circuit $\mathcal{E}$ for v. We first notice that all FFs inside $\overline{X}$ can be moved out by retiming. The retiming is: for each node u^d in $\overline{X}$, its retiming value is d; for all nodes in X, their retiming values are zero. Let $u^d \xrightarrow{e'} x$ be an edge in $E(X, \overline{X})$. (Note that u^d is a replicate of node u in N and e' is a replicate of edge e.) It can be verified that the number of FFs on e' is d after the retiming. The LUT derived from this cut is simply the subcircuit induced by the nodes in $\overline{X}$ with the FFs being removed. Let $\mathcal{L}$ denote the LUT. If u^d is in the node-set $V(X, \overline{X})$ of the cut, it means that u after passing through d FFs is an input to $\mathcal{L}$. As a result, the number of inputs to $\mathcal{L}$ is equal to the number of nodes in $V(X, \overline{X})$, which is k. Therefore, $\mathcal{L}$ is a k-LUT.

As an example, for the 3-cut indicated in the expanded circuit in Figure 5(d) as shown in Figure 6(a), Figure 6(b) shows the corresponding 3-LUT. The inputs to this 3-LUT are i_1, c passing through a FF, and i_2 passing through a FF.

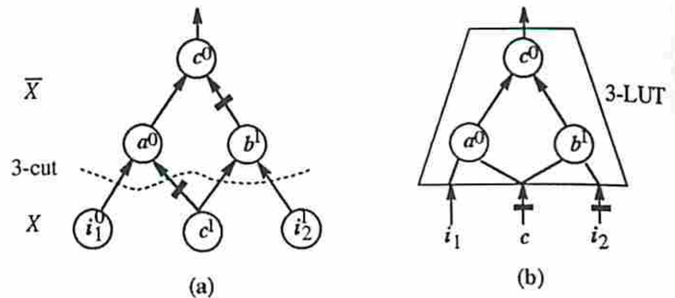


Figure 6: Derivation of a LUT from a cut.

Moreover, we can show that for any k-LUT there is a k-cut that can derive the LUT in this fashion, if the expanded circuit is $\mathcal{E}_v^{kn}$, where n is the number of nodes in N. Therefore, we have the following main result of this section:

Theorem 1 *It suffices to examine the k-LUTs for v that can be derived from the k-cuts in $\mathcal{E}_v^{kn}$.*

It can be shown the number of nodes in $\mathcal{E}_v^i$ is $O(ni)$ and the number of edges is $O(kni)$. In particular, the numbers of nodes and edges in $\mathcal{E}_v^{kn}$ are $O(kn^2)$ and $O(k^2n^2)$, respectively.

5 An algorithm for finding an optimal mapping solution

The way we solve Problem 1 is to solve its decision version as stated in the following:

Problem 2 *Given a target clock period ϕ, determine a mapping solution with a clock period of ϕ or less, whenever such a mapping solution exists.*

If we can solve Problem 2, we can do a binary search on ϕ to find a mapping solution with the minimum clock period.

We describe our algorithm for solving Problem 2 in this section. The algorithm has two phases: the labeling phase and the mapping phase. In the labeling phase, we compute a label (defined later) for each node in N. After we have computed all the labels and determined that there is a mapping solution with a clock period of ϕ or less, we then generate one such mapping solution in the mapping phase. In the next two subsections, we present the details of the two phases, separately.

5.1 The labeling phase

Let S be a mapping solution. We define a value (called l-value) for each LUT in S. To define the l-values, we use a graph whose topology is the same as that of S, and assign a weight $-\phi \cdot w_1(e)+1$ to an edge e, where $w_1(e)$ is the number of FFs on e in S. The *l-value* of a LUT in S is the maximum weight of the paths from the PIs to the LUT according to the new edge weights.

The *label* of a node in N is the *minimum* of the l-values of the k-LUTs for the node, generated according to Theorem 1.

For a node v in N, we will use $l^{opt}(v)$ to denote its label. We determine $l^{opt}(v)$ for each node v in N in this phase of the algorithm.

Our method for computing the labels is quite similar to a longest path algorithm. The approach is to compute a lower-bound on the value of each label and to repeatedly improve (increase) the lower-bound. The lower-bounds will be equal to the actual labels when no further improvement is observed for all the lower-bounds. Initially, the lower-bound for all PIs are zero and the lower-bounds for all other nodes are $-\infty$. Figure 7 shows the overall algorithm, where $l(v)$ denotes the lower-bound on $l^{opt}(v)$. Improving the lower-bounds is carried out by Procedure IMPROVE.

```
L_FIND(N, φ)
// V denotes the set of nodes in circuit N,
// w(e) denotes the number of FFs on edge e in N
1.    for each node v in V    // initialization
2.        if v is a PI
3.            then l(v) ← 0;
4.            else l(v) ← −∞;
5.    updated ← FALSE;
6.    for i ← 1 to |V|    // improve at most n times
7.        for each node v in V
8.            if IMPROVE(v) = TRUE, updated ← TRUE;
9.        if updated = FALSE, return success;
10.       updated ← FALSE;
11.   return failure;

IMPROVE(v)
a.    Determine l_new;
b.    if l(v) < l_new
c.        then
d.            l(v) ← l_new ;
e.            return TRUE;    // improved
f.    return FALSE;    // not improved
```

Figure 7: Algorithm for computing the labels.

The purpose of Procedure IMPROVE is to test whether we can improve the current lower-bound on the label of v, based on the current lower-bounds on all labels, and if so, to update the current lower-bound for v. l_{new} is the new lower-bound for v computed from the current lower-bounds.

Now the remaining issue is to determine l_{new}. Based on the discussion in Section 3, we have

$$l_{new} = \min_{(X,\overline{X})} \left(\max\{l(u) - \phi \cdot d + 1 \mid u^d \text{ is in } V(X,\overline{X})\}\right),$$

where the minimum is taken over all k-cuts in $\mathcal{E}_v^{kn}$.

We will use the above formula to compute l_{new}. Our approach is to study the corresponding decision problem, namely,

Problem 3 *Check whether $l_{new} \le L$ for a given integer L.*

We use network flow techniques to solve Problem 3. A flow network G is constructed from $\mathcal{E}_v^{kn}$ by applying to $\mathcal{E}_v^{kn}$ a standard network transformation, called *node-splitting* to reduce the problem of finding a k-cut to that of finding a cut with an edge capacity bound. To do so, each node in $\mathcal{E}_v^{kn}$ except v^0 is split into two nodes with a bridging edge between them. A supersource is added and connected to all the sources. The bridging edge for node u^d has capacity one if $l(u) - \phi \cdot d + 1 \le L$. All other edges in G has infinite capacity.

As an example, suppose for the circuit in Figure 1(a), we currently have $l(i_1) = l(i_2) = 0$, $l(a) = l(b) = 1$, and $l(c) = -\infty$, and the target clock period is one. In the expanded circuit for c in Figure 8(a), suppose we want to test whether $l_{new} \le 1$. For node b^1, $l(b) - \phi \cdot 1 + 1 = 1$, so the corresponding bridging edge has capacity one. On the other hand, for node a^0, $l(a) - \phi \cdot 0 + 1 = 2$, so the corresponding bridging edge has infinite capacity. Figure 8(b) shows the flow network, where the bridging edges for nodes i_1^0, i_2^1, c^1, and b^1 have unit capacity and all other edges have infinite capacity.

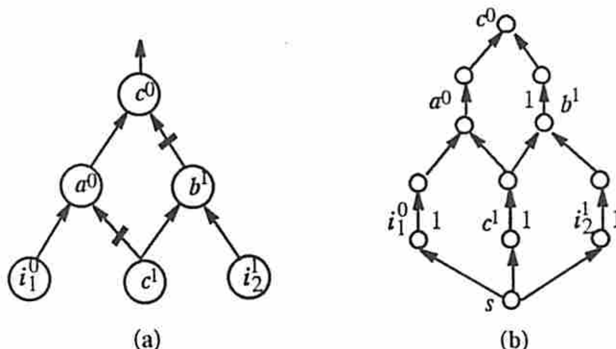


Figure 8: Construction of flow network.

The edge capacity of a cut is the sum of the capacities of the edges in the edge-set of the cut. The following result can be shown for the flow network G:

Lemma 1 *$l_{new} \le L$ iff G has a cut with edge capacity no more than k.* □

Based on the classical Max-flow Min-cut Theorem, G has a cut with edge capacity no more than k iff the maximum flow in G is at most k. We can, therefore, use an augmenting path algorithm for solving the max-flow problem to determine whether G has a cut with edge capacity no more than k in $O(k \cdot |E(G)|) = O(k^3n^2)$ time. Thus, we can determine whether $l_{new} \le L$ in $O(k^3n^2)$ time.

Obviously, l_{new} is from the following set

$$\{l(u) - \phi \cdot d + 1 \mid u^d \text{ is in } \mathcal{E}_v^{kn}\}$$

whose size is $O(kn^2)$, the number of nodes in $\mathcal{E}_v^{kn}$. We can first sort all the values in the set , and then use binary search to determine l_{new}. Overall, we have an $O(k^3n^2\log(kn))$-time algorithm for determining l_{new}.

L_FIND(N, ϕ) needs to call Procedure IMPROVE $O(n^2)$ times in the worst case. In summary, we have the following result:

Theorem 2 *The labels of all nodes in N can be determined in $O(k^3 n^4 \log(kn))$ time.* □

Remark: To guarantee that a mapping solution with the target clock period can always be found whenever there exists one, we need to use the expanded circuit $\mathcal{E}^{kn}$. This is the worst case scenario. In practice, we may use an expanded circuit $\mathcal{E}^i$ for an i considerably smaller than kn. For instance, for node c in the circuit in Figure 1(a), it can be shown that it is sufficient to use $\mathcal{E}_c^2$ (in Figure 5(d)) for examining the 3-LUTs for c. To make our algorithm flexible and to save computation time, we can use i as a control parameter so that the expanded circuit $\mathcal{E}^i$, instead of $\mathcal{E}^{kn}$ is used in Procedure IMPROVE.

5.2 The mapping phase

The purpose of this phase is to generate a mapping solution with a clock period of ϕ or less (if, of course, there is one such mapping solution).

The first step is to assemble a mapping solution from the LUTs corresponding to the cuts that realize the labels. To do so, we trace from the POs backward and to include those LUTs that are on paths from PIs to POs in the mapping solution. Specifically, we keep two lists D and U. D is the set of nodes in N whose k-LUTs have already been included in the partial mapping solution and U is the set of nodes whose k-LUTs are inputs to some k-LUTs in D and have not yet been included in the partial mapping solution. At the beginning, D consists of the PIs and U consists of the POs. At each iteration, a node v in U is removed and added to D. Let the k-LUT that realizes $l^{opt}(v)$ be $\mathcal{L}_v$ which is determined in the labeling phase. Then, if u after passing d FFs is an input to $\mathcal{L}$ we create an edge from $\mathcal{L}_u$ to $\mathcal{L}_v$ with weight d in S, and add u to U if it is not in D or U. This process stops when U becomes empty. Let S denote the resulting mapping solution. After the process is finished, D may not contain all the nodes in N. For those nodes not in D, they disappear because they are contained in some of the LUTs.

We now define a retiming r on S. For each LUT $\mathcal{L}_v$ in S, the retiming value is as follows:

$$r(\mathcal{L}_v) = \begin{cases} 0 & v \text{ is a PI or PO} \\ \lceil \frac{l^{opt}(v)}{\phi} \rceil - 1 & \text{otherwise.} \end{cases}$$

Let S_r denote the circuit obtained from S by applying retiming r. Note that by definition S_r is also a mapping solution of N. We have the following result:

Theorem 3 *The following three statements are equivalent:*

(i) *N has a mapping solution with a clock period of ϕ or less.*

(ii) *$l^{opt}(v) \leq \phi + 1$ for each PO v in N.*

(iii) *S_r has a clock period of ϕ or less.* □

Based on Theorem 3, we can check whether there is a PO whose label is larger than $\phi + 1$ after the labeling phase. If this is the case, the algorithm will not proceed to the mapping phase because there is simply no mapping solution with the target clock period ϕ. If for each PO, its label is less than or equal to $\phi + 1$, the algorithm simply return S_r since it meets the target clock period ϕ.

6 Experimental results

Our optimal clock period mapping algorithm has been implemented in the C language (referred to as SeqMapII). Experiments were carried out on sequential benchmark circuits in the LGSynth91 suite. In this section, we describe our experiments and summarize the results.

For comparison, we also implemented a technology mapping algorithm based on existing approaches which will be referred to as ComMap. ComMap maps a sequential circuit by mapping the combinational logic between FFs using FlowMap — a delay optimal technology mapping algorithm for combinational circuits [2]. ComMap also uses retiming as a pre-processing step as well as a post-processing step. Specifically, it retimes the initial circuit to the minimum clock period before applying FlowMap. It also retimes the mapping solution to the minimum clock period after FlowMap. The resulting circuit is then the output of ComMap.

We tested both ComMap and SeqMapII on a set of benchmark circuits using 5-LUTs. The results are summarized in Table 1. In Table 1, under column **initial** we list the number of gates and the number of FFs of each benchmark circuit (decomposed using *tech_decomp -a 2 -o 2* in **SIS**). Under column **ComMap**, we list the number of LUTs, the number of FFs, and the clock period (ϕ) of the mapping solution produced by ComMap. The same quantities are also listed for SeqMapII. For SeqMapII, we set the control parameter i, the depth of the expanded circuits used to form LUTs, to be 6 in the experiments. (Therefore, the clock periods of the mapping solutions produced by SeqMapII might not be minimum.) Even with such a small depth, SeqMapII consistently produced mapping solutions with smaller clock periods than that produced by ComMap as can be observed from the table. This clearly shows the advantage of the new approach. It can also be seen that the mapping solutions produced by SeqMapII usually have fewer LUTs than that produced by ComMap. This was also expected since SeqMapII can form LUTs by extending across FF boundaries. Overall, the mapping solutions produced by ComMap use 10% more LUTs, 33% larger clock periods, and 2% less FFs. For all the test circuits except s38417, the CPU times of our current implementation of SeqMapII were less than two minutes and in most cases only a few seconds on a SPARC 5 workstation with 32Mb memory. However, for s38417 it took SeqMapII close to 30 minutes due to the large size of the circuit and a larger reduction in the clock period. Overall, the CPU times of SeqMapII are about 10 times that of ComMap for the test circuits.

7 Conclusions

In this paper, we studied the FPGA technology mapping problem for sequential circuits in the most general setting. In our approach, retiming is fully integrated into the mapping process. As a result, the mapping solution space explored by our approach is much larger than what existing approaches are able to explore. Another way to understand our approach is that for our approach, there is no FF boundary at all, in the sense that if one circuit is obtained from another one by retiming, our algorithm will produce the same mapping solution for both circuits. In other words, where to place the

test circuit	Initial		ComMap			SeqMapII		
	gates	FFs	LUTs	FFs	ϕ	LUTs	FFs	ϕ
ex1	326	20	202	56	6	209	60	5
ex5	105	9	72	29	4	58	24	3
mult16a	261	16	75	58	3	38	55	2
mult32a	533	32	153	202	3	78	207	2
s344	109	15	50	40	3	36	36	2
s349	112	15	49	39	3	33	33	2
s382	148	21	73	49	3	64	43	2
s400	158	21	72	47	3	66	46	2
s444	169	21	77	51	3	64	43	2
s526	252	21	166	84	3	127	89	2
s526n	251	21	166	85	3	140	97	2
s953	348	29	196	61	5	202	54	4
s1488	734	6	339	63	5	266	25	4
s9234	2352	193	590	270	5	593	276	4
s15850	3852	522	1670	704	9	1627	818	8
s38417	8709	1583	4170	2503	8	3761	2507	6
Total			8120	4341	69	7362	4413	52
Ratio			1.10	.98	1.33	1	1	1

Table 1: Experimental results.

FFs in a circuit has no effect on our algorithm. On the other hand, for a mapping algorithm based on existing approaches, there always exist FF boundaries and signal dependencies across FF boundaries are severed. We further presented a polynomial mapping algorithm which can produce mapping solutions with the minimum clock periods.

References

[1] N. Bhat and D. Hill. Routable technology mapping for FPGAs. In *ACM/SIGDA Workshop on FPGAs*, pages 143–148, 1992.

[2] J. Cong and Y. Ding. FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Trans. on Computer-Aided Design*, 13:1-11, 1994.

[3] J. Cong and Y. Ding. On area/depth trade-off in LUT-based FPGA technology mapping. *IEEE Trans. on VLSI Systems*, 2:137–148, 1994.

[4] A. H. Farrahi and M. Sarrafzadeh. Complexity of the lookup-table minimization problem for FPGA technology mapping. *IEEE Trans. on Computer-Aided Design*, 13:1319–1332, 1994.

[5] R. J. Francis, J. Rose, and Z. Vranesic. Chortle-crf: Fast technology mapping for lookup table-based FPGAs. In *ACM/IEEE Design Automation Conf. (DAC)*, pages 227–233, 1991.

[6] R. J. Francis, J. Rose, and Z. Vranesic. Technology mapping for lookup table-based FPGAs for performance. In *Intl. Conf. on Computer-Aided Design (ICCAD)*, pages 568–571, 1991.

[7] K. Karplus. Xmap: A technology mapper for table-lookup FPGAs. In *ACM/IEEE Design Automation Conf. (DAC)*, pages 240–243, 1991.

[8] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing synchronous circuitry by retiming. In *Proc. 3rd Caltech Conf. on VLSI*, pages 87–116, 1983.

[9] A. Mathur and C. L. Liu. Performance driven technology mapping for lookup-table based FPGAs using the general delay model. In *ACM/SIGDA Workshop on Field Programmable Gate Arrays*, 1994.

[10] R. Murgai, R.K. Brayton, and A. Sangiovanni-Vincentelli. Sequential synthesis for table look up programmable gate arrays. In *ACM/IEEE Design Automation Conf. (DAC)*, pages 224–229, 1993.

[11] R. Murgai, N. Shenoy, R.K. Brayton, and A. Sangiovanni-Vincentelli. Improved logic synthesis algorithms for table look up architectures. In *Intl. Conf. on Computer-Aided Design (ICCAD)*, pages 564–567, 1991.

[12] P. Pan and C. L. Liu. Technology mapping of sequential circuits for LUT-based FPGAs for performance. In *ACM/SIGDA Intl. Symposium on Field-Programmable Gate Arrays*, pages 58–64, 1996.

[13] P. Sawkar and D. Thomas. Area and delay mapping for table-look-up based field programmable gate arrays. In *ACM/IEEE Design Automation Conf. (DAC)*, pages 368–373, 1992.

[14] M. Schlag, J. Kong, and P.K. Chan. Routability-driven technology mapping for lookup table-based FPGA's. *IEEE Trans. on Computer-Aided Design*, 13:13–26, 1994.

[15] U. Weinmann and W. Rosenstiel. Technology mapping for sequential circuits based on retiming techniques. In *Proc. European Design Automation Conf.*, pages 318–323, 1993.

[16] N.-S. Woo. A heuristic method for FPGA technology mapping based on the edge visibility. In *ACM/IEEE Design Automation Conf. (DAC)*, pages 248–251, 1991.

[17] Xilinx. *The Programmable Gate Arrays Data Book*. Xilinx, San Jose, CA, 1993.

[18] H. Yang and D. F. Wong. Edge-Map: Optimal performance driven technology mapping for iterative LUT based FPGA designs. In *Intl. Conf. on Computer-Aided Design (ICCAD)*, pages 150–155, 1994.

Structural Gate Decomposition for Depth-Optimal Technology Mapping in LUT-based FPGA Design

Jason Cong and Yean-Yow Hwang

Department of Computer Science
University of California, Los Angeles

Abstract

In this paper, we study the problem of decomposing gates in fanin-unbounded or K-bounded networks such that the K-input LUT mapping solutions computed by a depth-optimal mapper have minimum depth. We show (1) any decomposition leads to a smaller or equal mapping depth regardless the decomposition algorithm used, and (2) the problem is NP-hard for unbounded networks when $K \geq 3$ and remains NP-hard for K-bounded networks when $K \geq 5$. We propose a gate decomposition algorithm, named DOGMA, which combines level-driven node packing technique (Chortle-d) and the network flow based optimal labeling technique (FlowMap). Experimental results show that networks decomposed by DOGMA allow depth-optimal technology mappers to improve the mapping solutions by up to 11% in depth and up to 35% in area comparing to the mapping results of networks decomposed by other existing decomposition algorithms.

1. Introduction

The lookup-table (LUT) based FPGAs have been a popular technology for VLSI ASIC design and system prototyping. A K-input LUT (K-LUT) can implement any function of up to K variables. The goal of LUT-based FPGA technology mapping is to cover a given network using LUTs such that either area or delay is minimized or routability is maximized in the final LUT network. The delay of a network can be estimated by the number of levels (i.e. depth). Two factors affect the mapping solution depth: the gate decomposition before mapping and the mapping algorithm. Several LUT mapping algorithms have been proposed for depth minimization [6, 9, 2]. In particular, the FlowMap algorithm [2] guarantees a depth-optimal mapping solution for any K-bounded network. However, FlowMap can not be applied directly to unbounded networks. Gate decomposition can be classified into structural decomposition and Boolean decomposition. The structural decomposition replaces multi-fanin (simple) gates by fanin trees while the Boolean decomposition decomposes the functionality of gates. This paper focuses on structural decomposition for depth minimization in LUT mapping.

33rd Design Automation Conference®
Permission to make digital/hard copy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 96 - 06/96 Las Vegas, NV, USA
©1996 ACM 0-89791-779-0/96/0006..$3.50

Gate decomposition affects the mapping solution depth significantly. For example, assume $K=3$. The network N in Figure 1(a) is not K-bounded. If node v is decomposed in the way shown in Figure 1(b), there is no way to obtain a mapping solution of depth less than 3. However, if the decomposition shown in Figure 1(c) is carried out for node v, a mapping solution of depth equal to 2 can be obtained. Even for K-bounded networks, the depth of mapping solutions computed by FlowMap may decrease if gates are *further* decomposed before mapping [4].

Several gate decomposition routines have been used for LUT-mapping. The *tech_decomp* and the *speed_up* in SIS [10] and the *dmig* in [1] focus on minimizing the number of levels in the decomposed network. They do not directly minimize the depth of the *mapping solution.* Chortle-d [6] computes depth-optimal gate decomposition and mapping solutions for tree networks (may be unbounded) but produces suboptimal results for general networks. In this paper, we study the structural gate decomposition problem to decompose gates in a general network such that a mapping solution of minimum depth can be obtained.

The remainder of this paper is organized as follows. Section 2 defines basic terminology, presents properties of structural gate decomposition and formulates the problem. Section 3 gives the complexity results. A novel gate decomposition algorithm for depth-optimal mapping is presented in Section 4. Experimental results are presented in Section 5 and Section 6 concludes the paper.

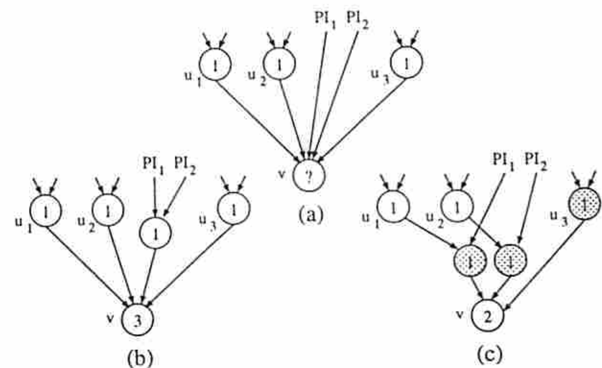


Figure 1 Decomposition of node v ($K=3$). (a) initial network, (b) decomposition yielding mapping depth = 3, (c) decomposition yielding mapping depth = 2.

2. Problem Formulation

Let K be the input size of an LUT. Let $input(v)$ be the set of fanin nodes of node v. A primary input (PI) node has no fanins and a primary output (PO) node has no outgoing edges. A network N is K-bounded if every node $v \in N$ satisfies $|input(v)| \le K$. Otherwise, it is an unbounded network. Given a subnetwork H, we use $input(H)$ to denote the set of distinct nodes outside H which supply inputs to nodes in H. Given a node v in network N, let N_v denote the subnetwork consisting of node v and all the predecessors of v. The *minimum mapping depth* of v in N, denoted $MMD_N(v)$, is defined as the minimum depth among all possible K-LUT mapping solutions of N_v. If N_v is unbounded, let $MMD_N(v) = \infty$. PI nodes have a mapping depth of 0. The minimum mapping depth of a network N, denoted $MMD(N)$, is the largest mapping depth among all PO nodes. Given a K-bounded network N, the FlowMap [2] algorithm computes $MMD_N(v)$ for every node $v \in N$ in polynomial time. A cut in N_v is a partition $(X_v, \bar{X}_v)$ of nodes in N_v such that PI nodes are in X_v and $v \in \bar{X}_v$. The cutset of a cut, denoted $n(X_v, \bar{X}_v)$, is defined as $input(\bar{X}_v)$. A cut is K-feasible if $|n(X_v, \bar{X}_v)| \le K$. The height of a cut, denoted $height(X_v, \bar{X}_v)$, is the maximum mapping depth for nodes in $n(X_v, \bar{X}_v)$. We have the following lemma based on results in [2].

Lemma 1 A node v has $MMD_N(v) = p$ if there is a K-feasible cut of height of $p-1$ in N_v but no K-feasible cut of height of $p-2$ or smaller exists.

Let node $v \in N$ satisfies $|input(v)| > 2$. Given a decomposition algorithm D, we define a *decomposition step* at v by D, denoted D_v, as follows: Decomposition step D_v (i) chooses two nodes u_1 and u_2 from $input(v)$, (ii) removes edges (u_1, v) and (u_2, v), (iii) introduces a new node w and new edges (u_1, w), (u_2, w), (w, v) and adds them to N. The resulting network is denoted as $D_v(N)$. For example, Figure 2(b) shows the result of one decomposition step at node v from Figure 2(a). Obviously, the introduced node w have the same gate type as v. We present two properties of the structural gate decomposition.

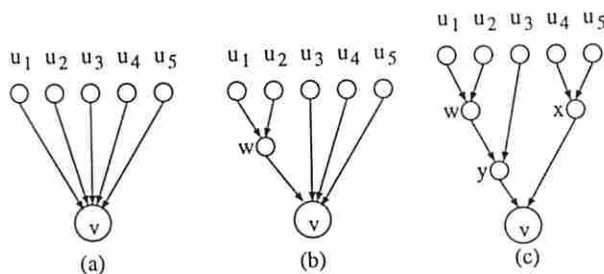


Figure 2 Decomposition of node v. (a) Before decomposition, (b) $D_v(N)$ after one decomposition step of D_v, (c) after a sequence of decomposition steps.

Lemma 2 Given a network N, any decomposition algorithm D, and any node $v \in N$, it must be true that $MMD(D_v(N)) \le MMD(N)$.

Lemma 3 If $MMD_N(u) = MMD_N(v)$ for all $u \in input(v)$ in a K-bounded network N, then $MMD(N) = MMD(D_v(N))$ for any decomposition algorithm D.

According to Lemma 2, the further a network is decomposed, the smaller the mapping depth might be. Therefore, we decompose every gate into a binary fanin tree. Figure 2(c) is a complete decomposition of node v. Every decomposition step introduces one intermediate node and it requires $|input(v)| - 2$ steps to decompose v. We formulate the following problems.

Structural Gate Decomposition for K-LUT Mapping (SGD/K) Given a simple-gate unbounded network N_∞, decompose N_∞ into a 2-input network N_2 such that for any other 2-input network decomposition N'_2 of N, $MMD(N_2) \le MMD(N'_2)$.

Structural Gate Decomposition for K-LUT Mapping of K-bounded Network (K-SGD/K) Given a simple-gate K-bounded network N_K, decompose N_K into a 2-input network N_2 such that for any other 2-input network decomposition N'_2 of N, $MMD(N_2) \le MMD(N'_2)$.

There are two issues related to the problem of gate decomposition. (1) A smaller depth might be obtained when several gates are decomposed *simultaneously* instead of *independently* [4]. This is because the intermediate nodes could be shared during the decomposition of multiple gates of the same functional type. (2) Gate decomposition can be performed before the mapping phase in a *two-step* approach or embedded into the mapping process being part of an *integrated* approach. For example, Chortle-d [6] is an integrated approach (since it decomposes gates and maps LUTs in an interleaving manner) while *dmig* + FlowMap in [2] uses a two-step approach. We can show that the *best* two-step approach produces the same optimal mapping depth as that by the *best* integrated approach [4]. In this paper, we consider only independent gate decompositions in a two-step approach. Nevertheless, our gate decomposition algorithm takes into account the impact of gate decomposition on mapping to obtain a decomposed network which is most suitable for FlowMap to achieve a minimum mapping depth.

3. Complexity of SGD/K and K-SGD/K Problems

In this section, we only state our complexity theorems. Complete proofs can be found in [4].

Theorem 1 The SGD/K problem is NP-hard for $K \ge 3$.

Theorem 2 The K-SGD/K problem is NP-hard for $K \ge 5$.

4. Gate Decomposition Algorithm for Depth-Optimal LUT Mapping

Our decomposition algorithm, named DOGMA (Depth-Optimal Gate decomposition for MApping),

combines the level-driven node packing technique in Chortle-d and the network flow based labeling technique in FlowMap. Given a network N, DOGMA decomposes nodes from PIs to POs in a topological order. Let $N(v)$ denote the network after decomposing the node v. DOGMA labels each node v by $MMD_{N(v)}(v)$ as follows. Nodes in $input(v)$ are grouped in such a way that each group consists of nodes with the same label (i.e. minimum mapping depth). Groups are processed in an ascending order according to their labels. For a group of nodes labeled p, nodes are packed into a minimum number of bins such that a K-feasible cut of height $p-1$ exists for the nodes in each bin (checked based on Lemma 1). Such a bin is called a *min-height K-feasible bin*. A node u_i is created for each bin B_i with fanins from nodes in B_i and a fanout to v. Node u_i will be given a label p. Note that according to Lemma 3, no matter u_i is further decomposed or not, the minimum mapping depth of the network is always the same. We then proceed to the group of a next higher label $p+1$. For each node u_i created in the previous step, a buffer node w_i (with label $p+1$) is created with u_i as input. (All the buffer nodes will be removed after decomposition). These buffer nodes together with nodes in the group of label $p+1$ are again packed into a minimal number of min-height K-feasible bins. We continue this process until all nodes are packed into one bin which corresponds to the node v.

DOGMA is similar to Chortle-d in that decomposition is done by packing nodes into a *minimal* number of min-height K-feasible bins. However, Chortle-d integrates gate decomposition with technology mapping, and computes mapping depth based on the partially generated LUT network. Since the fanin constraint is not a *monotone* clustering constraint [2], Chortle-d may obtain inaccurate node mapping depth. Besides, Chortle-d enumerates all packing combinations for nodes on reconvergent paths, which is quite expensive. In contrast, DOGMA computes mapping depth as well as packs nodes into bins (to be discussed) using the network flow based computation. The mapping depth is always accurate and the reconvergent paths are taken into account naturally.

The problem remains to be solved is the min-height K-feasible bin packing problem defined as follows.

Min-Height K-Feasible Bin Packing Problem Given a set $S_p \subseteq input(v)$ of nodes of minimum mapping depth p when decomposing node v, partition S_p into a minimal number of bins such that there is a K-feasible cut of height $p-1$ for nodes in each bin.

We shall give three heuristics and one exact method to solve the problem. Our heuristics are based on the max-flow algorithm and bin-packing heuristics. We define the total cut size $TC_p(S)$ of a set S of nodes with label p to be the size of the min-cut of height $p-1$ which separates S from all PIs. A set X of nodes of label p can be packed into one bin as long as $TC_p(X) \leq K$. Nodes are packed in a decreasing order of individual cut sizes. The first two heuristics, named MC-FFD and MC-BFD, to the min-height K-feasible bin packing problem are based on the first-fit-decreasing (FFD) and best-fit-decreasing (BFD), which are two heuristics for the bin packing problem [8]. The third method is the maximal-share-decreasing (MC-MSD) heuristic which packs nodes that can maximally share a cut together. The fourth method is inspired by the dynamic programming approach for the number partitioning problem [7]. Instead of partitioning numbers, we ask whether there is a way to partition the nodes in S_p into k subsets $X_1, X_2, \cdots, X_k$ such that $TC_p(X_i) \leq K$ $(1 \leq i \leq k)$. We can solve the problem by dynamic programming. By searching the minimal k $(k=2,3,\cdots)$, the min-height K-feasible bin packing problem can be solved optimally. We refer to this method as the MC-DP algorithm. The details of these algorithms can be found in [4].

5. Experimental Results

We have implemented the DOGMA algorithm with MC-FFD, MC-BFD, MC-MSD, and MC-DP packing methods using the C language and incorporated our implementation into the RASP FPGA synthesis system [5]. In our experiments, we optimize the MCNC benchmark circuits for area using standard SIS scripts, decompose them into simple gate networks, apply gate decomposition routines to obtain 2-input networks, and obtain the final LUT networks using a depth-optimal technology mapper. We choose $K=5$ in the experiments.

We compare the performance of our four methods for the min-height K-feasible bin packing in DOGMA and observe that the impact of the four methods on mapping results is almost the same. Since MC-FFD is faster than other three methods, DOGMA employs MC-FFD to solve the packing problem. We compare DOGMA with two decomposition routines: the *tech_decomp* in SIS [10] and the *dmig* in [1]. The *tech_decomp* routine is based on a balanced-tree heuristic which only minimizes the gate level locally. The *dmig* routine minimizes the gate level of the decomposed networks. The decomposed networks by the three algorithms are all mapped by CutMap [3], an enhancement of FlowMap. The results are shown in Table 1. We see CutMap produces the same or smaller depth for circuits decomposed by DOGMA. On average, DOGMA allows CutMap to achieve 10% and 4% depth reduction comparing to *tech_decomp* and *dmig*, respectively.

We compare DOGMA + CutMap with existing gate decomposition and depth-oriented mapping algorithms. The tested circuits are area-optimized MCNC benchmarks. The mappers TechMap-D [9], FlowMap [2], and CutMap [3] are used for comparison. The *dmig* was used in [2] while the *speed_up* was used in [9] and [3] to prepare 2-input networks for technology mapping. The results are in Table 2. Comparing results from [2, 3] with ours, we see gate decomposition routines *speed_up*, *dmig*, and DOGMA decompose gates equally well in terms of mapping depth. However, networks decomposed by DOGMA allow CutMap to reduce 16% of LUTs in the mapping solutions.

Circuit	*tech_decomp*		*dmig*		DOGMA	
	LUT	*d*	LUT	*d*	LUT	*d*
5xp1	24	3	23	3	24	3
9sym	66	5	66	5	59	5
apex2	154	6	155	5	151	5
apex4	770	7	792	6	770	5
clip	37	4	37	4	38	3
con1	3	2	3	2	3	2
duke2	160	5	173	4	177	4
e64	108	9	108	9	108	9
misex1	18	2	18	2	16	2
misex2	32	3	31	3	36	2
misex3	179	17	174	16	176	16
rd73	25	3	27	3	23	3
rd84	52	4	52	4	54	4
sao2	47	4	44	4	42	4
vg2	23	4	29	3	29	3
Total	1698	78	1732	73	1706	70
	-0.5%	+11%	+2%	+4%	1	1

Table 1 Comparison of *tech_decomp*, *dmig* and DOGMA.

In [9], TechMap-D obtained the smallest depth because it integrated logic synthesis into technology mapping. However, 35% more LUTs are generated comparing to DOGMA + CutMap.

	[9]	[2]	[3]	Ours
	speed_up	*dmig*	*speed_up*	DOGMA
	TechMap-D	FlowMap	CutMap	CutMap
Circuit	LUT(*d*)	LUT(*d*)	LUT(*d*)	LUT(*d*)
5xp1	17 (2)	22 (3)	23 (3)	24 (3)
9sym	9 (3)	60 (5)	62 (5)	59 (5)
9symml	9 (3)	55 (5)	58 (5)	50 (4)
C499	148 (4)	68 (4)	143 (5)	68 (4)
C880	213 (7)	124 (8)	205 (8)	98 (8)
alu2	197 (8)	155 (9)	144 (8)	138 (9)
apex6	252 (5)	238 (5)	233 (4)	231 (5)
apex7	86 (4)	79 (4)	80 (4)	68 (4)
count	71 (4)	31 (5)	69 (3)	31 (5)
des	1395 (8)	1310 (5)	986 (5)	938 (5)
duke2	175 (4)	174 (4)	178 (4)	173 (4)
misex1	18 (2)	16 (2)	15 (2)	16 (2)
rd84	16 (3)	46 (4)	45 (4)	53 (4)
rot	315 (6)	234 (7)	239 (6)	210 (7)
vg2	36 (4)	29 (3)	39 (4)	27 (3)
z4ml	9 (2)	5 (2)	12 (3)	5 (2)
Total	2966(69)	2646(75)	2531(73)	2189(74)
	+35%(-7%)	+21%(+1%)	+16%(-1%)	1 (1)

Table 2 Comparison of DOGMA + CutMap with previous results.

6. Conclusion

In this paper, we study the structural gate decomposition for depth-optimal LUT mapping. We show gate decomposition leads to a smaller or equal mapping depth regardless the decomposition algorithm used, and the problem is NP-hard for unbounded networks when $K \geq 3$ and remains NP-hard for K-bounded networks when $K \geq 5$. We propose a gate decomposition algorithm (DOGMA) for depth-optimal mapping. Experimental results show a reduction of up to 10% in mapping depth. Together with CutMap, we achieve comparable mapping depth with up to 35% reduction in area comparing to previous results.

Acknowledgement

This work is partially supported by NSF Young Investigator (NYI) Award MIP-9357582, and grants from Xilinx, Xerox PARC, and AT&T Microelectronics under NSF NYI and California MICRO programs. The authors would like to thank Dr. Robert Francis for his helpful discussions.

References

[1] Chen, K. C., J. Cong, Y. Ding, A. B. Kahng, and P. Trajmar, "DAG-Map: Graph-based FPGA Technology Mapping for Delay Optimization," *IEEE Design and Test of Computers*, pp. 7-20, Sep. 1992.

[2] Cong, J. and Y. Ding, "An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs," *IEEE Trans. on Computer-Aided Design*, Vol. **13**, pp. 1-12, Jan. 1994.

[3] Cong, J. and Y.-Y. Hwang, "Simultaneous Depth and Area Minimization in LUT-Based FPGA Mapping," *Proc. ACM 3rd Int'l Symp. on FPGA*, pp. 68-74, Feb. 1995.

[4] Cong, J. and Y.-Y. Hwang, "Structural Gate Decomposition for Depth-Optimal Technology Mapping in LUT-based FPGA," in *UCLA Computer Science Dept. Tech. Report CSD-950045*, (December 1995).

[5] Cong, J., J. Peck, and Y. Ding, "RASP: A General Logic Synthesis System for SRAM-based FPGAs," *Proc. ACM 4th Int'l Symp. on FPGA*, Feb. 1996.

[6] Francis, R. J., J. Rose, and Z. Vranesic, "Technology Mapping of Lookup Table-Based FPGAs for Performance," *Proc. IEEE Int'l Conf. on CAD*, pp. 568-571, Nov. 1991.

[7] Garey, M. and D. Johnson, *Computer and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco (1979).

[8] Horowitz, E. and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Potomac, Maryland (1978).

[9] Sawkar, P. and D. Thomas, "Performance Directed Technology Mapping for Look-Up Table Based FPGAs," *Proc. 30th ACM/IEEE Design Automation Conf.*, pp. 208-212, June 1993.

[10] Sentovich, E., K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephen, R. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," *U.C.Berkeley Technical Report UCB/ERL M92/41*, May, 1992.

A Boolean Approach to Performance-Directed Technology Mapping for LUT-Based FPGA Designs

Christian Legl* Bernd Wurth** Klaus Eckl*
*Institute of Electronic Design Automation, Technical University of Munich, 80290 Munich, Germany
**Synopsys, Inc., 700 E. Middlefield Rd., Mountain View, CA 94043

Abstract — This paper presents a novel, Boolean approach to LUT-based FPGA technology mapping targeting high performance. As the core of the approach, we have developed a powerful functional decomposition algorithm. The impact of decomposition is enhanced by a preceding collapsing step. To decompose functions for small depth and area, we present an iterative, BDD-based variable partitioning procedure. The procedure optimizes the variable partition for each bound set size by iteratively exchanging variables between bound set and free set, and finally selects a good bound set size. Our decomposition algorithm extracts common subfunctions of multiple-output functions, and thus further reduces area and the maximum interconnect lengths. Experimental results show that our new algorithm produces circuits with significantly smaller depths than other performance-oriented mappers. This advantage also holds for the actual delays after placement and routing.

1 Introduction

An important class of FPGAs is based on the lookup-table (LUT) as the basic programmable logic block. A k-LUT implements any Boolean function of up to k variables. The LUTs are wired by various kinds of programmable interconnects [1]. Minimizing the delay of LUT-based FPGA designs is an important task because the programmable interconnects introduce extra delay compared with conventional gate array or standard cell technologies. The performance of an FPGA design is determined by the number of LUTs and the interconnect delays on the critical path.

Performance-driven technology mapping for LUT-based FPGAs is to transform a Boolean network, which has been produced in the technology independent logic optimization phase, into a functionally equivalent LUT network with minimum circuit delay. Technology mapping of a Boolean network is usually performed in two steps: The first step is the decomposition of nodes with more than k inputs into smaller nodes with k or less inputs. The resulting network is called *k-bounded*. A subsequent covering step finds a circuit of LUTs covering the k-bounded network.

A variety of technology mapping algorithms tackle performance optimization by minimizing the depth of k-bounded networks. Chortle-d, which is based on tree decomposition and bin-packing, is depth-optimal for trees [2]. DAG-Map heuristically minimizes the depth of a general k-bounded network [3]. The FlowMap algorithm is a significant advance since it guarantees a depth-optimal covering for general k-bounded networks [4].

However, minimizing the network depth does not consider the interconnect delays. Since the maximum interconnect length on the chip is correlated with the circuit area, minimizing the area also contributes to delay minimization. This is considered in the FlowMap-r [5] algorithm, which achieves depth-optimal mappings as FlowMap but reduces the number of LUTs. Recent algorithms improve on the FlowMap algorithm by modeling interconnect delay more accurately. This is done by assigning different delays to different nets (*nominal delay model* [6]) or even to the different interconnections of the same net [7]. The combination of the technology mapping and layout synthesis phases has been proposed to achieve small interconnect delays and improve routability [8, 9].

Another class of performance-directed algorithms performs Boolean operations during the covering step. Collapsing of critical nodes and re-decomposition are employed in MIS-pga(delay) [8], TechMap-D [10] and FlowSYN [11]. The FlowMap-based FlowSYN algorithm, which uses an efficient functional decomposition method, outperforms FlowMap-r in terms of circuit depth and area.

All these performance-directed mapping algorithms concentrate on the covering step. To obtain a k-bounded network in the decomposition step, the DMIG algorithm [3] and SIS-algorithms [12] like *xl_k_decomp*, *speed_up*, and *tech_decomp* are used. The decomposition step typically yields a network in which each gate has at most two inputs. This maximizes the flexibility during the covering step. A reason for the little attention given to the decomposition step is the fact that technology independent logic optimization generates Boolean networks with relatively small nodes. Thus, decomposition has only local effect, whereas a state-of-the-art covering step can deal with the entire network and therefore dominates the final result.

In this paper, we present a novel, Boolean approach to performance-directed technology mapping. As a first step, delay-driven collapsing is performed. In contrast to previous mapping approaches, large network portions are collapsed such that the decomposition step can have a significant impact. As the core of our approach, we have developed a powerful, functional decomposition algorithm that creates k-bounded networks with small depth and area. The main components of the decomposition algorithm are: First, a BDD-based, iterative variable partitioning procedure that efficiently evaluates a large number of variable partitions for their effect on circuit depth and area. Second, we have developed cost functions that estimate the delay and area of Boolean functions after decomposition and determine the effectiveness of the variable partitioning procedure. Third, we use a multiple-output decomposition approach which extracts common subfunctions.

The rest of the paper is organized as follows. Section 2 reviews the state of the art in functional decomposition. In Section 3, we present our performance-directed variable partitioning procedure. Section 4 describes the overall approach. We show experimental results in Section 5, and conclude the paper in Section 6.

2 Review of Functional Decomposition

Single-output decomposition. We first review the functional single-output decomposition which is based on the theory of Curtis [13], Roth and Karp [14]. Functional single-output decomposition breaks a function $f(\mathbf{x},\mathbf{y})$ into the composition function $g(\mathbf{v},\mathbf{y})$ and the subfunction vector $\mathbf{d}(\mathbf{x}) = (d_1(\mathbf{x}),\ldots,d_c(\mathbf{x}))$ such that $f(\mathbf{x},\mathbf{y}) = g(\mathbf{d}(\mathbf{x}),\mathbf{y})$. We deal with disjoint decompositions, where the bound set $BS = \{x_1,\ldots,x_b\}$ and the free set $FS = \{y_1,\ldots,y_{n-b}\}$ are disjoint sets where b is the size of the bound set and n is the number of inputs of f. In non-trivial decompositions, composition function g as well as the subfunctions d_i have fewer inputs than the original function f. Therefore, functional decomposition can be recursively used to compute k-bounded networks.

Two problems must be solved to perform functional decomposition. First, the input variables of f must be partitioned into the bound set and the free set. This is the *variable partitioning problem*. Second, given a variable partition, a minimum number c of subfunctions d_i must be computed. This number c depends on the variable partition.

We deal with the second problem first. To compute subfunctions d_i, the notion of *compatible* BS-vertices was introduced [14]. Two BS-vertices $\hat{\mathbf{x}}_v \in \{0,1\}^b$ and $\hat{\mathbf{x}}_w \in \{0,1\}^b$ are *compatible*, denoted by $\hat{\mathbf{x}}_v \sim \hat{\mathbf{x}}_w$, if and only if $\forall \hat{\mathbf{y}} \in \{0,1\}^{n-b} : f(\hat{\mathbf{x}}_v,\hat{\mathbf{y}}) = f(\hat{\mathbf{x}}_w,\hat{\mathbf{y}})$. For completely specified functions, compatibility is an equivalence relation, which partitions the set of BS-vertices into *compatible classes*. The number of compatible classes is denoted by ℓ. The *decomposition condition* states that a decomposition with the subfunction vector $\mathbf{d}$ exists if and only if $\forall \hat{\mathbf{x}}_v, \hat{\mathbf{x}}_w \in \{0,1\}^b : \hat{\mathbf{x}}_v \not\sim \hat{\mathbf{x}}_w \Longrightarrow \mathbf{d}(\hat{\mathbf{x}}_v) \neq \mathbf{d}(\hat{\mathbf{x}}_w)$, i.e., different

33rd Design Automation Conference®
Permission to make digital/hard copy of all or part of this work for personal or class-room use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 96 - 06/96 Las Vegas, NV, USA
©1996 ACM 0-89791-779-0/96/0006..$3.50

codes $\mathbf{d}(\hat{\mathbf{x}})$ must be assigned to incompatible BS-vertices. Thus, the minimum number c of subfunctions is $c = \lceil \log_2 \ell \rceil$. A simple method to compute subfunctions and to fulfill the decomposition condition is to assign a unique code of length c to each compatible class.

It is obvious that the computation of ℓ is an important subtask of functional decomposition. Roth and Karp [14] described how ℓ is computed using a SOP representation of the function f. Lai et al. showed that the computation is significantly sped up if the function f is represented by a BDD in which the bound set variables are ordered before the free set variables [15]. Lai introduced a set of BDD nodes called $cut_set(f,b)$ comprising all BDD nodes that have a level greater than b and a predecessor with a level less than or equal to b. Each node $v \in cut_set(f,b)$ is in a one-to-one correspondence to a compatible class. Thus, the number of compatible classes is given by the cardinality of $cut_set(f,b)$.

We now give a brief review of previous approaches to the variable partitioning problem. Note that all these approaches target area. During technology mapping for k-LUT architectures, usually BS cardinality k is chosen. The SOP-based functional decomposition method implemented in SIS either selects the first variable partition with BS size k that yields a non-trivial decomposition [16], or enumerates all partitions of fixed size [17]. The enumerative approach was adapted for BDD-based functional decomposition by Lai et al. [15] and Sasao [18]. Since enumeration is very expensive, it is not applicable for functions with many variables. Recently, a heuristic was proposed which directly constructs a BS of fixed size from the SOP representation of f [19]. In contrast to the approaches mentioned above, Schlichtmann proposed a BDD-based variable partitioning approach that also selects a good BS size [20]. Our new performance-directed variable partitioning presented in Section 3 is based on this approach.

Multiple-output decomposition. We briefly summarize functional multiple-output decomposition. Its goal is to compute subfunctions d_i that can be used for several outputs. Functional multiple-output decomposition breaks a multiple-output function $\mathbf{f}(\mathbf{x},\mathbf{y}) = (f_1,\ldots,f_m)$ into the composition function vector $\mathbf{g}(\mathbf{v},\mathbf{y}) = (g_1,\ldots,g_m)$ and the subfunction vector $\mathbf{d}(\mathbf{x}) = (d_1,\ldots,d_q)$ such that $\mathbf{f}(\mathbf{x},\mathbf{y}) = \mathbf{g}(\mathbf{d}(\mathbf{x}),\mathbf{y})$. Each composition function output g_k depends on a subset of the q subfunctions d_i. Precisely, g_k depends on $c_k = \lceil \log_2 \ell_k \rceil$ subfunctions d_i, where ℓ_k is the number of compatible classes of function $f_k(\mathbf{x},\mathbf{y})$. This guarantees that multiple-output decomposition of a vector $\mathbf{f}$ is at least as good as single-output decomposition of each output of $\mathbf{f}$ with respect to a given variable partition.

Multiple-output functional decomposition has the advantage that by extracting common subfunctions it performs a task that is typically confined to the logic optimization stage before technology mapping.

The problem faced during multiple-output decomposition is the usually very large number of possible subfunctions for each output. To cope with this problem, we use the multiple-output decomposition approach as described in [21]. Additionally, we compute subfunctions with minimal support [22].

3 Performance-Directed Variable Partitioning

For ease of explanation, we first describe the variable partitioning procedure for a single-output function f. Our goal is to partition the variables of f into bound set variables $\mathbf{x}$ and free set variables $\mathbf{y}$ such that the arrival time $at(g)$ at the output of composition function g is minimal. We use the unit delay model, i.e., each LUT has a delay of 1 unit. The second goal of our variable partitioning procedure is to reduce the LUT count needed to implement function f. Minimizing the LUT count reduces the maximum interconnect length on the FPGA chip and thus also affects performance.

We illustrate the variable partitioning problem with an example. To obtain a small arrival time $at(g)$, one might intuitively assign early arriving inputs to the bound set, and inputs with large arrival times to the free set.

Example 1 Figure 1 shows an example with the bound set size 3. Please assume that we want to achieve a 3-LUT implementation. The numbers at the inputs denote arrival times. Each of the resulting subfunctions d_1 and d_2 has 3 inputs and can be implemented by a single 3-LUT, thus we have propagation delays $dt(d_i) = 1$ and $at(d_i) = 3$. Since the composition function g has 4 inputs, it has to be decomposed further, and we have an estimated propagation delay $dt(g) = 2$ and arrival time $at(g) = 6$. Note that we must assume a propagation delay of 2 for each path through g since we do not know at this point how g will be decomposed.

It is easily recognized that using the variable with arrival time 3 in the bound set and one of the inputs with arrival time 2 in the free set would not increase the maximum arrival time of the inputs of g, which is 4 anyway. Thus, there are several variable partitions of bound set size 3, all of which should be evaluated to possibly reduce the number c of subfunctions d_i. Note that a variable partition for which $c = 1$ reduces the number of inputs of g to 3, thus decreasing the arrival time $at(g)$ to 5. □

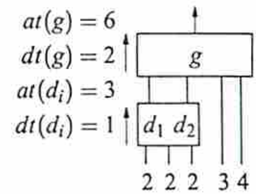


Figure 1: Delay oriented decomposition.

The example shows that the variable partition must take into account the arrival times of the input variables, the number $c = \lceil \log_2 \ell \rceil$ of subfunctions, and the estimated propagation delays of the resulting functions d_i and g, which depend on the BS size and c, respectively.

We separate the variable partitioning problem into two subtasks. First, we determine an optimal variable partition VP_i for each bound set size i. Then we select the best partition among all VP_i. A solution of the two subtasks requires proper cost functions.

To solve the first subtask, we propose an iterative heuristic. Each iteration step involves a cost-reducing exchange of a BS and a FS variable. Note that the BS size is given and is not allowed to change. A greedy method would require $|BS| \cdot |FS|$ tentative variable exchanges in an iteration step to find the best exchange of a BS and a FS variable. This is too expensive if the number of variables is large. Therefore, we compute the BS variable that yields the lowest costs if moved to the FS, and similarly we compute the FS variable that yields the lowest costs if moved to the BS. The *best* BS and the *best* FS variables found in such a way are exchanged if this reduces the costs. An iteration step then requires only $|BS| + |FS|$ tentative variable exchanges to find a good (and possibly suboptimal) exchange.

We employ cost functions for delay and area to evaluate tentative variable exchanges and to solve the second subtask mentioned above. The cost functions estimate the arrival time of the composition function g and the total LUT count of the resulting functions d_i and g. The only information used as input of the delay and area cost functions is the BS size b, the number of compatible classes ℓ, the maximum arrival time among the BS variables, denoted by $at_{max}(\mathbf{x})$, and the maximum arrival time among the FS variables, denoted by $at_{max}(\mathbf{y})$. Let us first introduce the *function propagation delay estimate FDE* and the *function area estimate FAE* of a Boolean function $h(\mathbf{x})$.

Definition 1 *The* **function propagation delay estimate** *FDE is a measure of the propagation delay of a Boolean function $h(\mathbf{x})$ in the unit delay model. It is defined by*

$$FDE(h(\mathbf{x})) = \begin{cases} 1 & : \ |\mathbf{x}| \le k \\ |\mathbf{x}| - k + 1 & : \ |\mathbf{x}| > k \end{cases} \qquad (1)$$

where $|\mathbf{x}|$ denotes the number of variables that h depends on, and k is the number of inputs of a LUT.

The *FDE* is the depth of a LUT network obtained by the Shannon decomposition of h and thus is a worst-case estimate of the function's propagation delay in the unit delay model.

For area we have examined various estimates that are linear, quadratic or exponential in the number of variables. Our experiments have shown that the best results are obtained using the same linear estimate as for the function propagation delay estimate.

Definition 2 *The* **function area estimate** *FAE is a measure of the area of a Boolean function $h(\mathbf{x})$ in terms of LUTs. It is defined by*

$$FAE(h(\mathbf{x})) = \begin{cases} 1 & : \ |\mathbf{x}| \le k \\ |\mathbf{x}| - k + 1 & : \ |\mathbf{x}| > k \end{cases} \qquad (2)$$

We define the **delay cost** function as

$$DC = \max\left[(at_{max}(\mathbf{x}) + FDE(\mathbf{d})), at_{max}(\mathbf{y})\right] + FDE(g), \qquad (3)$$

which is a simple computation of the arrival time $at(g)$ as described in standard literature.

The **area cost** function

$$AC = \sum_{j=1}^{c} FAE(d_j) + FAE(g) \qquad (4)$$

sums up the area estimates for the resulting functions. Note that we must know the number $c = \lceil \log_2 \ell \rceil$ of subfunctions to compute the delay and area estimates $FDE(g)$ and $FAE(g)$.

It has been shown in Section 2 that computing the number ℓ of compatible classes is simple if f is represented by a BDD and the BS variables are ordered before the FS variables. In this case, we have $\ell = |cut_set(f,b)|$. Thus, the delay and area cost functions can be evaluated very fast for a given variable partition. Representing function f by a BDD additionally has the advantage that variable moves and thus variable exchanges can be performed rapidly. If, e.g., a variable on level i shall be moved to level j, $j-i$ adjacent variable swaps must be performed. Adjacent variable swaps modify the BDD only on the levels of the swaped variables and are therefore carried out rapidly.

Let us resume the discussion of our variable partitioning procedure. We now describe in more detail the computation of the *best* BS and the *best* FS variable for a bound set size b. To find the best BS variable, the topmost BS variable is tentatively moved to the FS. This is done in the BDD by a variable move (a sequence of adjacent variable swaps) from level 1 to level b. Thus, the variable previously on level 2 is on level 1 now, the variable previously on level b is on level $b-1$ and the variable previously on level 1 is now on level b. The BDD is then traversed to compute $cut_set(f,b-1)$; delay and area costs are evaluated and stored. The variable move from level 1 to level b is repeated for the remaining $b-1$ BS variables. The *best* BS variable is the variable with minimal area cost among all BS variables with minimal delay cost. The *best* FS variable is computed similarly by a sequence of variable moves (from level n to level $b+1$) and BDD traversals. After exchanging the best BS and FS variable, the BDD is traversed once again to compute $cut_set(f,b)$ and to check if the costs have actually been reduced by the exchange.

Example 2 We illustrate the computation of the best BS and FS variable for the function $f(\mathbf{z}) = z_1z_2z_3z_4 + \bar{z}_1\bar{z}_5$ and a bound set size of 3. Figure 2 a) shows the initial BDD of function f where the variables z_i are ordered according to their arrival times $at(z_i)$, which are indicated by the numbers next to the corresponding BDD nodes. First, we have to compute the costs for the initial variable partition $BS = \{z_1,z_2,z_3\}$ and $FS = \{z_4,z_5\}$. For ease of explanation, we only consider delay costs. In Figure 2 a), all nodes that have a level greater than 3 and a predecessor with a level less than or equal to 3 are shaded. These nodes comprise the $cut_set(f,3)$. We need $c = 2$ subfunctions, as $l = |cut_set(f,3)| = 3$. Note that we have the same decomposition structure and the same arrival times of the BS and the FS variables as in Figure 1 of Example 1. Therefore, we have a delay cost of $DC = 6$. Now, we have to compute the best BS variable. We move z_1 from level 1 to level 3. The obtained BDD is shown in Figure 2 b). The delay cost after moving z_1 to the free set is determined by evaluating the delay cost function DC for the $BS = \{z_2,z_3\}$. As there are 2 nodes in $cut_set(f,2)$, only one subfunction is needed, which can be implemented by a single 3-LUT. Therefore, the estimated propagation delay of this subfunction is $FDE(d) = 1$. The resulting composition function g depends on 4 variables. Thus, $FDE(g)$ evaluates to 2. Using Equation (3) and the maximum arrival times of the BS and FS variables, we obtain $DC = \max[2+1,4]+2 = 6$. If z_2 and z_3, respectively, are moved to the free set, we get delay costs of 7. Thus, the best BS variable is z_1.

The computation of the best FS variable is done similarily. Moving z_5 to the bound set yields a delay cost of 7, whereas moving z_4 to the bound set yields a delay cost of 6. Thus, the best FS variable is z_4.

Now, the best BS and FS variables are exchanged as shown in the BDD of Figure 2 c) in order to check if a cost reduction is achieved. The resulting BDD has 2 nodes in $cut_set(f,3)$ so that we need one subfunction. The number of inputs to the composition function g is 3. Thus, $FDE(d)$ and $FDE(g)$ evaluate to 1. We obtain $DC = 5$. Thus, the delay costs are reduced from 6 for the initial partition to 5 for the new variable partition $BS = \{z_2,z_3,z_4\}$ and $FS = \{z_1,z_5\}$. □

Iterative variable exchanges are performed for a given BS size b to find an optimal variable partition VP_b. However, a closer look reveals that we can gather information that is useful for other BS sizes during the iterative procedure. Note that the BDD is completely traversed $|BS|+|FS|+1$ times for each variable exchange. Although each BDD traversal is carried out to compute the cut set for a specific BS size b, we can gather *all* cut sets $cut_set(f,i)$, $i = 2,\ldots,n-1$, with only small additional computational effort. For each BS size i, we compare the delay and area costs, as computed using the current variable order, with delay and area costs stored for VP_i. If a cost reduction is achieved, then VP_i is replaced by the variable partition determined by the current variable order. This method yields a coupling between the iterative procedures performed for each BS size.

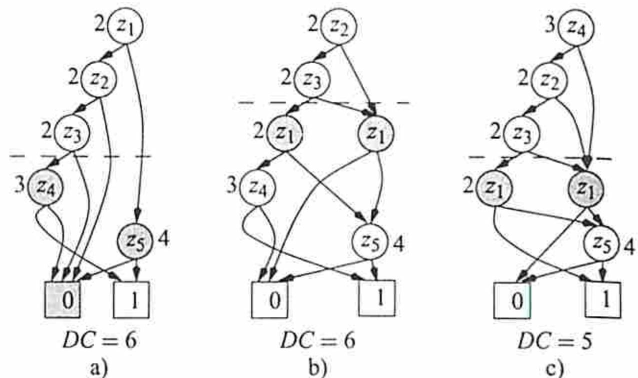


Figure 2: BDDs of Example 2.

Multiple-output decomposition of function vector $\mathbf{f}(\mathbf{x},\mathbf{y}) = (f_1,\ldots,f_m)$ requires a slight modification of this algorithm. First, we have a BDD with several roots, one for each output f_j. Second, the delay and area cost functions must be modified:
We use the **multiple-output delay cost** function

$$MDC = \max[DC_1,\ldots,DC_m], \tag{5}$$

where DC_j denotes the delay cost for output f_j. The **multiple-output area cost** function sums up the area estimates AC_j for each output f_j:

$$MAC = \sum_{j=1}^{m} AC_j. \tag{6}$$

4 Algorithm Overview

In this section we describe the overall algorithm for performance-directed technology mapping. The algorithm consists of three steps, i.e., collapsing, decomposition, and covering.

We first try to completely collapse the circuit within a given limit of CPU time. If collapsing is possible, the decomposition step starts from the obtained flattened circuit. Otherwise, a depth-oriented partial collapsing is performed by applying the SIS-command *reduce_depth -r -d d_value* [23]. This command, which is based on Lawler's clustering algorithm, first clusters nodes and then collapses each cluster such that the resulting network has the specified depth *d_value* and the cluster size is minimal. Since this command should only be used on a network with nodes of comparable complexity, we first decompose the nodes of the original network into nodes with at most k inputs using our performance-directed decomposition approach. We then apply *reduce_depth* to the decomposed network to obtain a network with depth 3 or 4. These networks are used as the starting point for the final performance-directed decomposition.

For the decomposition step, functional multiple-output decomposition as described in Section 2 and the variable partitioning algorithm of Section 3 are used. Only nodes of the network with more than k inputs are decomposed. As candidates to be decomposed we select only these nodes for which all nodes in the transitive fanin have at most k inputs. This guarantees accurate arrival times at the inputs of the considered nodes.

After all nodes in the network have been decomposed into nodes with at most k inputs, we do a simple covering step. A node is collapsed into its successors if each successor does not have more than k inputs after collapsing.

5 Experiments

Depth and Area Results. We implemented our new approach called *BoolMap-D* and integrated it into the synthesis tool TOS[TUM]. We compared *BoolMap-D* with two other performance-directed technology mappers, i.e., *FlowMap-r* [5] and *FlowSYN* [11]. We used the same set of benchmark circuits as in [5, 11], which are given in the first column of Table 1. In columns 2 to 5, we repeat the LUT count and depth results for *FlowMap-r* and *FlowSYN* from [5, 11]. The columns titled *BoolMap-D* show the results of our algorithm as described in Section 4. CPU times of column 8 are measured on a

Table 1: TECHNOLOGY MAPPING FOR 5-LUT BASED FPGAS

circuit	FlowMap-r[5] #LUT	depth	FlowSYN[11] #LUT	depth	**BoolMap-D** **#LUT**	**depth**	**CPU/s**
5xp1	23	3	20	2	13	2	3.6
9sym	61	5	7	3	7	3	1.4
9symml	58	5	7	3	7	3	1.4
C499*	151	5	133	5	101	4	627.8
C880*	211	8	232	8	146	7	62.3
alu2	148	8	113	6	43	4	38.5
alu4	245	10	249	9	268	7	950.0
apex6	232	4	257	4	189	4	139.1
apex7	80	4	89	4	78	3	70.3
count	73	3	75	3	42	2	30.2
des*	1087	5	893	4	594	3	1787.1
duke2	187	4	187	4	193	5	346.9
misex1	15	2	15	2	15	2	1.1
rd84	43	4	13	3	10	2	3.6
rot*	243	6	262	6	228	6	99.0
vg2	38	4	45	4	30	4	22.4
z4ml	13	3	6	2	5	2	0.6
sum	2908	83	2603	72	1969	63	-
perc.	-	-	100%	100%	-24.4%	-12.5%	-

DEC AlphaStation 250 4/266. All circuits that have been partially collapsed are marked with an asterisk in Table 1. For the marked circuits, the CPU times include initial decomposition, partial collapsing (*reduce_depth*), the final performance-directed decomposition, and covering. For the other circuits, CPU time is spent for collapsing, performance-directed decomposition, and covering.

BoolMap-D outperforms *FlowMap-r* and *FlowSYN* with respect to the circuit depth by 24.1% and 12.5%, respectively. Furthermore, a reduction in LUT count of 32.3% and 24.4% compared to *FlowMap-r* and *FlowSYN* is achieved. There is only one circuit, `duke2`, for which *BoolMap-D* produces a larger depth than *FlowMap-r* or *FlowSYN*. Compared to *FlowMap-r*, *BoolMap-D* produces 12 circuits with smaller depth and 4 circuits with equal depth. Compared to *FlowSYN*, *BoolMap-D* produces 8 circuits with smaller depth and 8 circuits with equal depth. For all circuits except for `alu4` and `duke2`, *BoolMap-D* produces circuits with fewer LUTs than *FlowMap-r* and *FlowSYN*, respectively.

Delay after Placement and Routing. To show the effectiveness of *BoolMap-D* in reducing the circuit delay after placement and routing, we implemented all designs (except `apex6`, `des`, and `rot`) obtained with *BoolMap-D* on Xilinx XC3000 FPGAs. The circuits `apex6` and `rot` have too many I/O pins to be implemented on a single XC3000 FPGA, and circuit `des` has too many CLBs [1]. We used the Xilinx tool *apr* for placement and routing. For each design, we selected a Xilinx XC3000 chip which yields about 80% cell utilization as proposed in [1]. The circuits obtained with *BoolMap-D* could be routed easily. In fact, all designs were routed in the first routing attempt.

We compare our results with the results of a FlowMap-type algorithm that aims at minimizing the nominal delay of a circuit [6]. We used the same set of benchmark circuits as in [6]. Column 2 of Table 2 shows the type of the used Xilinx XC3000 chip. Columns 3 and 4 repeat the number of CLBs and the actual delay after placement and routing for the *Nominal Delay Algorithm* [6]. The columns titled *BoolMap-D* show the results for our approach. As in [6], we measured the actual circuit delays using the Xilinx tool *xdelay*. The last column gives the relative reduction of the circuit delay achieved by *BoolMap-D*. The circuit delay is reduced by 27.8% on average.

6 CONCLUSION AND FUTURE WORK

In this paper, we have presented *BoolMap-D*, a Boolean approach to simultaneously minimize depth and area during LUT-based FPGA technology mapping. In the first step, large network portions are collapsed. Collapsing is motivated by the idea that decomposition has then a greater potential to determine a new network structure with small depth and area.

Functional decomposition is applied to the collapsed networks. We have presented an effective heuristic solution of the variable partitioning problem targeting small circuit depth in the first place and small area in the second place.

Compared to the mapping algorithms FlowMap-r and FlowSYN, BoolMap-D reduces the depth of LUT networks by 24.1% and 12.5% on average, and the number of LUTs by 32.3% and 24.4%, respectively. We also placed and routed Xilinx FPGA designs, and achieved

Table 2: DELAYS AFTER PLACEMENT AND ROUTING

Circuit	XC3000 part#	Nom.Del.Alg [6] #CLB	actual delay	**BoolMap-D** **#CLB**	**actual delay**	**red.**
9sym	3020PC68	50	94.3	6	**59.0**	37.4%
C880	3090PQ208	195	208.3	117	**143.8**	31.0%
alu2	3064PC84	149	200.1	37	**85.0**	57.5%
apex7	3042PP132	65	93.2	59	**80.7**	13.4%
count	3020PC68	60	79.8	31	**65.6**	17.8%
vg2	3020PC68	39	78.1	25	**70.7**	9.5%
aver.						27.8%

delay reductions of 27.8% on average compared with a FlowMap-type nominal delay algorithm.

ACKNOWLEDGMENT

The authors are very grateful to Prof. Kurt J. Antreich and Dr. Ulf Schlichtmann for many valuable discussions.

REFERENCES

[1] Xilinx Inc., San Jose, CA-95125, *The Programmable Logic Data Book*, 1994.

[2] R. J. Francis, J. Rose, and Z. Vranesic, "Technology mapping of lookup table-based FPGAs for performance," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 568–571, 1991.

[3] K.-C. Chen, J. Cong, Y. Ding, A. B. Kahng, and P. Trajmar, "DAG-Map: Graph-based FPGA technology mapping for delay optimization," *IEEE Design & Test of Computers*, pp. 7–20, Sept. 1992.

[4] J. Cong and Y. Ding, "Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD*, vol. 13, pp. 1–12, Jan. 1994.

[5] J. Cong and Y. Ding, "On area/depth trade-off in LUT-based FPGA technology mapping," in *30th ACM/IEEE Design Automation Conference (DAC)*, pp. 213–218, 1993.

[6] J. Cong and Y. Ding, "On nominal delay minimization in LUT-based FPGA technology mapping," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 82–88, Feb. 1995.

[7] H. Yang and D. F. Wong, "Edge-Map: Optimal performance driven technology mapping for iterative LUT based FPGA designs," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 150–155, 1994.

[8] R. Murgai, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Performance directed synthesis for table look up programmable gate arrays," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 572–575, 1991.

[9] N. Togawa, M. Sato, and T. Ohtsuki, "Maple: A simultaneous technology mapping, placement, and global routing algorithm for field-programmable gate arrays," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 156–163, Nov. 1994.

[10] P. Sawkar and D. Thomas, "Performance directed technology mapping for look-up table based FPGAs," in *30th ACM/IEEE Design Automation Conference (DAC)*, pp. 208–212, 1993.

[11] J. Cong and Y. Ding, "Beyond the combinatorial limit in depth minimization for LUT-based FPGA designs," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 110–114, Nov. 1993.

[12] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Sequential circuit design using synthesis and optimization," in *IEEE International Conference on Computer Design (ICCD)*, pp. 328–333, Oct. 1992.

[13] H. A. Curtis, "A generalized tree circuit," *Journal of the ACM*, vol. 8, pp. 484–496, 1961.

[14] J. P. Roth and R. M. Karp, "Minimization over boolean graphs," *IBM Journal of Research and Development*, pp. 227–238, 1962.

[15] Y. Lai, M. Pedram, and S. Vrudhula, "BDD based decomposition of logic functions with application to FPGA synthesis," in *30th ACM/IEEE Design Automation Conference (DAC)*, pp. 642–647, June 1993.

[16] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Logic synthesis for programmable gate arrays," in *27th ACM/IEEE Design Automation Conference (DAC)*, pp. 620–625, June 1990.

[17] R. Murgai, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Improved logic synthesis algorithms for table look up architectures," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 564–567, Nov. 1991.

[18] T. Sasao, *Logic Synthesis and Optimization*. Kluwer Academic Publishers, Boston/London/Dordrecht, 1993.

[19] W.-Z. Shen, J.-D. Huang, and S.-M. Chao, "Lambda set selection in Roth-Karp decomposition for LUT-based FPGA technology mapping," in *32nd ACM/IEEE Design Automation Conference (DAC)*, pp. 65–69, 1995.

[20] U. Schlichtmann, "Boolean matching and disjoint decomposition," in *IFIP Workshop on Logic and Architecture Synthesis*, pp. 83–102, Dec. 1993.

[21] B. Wurth, K. Eckl, and K. Antreich, "Functional multiple-output decomposition: Theory and an implicit algorithm," in *32nd ACM/IEEE Design Automation Conference (DAC)*, pp. 54–59, June 1995.

[22] C. Legl, B. Wurth, and K. Eckl, "An implicit algorithm for support minimization during functional decomposition," in *European Design and Test Conference (EDAC/ETC/EUROASIC)*, March 1996.

[23] H. Touati, H. Savoj, and R. K. Brayton, "Delay optimization of combinational logic circuits by clustering and partial collapsing," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 188–191, 1991.

Chapter 1

Introduction

1.1 CAD Systems

Computer-aided design (CAD) systems have been used since the inception of the integrated circuits. The goal of these systems is to automatically transform the high level (behavioral) description to physical description while producing near-optimal results that meet the specifications set by the designer. CAD techniques have reached a fairly high level of maturity in many areas, but reduction in device feature size, increase in circuit integration, introduction of new design styles such as FPGAs, and renewed emphasis on alternative objective functions such as circuit power demand CAD tools with increased capability.

A CAD program can be divided into three steps, behavioral synthesis, logic synthesis, and physical design. Behavioral synthesis transforms an algorithmic or behavioral description into a set of interconnected modules and control logic. Logic synthesis, which fits between behavioral synthesis and physical design, takes the register-transfer level description and provides automatic synthesis of gate-level netlists. Physical design provides automatic circuit partitioning, placement and routing, gate and wire sizing, power and ground distribution, and clock routing.

1.2 Logic Synthesis

The goal of logic synthesis is to convert a register-transfer level specification into a gate-level implementation. Logic synthesis is divided into two-level synthesis and multi-level synthesis.

Two-level synthesis has mainly been used to synthesize programmable logic arrays (PLAs). Because the nature of PLA architecture, the optimization methods are focused on minimizing the number of product terms and literals. Minimum-area two-level synthesis has been well developed and is considered to be well-understood [13].

In contrast, multi-level synthesis is less structured, more difficult, and relatively new. Because multilevel logic can often result in a faster and smaller implementation of a function than two-level logic, synthesis of multi-level logic has received considerable attention over the past decade (e.g., [10, 12, 9, 6]).

Most multi-level synthesis systems contain two steps: a technology-independent step that manipulates and optimizes Boolean functions and a technology-mapping step that maps Boolean functions into a set of gates in a specific target technology. The technology-independent phase is further divided into two sub-steps: logic restructuring that identifies common sub-logic to produce a near-optimal structure and logic minimization that optimizes the logic with respect to the structure obtained in the previous step.

1.2.1 Logic Restructuring

There are two methods for identifying common sub-logic: algebraic and Boolean. The algebraic method is fast because the logic function is represented and manipulated as an algebraic expression. Some optimality may however be lost because Boolean identities are not exploited by the algebraic method. In comparison, the Boolean method is slow, but tends to produce better results.

The algebraic approach is based on the *division* operation, namely, rewriting a function f as $qd + r$ where q, d, and r are the quotient, divisor and remainder, respectively. The theory of division was studied by Brayton and McMullen [14] and well developed in the MIS package [11]. The identification of common sub-logic is to extract common subexpressions as divisors. Because the number of divisors is huge, usually only a subset of the divisors are used. For example, *kernels* (cube-free primary divisors) are used in [11] while double- and single-cube divisors are used in [63]. Division can be also carried out by coalgebraic [32] and Boolean [11] methods.

The Boolean approach is based on the *decomposition* operation, namely, rewriting a function $f(X, Y)$ as $f'(g(X), Y)$ where the number of inputs of f' is smaller than that of f. The theory of decomposition was pioneered by Ashenhurst [4], Curtis [21] and Roth and Karp [51]. For representing functions, Karnaugh maps are used in [4, 21, 30], cubes are used in [34, 36, 52] and ordered binary decision diagrams (OBDDs) [15] are used in [16, 43, 55]. Most of these methods, except [36] and [30], only address single output functions.

A Boolean method for extracting common subfunctions was proposed by Karp [36]. He presented an algorithm for identifying a common subfunction between two functions based on the partitioning of compatible classes [52]. This approach has two shortcomings: first, it does not apply to more than two functions and second, it does not identify more than one shared subfunction. A new Boolean extraction algorithm based on Karnaugh maps was recently proposed in [30]. Because of the size complexity of the Karnaugh map representation, this approach is only applicable to functions with small number of inputs. Compared to [36], our proposed methods in this thesis can identify multiple (≥ 2) shared functions among multiple (≥ 2) functions. Complexity of our methods depends on the size of the bound set while that of the approach in [36] depends on the number of

compatible classes. In practical applications, size of the bound sets considered are much smaller than the number of compatible classes.

Shen and McKellar [61] proposed an algorithm for obtaining all simple disjunctive decompositions of a Boolean function. They construct a *decomposition graph* based on a necessary condition for the existence of simple disjunctive decompositions. Only the k-complete subgraphs of the decomposition graph need to be checked for decomposability. The construction of the decomposition graph is based on a *mod 2 map* rather than a Karnaugh map. The mod 2 map uses the Reed-Muller canonical form of a Boolean function. Even though a small number of bound sets are to be examined, their method still requires $O(2^n)$ computations.

1.2.2 Technology Mapping

Technology mapping is a process of transforming an optimized Boolean network into a netlist of gates or devices that are available from a semiconductor vendor. For application specific integrated circuits (ASICs vendor, a technology-based gates is a collection of standard cells; for field programmable gate arrays (FPGAs), a technology-based gates is a collection of basic logic blocks.

Translating a netlist of generic components into a cell library or logic blocks is straight forwards. But the challenge lies in maximally utilizing the components in the library such that the resulting netlist realizes its area, delay and power goals.

The early stage of technology mapping for FPGAs used the basic logic blocks as a library of basic cells and mapped the circuits as a conventional library-based mapper [38, 22]. However, it has been observed that these techniques are not suitable for technology mapping to FPGA architectures [24]. In recent years, several approaches for the FPGA technology mapping have been proposed [24, 45, 2, 25, 37, 23, 64, 46, 58, 20]. *chortle* [24] divides the Boolean network into a forest of tree and determines an optimal mapping of each tree. *chortle-crf* [25] employs bin-packing

to choose gate-level decompositions and exploits re-convergent paths and replication at multiple-fanout nodes. Xmap [37] use if-then-else dag as a decomposition of function and uses a covering procedure to map it. *mis-pga (new)* uses set of decomposition techniques, cofactoring, AND-OR decomposition, disjoint decomposition, cube-packing, to optimizing the mapping results. FlowMap [20] is based on a topological labeling algorithm to minimize the depth of the mapping results.

1.3 Overview

In this thesis, we describe OBDD-based algorithms for function decomposition of Boolean functions. We then present methods for identifying common subfunctions of multiple-output functions. We use OBDDs to represent functions so that our methods can be concisely and effectively carried out. Finally, these methods are applied to Look-Up Table (LUT) based FPGA synthesis.

The remainder of the thesis is organized as follows. Section 2 gives overview of function decomposition, OBDD, and some previous works. Section 3 presents OBDD-based algorithms for identifying common sub-logic among multiple Boolean functions. In Section 4, we present the decomposition techniques that minimize the delay of the network and the switching activity (energy) of resulting network. We also present the decomposition techniques that decompose function into special classes functions (symmetric and unate functions). Section 5 shows the application of these ideas and algorithms to the synthesis of different architecture of LUT-based FPGA devices. Experimental results and contributions are given in Section 6 and Section 7.

Chapter 2

Background

Terminology and definitions related to function decomposition and Ordered binary decision diagram (OBDD) data structure are given in this section. In addition, a brief description of previous work [39] performed on this subject is given.

2.1 Function Decomposition

The function decomposition operation is rewriting a function $f(X,Y)$ as $f'(g(X),$ $Y)$ where the number of inputs of f' is smaller than that of f. The theory of decomposition was introduced by Ashenhurst [4], Curtis [21] and Roth and Karp [51]. The motivation for using function decomposition in logic synthesis is to reduce the complexity of the problem by a divide-and-conquer paradigm: A function is decomposed into a set of smaller functions such that each of them is easier to synthesize.

Definition 2.1.1 A function $f(x_0,\ldots,x_{n-1})$ is said to be *decomposable* under *bound set* $\{x_0,\ldots,x_{i-1}\}$ and *free set* $\{x_{i-s},\ldots,x_{n-1}\}, 0 < i < n, 0 \leq s$ if f can be transformed to $f'(g_0(x_0,\ldots,x_{i-1}),\ldots,g_{j-1}(x_0,\ldots,x_{i-1}),x_{i-s},\ldots,x_{n-1})$, where $0 < j < i-s$. If s equals 0 then it is *disjunctively decomposable*; otherwise, it is *non-disjunctively decomposable*. Function f' is referred as the *f-function* and each g_i is referred as a *g-function*. The reduction in variable support is equal to

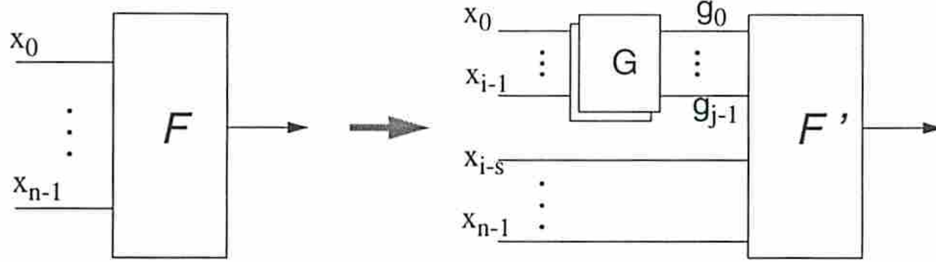


Figure 2.1: Function decomposition

$i-(j+s)$. The above transformation is referred as *decomposition.* If only some of the g-functions are formed, then f is partially decomposed.

A graphical representation of function decomposition is shown in Figure 2.1.

The decomposition chart of a Boolean function is an arrangement of the Karnaugh map where columns correspond to the variables in the bound set and rows correspond to the variables in the free set [3, 21]. The number of distinct column vectors is referred as the *column multiplicity.*

Definition 2.1.2 Given a Boolean function f, a bound set B, and a decomposition chart C with respect to f and B, the *column_vector* $\mathcal{V}^f$ of f with respect to B is defined as $\mathcal{V}^f = \langle v_{2^{|B|}-1}, \ldots, v_0 \rangle$ where $v_0 = 0$ and $v_i = j$ if v_i is the jth distinct column from the right of the decomposition chart. Each v_i is called *column_id.* The *column_set* $\mathcal{S}^f$ of f with respect to B is $\{0, \ldots, k-1\}$ if there are k distinct columns in the decomposition chart. The k is referred to as the number of *compatible classes* in [51]. The *bit_size* of $\mathcal{V}^f$ is defined as *bit_size*$(\mathcal{V}^f)$ $= \lceil \log_2 \mid \mathcal{S}^f \mid \rceil$ and equals the number of *g-functions* needed in the decomposed function f'. Furthermore the bound set size is defined as *bset_size*$(\mathcal{V}^f) = \mid B \mid$.

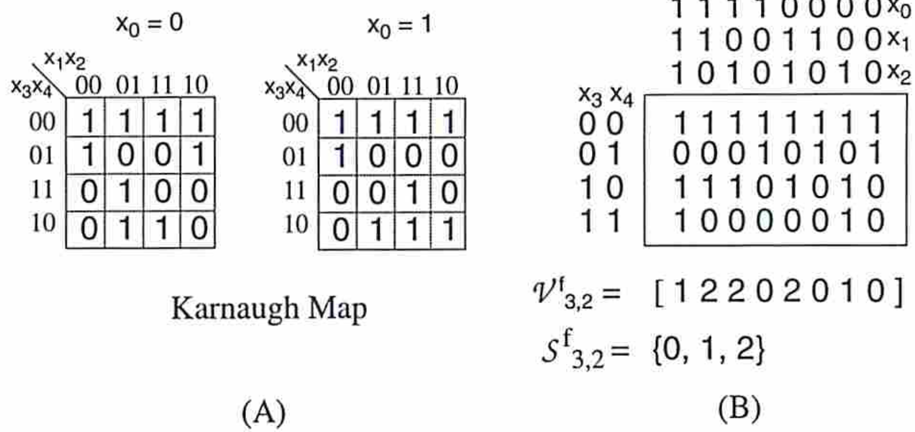
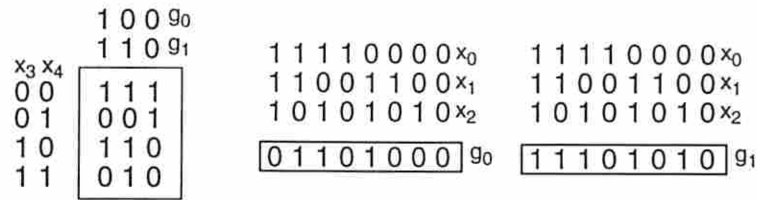


Figure 2.2: The Karnaugh map and the decomposition chart

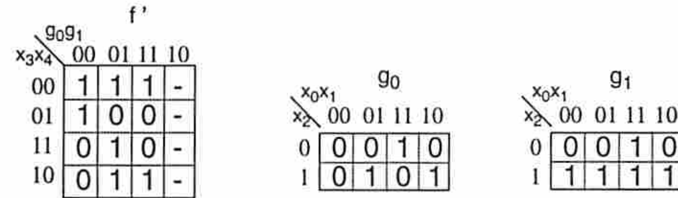
Theorem 2.1.1 [21] A function $f(x_0, \ldots, x_{n-1})$ can be transformed to $f'(g_0(x_0, \ldots, x_{i-1}), \ldots, g_{j-1}(x_0, \ldots, x_{i-1}), x_i, \ldots, x_{n-1})$ if and only if its decomposition chart has at most 2^j distinct column vectors.

We use $\mathcal{V}_{i,j}$ to denote a column_vector with bound set size i and bit_size j, and $\mathcal{S}_{i,j}$ to denote the column_set of $\mathcal{V}_{i,j}^f$.

Example 2.1.1 Let $f = \bar{x_0}\bar{x_1}\bar{x_2}\bar{x_3} + \bar{x_0}\bar{x_1}x_2x_3 + \bar{x_0}x_1\bar{x_2}\bar{x_3} + \bar{x_0}x_1x_2\bar{x_4} + x_0\bar{x_1}\bar{x_2}\bar{x_3} + x_0\bar{x_1}x_2\bar{x_4} + x_0x_1\bar{x_2}\bar{x_4} + x_0x_1x_2x_3 + \bar{x_3}\bar{x_4}$, the corresponding Karnaugh map is shown in Figure 2.2.A and the decomposition chart of $f(x_0, x_1, x_2, x_3, x_4)$ with respect to the bound set $\{x_0, x_1, x_2\}$ and the free set $\{x_3, x_4\}$ is shown in Figure 2.2.B. Since there are three distinct columns, namely $[1100]^t$, $[1011]^t$ and $[1010]^t$, it requires two g-functions. Thus, $\mathcal{V}^f = [12202010]$, $\mathcal{S}^f = \{0, 1, 2\}$ with $bset_size(\mathcal{V}^f) = 3$, $bit_size(\mathcal{V}^f) = 2$ and the number of compatible classes (column_set size) is $|\ \mathcal{S}\ | = 3$. Since the $bit_size(\mathcal{V}^f) = 2$, we use two variables g_0 and g_1 to encode each column_id in $\mathcal{V}^f$. If we encode column_id 0 as $g_0g_1 = 00$, column_id 1 as $g_0g_1 = 01$, and column_id 2 as $g_0g_1 = 11$, then we have



(A) Decomposed charts



(B) Karnaugh Map

Figure 2.3: Function decomposition

$$
\begin{aligned}
\mathcal{V}^f &= \quad 1\ 2\ 2\ 0\ 2\ 0\ 1\ 0 \\
g_0 &= [\,0\ 1\ 1\ 0\ 1\ 0\ 0\ 0\,] \\
g_1 &= [\,1\ 1\ 1\ 0\ 1\ 0\ 1\ 0\,]
\end{aligned}
$$

Then the decomposition chart can be reduced with two charts that map bound set variables x_0, x_1, x_2 to g_0 and g_1 as shown in Figure 2.3.A. These charts can transform to their Karnaugh maps (Figure 2.3.B), then we have function f decomposed into:

$$
\begin{aligned}
f' &= \bar{g_0}\bar{g_1}\bar{x_3} + g_1\bar{x_4} + \bar{g_0}g_1x_3 \\
g_0 &= \bar{x_0}x_1x_2 + x_0\bar{x_1}x_2 + x_0x_1\bar{x_2} \\
g_1 &= x_2 + x_0x_1
\end{aligned}
$$

□

2.2 Ordered Binary Decision Diagrams (OBDDs)

OBDDs are a graphical representation of Boolean functions which are compact and canonical. Because of these properties, many Boolean operations (e.g., function decomposition) can be carried out effectively using OBDDs.

Definition 2.2.1 [15] An OBDD is a directed acyclic graph consisting of two types of nodes. A *nonterminal* node $\mathbf{v}$ is represented by a 3-tuple $\langle variable(\mathbf{v}), child_l(\mathbf{v}), child_r(\mathbf{v})\rangle$ where $variable(\mathbf{v}) \in \{x_0, \ldots, x_{n-1}\}$. A terminal node $\mathbf{v}$ is either $\mathbf{0}$ or $\mathbf{1}$. There exist an index function $index(x) \in \{0, \ldots, n-1\}$ such that for every nonterminal node $\mathbf{v}$, either $child_l(\mathbf{v})$ is a terminal node or $index(\ variable\ (\mathbf{v})) < index(\ variable(\ child_l(\mathbf{v})))$, and either $child_r(\mathbf{v})$ is a terminal node or $index(variable(\mathbf{v})) < index(\ variable(\ child_r(\mathbf{v})))$. There is no nonterminal node $\mathbf{v}$ such that $child_l(\mathbf{v}) = child_r(\mathbf{v})$, and there are no two nonterminal nodes $\mathbf{u}$ and $\mathbf{v}$ such that $\mathbf{u} = \mathbf{v}$. The function denoted by $\langle x, \mathbf{v}_l, \mathbf{v}_r\rangle$ is $xf_l + \bar{x}f_r$ where f_l and f_r are the functions denoted by $\mathbf{v}_l$ and $\mathbf{v}_r$, respectively. The functions denoted by $\mathbf{0}$ and $\mathbf{1}$ are the constant function 0 and 1, respectively.

We use the following notation.

1. The left edge of a node represent 1 or the true edge and the right edge represents 0 or the false edge.
2. $\mathbf{v}$ represents both a OBDD node and the OBDD rooted by node $\mathbf{v}$.
3. $index(\mathbf{v})$: the index of the variable associated with node $\mathbf{v}$. If $\mathbf{v}$ is a terminal node, then $index(\mathbf{v}) = n$.
4.

$$l_child(\mathbf{v}, i) = \begin{cases} child_l(\mathbf{v}) & \text{if } index(\mathbf{v}) = i, \\ \mathbf{v} & \text{otherwise.} \end{cases}$$

$$r_child(\mathbf{v}, i) = \begin{cases} child_r(\mathbf{v}) & \text{if } index(\mathbf{v}) = i, \\ \mathbf{v} & \text{otherwise.} \end{cases}$$

5. When $B = \{x_0, \ldots, x_{i-1}\}$ represents a bound set, $index(x_0) < \ldots < index(\ x_{i-1})$, $head(B) = x_0$ and $last(B) = x_{i-1}$.

6. $new_bdd(x, \mathrm{l}, \mathrm{r})$ returns a BDD node v such that $variable(\mathrm{v}) = x$, $child_l(\mathrm{v}) = \mathrm{l}$ and $child_r(\mathrm{v}) = \mathrm{r}$.

Definition 2.2.2 Given an OBDD node v representing $f(x_0, \ldots, x_{n-1})$ and a bit vector $\langle b_0, \ldots, b_{i-1} \rangle$, the function *eval* is defined as

$$eval(\mathrm{v}, \langle\rangle) = \mathrm{v},$$
$$eval(\mathrm{v}, \langle b_0, \ldots, b_{i-1} \rangle) = \mathrm{v}',$$

where v' is the OBDD representing function $f(b_0, \ldots, b_{i-1}, x_i, \ldots, x_{n-1})$. When i is known, we also use $eval(\mathrm{v}, p)$ for $eval(\mathrm{v}, \langle b_0, \ldots, b_{i-1} \rangle)$ where $p = 2^{i-1} b_0 + \ldots + 2^0 b_{i-1}$.

2.3 OBDD-based Function Decomposition

The function decompositions can be solved more efficiently by using the OBDD-based representation [43]. In this section, the function decomposition that based on the OBDD data structure is described.

Definition 2.3.1 Given an OBDD v representing $f(x_0, \ldots, x_{n-1})$ with variable ordering $x_0, \ldots, x_{n-1}$ and bound set $B = \{x_0, \ldots, x_{i-1}\}$, we define

$$cut_set(\mathrm{v}, B) \quad = \quad \{\mathrm{u} \mid \mathrm{u} = eval(\mathrm{v}, p), 0 \leq p < 2^i\}.$$

In the above definition, each element in $cut_set(\mathrm{v}, B)$ corresponds to a distinct column in Ashenhurst-Curtis decomposition charts [4, 21]. Furthermore, $\lceil \log_2 \mid cut_set(\mathrm{v}, B) \mid \rceil$ determines the minimum number of g-functions required for a decomposition of f under B. The result of $cut_set(\mathrm{v}, B)$ is $\mathcal{S}^f$ which is the *column_set* in the decomposition chart. We use $\mathcal{S}^f$ to denote a *cut_set* of f with bound set size $bset_size(\mathcal{S}^f)$ and bit size $bit_size(\mathcal{S}^f)$.

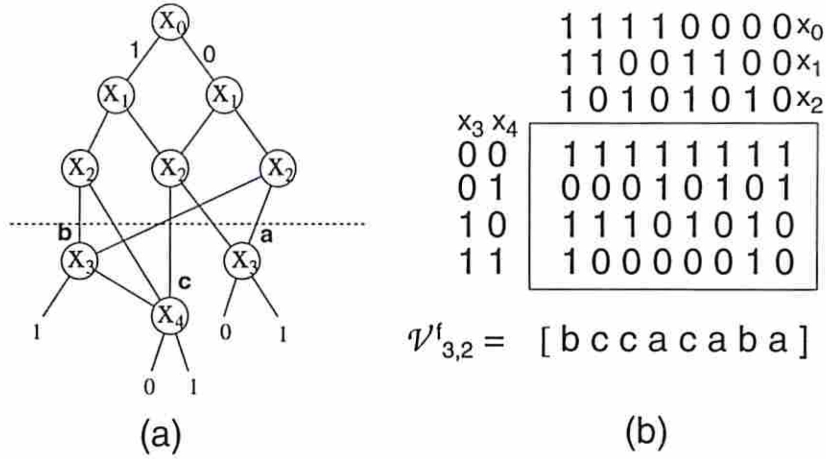


Figure 2.4: A function represented in (a) OBDD and (b) decomposition chart.

Example 2.3.1 The OBDD representation and decomposition chart of the function in Example 2.1.1 are shown in Figure 2.4 (a) and (b) respectively. In the OBDD representation, $cut_set(\mathbf{f}, \{x_0, x_1, x_2\}) = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$. Nodes **a**, **b**, and **c** correspond to distinct columns 1100, 1010, and 1011 respectively. Since there are three distinct columns f is not simple decomposable under bound set $\{x_0, x_1, x_2\}$ and free set $\{x_3, x_4\}$. □

When the bound variables are on the top of the OBDDs, the computation of the cut_set is straightforward as shown next. The time complexity of computing cut_sets depends on the size of the OBDD representation.

```
cut_set(v, B) /* B is on the top of the OBDD */
{
   if (index(v) > index(last(B))) return({ v });
   else return( cut_set(child_l(v), B) ∪ cut_set(child_r(v), B) );
}
```

To move a bound variable x to the top of an OBDD, we carry out $new_bdd(x, \mathbf{f_x}, \mathbf{f_{\overline{x}}})$ where $\mathbf{f_x}$ and $\mathbf{f_{\overline{x}}}$ are the cofactors of f with respect to x and $\overline{x}$, respectively.

In the worst case, both $\mathbf{f_x}$ and $\mathbf{f_{\overline{x}}}$ have about the same size as that of $\mathbf{f}$. Thus, moving a variable to the top may double the size of an OBDD. To move the bound variables to the top is therefore practical only for small bound set size.

It is clear that the computation of cut_sets of all 2^n bound sets is very expensive. However, in practical applications, we need compute the cut_sets of C_k^n bound sets where k is a small number such as 4 or 5. The time complexity of computing the cut_sets of C_k^n bound sets is then $O(n^k m)$ while the space complexity is $O(2^k m)$ where m is the size of an OBDD.

Definition 2.3.2 Given an OBDD v representing $f(x_0, \ldots, x_{n-1})$ with variable ordering $x_0 < \ldots < x_{n-1}$ and bound set $B = \{x_0, \ldots, x_{i-1}\}$, we define

$$\mathcal{V}^f = cut_vector(\mathrm{v}, B) \quad = \quad [eval(\mathrm{v}, 2^i - 1), \ldots, eval(\mathrm{v}, 0)].$$

In the following procedure for $cut_vector(\mathrm{v}, B)$, we assume that the bound variables B are on top of the OBDD. In addition, if $B = \{x_0, \ldots, x_{i-1}\}$, then $index(x_0) < \ldots < index(x_{i-1})$, $head(B) = x_0$, and $rest(B) = \{x_1, \ldots, x_{i-1}\}$.

```
cut_vector(v, B)
{
   if (B == φ) return(⟨v⟩);
   if (index(v) == index(head(B)))
        return(concatenate(cut_vector(child_l(v), rest(B)),
                  cut_vector(child_r(v), rest(B)));
   else       /* index(v) > index(head(B)) */
        return(concatenate(cut_vector(v, rest(B)), cut_vector(v, rest(B)));
}
```

We use $\mathcal{V}^f$ to denote a *cut_vector* of f with bound set size $bset_size(\mathcal{V}^f)$ and bit size $bit_size(\mathcal{V}^f)$; the *cut_vector* is the same as the *column_vector*, the only

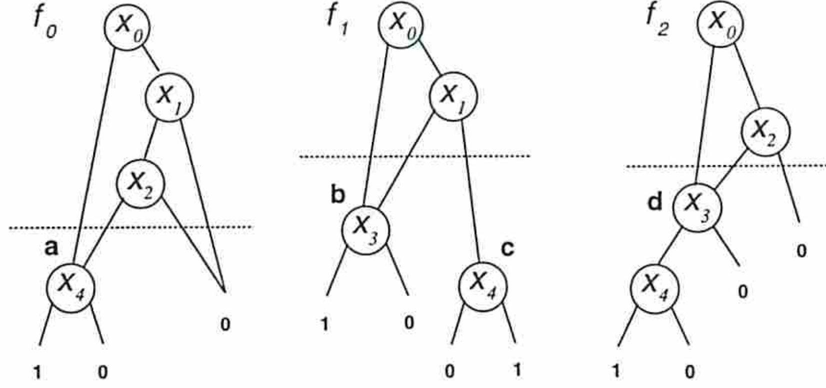


Figure 2.5: An example for operator *cut_vector* and *cut_set*.

difference being that the former is based on the OBDD representation while the later is based on the decomposition chart representation.

Example 2.3.2 The OBDD representation of a multiple-output Boolean function is shown in Figure 2.5. With the bound set $B = \{x_0, x_1, x_2\}$, we have the following cut_vectors and cut_sets:

$$\mathcal{V}_{3,2}^{f_0} = cut_vector(\mathrm{f}_0, B) = [\mathbf{a}, \mathbf{a}, \mathbf{a}, \mathbf{a}, \mathbf{a}, \mathbf{0}, \mathbf{0}, \mathbf{0}],$$
$$\mathcal{V}_{3,2}^{f_1} = cut_vector(\mathbf{f}_1, B) = [\mathbf{b}, \mathbf{b}, \mathbf{b}, \mathbf{b}, \mathbf{b}, \mathbf{b}, \mathbf{c}, \mathbf{c}], \text{ and}$$
$$\mathcal{V}_{3,2}^{f_2} = cut_vector(\mathbf{f}_2, B) = [\mathbf{d}, \mathbf{d}, \mathbf{d}, \mathbf{d}, \mathbf{d}, \mathbf{0}, \mathbf{d}, \mathbf{0}].$$

$$\mathcal{S}_{3,2}^{f_0} = cut_set(\mathrm{f}_0, B) = \{\mathbf{a}, \mathbf{0}\},$$
$$\mathcal{S}_{3,2}^{f_1} = cut_set(\mathbf{f}_1, B) = \{\mathbf{b}, \mathbf{c}\}, \text{ and}$$
$$\mathcal{S}_{3,2}^{f_2} = cut_set(\mathbf{f}_2, B) = \{\mathbf{d}, \mathbf{0}\}.$$

2.3.1 Disjunctive Decomposition

We show how to perform the disjunctive decomposition of a function $f(x_0, \ldots, x_{n-1}) = f'(g_0(x_0, \ldots, x_{i-1}), \ldots, g_{j-1}(x_0, \ldots, x_{i-1}), x_i, \ldots, x_{n-1})$ directly on its OBDD representation.

Algorithm *decomp*:

Given a function f represented in an OBDD $\mathbf{v_f}$ and a bound set B, a disjunctive decomposition with respect to B is carried out by the following steps:

1. Compute the cut_set with respect to B. Let $cut_set(\mathbf{v}, B) = \{\mathbf{u_0}, \ldots, \mathbf{u_{k-1}}\}$.
2. Encode each node in the cut_set by $\lceil \log_2 k \rceil = j$ bits. Let the encoding of $\mathbf{u_q}$ be q.
3. Construct $\mathbf{v_{f'}}$ to represent function f' by replacing the top part of $\mathbf{v_f}$ by a new set of variables $g_0, \ldots, g_{j-1}$ such that $eval(\mathbf{v_{f'}}, q) = \mathbf{u_q}$ for $0 \leq q < k-1$, $eval(\mathbf{v_{f'}}, q) = \mathbf{u_{k-1}}$ for $k-1 \leq q < 2^j$.
4. Construct $\mathbf{v_{g_p}}$'s to represent g_p's, $0 \leq p < j$ by replacing each node $\mathbf{u}$ with encoding $b_0, \ldots, b_{j-1}$ in the cut_set by terminal node $\mathbf{b_p}$.

Example 2.3.3 Consider a function $f = x_1x_3x_5x_6 + x_1x_3\bar{x}_5x_7 + x_1\bar{x}_3x_4x_8 + x_1\bar{x}_3\bar{x}_4x_9 + \bar{x}_1x_2x_4x_8 + \bar{x}_1x_2\bar{x}_4x_9 + \bar{x}_1\bar{x}_2x_{10}$. Let bound set $B = \{x_1, x_2, x_3, x_4, x_5\}$. We construct the OBDD for function f with variable ordering $x_1 < x_2 \cdots < x_{10}$ (Figure 2.6.A). Function f can be decomposed to functions f' and g_0, g_1, g_2 by cutting the OBDD between the bound set and free set. The resulting functions are (Figure 2.6.B):

$f' = g_0g_1g_2x_6 + g_0g_1\bar{g}_2x_7 + g_0\bar{g}_1g_2x_8 + g_0\bar{g}_1\bar{g}_2x_9 + \bar{g}_0x_{10}$

$g_0 = x_1 + x_2$

$g_1 = x_1x_3 + \bar{x}_1\bar{x}_2$

$g_2 = x_1x_3x_5 + x_1\bar{x}_3x_4 + \bar{x}_1x_2x_4 + \bar{x}_1\bar{x}_2$

2.3.2 Nondisjunctive Decomposition

Before describing how to perform non-disjunctive decomposition based on OBDD representation, we extend the concept of *cut_set* in the following definition.

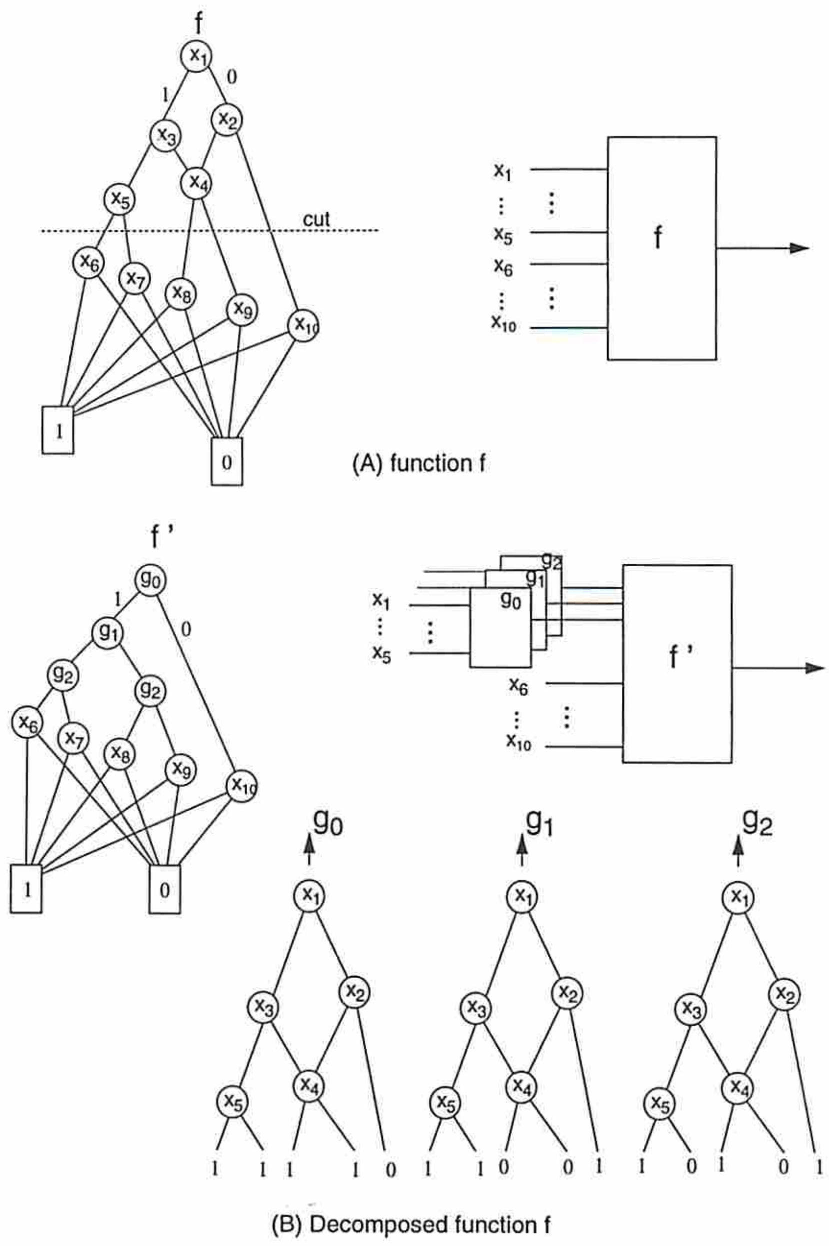


Figure 2.6: OBDD function decompositions, B = {x1, x2, x3, x4, x5}

Definition 2.3.3 Let $R = \{x_0, \ldots, x_{s-1}\}$, $S = \{x_s, \ldots, x_{i-1}\}$, and $T = \{x_i, \ldots, x_{n-1}\}$, $0 < s < i < n$. Given an OBDD $\mathbf{v}$ representing $f(x_0, \ldots, x_{n-1})$, a bound set $R \cup S$, and a free set $S \cup T$, we define

$$cut_set_nd(\mathbf{v}, R, S, p) = \{eval(\mathbf{w}, p) \mid \mathbf{w} \in cut_set(\mathbf{v}, R)\},$$

where $0 \leq p < 2^{|S|}$.

With the above definition, $cut_set(\mathbf{v}, B)$ can be represented by $cut_set_nd(\mathbf{v}, B, \phi, 0)$.

Example 2.3.4 The OBDD in Figure 2.4 (a) has

$$\begin{aligned}
&cut_set_nd(\mathbf{f}, \{x_0, x_1\}, \{x_2\}, 0) = \{\mathbf{a}, \mathbf{b}\},\\
&cut_set_nd(\mathbf{f}, \{x_0, x_1\}, \{x_2\}, 1) = \{\mathbf{b}, \mathbf{c}\},\\
&cut_set_nd(\mathbf{f}, \{x_0\}, \{x_1, x_2\}, 0) = \{\mathbf{a}\},\\
&cut_set_nd(\mathbf{f}, \{x_0\}, \{x_1, x_2\}, 1) = \{\mathbf{b}, \mathbf{c}\},\\
&cut_set_nd(\mathbf{f}, \{x_0\}, \{x_1, x_2\}, 2) = \{\mathbf{a}, \mathbf{b}\}, \text{ and}\\
&cut_set_nd(\mathbf{f}, \{x_0\}, \{x_1, x_2\}, 3) = \{\mathbf{b}, \mathbf{c}\}.
\end{aligned}$$

□

The non-disjunctive decomposition algorithm [41] (*decomp_nd*) is carried out in a similar fashion to the *decomp* algorithm (see [41] for details), but uses *cut_set_nd* instead of *cut_set* to construct the f' and g-functions.

Example 2.3.5 One possible non-disjunctive decomposition of the OBDD in Figure 2.4 (a) with respect to the bound set $\{x_0, x_1, x_2\}$ and the free set $\{x_2, x_3, x_4\}$ is shown in Figure 2.7. In this decomposition, we use the following coding: $\{\mathbf{a} = \mathbf{u}_{0,0}, \mathbf{b} = \mathbf{u}_{0,1}\} = cut_set_nd(\mathbf{v_f}, \{x_0, x_1\}, \{x_2\}, 0)$ and $\{\mathbf{c} = \mathbf{u}_{1,1}, \mathbf{b} = \mathbf{u}_{1,0}\} = cut_set_nd(\mathbf{v_f}, \{x_0, x_1\}, \{x_2\}, 1)$. □

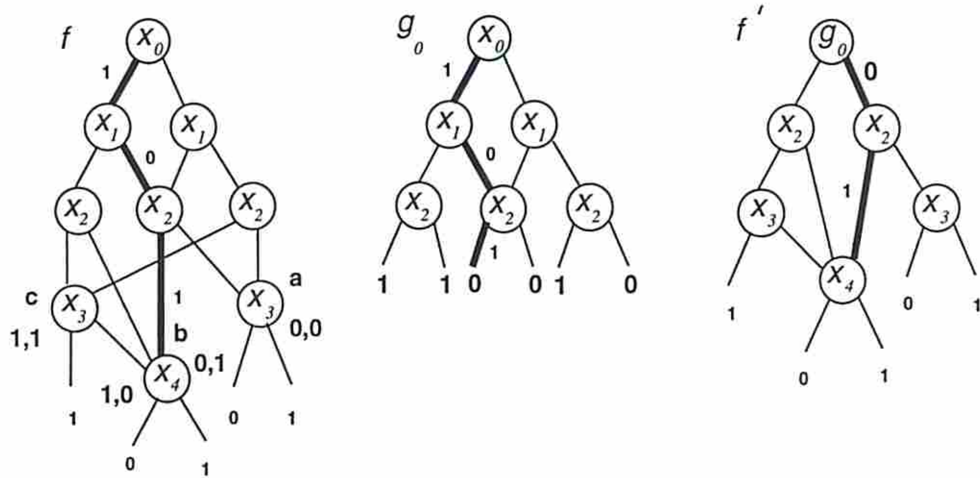


Figure 2.7: An example of non-disjunctive decomposition.

Chapter 3

Common Subfunction Extraction

In this section, we present four methods to extract common subfunctions from multiple Boolean functions. The first method (column encoding) is based on the stacking of the decomposition charts of the individual outputs; the second method (uni-code shared subfunction encoding) is based on the examination of all possible g-functions that can be generated; the third method (multi-code shared subfunction encoding) is the extension of the second method to allow the multi-code assignment to each g-function; the fourth method (graph-based shared subfunction encoding) is based on graph bipartitioning technique and can handle decomposition with respect to large bound sets.

3.1 Column Encoding

Our first method is called *column encoding* which is carried out as follows: we first stack up the decomposition charts for individual functions and then encode the distinct column patterns. This is equivalent to finding a common encoding for all functions.

Example 3.1.1 Consider a multiple-output function F: $f_0 = x_0\bar{x_4} + x_1x_2\bar{x_4}$, $f_1 = x_0\bar{x_3} + x_1\bar{x_3} + \bar{x}_0\bar{x}_1x_4$ and $f_2 = x_0\bar{x_3}\bar{x_4} + x_2\bar{x_3}\bar{x_4}$ with bound set $B = \{x_0, x_1, x_2\}$.

We stack the decomposition charts of F as shown in Figure 3.1 (a). Since there are four distinct column patterns, we use two bits to encode each column pattern. This is shown in Figure 3.1 (b) which defines the two g-functions g_0 and g_1. The f-functions are determined from the map of Figure 3.1 (c) which is obtained by combining identical columns of Figure 3.1 (a).

To see the decomposition of $f_k(x_0, x_1, x_2, x_3, x_4)$ as $f'_k(g_0(x_0, x_1, x_2), g_1(x_0, x_1, x_2), x_3, x_4)$, $k = 0, 1, 2$, consider the following evaluation: when $g_0 = 1$ and $g_1 = 0$ (the sixth column from the left in Figure 3.1 (b)), the corresponding columns of f_0, f_1, and f_2 are [0000], [1100], and [0000], respectively (the sixth column from the left in Figure 3.1 (a)), and the corresponding columns of f'_0, f'_1, and f'_2 are also [0000], [1100], and [0000], respectively (the second column from the left in Figure 3.1 (c)).

Thus, after multiple-output decomposition with respect to the bound set $\{x_0, x_1, x_2\}$, we have the following g- and f-functions:

$$
\begin{aligned}
g_0(x_0, x_1, x_2) &= x_0 + x_1, \\
g_1(x_0, x_1, x_2) &= x_0 + x_2, \\
f'_0(g_0, g_1, x_3, x_4) &= g_0 g_1 \bar{x}_4, \\
f'_1(g_0, g_1, x_3, x_4) &= g_0 \bar{x}_3 + \bar{g}_0 x_4, \text{ and} \\
f'_2(g_0, g_1, x_3, x_4) &= g_1 \bar{x}_3 \bar{x}_4.
\end{aligned}
$$

□

Note that the above method can identify common subexpressions that algebraic division based methods cannot. After the above decomposition, the literal count of the resulting circuit is 14. On the other hand, the best we could achieve by the algebraic method, is 16 as shown next:

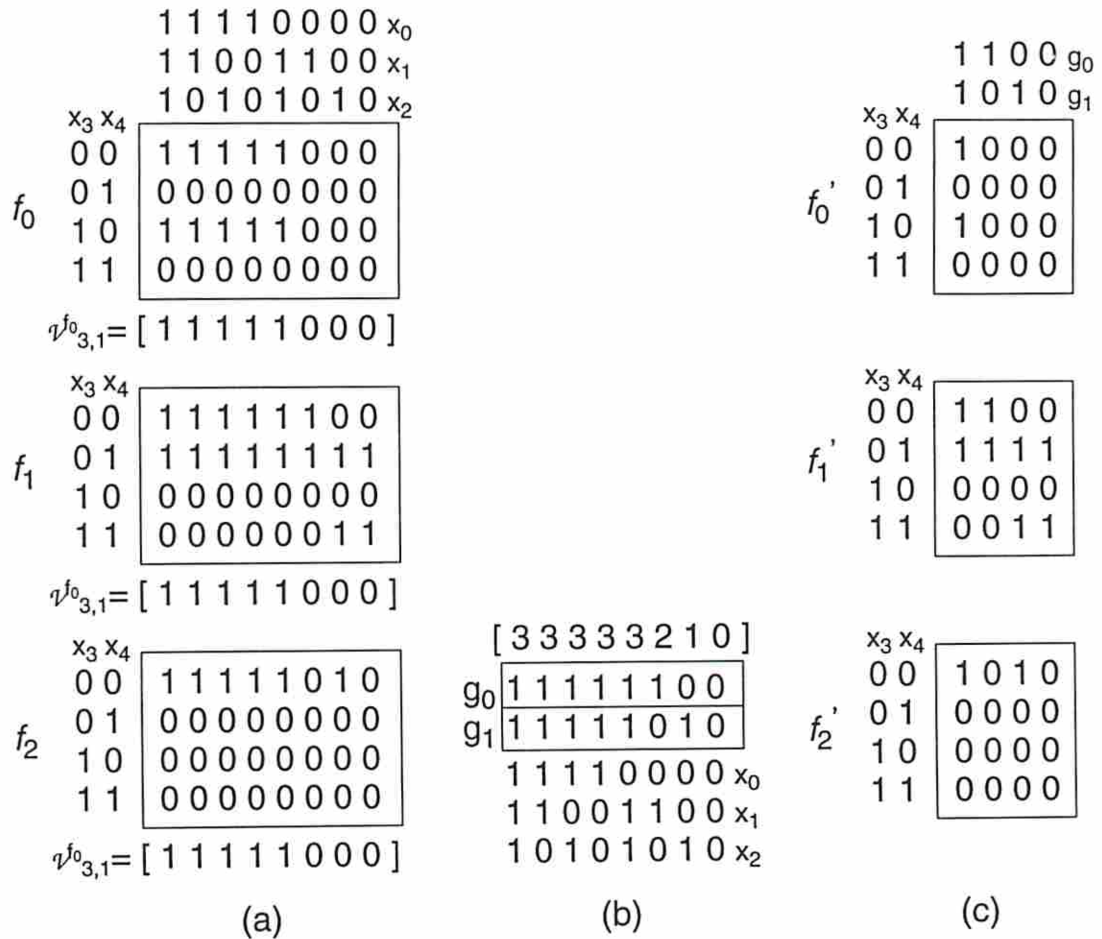


Figure 3.1: An example of multiple-output decomposition.

$$y_0 = x_0 + x_1,$$
$$y_1 = x_0 + x_2,$$
$$f_0' = x_0\bar{x_4} + x_1x_2\bar{x_4},$$
$$f_1' = y_0\bar{x_3} + \bar{y}_0x_4, \text{ and}$$
$$f_2' = y_1\bar{x_3}\bar{x_4}.$$

Lemma 3.1.1 Given a multiple-output Boolean function $F = \langle f_0, \ldots, f_{m-1} \rangle$ on variable set X and bound set $B \subset X$, if the column multiplicity of the stacking of individual decomposition charts is k such that $2^{j-1} < k \leq 2^j$, then F can be transformed to the following:

$$\langle f_0'(g_0(B), \ldots, g_{j-1}(B), X - B), \ldots, f_{m-1}'(g_0(B), \ldots, g_{j-1}(B), X - B) \rangle.$$

Proof: Omitted. □

Definition 3.1.1 Given cut_vectors $[v_{0,2^i-1}, \ldots, v_{0,0}], \ldots [v_{m-1,2^i-1}, \ldots, v_{m-1,0}]$, the operator *column_encode* is defined as:

$$column_encode([v_{0,2^i-1}, \ldots, v_{0,0}], \ldots [v_{m-1,2^i-1}, \ldots, v_{m-1,0}]) = [v_{m,2^i-1}, \ldots, v_{m,0}]$$

where $v_{m,0} = 0$ and $v_{m,p} = q$ if $[v_{0,p}, \ldots, v_{m-1,p}]$ is the qth distinct m-tuple of these cut_vectors.

Example 3.1.2 *column_encode*([1 1 1 1 1 0 0 0], [1 1 1 1 1 1 0 0], [2 2 2 2 2 1 0 0]) = [2 2 2 2 2 1 0 0].

Definition 3.1.2 Operator *select* is defined as

$$select(j, [v_{0,2^i-1}, \ldots, v_{0,0}], \ldots, [v_{m-1,2^i-1}, \ldots, v_{m-1,0}]) = [v_{0,k}, \ldots, _{m-1,k}]$$

where $[v_{0,k}, \ldots, v_{m-1,k}]$ is the jth distinct m-tuple of $[v_{0,0}, \ldots, v_{m-1,0}]$, $\ldots$, $[v_{0,2^i-1}, \ldots, v_{m-1,2^i-1}]$. If j is greater than the number of distinct m-tuples, then $[v_{0,k}, \ldots, v_{m-1,k}]$ is the last distinct m-tuple.

Example 3.1.3 Let $\mathcal{V}^{f_0}$ = [2 2 1 1 1 0 0 0] and $\mathcal{V}^{f_1}$ = [2 2 2 2 2 1 1 0], then we have the following:

$column_encode(\mathcal{V}^{f_0}, \mathcal{V}^{f_1}) = [\ 3\ 3\ 2\ 2\ 2\ 1\ 1\ 0\ ]$,
$select(0, \mathcal{V}^{f_0}, \mathcal{V}^{f_1}) = [0, 0]$,
$select(1, \mathcal{V}^{f_0}, \mathcal{V}^{f_1}) = [0, 1]$,
$select(2, \mathcal{V}^{f_0}, \mathcal{V}^{f_1}) = [1, 2]$, and
$select(3, \mathcal{V}^{f_0}, \mathcal{V}^{f_1}) = [2, 2]$. □

Given a multiple output function $\langle f_0, \ldots, f_{m-1} \rangle$ represented by a vector of OBDDs and a bound set $\{x_0, \ldots, x_{i-1}\}$, after the computation of cut_vectors and column encoding, the g- and f- functions are constructed as follows: For any input pattern $b = b_0, \ldots, b_{i-1}$, if the evaluation of b on f_k, $0 \leq k < m$, ends at node $\mathbf{v}$ with encoding $e_0, \ldots, e_{j-1}$, then we let $g_0(b), \ldots, g_{j-1}(b)$ produce function values $e_0, \ldots, e_{j-1}$ and $f'_k(g_0(b), \ldots, g_{j-1}(b), x_i, \ldots, x_{n-1})$ result in node $\mathbf{v}$. Consequently, $f_k(b_0, \ldots, b_{i-1}, x_i, \ldots, x_{n-1}) = f'_k(g_0(b_0, \ldots, b_{i-1}), \ldots, g_{j-1}(b_0, \ldots, b_{i-1}), x_i, \ldots, x_{n-1})$ for every input pattern $b_0, \ldots, b_{i-1}$ and $0 \leq k < m$. The following procedure gives the details of our algorithm.

Algorithm *decomp_mo_ce*:

Given a vector of OBDDs $\langle \mathbf{v}_0, \ldots, \mathbf{v}_{m-1} \rangle$ representing $\langle f_0(x_0, \ldots, x_{n-1}), \ldots, f_{m-1}(x_0, \ldots, x_{n-1}) \rangle$ with variable ordering $x_0, \ldots, x_{n-1}$ and a bound set $B = \{x_0, \ldots, x_{i-1}\}$.

1. Compute $V_k = cut_vector(\mathbf{v}_k, B) = [\mathbf{u}_{k,2^i-1}, \ldots, \mathbf{u}_{k,0}]$, $0 \leq k < m$.

2. Compute $\mathcal{V} = column_encode(V_0, \ldots, V_{m-1})$. Encode each element v_p, $(0 \leq p < 2^i)$ of $\mathcal{V}$ by j bits $d_{p,0} \ldots d_{p,j-1}$ such that $v_p = 2^{j-1} d_{p,0} + \ldots + 2^0 d_{p,j-1}$.

3. Construct each g-function $g_q(x_0, \ldots, x_{i-1})$, $0 \leq q < j$, as

   $$g_q(x_0, \ldots, x_{i-1}) = [d_{2^i-1,q} \ldots d_{0,q}] \quad \text{(truth table of } g_q)$$

   where $g_q(b_0, \ldots, b_{i-1}) = d_{p,q}$ if $2^{i-1} b_0 + \ldots + 2^0 b_{i-1} = p$.

4. Compute $select(r, V_0, \ldots, V_{m-1}) = [\mathbf{u}_{0,s_r}, \ldots, \mathbf{u}_{m-1,s_r}]$, $0 \leq r < 2^j$, $0 \leq s_r < 2^i$, s_r is any l such that $v_l = r$.

5. Construct each f-function $f'_k(g_0,\ldots,g_{j-1},x_i,\ldots,x_{n-1})$, $0 \le k < m$, as

$$f'_k(b_0,\ldots,b_{j-1},x_i,\ldots,x_{n-1}) = [\mathbf{u}_{k,s_{2^j-1}} \ldots \mathbf{u}_{k,s_0}],$$

where $f'_k(b_0,\ldots,b_{j-1},x_i,\ldots,x_{n-1}) = \mathbf{u}_{s_r,k}$ if $2^{j-1}b_0 + \ldots + 2^0 b_{j-1} = r$.

Example 3.1.4 The application of *decomp_mo_ce* on the multiple-output function in Ex. 3.1.1 is summarized as follows:

1. $cut_vector(f_0, B) = [\mathbf{aaaaa000}] = V_0$,
   $cut_vector(f_1, B) = [\mathbf{bbbbbbcc}] = V_1$,
   $cut_vector(f_2, B) = [\mathbf{ddddd0d0}] = V_2$ (see Figure 2.5),

2. $column_encode(V_0, V_1, V_2) = [\ 3\ 3\ 3\ 3\ 3\ 2\ 1\ 0\ ] \Rightarrow [\ 11,\ 11,\ 11,\ 11,\ 11,\ 10,\ 01,\ 00\ ]$,

3. $g_0(x_0, x_1, x_2) = [\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ ]$,
   $g_1(x_0, x_1, x_2) = [\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ 0\ ]$,

4. $select(0, V_0, V_1, V_2) = [\mathbf{0}, \mathbf{c}, \mathbf{0}]$,
   $select(1, V_0, V_1, V_2) = [\mathbf{0}, \mathbf{c}, \mathbf{d}]$,
   $select(2, V_0, V_1, V_2) = [\mathbf{0}, \mathbf{b}, \mathbf{0}]$,
   $select(3, V_0, V_1, V_2) = [\mathbf{a}, \mathbf{b}, \mathbf{d}]$,

5. $f'_0(g_0, g_1, x_3, x_4) = [\ a\ 0\ 0\ 0\ ]$,
   $f'_1(g_0, g_1, x_3, x_4) = [\ b\ b\ c\ c\ ]$, and
   $f'_2(g_0, g_1, x_3, x_4) = [\ d\ 0\ d\ 0\ ]$.

The resulting g- and f-functions are shown in Figure 3.2 (a) and (b), respectively. To see $f_k(x_0, x_1, x_2, x_3, x_4) = f'_k(g_0(x_0, x_1, x_2), g_1(x_0, x_1, x_2), x_3, x_4)$, consider the evaluation of $x_0 = 0$, $x_1 = 1$, and $x_2 = 0$ on f_1, g_0, g_1, and f'_1 as an example:

$$f_1(0, 1, 0, x_3, x_4) = \mathbf{b} = x_3,$$
$$g_0(0, 1, 0) = 1,$$
$$g_1(0, 1, 0) = 0, \text{ and}$$
$$f'_1(1, 0, x_3, x_4) = \mathbf{b} = x_3.$$

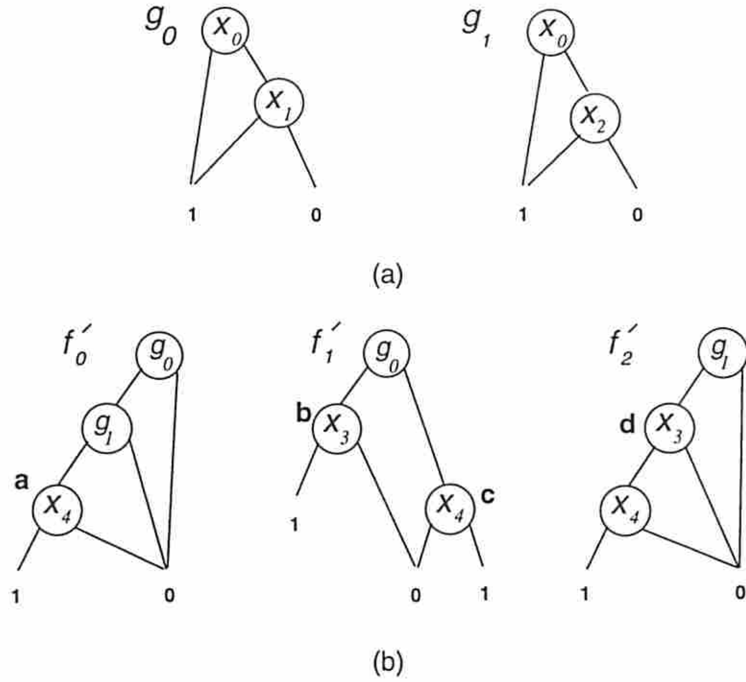


Figure 3.2: An example of multiple-output decomposition in OBDD representation.

□

The following lemmas prove the correctness of *decomp_mo_ce*.

Lemma 3.1.2 $function_encode([v_{0,2^i-1}, \ldots, v_{0,0}], \ldots, [v_{m-1,2^i-1}, \ldots, v_{m-1,0}]) = [u_{2^i-1}, \ldots, u_0]$ where $u_0 = 0$ and $u_p = q$ if $[v_{0,p}, \ldots, v_{m-1,p}]$ is the qth distinct m-tuple of $[v_{0,0}, \ldots, v_{m-1,0}], \ldots, [v_{0,2^i-1}, v_{m-1,2^i-1}]$.

Proof by contradiction: If $v_{\ell,j} \neq v_{\ell,k}$ for some ℓ, then $[v_{0,j}, \ldots, v_{\ell,j}, \ldots, v_{m-1,j}]$ is distinct from $[v_{0,k}, \ldots, v_{\ell,k}, \ldots, v_{m-1,k}]$ which implies that $v_j \neq v_k$. □

Lemma 3.1.3 The *decomp_mo_ce* algorithm performs the following transformation

$$f_k(x_0, \ldots, x_{n-1}) = f_k'(g_0(x_0, \ldots, x_{i-1}), \ldots, g_{j-1}(x_0, \ldots, x_{i-1}), x_i, \ldots, x_{n-1}),$$

where $0 \leq k < m, 0 < i < n$.

Proof: What we need to show is for all $b_0, \ldots, b_{i-1} \in \{0,1\}$, $0 \leq k < m$, $f_k(b_0, \ldots, b_{i-1}, x_i, \ldots, x_{n-1}) = f'_k(g_0(b_0, \ldots, b_{i-1}), \ldots, g_{j-1}(b_0, \ldots, b_{i-1}), x_i, \ldots, x_{n-1})$. Let $cut_vector(\mathbf{v}_k, \{x_0, \ldots, x_{i-1}\}) = [\mathbf{u}_{k,2^i-1}, \ldots, \mathbf{u}_{k,0}]$ and $p = 2^{i-1}b_0 + \ldots + 2^0 b_{i-1}$ for an arbitrary bit vector $b_0, \ldots, b_{i-1}$.

From Definition 2.2.2, $f_k(b_0, \ldots, b_{i-1}, x_i, \ldots, x_{n-1}) = eval(\mathbf{v}_k, [b_0, \ldots, b_{i-1}]) = \mathbf{u}_{k,p}$.

From step 3 of *decomp_mo_ce*:

$$g_q(b_0, \ldots, b_{i-1}) = d_{p,q}, 0 \leq q < j, \text{ and}$$

$$2^{j-1}g_0(b_0, \ldots, b_{i-1}) + \ldots + 2^0 g_{j-1}(b_0, \ldots, b_{i-1})$$

$$= 2^{j-1}d_{p,0} + \ldots + 2^0 d_{p,j-1}$$

$$= v_p.$$

From step 5 of *decomp_mo_ce*:

$$f'_k(g_0(b_0, \ldots, b_{i-1}), \ldots, g_{j-1}(b_0, \ldots, b_{i-1}), x_i, \ldots, x_{n-1})$$

$$= f'_k(d_{p,0}, \ldots, d_{p,j-1}, x_i, \ldots, x_{n-1})$$

$$= \mathbf{u}_{k,s_{v_p}} \qquad (2^{j-1}d_{p,0} + \ldots + 2^0 d_{p,j-1} = v_p),$$

where $s_{v_p} = \ell$ such that $v_\ell = v_p$.

From Lemma 3.1.2 $v_\ell = v_p$ implies that $\mathbf{u}_{k,s_{v_p}} = \mathbf{u}_{k,p}$. □

3.1.1 Output Grouping

In practice, it is unlikely that a multiple-output function is decomposable. For example, if we directly apply column encoding to every output, then the resulting column_vector for the stacked decomposition chart will often be $\mathcal{V}_{i,i}$. Output partitioning is thus useful to improve decomposability. We partition the outputs into groups such that the column_vector of the stacked decomposition chart for each group corresponds to a decomposable function (i.e., the number of required g-functions required is less than size of the bound set) and the total number of g-functions required to implement all groups is minimum. This problem is formulated as follows.

Definition 3.1.3 Given a set of column_vectors $\mathcal{V}_{i,j_k}^{f_k}$'s with respect to m Boolean functions $F = \langle f_0, \ldots, f_{m-1} \rangle$ and bound set B, partition this set into $P_0, \ldots, P_{\ell-1}$ such that the resulting column_vector $\mathcal{V}_{i,j_q}^{q}$ of each P_q satisfies $i > j_q$ and $\sum_{q=0}^{l-1} j_q$ is minimum.

We use the following greedy algorithm to solve this problem.

Algorithm *output_grouping*:
Assume every column_vector $\mathcal{V}_{i,j_k}^{f_k}$ satisfies $i > j_k$.

1. Order $\mathcal{V}_{i,j_k}^{f_k}$ in non-increasing order of $\mid \mathcal{S}_{i,j_k}^{f_k} \mid$. Initialize $\mathcal{V}_{i,j_0}^{0}$ to the null set.

2. Starting from the first element of the above list, merge as many column_vectors $\mathcal{V}_{i,j_k}^{f_k}$'s as possible into $\mathcal{V}_{i,j_0}^{0}$ as long as column_encode($\mathcal{V}_{i,j_0}^{0}, \mathcal{V}_{i,j_k}^{f_k}$) $= \mathcal{V}_{i,j_r}$ satisfies $i > j_r$. As soon as $i \le j_r$, initiate a new group of outputs. Repeat until all column_vectors are processed.

The above algorithm is based on the following observation: A $\mathcal{V}_{i,j_k}^{f_k}$ with larger $\mathcal{S}_{i,j_k}^{f_k}$ has better chance to *contain* another $\mathcal{V}_{i,k_l}^{f_l}$ with smaller $\mathcal{S}_{i,k_l}^{f_l}$. For example, if we have column_encode([2 2 2 1 1 0 0 0], [3 3 3 2 2 1 1 0]) = [3 3 3 2 2 1 1 0], then all the g-functions required for the first $\mathcal{V}_{3,2}$ are contained in those for the second $\mathcal{V}_{3,2}$. Thus, by putting these two column_vectors into the same set, we only need two g-functions for both functions.

Example 3.1.5 Given a set of column_vectors as following:

$\mathcal{V}^{f_0} = [\ 2\ 2\ 2\ 2\ 1\ 1\ 0\ 0\ ]$,
$\mathcal{V}^{f_1} = [\ 3\ 3\ 2\ 2\ 2\ 2\ 1\ 0\ ]$,
$\mathcal{V}^{f_2} = [\ 2\ 2\ 1\ 1\ 1\ 1\ 0\ 0\ ]$,
$\mathcal{V}^{f_3} = [\ 2\ 2\ 2\ 1\ 1\ 0\ 0\ 0\ ]$,
$\mathcal{V}^{f_4} = [\ 2\ 2\ 2\ 2\ 2\ 2\ 1\ 0\ ]$,

$\mathcal{V}^{f_5} = [\ 2\ 2\ 2\ 1\ 1\ 1\ 0\ 0\ ]$.

If we apply column_encode on every outputs without using output_grouping algorithm, then column_encode($\mathcal{V}^{f_0}, \ldots, \mathcal{V}^{f_5}$) = $[\ 6\ 6\ 5\ 4\ 3\ 2\ 1\ 0\ ]$ which is $\mathcal{V}_{3,3}$.

If we apply output_grouping algorithm, then three groups will be produced.

group 1: column_encode($\mathcal{V}^{f_1}, \mathcal{V}^{f_4}$) = $[\ 3\ 3\ 2\ 2\ 2\ 2\ 1\ 0\ ] = \mathcal{V}_{3,2}$.

group 2: column_encode($\mathcal{V}^{f_0}, \mathcal{V}^{f_2}$) = $[\ 3\ 3\ 2\ 2\ 1\ 1\ 0\ 0\ ] = \mathcal{V}_{3,2}$.

group 3: column_encode($\mathcal{V}^{f_3}, \mathcal{V}^{f_5}$) = $[\ 3\ 3\ 3\ 2\ 2\ 1\ 0\ 0\ ] = \mathcal{V}_{3,2}$. □

3.2 Uni-code Shared Subfunction Encoding

After computing the column vectors of a multiple output function with respect to a bound set B, it is possible to develop a decomposition scheme that minimizes the number of required g-functions by sharing these functions among the original functions as described next.

Example 3.2.1 Given two Boolean functions $f_1 = \bar{x}_3\bar{x}_4\bar{x}_5 + \bar{x}_1\bar{x}_2x_3x_5 + \bar{x}_1\bar{x}_2x_3x_4 + \bar{x}_1\bar{x}_3x_4x_5 + x_1\bar{x}_2\bar{x}_4\bar{x}_5 + x_1\bar{x}_3\bar{x}_4 + \bar{x}_2x_3x_4x_5$ and $f_2 = \bar{x}_1\bar{x}_2x_3\bar{x}_4\bar{x}_5 + \bar{x}_1x_2x_3x_4 + \bar{x}_2\bar{x}_3x_4\bar{x}_5 + x_2\bar{x}_3x_4\bar{x}_5 + x_1x_2x_3\bar{x}_4\bar{x}_5 + x_1\bar{x}_2\bar{x}_3\bar{x}_4x_5 + x_1\bar{x}_2x_4\bar{x}_5$, let bound set $B = \{x_1, x_2, x_3\}$ and free set $\{x_4, x_5\}$. Decomposition charts for these two functions are shown in Figure 3.3. The column_vectors are $\mathcal{V}^{f_1}_{3,2} = [\ 2\ 3\ 2\ 0\ 1\ 2\ 1\ 0\ ]$ and $\mathcal{V}^{f_2}_{3,2} = [\ 1\ 0\ 1\ 3\ 2\ 1\ 1\ 0\ ]$, and *column_encode*($\mathcal{V}^{f_1}_{3,2}, \mathcal{V}^{f_2}_{3,2}$) = $[\ 2\ 5\ 2\ 4\ 3\ 2\ 1\ 0\ ] = \mathcal{V}_{3,3}$.

If we encode 0 as 00, 1 as 01, 2 as 10, and 3 as 11, then we have

$$\begin{array}{cc} 2\ 3\ 2\ 0\ 1\ 2\ 1\ 0 & 1\ 0\ 1\ 3\ 2\ 1\ 1\ 0 \\ [\ 1\ 1\ 1\ 0\ 0\ 1\ 0\ 0\ ] & [\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ ] \\ [\ 0\ 1\ 0\ 0\ 1\ 0\ 1\ 0\ ] & [\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 0\ ] \end{array}$$

which requires four g-functions: $g_1 = [\ 1\ 1\ 1\ 0\ 0\ 1\ 0\ 0\ ]$, $g_2 = [\ 0\ 1\ 0\ 0\ 1\ 0\ 1\ 0\ ]$, $g_3 = [\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ ]$ and $g_4 = [\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 0\ ]$. The resulting functions are:

$$f_1' = \bar{g_1}g_2\bar{x_4} + g_2\bar{x_4}x_5 + g_1g_2x_4 + g_1x_4x_5 + g_1\bar{g_2}\bar{x_4}\bar{x_5}$$
$$g_1 = \bar{x_2}x_3 + \bar{x_1}\bar{x_3}$$
$$g_2 = x_1\bar{x_3} + \bar{x_1}\bar{x_2}x_3$$
$$f_2' = \bar{g_3}\bar{g_4}\bar{x_4}\bar{x_5} + g_3\bar{g_4}\bar{x_4}x_5 + g_4x_4\bar{x_5} + g_3g_4x_4 + g_3x_4\bar{x_5}$$
$$g_3 = x_1\bar{x_2}\bar{x_3} + \bar{x_1}x_2x_3$$
$$g_4 = \bar{x_1}\bar{x_3} + \bar{x_1}x_2 + x_2\bar{x_3} + x_1\bar{x_2}x_3$$

If we encode 0 as 00, 1 as 11, 2 as 01, and 3 as 10 for $\mathcal{V}_{3,2}^{f_1}$, and 0 as 00, 1 as 01, 2 as 11, and 3 as 10 for $\mathcal{V}_{3,2}^{f_2}$, then we have

$$\begin{array}{cc} 2\,3\,2\,0\,1\,2\,1\,0 & 1\,0\,1\,3\,2\,1\,1\,0 \\ [\,0\,1\,0\,0\,1\,0\,1\,0\,] & [\,0\,0\,0\,1\,1\,0\,0\,0\,] \\ [\,1\,0\,1\,0\,1\,1\,1\,0\,] & [\,1\,0\,1\,0\,1\,1\,1\,0\,] \end{array}$$

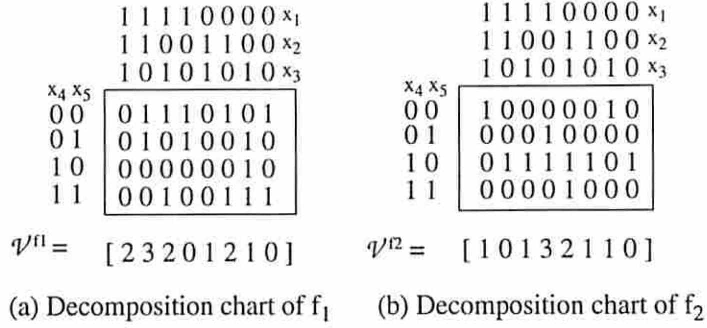
which requires only three g-functions: $g_1 = [\,0\;1\;0\;0\;1\;0\;1\;0\,]$, $g_2 = [\,1\;0\;1\;0\;1\;1\;1\;0\,]$ and $g_3 = [\,0\;0\;0\;1\;1\;0\;0\;0\,]$. The resulting functions are:

$$f_1' = g_2\bar{x_4}\bar{x_5} + g_1\bar{x_4}x_5 + \bar{g_1}g_2x_4x_5 + g_1\bar{g_2}x_4$$
$$g_1 = \bar{x_1}\bar{x_2}x_3 + x_1\bar{x_3}$$
$$g_2 = \bar{x_3} + x_1\bar{x_2}$$
$$f_2' = \bar{g_3}\bar{g_2}\bar{x_4}\bar{x_5} + g_3g_2\bar{x_4}x_5 + g_2x_4\bar{x_5} + g_3\bar{g_2}x_4$$
$$g_3 = x_1\bar{x_2}\bar{x_3} + \bar{x_1}x_2x_3$$

□

To achieve the above type of subfunction sharing, we present a *shared subfunction encoding* scheme as follows: For each output function, we compute all g-functions which can be produced from every possible encoding. We then identify the minimum number of g-functions which produce valid encodings for every function with respect to the given bound set.

The optimum g-function sharing can only be found if we allow the assignment of multiple function values (codes) to the same pattern (multi-coding). This is,



(a) Decomposition chart of f_1 (b) Decomposition chart of f_2

Figure 3.3: Decomposition charts of f_1 and f_2

however, very expensive. We trade some optimality for computational efficiency by requiring that a unique function value (code) be assigned to each pattern (uni-coding). In this case, for $\mathcal{S}_{i,j}$ with set size k, there will be $C_k^{2^j} k!$ different encodings. In this section, we only consider uni-coding.

Although there are many different encodings, the number of different g-functions which can be generated from these encodings is small as described next.

3.2.1 Permissible G-functions

Definition 3.2.1 Given *column_vector* $\mathcal{V}_{i,j}$ and *column_set* $\mathcal{S}_{i,j}$ for some bound set, let $\mathcal{S}_{i,j}$ be partitioned into S_0 and S_1 such that $0 \in S_0$, $\mid S_0 \mid \leq 2^{j-1}$ and $\mid S_1 \mid \leq 2^{j-1}$. A *permissible g-function encoding* (pg-code) of $\mathcal{V}_{i,j}$ with respect to S_0 and S_1, denoted by $pg_{\mathcal{V}_{i,j},S_0,S_1}$, is defined as:

$$pg_{\mathcal{V}_{i,j},S_0,S_1} = [b_{2^i-1} \dots b_0],$$

where $b_p = 0$ if $v_p \in S_0$ and $b_p = 1$ otherwise, $0 \leq p < 2^i$. If $S_0 \cap S_1 = \phi$ then $pg_{\mathcal{V}_{i,j},S_0,S_1}$ is a *uni-code* assignment, otherwise it is a *multi-code* assignment.

The uni-code *pg-set* of $\mathcal{V}_{i,j}$ is the set of all *pg*-code's of $\mathcal{V}_{i,j}$ obtained by enumerating all two-way partitions of $\mathcal{S}_{i,j}$ into S_0 and S_1 satisfying the conditions stated

above. The restrictions on $\mid S_0 \mid$ and $\mid S_1 \mid$ are needed to ensure that a valid j-bit encoding of the nodes in the *column_set* of f with respect to B can be found.

Example 3.2.2 Let $\mathcal{V}_{3,2} = [2,2,1,1,1,1,0,0]$, then

$$\begin{aligned} pg_{\mathcal{V}_{3,2},\{0\},\{1,2\}} &= [11111100], \\ pg_{\mathcal{V}_{3,2},\{0,1\},\{2\}} &= [11000000] \text{ and} \\ pg_{\mathcal{V}_{3,2},\{0,2\},\{1\}} &= [00111100]. \end{aligned}$$

The pg-set of $\mathcal{V}_{3,2}$ is $\{[11111100],[11000000],[00111100]\}$. Note that we need not consider $pg_{\mathcal{V}_{3,2},\{1\},\{0,2\}}$ because it is equal to $\overline{pg_{\mathcal{V}_{3,2},\{0,2\},\{1\}}}$ and that is why we force $0 \in S_0$ at $\mathcal{S}_{i,j}$ partition in definition 3.2.1. □

The cardinality of the pg-set of $\mathcal{V}_{i,j}$ is given by the following equation:

$$\mid pg_set \mid \;=\; C^{k-1}_{k-2^{j-1}-1} + \ldots + C^{k-1}_{2^{j-1}-1} = \sum_{l=k-2^{j-1}}^{2^{j-1}} C^{k-1}_{l-1}$$

where $2^{j-1} <\mid \mathcal{S}_{i,j} \mid = k \leq 2^j$. Because neither $\mid S_0 \mid$ nor $\mid S_1 \mid$ can exceed 2^{j-1}, the minimum and maximum sizes of S_0 are $k - 2^{j-1}$ and 2^{j-1}, respectively. $k-1$ and $l-1$ are used because $0 \in S_0$.

Example 3.2.3 For $\mid \mathcal{S}_{i,3} \mid = k = 3, 4, 6$, and 8, the cardinalities of their pg-sets are computed as follows:

$$\begin{aligned} k = 3:&\quad C^2_0 + C^2_1 = 1 + 2 = 3, \\ k = 4:&\quad C^3_1 = 3, \\ k = 6:&\quad C^5_1 + C^5_2 + C^5_3 = 5 + 10 + 10 = 25, \text{ and} \\ k = 8:&\quad C^7_3 = 35. \end{aligned}$$

□

Note that there are $C^{2^j}_k k! = 40,320$ different encodings for $\mid \mathcal{S}_{i,3} \mid = 8$ while only 35 different pg-codes can be generated.

Because each pg-code defines a partial (1-bit) encoding, a complete encoding of $\mathcal{S}_{i,j}$ is determined by selecting exactly j pg-code's. However, not every subset

of a *pg*-set, forms a valid partial encoding. This leads to the following definition of *compatibility of pg-code's.*

Definition 3.2.2 A *k-bit partition* P^k of $\mathcal{S}_{i,j}$ is defined as a partitioning of $\mathcal{S}_{i,j}$ into $P^k = \{S_0, \ldots, S_{2^k-1}\}$, $k \leq j$ such that $\forall S_q \in P^k, \mid S_q \mid \leq 2^{j-k}$.

Definition 3.2.3 A *partial k-bit encoding* of P^k is defined as follows:

$$\text{if } s \in S_p, \text{ then } s \text{ is encoded as } b_0 \ldots b_{k-1} x_k \ldots x_{j-1}$$

where $2^{k-1}b_0 + \ldots + 2^0 b_{k-1} = p$ and $x_k, \ldots, x_{j-1}$ are unassigned bits.

Thus, a k-bit partition defines a k-bit encoding of the $\mathcal{S}_{i,j}$. Note that if there is a S_q such that $\mid S_q \mid > 2^{j-k}$, then we cannot find a $j-k$ bit encoding of S_q and, therefore, cannot generate a valid j bit encoding of $\mathcal{S}_{i,j}$.

Definition 3.2.4 Given k- and l-bit partitions $P^k = \{S_0, \ldots, S_{2^k-1}\}$ and $Q^l = \{T_0, \ldots, T_{2^l-1}\}$ of $\mathcal{S}_{i,j}$ and assuming $k + l \leq j$, we define a merge operator $\mathcal{M}$ as follows:

$$\mathcal{M}(P^k, Q^l) = \{R_0, \ldots, R_{2^{k+l}-1}\},$$

where $R_{2^l p+q} = S_p \cap T_q, S_p \in P^k, T_q \in Q^l$. If $\mathcal{M}(S^k, T^l)$ is a $(k+l)$-bit-partition of $\mathcal{S}_{i,j}$, then P^k and Q^l are *compatible.* That is, if every $R_z \in \mathcal{M}(P^k, Q^l)$ satisfies $\mid R_z \mid \leq 2^{k+l}$, then P^k and Q^l are compatible.

In other words, if P^k and Q^l are compatible, then they can be used together to produce a $(k+l)$-bit encoding of $\mathcal{S}_{i,j}$.

Example 3.2.4 Let $\mathcal{S}_{5,4} = \{0,1,2,3,4,5,6,7,8,9\}$. some of 1-bit partitions of $\mathcal{S}_{5,4}$ are:

$$\begin{aligned} P^1 &= \{\{0,1,2,3,4\},\{5,6,7,8,9\}\}, \\ Q^1 &= \{\{0,1,2,5,6,7\},\{3,4,8,9\}\}, \\ U^1 &= \{\{0,1,2,3,4,5\},\{6,7,8,9\}\}, \\ V^1 &= \{\{0,2,4,6,8\},\{1,3,5,7,9\}\}. \end{aligned}$$

We have:

$$\begin{aligned}
\mathcal{M}(P^1, Q^1) &= \{\{0,1,2\},\{3,4\},\{5,6,7\},\{8,9\}\} = P^2, \\
\mathcal{M}(Q^1, U^1) &= \{\{0,1,2,5\},\{6,7\},\{3,4\},\{8,9\}\}, \\
\mathcal{M}(P^1, U^1) &= \{\{0,1,2,3,4\},\{\},\{5\},\{6,7,8,9\}\}, \\
\mathcal{M}(P^2, V^1) &= \{\{0,2\},\{1\},\{4\},\{3\},\{6\},\{5,7\},\{8\},\{9\}\}, \\
\mathcal{M}(P^2, U^1) &= \{\{0,1,2\},\{\},\{3,4\},\{\},\{5\},\{6,7\},\{\},\{8,9\}\}.
\end{aligned}$$

P^1 and Q^1, Q^1 and U^1, and P^2 and V^1 are compatible, but P^1 and U^1, and P^2 and U^1 are not compatible. Note that this example shows that compatibility relation is *not* transitive.

P^1 defines the following encoding:

$0, 1, 2, 3$ and 4 are encoded by $0g_1g_2g_3$,

$5, 6, 7, 8$ and 9 are encoded by $1g_1g_2g_3$.

P^2 defines the following encoding:

0, 1 and 2 are encoded by $00g_2g_3$,

3 and 4 are encoded by $01g_2g_3$,

5, 6 and 7 are encoded by $10g_2g_3$,

8 and 9 are encoded by $11g_2g_3$.

□

3.2.2 Minimum Subfunction Covering Problem

After generating the *pg*-sets for individual Boolean functions, we select a minimum number of *pg*-functions which produce valid, complete encoding of each function. This decision version of this problem is formulated as follows:

Definition 3.2.5 *Minimum subfunction covering problem:*
Instance: Collection PGS of *pg*-sets $\{pgs_1,\ pgs_2, \ldots,\ pgs_n\}$ and $G = \cup_{i=1}^{n} pgs_i$ and positive integer J. **Question:** Does G contain a subset G' with $\mid G' \mid \leq J$ and

such that G' contains at least a subset of size j which induces a j-bit-partition of pgs_i whenever bit_size(pgs_i) $= j$?

Note that when bit_size(pgs_i) $= 1$, for all $pgs_i \in PGS$, then the compatibility requirement (that is, the requirement that the j elements within G' induce a j-bit-partition of pgs_i) is automatically satisfied.

The *hitting set problem* is an NP-complete problem and is defined as:

Definition 3.2.6 [28] *Hitting set problem:*
Instance: Collection C of subsets of a set S, positive integer K. **Question:** Does S contain a *hitting set* for C of size K or less, that is, a subset $S' \subseteq S$ with $| S' | \leq K$ and such that S' contains at least one element from each subset in C?

Lemma 3.2.1 The minimum subfunction covering problem is NP-complete.

Proof: We transform the hitting set problem to a restricted version of the minimum subfunction covering problem where bit_size(pgs_i) $= 1$ for all $pgs_i \in PGS$. Let collection C of subsets of a set S and positive integer K constitute an arbitrary instance of hitting set. The basic unit of the instance of hitting set are the elements of C. For each element i of C, create a *pg*-set pgs_i of PGS. The instance of subfunction covering problem is completely specified by

Subfunction covering problem		Hitting set problem
PGS	$=$	C
G	$=$	S
J	$=$	K

It is easy to see that this instance can be constructed in polynomial time and that S contains a hitting set for C of size K or less if and only if a subset G' of G exists with $| G' | \leq J$ and such that G' contains at least one element from each subset in PGS. □

We use the following greedy algorithm for solving the minimum subfunction covering problem.

Algorithm *decomp_mo_sse*:

Given a vector of OBDDs $\langle \mathrm{v}_0, \ldots, \mathrm{v}_{m-1} \rangle$ representing $\langle f_0(x_0, \ldots, x_{n-1}), \ldots, f_{m-1}(x_0, \ldots, x_{n-1}) \rangle$ with variable ordering $x_0, \ldots, x_{n-1}$ and a bound set $B = \{x_0, \ldots, x_{i-1}\}$.

1. Compute $V_k = cut_vector(\mathrm{v}_k, B) = [\mathrm{u}_{k,2^i-1}, \ldots, \mathrm{u}_{k,0}]$, $0 \leq k < m$.

2. Compute $\mathcal{V}_{i,j} = column_encode(V_0, \ldots, V_{m-1})$.

3. Compute the *pg*-set corresponding to each $\mathcal{V}_{i,j}$. Annotate each *pg*-set with a value count initialized to its bit_size (this is for indicating when a function has a complete encoding) and a 0-bit-partition *bp* initialized to its column_set (this is for checking compatibility).

4. Find a *pg*-code, g, that occurs in the *pg*-sets most frequently.

5. For each *pg*-set that contains g, decrease its *count* by 1. If $count = 0$, remove this set.

6. For each *pg*-set that contains g, perform $bp = \mathcal{M}(bp, g)$ and remove any *pg* that is not compatible with $\mathcal{M}(bp, g)$.

7. Repeat steps 4-6 until every *pg*-set is removed. Then, return the set of *bp*'s, which defines the encoding for each *pg*-set.

3.2.3 Minimum Support for G-functions

A g-function encoding scheme that minimizes the number of support variables for individual g-function is described in this section. This scheme reduces the logic complexity of individual g-functions.

Definition 3.2.7 Given a function $f(x_0, \ldots, x_{n-1}) = f'(g_0(x_0, \ldots, x_{i-1}), \ldots, g_{j-1}(x_0, \ldots, x_{i-1}), x_i, \ldots, x_{n-1})$, the total support size of the g-function is given by:

$$supp_size_g_func(\{g_0, \ldots, g_{j-1}\}) = \sum_{k=0}^{j-1} | \, supp(g_k) \, | \tag{3.1}$$

By properly choosing the g-function encodings, this support size can be minimized, thus minimizing the logic complexity of the g-functions.

Example 3.2.5 Consider a function $f = x_0x_1x_3x_4x_5 + x_1x_3\bar{x}_4x_6 + x_1\bar{x}_3x_4x_7 + x_1\bar{x}_3\bar{x}_4x_8 + \bar{x}_1x_2x_4x_7 + \bar{x}_1x_2\bar{x}_4x_8 + \bar{x}_1\bar{x}_2x_9$. Let bound set $B = \{x_1, x_2, x_3, x_4\}$. We construct the OBDD for function f. Function f can be decomposed to functions f' and g_0, g_1, g_2. Figure 3.4 shows two different g-function encoding schemes that result in different $supp_size_g_func(\{g_0, g_1, g_2\})$. The OBDD representation of resulting functions in first encoding scheme (scheme I) is shown in Figure 3.4.A Their Boolean representation is given as:

$f' = g_0\bar{g}_1x_5 + \bar{g}_0g_1x_6 + g_0g_1g_2x_7 + g_0g_1\bar{g}_2x_8 + \bar{g}_0\bar{g}_1x_9$

$g_0 = x_1x_4 + x_1\bar{x}_3 + \bar{x}_1x_2$

$g_1 = x_1\bar{x}_4 + x_1\bar{x}_3 + \bar{x}_1x_2$

$g_2 = x_1x_3\bar{x}_4 + x_1\bar{x}_3x_4 + \bar{x}_1x_2x_4$

Then,

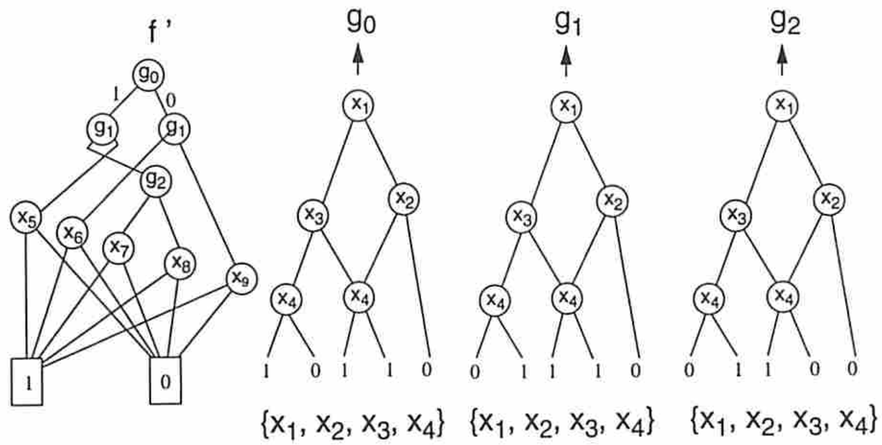
$$\begin{aligned} supp_size_g_func(\{g_0, g_1, g_2\}) &= | \{x_1, x_2, x_3, x_4\} | + | \{x_1, x_2, x_3, x_4\} | \\ &\quad + | \{x_1, x_2, x_3, x_4\} | \\ &= 4 + 4 + 4 = 12 \end{aligned}$$

The OBDD representation of resulting functions in second encoding scheme (scheme II) is shown in Figure 3.4.B with reduced support g-functions. Their Boolean representation is given as:

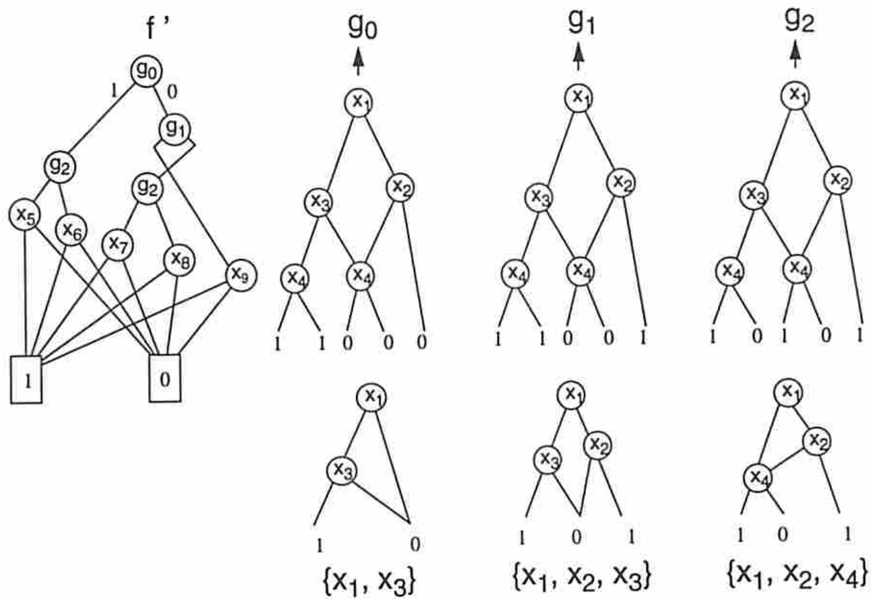
$f' = g_0g_2x_5 + g_0\bar{g}_2x_6 + \bar{g}_0\bar{g}_1g_2x_7 + \bar{g}_0\bar{g}_1\bar{g}_2x_8 + \bar{g}_0g_1x_9$

$g_0 = x_1x_3$

$g_1 = x_1x_3 + \bar{x}_1\bar{x}_2$



(A) g-function encoding scheme I



(B) g-function encoding scheme II

Figure 3.4: Resulting g-functions of two encodings

$g_2 = \bar{x_1}\bar{x_2} + x_4$

Here,

$$
\begin{aligned}
supp_size_g_func(\{g_0, g_1, g_2\}) &= \mid \{x_1, x_3\} \mid + \mid \{x_1, x_2, x_3\} \mid + \mid \{x_1, x_2, x_4\} \mid \\
&= 2 + 3 + 3 = 8
\end{aligned}
$$

The problem of finding the g-function encoding so that $supp_size_g_func$ is minimized can be formulated as the minimum subfunction covering problem (Definition 3.2.5) with a new cost function, $supp(pg)$, for each pg-code, pg. The algorithm *decomp_mo_sse* described in Section 3.2 can be used here with line 4 modified as follows:

> 4. Find a pg-code, g, that occurs in the pg-sets most frequently.
>   If there is a tie, then choose the pg-code with minimum $supp(g)$.

3.3 Multi-code Shared Subfunction Encoding

More shared subfunctions can be found by allowing multi-code assignment ("0" and "1") on a same column_id.

Definition 3.3.1 Given a column_vector $\mathcal{V}_{i,j}$ with column_set $\mathcal{S}_{i,j}$, the *residue* of $\mathcal{V}_{i,j}$ (denoted by $R(\mathcal{V}_{i,j})$) is $2^j - \mid \mathcal{S}_{i,j} \mid$.

Lemma 3.3.1 Column_vector $\mathcal{V}^f_{i,j}$ has a j-bit multi-code assignment if and only if $R(\mathcal{V}^f_{i,j}) > 0$.

Proof: Omitted. □

Note that the number of shared elements between S_0 and S_1 must be less than or equal to $R(\mathcal{V}_{i,j})$, that is, $\mid S_0 \cap S_1 \mid \leq R(\mathcal{V}_{i,j})$, in order to produce a valid j-bit encoding.

Example 3.3.1 Given a Boolean function F with two outputs f_1 and f_2, let

$$\mathcal{V}^{f1}_{4,3} = [\,3\;3\;5\;5\;0\;0\;4\;0\;3\;3\;2\;2\;3\;2\;1\;0\,] \quad \mathcal{S}^{f1}_{4,3} = \{0,1,2,3,4,5\}$$
$$\mathcal{V}^{f2}_{4,3} = [\,1\;0\;7\;7\;5\;5\;6\;4\;5\;4\;0\;0\;3\;2\;1\;0\,] \quad \mathcal{S}^{f2}_{4,3} = \{0,1,2,3,4,5,6,7\}$$

under some bound set B. Because $\mid \mathcal{S}^{f1} \mid= 6$ and $\mid \mathcal{S}^{f2} \mid= 8$, $R(\mathcal{V}^{f1}_{4,3}) = 2^3 - 6 = 2$ and $R(\mathcal{V}^{f2}_{4,3}) = 2^3 - 8 = 0$. Since $R(\mathcal{V}^{f1}_{4,3}) > 0$ and $R(\mathcal{V}^{f2}_{4,3}) = 0$, only $\mathcal{V}^{f1}_{4,3}$ has the potential for admitting a multi-code shared subfunction encoding.

There is no uni-code shared subfunction encoding for $\mathcal{V}^{f1}_{4,3}$ and $\mathcal{V}^{f2}_{4,3}$ for the following reason. By inspection, column_id 3 in $\mathcal{V}^{f1}$ aligns with column_id's 0, 1, 3, 4 and 5 in $\mathcal{V}^{f2}$.

$$\mathcal{V}^{f1}_{4,3} = [33------33--3---]$$
$$\mathcal{V}^{f2}_{4,3} = [10------54--3---]$$

In order to produce a uni-code shared g-function encoding, column_id's 0, 1, 3, 4 and 5 of $\mathcal{V}^{f2}$ must be assigned either all "1" or all "0" code in order to match the code for column_id 3 in $\mathcal{V}^{f1}$. However, assigning the same code to these column_id's violates the permissible g-function encoding constraint as 5 distinct columns cannot be encoded by the remaining two bits.

Instead if we partition $\mathcal{V}^{f2}$ into $S_0 = \{0,1,4,5\}$ and $S_1 = \{2,3,6,7\}$ which produces encoding [1 1 0 0 1 1 0 1 1 1 1 1 0 0 1 1] and assign this code to $\mathcal{V}^{f1}$, then we obtain a non-disjoint partitioning of $\mathcal{V}^{f1}$ into $S_0 = \{0,1,2,3\}$ and $S_1 = \{2,3,4,5\}$. By assigning columns in S_0 to 0 and columns in S_1 to 1, we achieve a multi-code shared subfunction encoding between $\mathcal{V}^{f1}$ and $\mathcal{V}^{f2}$. □

Lemma 3.3.2 [51] Given column_vector $\mathcal{V}_{i,j}$ and column_set $\mathcal{S}_{i,j}$ with $\mid \mathcal{S}_{i,j} \mid= k$, the number of possible multi-code assignments is

$$\frac{2^j!}{j!}\sum_{z=k}^{z=2^j}\frac{p(z)}{(2^j-z)!}. \tag{3.2}$$

where $p(z)$ is the number of ways of partitioning $\mathcal{S}_{i,j}$ into z nonempty sets.

Example 3.3.2 Given column_vector $\mathcal{V}^f_{4,3} = [\,3\;3\;5\;5\;0\;0\;4\;0\;3\;3\;2\;2\;3\;2\;1\;0\,]$, here p(6) = 1, p(7) = 26, p(8) = 228. The number of possible multi-code assignments is 1,535,520. The number of uni-code assignments for $\mathcal{V}^f_{4,3}$ is however only 25 [40]. □

Because of the exponential number of different multi-code assignments for each column_vector, it is impractical to find possible subfunction sharing directly. We thus use following lemma to relax the multi-code shared subfunction encoding problem into the uni-code shared subfunction encoding problem.

Lemma 3.3.3 Given column_vector $\mathcal{V}^{f1}$ and a one bit encoding pg of $\mathcal{V}^{f2}$, the multi-code shared subfunction encoding of $\mathcal{V}^{f1}$ is equivalent to the uni-code shared subfunction encoding of $\mathcal{U}^{f1}$ where $\mathcal{U}^{f1} = column_encode(\mathcal{V}^{f1}, pg)$.

Proof: Omitted. □

Example 3.3.3 Given column_vector $\mathcal{V}^{f1}_{4,3} = [\,3\;3\;5\;5\;0\;0\;4\;0\;3\;3\;2\;2\;3\;2\;1\;0\,]$ and $pg = [\,1\;1\;0\;0\;1\;1\;0\;1\;1\;1\;1\;1\;0\;0\;1\;1\,]$:

$$
\begin{aligned}
\mathcal{U}^f_{4,3} &= column_encode(\mathcal{V}^f_{4,3}, pg) \\
&= ([\,3\;3\;5\;5\;0\;0\;4\;0\;3\;3\;2\;2\;3\;2\;1\;0\,], [\,1\;1\;0\;0\;1\;1\;0\;1\;1\;1\;1\;1\;0\;0\;1\;1\,]) \\
&= [\,5\;5\;7\;7\;0\;0\;6\;0\;5\;5\;4\;4\;3\;2\;1\;0\,].
\end{aligned}
$$

Subsequently, $pg = [\,1\;1\;0\;0\;1\;1\;0\;1\;1\;1\;1\;1\;0\;0\;1\;1\,]$ partitions $\mathcal{U}^f_{4,3}$ into $S_0 = \{0, 1, 4, 5\}$ and $S_1 = \{2, 3, 6, 7\}$ which is an uni-code assignment ($S_0 \cap S_1 = \phi$). □

We use the following algorithm to find multi-code shared subfunction encodings among $\mathcal{V}^f$'s.

Algorithm *multi_code_shared_subfunction_encodings*:

Given a list VL^f of $\mathcal{V}^f$s:

1. Calculate $R(\mathcal{V}^f)$ for each $\mathcal{V}^f$ in VL^f.

2. Order VL^f in increasing order of $R(\mathcal{V}^f)$.

3. Pick the first column_vector $\mathcal{V}^{f_p}_{i_p,j_p}$ from the VL^f, produce the uni-code assignment *pg*-set of $\mathcal{V}^{f_p}_{i_p,j_p}$ and remove $\mathcal{V}^{f_p}_{i_p,j_p}$ from VL^f. Denote this pg-set by pgs_p.

4. For each pg_k in pgs_p do

```
{
    For each V^{fq}_{iq,jq} in VL^f that has R(V^{fq}_{iq,jq}) > 0 do
    {
        Divide V^{fq}_{iq,jq} into S0 and S1 according to pg_k.
        /* Check if this leads to a permissible g-function code of V^{fq}_{iq,jq} */
        if (| S0 |≤ 2^{jq-1} and | S1 |≤ 2^{jq-1}) then
         {
            V^{fq}_{iq,jq} = column_encode(V^{fq}_{iq,jq}, pg_k)
            Update R(V^{fq}_{iq,jq}) value and reposition V^{fq}_{iq,jq} in VL^f.
         }
    }
}
```

5. Repeat steps 3 - 4 until all $\mathcal{V}^f$'s are processed.

6. Use the *minimum_subfunction_covering* algorithm to find a minimum set of *pg*-sets such that each $\mathcal{V}^{f_q}_{i_q,j_q}$ has a j_q-bit encoding.

3.4 Common Subfunction Extraction for Large Bound Sets

The common subfunctions with large variable support sizes have more logic sharing compared with the common subfunctions with smaller variable support sizes. To extract common subfunctions with large variable support sizes using function decomposition, a shared subfunction encoding scheme that can handle the encoding problem for subfunctions with large support size is needed. We first discuss the complexity of the shared subfunction encoding for various bound set sizes, then introduce a new graph-based encoding scheme which can deal with any size of common subfunctions extraction.

3.4.1 Encoding Complexity

In the previous section, the shared subfunction encoding scheme first generates the uni-code *pg*-set for each individual Boolean function. Next, it searches for the common subfunctions among these *pg*-sets. This method is practical only when either the bound set size is small (≤ 5) or the compatible class size $\mid \mathcal{S} \mid$ is small (≤ 8). The following paragraphs describe the complexity of this scheme.

Lemma 3.4.1 Given column_vector $\mathcal{V}$ and column_set $\mathcal{S}$ with $bit_size(\mathcal{S}) = j$ and the number of compatible classes $\mid \mathcal{S} \mid = k$, the number of different g-functions which can be generated is equal to the cardinality of the uni-code *pg*-set of $\mathcal{V}$ which is given by the following equation:

$$\mid pg_set \mid \;=\; C^{k-1}_{k-2^{j-1}-1} + \ldots + C^{k-1}_{2^{j-1}-1} = \sum_{l=k-2^{j-1}}^{2^{j-1}} C^{k-1}_{l-1}$$

where $2^{j-1} <| \mathcal{S} |= k \leq 2^j$. Because neither $| S_0 |$ nor $| S_1 |$ can exceed 2^{j-1}, the minimum and maximum sizes of S_0 are $k - 2^{j-1}$ and 2^{j-1}, respectively. $k-1$ and $l-1$ are used because $0 \in S_0$ (Definition 3.2.1).

Example 3.4.1 Let $\mathcal{V} = [\ 2\ 2\ 1\ 1\ 1\ 1\ 0\ 0\ ]$, then

$$\begin{aligned} pg_{\mathcal{V},\{0\},\{1,2\}} &= [\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ ], \\ pg_{\mathcal{V},\{0,1\},\{2\}} &= [\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ ], \text{ and} \\ pg_{\mathcal{V},\{0,2\},\{1\}} &= [\ 0\ 0\ 1\ 1\ 1\ 1\ 0\ 0\ ]. \end{aligned}$$

The *pg*-set of $\mathcal{V}$ is $\{[\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ ], [\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ ], [\ 0\ 0\ 1\ 1\ 1\ 1\ 0\ 0\ ]\}$. Note that we need not consider $pg_{\mathcal{V},\{1\},\{0,2\}}$ because it is equal to $\overline{pg_{\mathcal{V},\{0,2\},\{1\}}}$. □

Example 3.4.2 For $| \mathcal{S} |= k = 3, 4, 5, 6, 7$, and 8, the cardinalities of their *pg*-sets are:

$$\begin{aligned} k = 3{:}\quad & C_0^2 + C_1^2 = 1 + 2 = 3, \\ k = 4{:}\quad & C_1^3 = 3, \\ k = 5{:}\quad & C_0^4 + C_1^4 + C_2^4 + C_3^4 = 1 + 4 + 6 + 4 = 15, \\ k = 6{:}\quad & C_1^5 + C_2^5 + C_3^5 = 5 + 10 + 10 = 25, \\ k = 7{:}\quad & C_3^6 + C_4^6 = 20 + 15 = 35, \\ k = 8{:}\quad & C_3^7 = 35. \end{aligned}$$

□

Given a bound set B of size i, $| B |= i$, function f and the corresponding compatible class size (column_set size) $| \mathcal{S} |= k$, k must be less than or equal to 2^{i-1} ($k \leq 2^{i-1}$) for function f to be decomposable. Table 3.1 shows that size of the *pg*-set is exponential in the column_set size. Generating the *pg*-set for each individual function prior to searching for the shared subfunctions is therefore impractical for large k.

B	k = |$\mathcal{S}$|	*pg*-set size
4	8	35
5	9	255
5	10	501
5	11	957
5	12	1749
5	13	3003
5	14	4719
5	15	6435
5	16	6435
6	17	65535
...	...	...
6	27	50,480,055
6	28	87,922,215
6	29	145,422,675
6	30	222,981,435
6	31	300,540,195
6	32	300,540,195
7	64	9.16×10^{17}

Table 3.1: *pg*-set size versus column_set size k

3.4.2 An Encoding Scheme for Large Compatible Class Size

The problem with large column_set size is that the number of *pg* functions in their corresponding *pg*-sets becomes exponentially large. A graph-based encoding scheme which can find shared subfunctions without generating the *pg*-sets will be described in this section.

Definition 3.4.1 A *bipartite* graph $G = (V^s, V^t, E)$ is a graph where the vertices can be partitioned into two sets, V^s and V^t such that edges of G exist only between vertices of V^s and V^t.

We can use a bipartite graph to represent the compatibility relation between two column_vectors.

Definition 3.4.2 Given column_vectors $\mathcal{V}^{f_1} = \langle v_{11}, v_{12}, \ldots, v_{1N}\rangle$, $\mathcal{V}^{f_2} = \langle v_{21}, v_{22}, \ldots, v_{2N} \rangle$ and the corresponding column_sets $\mathcal{S}^{f_1}$, $\mathcal{S}^{f_2}$, we construct a bipartite graph $G = (V^s, V^t, E)$ as follows: $V^s = \mathcal{S}^{f_1}$, $V^t = \mathcal{S}^{f_2}$ and there exists an edge

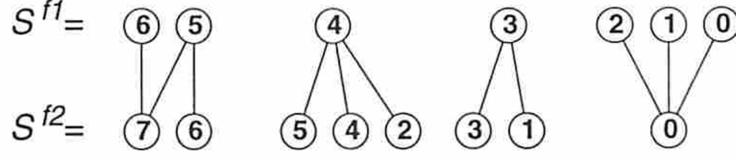


Figure 3.5: (Code) dependency graph

between $s_i \in V^s$ and $s_j \in V^t$ exactly if there is a column index k such that the kth element of $\mathcal{V}^{f_1}$ is s_i and the kth element of $\mathcal{V}^{f_2}$ is s_j. This graph will be referred to as *(code) dependency graph.*

Example 3.4.3 Given column_vectors $\mathcal{V}^{f_1}$ and $\mathcal{V}^{f_2}$,

$$\begin{aligned} \mathcal{V}^{f_1} &= [\,6\;5\;5\;5\;5\;4\;4\;4\;3\;3\;4\;3\;2\;1\;0\;0\,]\; \mathcal{S}^{f1} = \{0,1,2,3,4,5,6\} \\ \mathcal{V}^{f_2} &= [\,7\;6\;6\;7\;6\;5\;4\;2\;1\;3\;2\;1\;0\;0\;0\;0\,]\; \mathcal{S}^{f2} = \{0,1,2,3,4,5,6,7\} \end{aligned}$$

the dependency graph G is described as $V^s = \{0^{f_1}, 1^{f_1}, 2^{f_1}, 3^{f_1}, 4^{f_1}, 5^{f_1}, 6^{f_1}\}$, $V^t = \{0^{f_2}, 1^{f_2}, 2^{f_2}, 3^{f_2}, 4^{f_2}, 5^{f_2}, 6^{f_2}, 7^{f_2}\}$, $E = \{(0^{f_1}, 0^{f_2}), (1^{f_1}, 0^{f_2}), (2^{f_1}, 0^{f_2}), (3^{f_1}, 1^{f_2}), (3^{f_1}, 3^{f_2}), (4^{f_1}, 2^{f_2}), (4^{f_1}, 4^{f_2}), (4^{f_1}, 5^{f_2}), (5^{f_1}, 6^{f_2}), (5^{f_1}, 7^{f_2}), (6^{f_1}, 7^{f_2})\}$ (cf. Figure 3.5). □

The significance of the dependency graph is that for any g-function that is shared between f_1 and f_2, the elements of $\mathcal{V}^{f_1}$ and $\mathcal{V}^{f_2}$ which correspond to connected vertices of this graph must be assigned the same code (logic value "0" or "1").

Lemma 3.4.2 Given the dependency graph $G = (\mathcal{S}^{f_1}, \mathcal{S}^{f_2}, E)$ corresponding to two column_vectors, $\mathcal{V}^{f_1}$ and $\mathcal{V}^{f_2}$, if G is connected, then f_1 and f_2 do not have a uni-code shared subfunction with respect to the given bound set.

Proof: Pick any vertex v_i from one of the sets, if we assign this v_i to "0", in order to produce shared code between $\mathcal{S}^{f_1}$ and $\mathcal{S}^{f_2}$, the elements that are equal to v_i in $\mathcal{V}^{f_1}$ and the corresponding (same index) elements in $\mathcal{V}^{f_2}$ must be assigned

to "0". Let's assume these elements are $u_1, \ldots, u_m$. Now all entries of $\mathcal{V}^{f_2}$ that contain an element of $u_1, \ldots, u_m$ must be assigned to "0". But this implies that the corresponding (same index) elements of $\mathcal{S}^{f_1}$ must be assigned to "0" and so on. Therefore, if the graph is connected and for uni-code assignment, then the encoding of $\mathcal{S}^{f_1}$ and $\mathcal{S}^{f_2}$ will be all "0" vector, which is not valid. Thus f_1 and f_2 have no uni-code shared subfunction. □

Lemma 3.4.3 Given column_vectors $\mathcal{V}^s$, $\mathcal{V}^t$ with their column_sets $\mathcal{S}^s$, $\mathcal{S}^t$ and the corresponding dependency graph $G = (\mathcal{S}^s, \mathcal{S}^t, E)$, if there exists a partitioning that divides G into $G_0 = (\mathcal{S}^{s_0}, \mathcal{S}^{t_0}, E_0)$ and $G_1 = (\mathcal{S}^{s_1}, \mathcal{S}^{t_1}, E_1)$ such that each subset of vertices fulfills the size constraints: $\mid \mathcal{S}^{s_0} \mid \leq 2^{bit_size(\mathcal{S}^s)-1}$, $\mid \mathcal{S}^{s_1} \mid \leq 2^{bit_size(\mathcal{S}^s)-1}$, $\mid \mathcal{S}^{t_0} \mid \leq 2^{bit_size(\mathcal{S}^t)-1}$, and $\mid \mathcal{S}^{t_1} \mid \leq 2^{bit_size(\mathcal{S}^t)-1}$, then a uni-code shared subfunction between $\mathcal{S}^s$ and $\mathcal{S}^t$ can be found by giving "0" ("1") code to elements of $\mathcal{V}^s$ that appear in $\mathcal{S}^{s_0}$ ($\mathcal{S}^{s_1}$). Similarly, elements of $\mathcal{V}^t$ that appear in $\mathcal{S}^{t_0}$ ($\mathcal{S}^{t_1}$) are assigned to "0" ("1"), respectively.

Proof: Follows from Lemma 3.4.2. □

Example 3.4.4 Continuing with example 3.4.3, the dependency graph G can be divided into two sub-graphs (Figure 3.6.(a)) where each sub-graph satisfies the size constraints. Thus there is a uni-code subfunction encoding [0 0 0 0 0 1 1 1 0 0 1 0 1 1 1 1] by assigning the {3, 5, 6} elements of $\mathcal{V}^{f_1}$ to "0" and the {0, 1, 2, 4} elements of $\mathcal{V}^{f_1}$ to "1" Similarly, the {1, 3, 6, 7} elements of $\mathcal{V}^{f_2}$ are assigned to "0" and the {0, 2, 4, 5} elements of $\mathcal{V}^{f_2}$ to "1" (Figure 3.6.(b)).

$$\begin{aligned}
\mathcal{V}^{f_1} &= [\, 6\ 5\ 5\ 5\ 5\ \underline{4\ 4\ 4}\ 3\ 3\ \underline{4}\ 3\ \underline{2\ 1\ 0\ 0}\,] \\
\mathcal{V}^{f_2} &= [\, 7\ 6\ 6\ 7\ 6\ \underline{5\ 4\ 2}\ 1\ 3\ \underline{2}\ 1\ \underline{0\ 0\ 0\ 0}\,] \\
shared_code &= [\, 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 1\ 0\ 1\ 1\ 1\ 1\,]
\end{aligned}$$

□

Definition 3.4.3 *The Dependency Graph Partitioning Problem*

Given column_vectors $\mathcal{V}^s$, $\mathcal{V}^t$ and the corresponding dependency graph $G = (\mathcal{S}^s$,

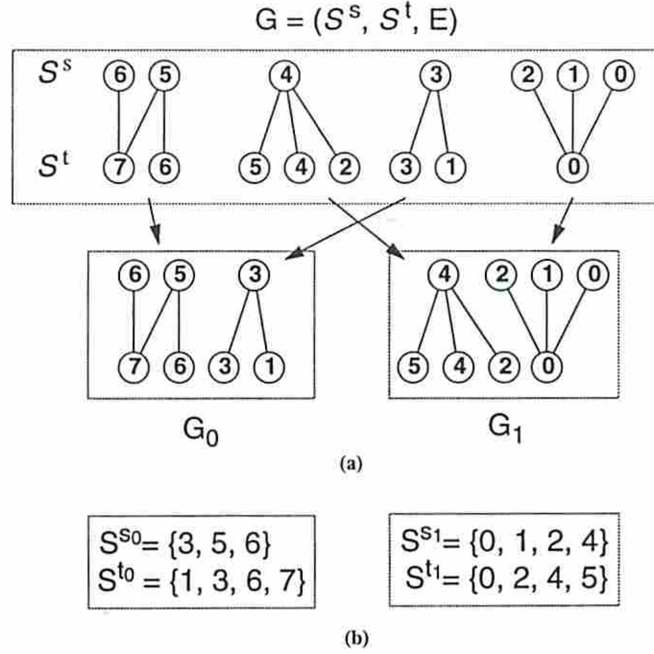


Figure 3.6: Partitioned graph

$\mathcal{S}^t$, E), find a two-way partition $G_0 = (\mathcal{S}^{s_0}, \mathcal{S}^{t_0}, E_0)$, $G_1 = (\mathcal{S}^{s_1}, \mathcal{S}^{t_1}, E_1)$ such that $| \mathcal{S}^{s_0} | \leq 2^{bit_size(\mathcal{V}^s)-1}$, $| \mathcal{S}^{s_1} | \leq 2^{bit_size(\mathcal{V}^s)-1}$, $| \mathcal{S}^{t_0} | \leq 2^{bit_size(\mathcal{V}^t)-1}$, $| \mathcal{S}^{t_1} | \leq 2^{bit_size(\mathcal{V}^t)-1}$ and there are no edges between G_0 and G_1.

Assume each connected sub-graph $sg_i = (V^{s_i}, V^{t_i}, E)$ in G as an item and the final partitions G_0 and G_1 as two bins, bin_0 and bin_1, each with a capacity of $\langle 2^{bit_size(\mathcal{V}^s)-1}, 2^{bit_size(\mathcal{V}^t)-1} \rangle$ (Figure 3.7). The problem is to put all sg_i's with size $\langle | V^{s_i} |, | V^{t_i} | \rangle$ into these two bins while satisfying the two-dimensional capacity constraints. This is exactly the *two-dimensional bin-packing problem* which is NP-complete in the strong sense [29]. We use the following greedy algorithm to solve this partitioning problem.

Algorithm *bipartition_dependency_graph*:
Given a graph $G = (V^s, V^t, E)$, and let $C_s = 2^{(\lceil log_2 |V^s| \rceil - 1)}$ and $C_t = 2^{(\lceil log_2 |V^t| \rceil - 1)}$.

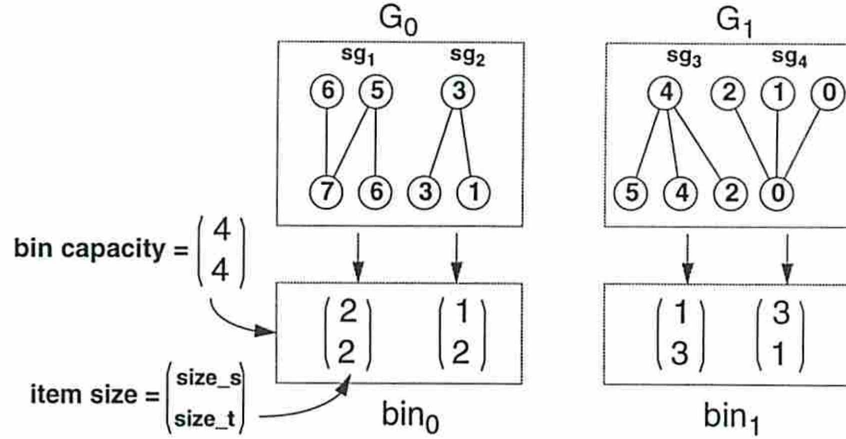


Figure 3.7: Conversion to bin packing problem

1. Separate G into N sub-graphs $SG = (sg_1, sg_2, \ldots, sg_N)$ where each sg_i is a connected graph but every pair of sub-graphs are disjoint. If there is only one graph in SG, then return ϕ. /* No solution (lemma 3.4.2) */

2. For i = 1 to N do

   {

   Let $sg_i = (V^{s_i}, V^{t_i}, E) \in SG$;

   If $(size(V^{s_i}) > C_s \mid\mid size(V^{t_i}) > C_t)$, then return ϕ. /* No solution */

   }

3. Set $B_0_size_s = 0$, $B_1_size_s = \sum_{i=1}^{N} size(V^{s_i})$

   Set $B_0_size_t = 0$, $B_1_size_t = \sum_{i=1}^{N} size(V^{t_i})$

4. Pick sg_i which has $max(size(V^{s_i}) + size(V^{t_i}) - \mid size(V^{s_i}) - size(V^{t_i}) \mid)$ and $(B_0_size_s + size(V^{s_i})) \leq C_s$, $(B_0_size_t + size(V^{t_i})) \leq C_t$

5. Put sg_i into B_0 and update

   $B_0_size_s = B_0_size_s + size(V^{s_i})$,

   $B_0_size_t = B_0_size_t + size(V^{t_i})$,

   $B_1_size_s = B_1_size_s - size(V^{s_i})$,

   $B_1_size_t = B_1_size_t - size(V^{t_i})$.

6. Repeat steps 4 and 5 until no such sg_i exists in SG.

7. Put the rest of sg's in SG into B_1.

8. If ($B_1_size_s \leq C_s$ and $B_1_size_t \leq C_t$) then

   return $\{B_0, B_1\}$. /* partition result */

   else

   return ϕ /* No solution */.

Step 1 can be performed in time linear in the number of edges in G. Step 2 checks the feasibility of each sub-graph generated in step 1. Step 3 initializes the size constraints for the two parts (bins). We start by putting all sg_i's into B_1. Next, we pick sg_i from B_1 and move it to B_0. The selected sg_i is the one that maximizes the minimum of $| V^{s_i} |$ and $| V^{t_i} |$. This approximately corresponds to picking a sg_i with maximum vertex count, yet minimum difference between its two vertex sets.

We update the size constraints in step 5 after we move sg_i to B_0 and repeat the process until no more sg_i's can be moved to B_0. We check if B_1 also satisfies its constraints in the end. If so, we have a valid partition, otherwise no solution has been found. Steps 4 to 6 can be performed in $O(N^2)$ where N is the number of disconnected sub-graphs in G. Since other steps can be performed in linear time, this algorithm has time complexity of $O(N^2)$. The worst case of N is equal to $| \mathcal{S} |$, thus the worst case time complexity of this algorithm is still polynomial in terms of the compatible class size $| \mathcal{S} |$.

We use the following algorithm, *generate_pg_sets*, to generate the *pg*-sets for all Boolean functions. This algorithm will produce exactly the same results as the shared subfunction encoding algorithm described in Section 3.2 for small compatible class size ($| \mathcal{S} | \leq 8$), but will use the bipartition_dependency_graph algorithm to search for common subfunctions when $| \mathcal{S} | > 8$.

Definition 3.4.4 Operator *Complete_encoding* is defined as follows:
$Complete_encoding(\mathcal{V}_{i,j}, \mathcal{S}_{i,j}, pg_1) = \{pg_2, \ldots, pg_j\}$ where $\{pg_1, \ldots, pg_j\}$ partitions $\mathcal{S}_{i,j}$ into $P^j = \{S_0, \ldots, S_{2^j-1}\}$ such that $\forall S_q \in P^j, |S_q| \leq 1$.

Algorithm *generate_pg_sets*:

Given a set of column_vectors $\mathcal{V}^1, \mathcal{V}^2, \ldots \mathcal{V}^N$ and their column_sets $\mathcal{S}^1, \mathcal{S}^2 \ldots \mathcal{S}^N$.

1. Separate column_sets into two groups, *Group_1* has all $\mathcal{S}^i$s with $|\mathcal{S}^i| \leq 8$ and *Group_2* has all $\mathcal{S}^i$s with $|\mathcal{S}^i| > 8$.

2. For each $\mathcal{S}^i$ in *Group_1*, generate its pg_i-set. /* $|\mathcal{S}^i| \leq 8$ */

3. For each pair of $\langle \mathcal{S}^i, \mathcal{S}^j \rangle$ in *Group_2*, do /* $|\mathcal{S}^i| > 8$, $|\mathcal{S}^j| > 8$ */

    {

    3.1 Create $G = (\mathcal{S}^i, \mathcal{S}^j, E)$, $C_s = 2^{(\lceil log_2|\mathcal{S}^i|\rceil - 1)}$, $C_t = 2^{(\lceil log_2|\mathcal{S}^j|\rceil - 1)}$.

    3.2 Let G_set = bipartition_dependency_graph(G, C_s, C_t);

    3.3 If G_set = $\{G_0, G_1\}$ then

    /* G was divided into two subgraphs G_0 and G_1 */

    {

    3.4 pg-code = $[c_0, c_1, \ldots, c_N]$

    where c_k = "0" if the kth element of $\mathcal{V}^i$ is in $\mathcal{S}^{i_0}$,

    otherwise c_k = "1".

    3.5 Insert this pg-code into pg_i-set and pg_j-set.

    }

    }

4. For each pair of $\langle \mathcal{S}^i, \mathcal{S}^j \rangle$ such that $\mathcal{S}^i \in$ *Group_1*, $\mathcal{S}^j \in$ *Group_2*, do

    {

    4.1 For each pg_k code in pg_i-set,

    4.2 If pg_k is a *permissible g-function encoding* of $\mathcal{S}^j$,

    then insert pg_k into pg_j-set.

}

5. For each $\mathcal{S}^i$ in $Group_2$, do /* | $\mathcal{S}^i$ |> 8 */

   {

   Set $complete_code_set = \phi$

   For each pg_k in pg_i-set, do

   {

   5.1 $new_codes = Complete_encoding(\mathcal{V}^i, \mathcal{S}^i, pg_k)$

   5.2 $complete_code_set = complete_code_set + new_codes,$

   }

   Append $complete_code_set$ to pg_i-set.

   }

□

In step 1, column_sets are put into two groups according to their size (> 8 or ≤ 8). In step 2, we produce all possible encodings for $\mathcal{S}^i$'s whose column_set size is ≤ 8. Step 3 produces shared codes between $\mathcal{S}^i$, $\mathcal{S}^j$ when $|\ \mathcal{S}^i\ | > 8$ and $|\ \mathcal{S}^j\ | > 8$. In steps 3.1 and 3.2, we prepare the dependency graph, constraints C_s, C_t and call on the bipartition_dependency_graph algorithm to find a bipartitioning solution. If such a solution exists, we produce the shared code and insert it in the *pg*-sets of $\mathcal{S}^i$ and $\mathcal{S}^j$ in steps 3.4 and 3.5. Step 4 finds shared codes between $\mathcal{V}^i$ and $\mathcal{V}^j$ when $|\ \mathcal{S}^i\ | > 8$ but $|\ \mathcal{S}^j\ | \leq 8$ by searching through pg_j-set for possible shared codes between $\mathcal{V}^i$ and $\mathcal{V}^j$. In step 5, we produce the complete encodings for each shared code in the *pg*-set obtained in steps 3 and 4.

After generating the *pg*-sets for all functions using *generate_pg_sets* algorithm, we select a minimum number of *pg*-functions which produce valid, complete encoding of each function. This problem is formulated as *Minimum subfunction covering*

problem (Definition 3.2.5) described in Section 3.2. We use the following greedy algorithm for solving the minimum subfunction covering problem.

Algorithm *shared_subfunction_encoding_covering*:
Given a collection of *pg*-sets in which each *pg*-set is annotated with a value *count* initialized to its bit_size (this is for indicating when a function has a complete encoding) and a code set *cs* initialized to ϕ.

1. Find a *pg*-code, g, that occurs in the *pg*-sets most frequently. If there is a tie, then choose the *pg*-code with minimum $supp(g)$ and put it in the code set cs.

2. For each *pg*-set that contains g, decrease its *count* by 1. If $count = 0$, remove this set.

3. For each *pg*-set that contains g, remove any *pg*-code that can not produce the valid encodings with g for corresponding column_vector $\mathcal{V}$.

4. Repeat steps 1-3 until every *pg*-set is removed. Then, return the code set cs, which defines the encoding for all *pg*-sets.

3.4.3 Graph-based Encoding Using Multi-code Assignments

In this section, we focus on solving the dependency graph non-disjunctive partitioning problem to find multi-code shared subfunction encoding. More shared subfunctions can be found by allowing multi-code assignments. The following example shows that shared subfunctions can be found by using multi-code assignments, but not using uni-code assignments.

Example 3.4.5 Given column_vectors $\mathcal{V}^{f_1}$ and $\mathcal{V}^{f_2}$,

$$\begin{aligned} \mathcal{V}^{f_1} &= [\,6\;5\;5\;5\;5\;4\;4\;4\;3\;3\;4\;3\;2\;1\;0\;0\,]\; \mathcal{S}^{f1} = \{0,1,2,3,4,5,6\} \\ \mathcal{V}^{f_2} &= [\,7\;6\;6\;7\;6\;5\;3\;4\;1\;3\;2\;1\;0\;0\;0\;0\,]\; \mathcal{S}^{f2} = \{0,1,2,3,4,5,6,7\} \end{aligned}$$

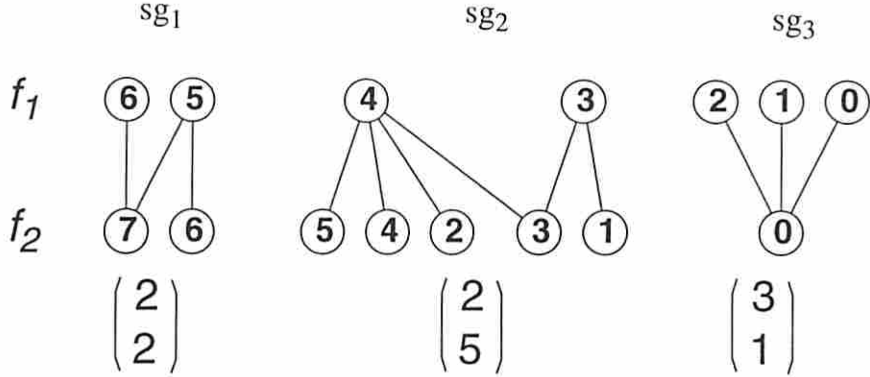


Figure 3.8: Dependency graph with large subgraph

The dependency graph G with three connected sub-graphs, $sg_1 = (V^{s_1}, V^{t_1}, E)$, $sg_2 = (V^{s_2}, V^{t_2}, E)$, $sg_3 = (V^{s_3}, V^{t_3}, E)$, is shown in Figure 3.8. It is clear to see that there is no such partitioning that divides G into $G_0 = (\mathcal{S}^{s_0}, \mathcal{S}^{t_0}, E_0)$ and $G_1 = (\mathcal{S}^{s_1}, \mathcal{S}^{t_1}, E_1)$ such that each subset of vertices fulfills the size constraints: $\mid \mathcal{S}^{s_0} \mid \leq 4$, $\mid \mathcal{S}^{s_1} \mid \leq 4$, $\mid \mathcal{S}^{t_0} \mid \leq 4$, and $\mid \mathcal{S}^{t_1} \mid \leq 4$, because the size of connected sub-graph, sg_2, alone has excess the size constraints ($\mid V^{t_2} \mid = 5 \not\leq 4$). Thus there is no uni-code assignment shared subfunction can be found in these two functions.

If we divide sub-graph sg_2 into two connected sub-graphs sg_4 and sg_5 by duplicating vertex "4", then a partitioning that divides G into G_0, G_1 (Figure 3.9) and also fulfills the size constraints can be found (e.g., $G_0 = \{sg_3, sg_4\}$, $G_1 = \{sg_1, sg_5\}$). □

Lemma 3.4.4 Given a dependency graph G and connected sub-graph $sg_i = (V^{s_i}, V^{t_i}, E)$ within it, dividing sg_i into two connected sub-graphs sg_{i_0} and sg_{i_1} by duplicating vertex $v \in V^{s_i}$ or $\in V^{t_i}$ will allow possible multi-code assignment in this column_id v in the given column_vector.

Proof: If in the final solution of bin packing, sg_{i_0} and sg_{i_1} are assigned to different bins (e.g., bin_0 and bin_1), then the duplicated vertex v which corresponds to the

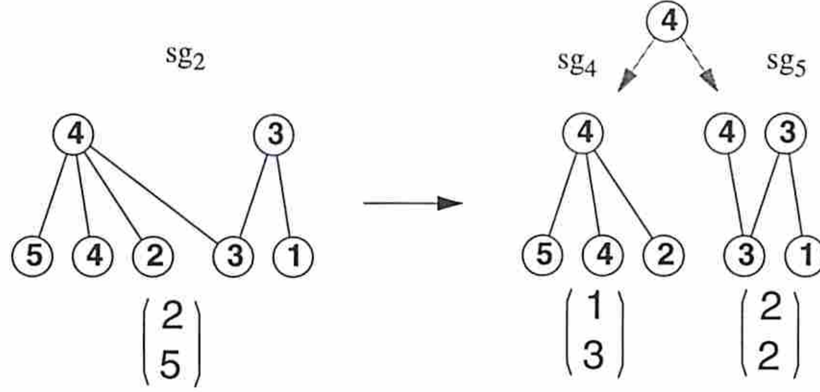


Figure 3.9: Divisible item (multi-code assignment)

column_id v in the column_vector can be assigned to "0" or "1" according to the bin it belongs to. □

The number of different multi-code assignments for a given column_vector is exponential as we described in Section 3.3. Since most of the connected bipartite graphs in our experiment benchmark circuits have tree structure, we only discuss the partitioning problem for connected acyclic bipartite graphs (tree structures) here.

The number of different ways to partition a connected acyclic bipartite graph is given as follows:

Lemma 3.4.5 Given a connected acyclic bipartite graph G with vertex $v_0, \ldots, v_{n-1}$ and $degree(v_i)$ is the number of edges connected to v_i, the number of possible ways to partition this graph into two graphs by duplicating any one vertex is given by:

$$num_partition(G) = \frac{1}{2}\sum_{i=0}^{n-1}\sum_{j=0}^{degree(v_i)-1} C_j^{degree(v_i)} \tag{3.3}$$

Proof: For each vertex v_i, the number of ways to duplicate this vertex is the number of ways to partition the edges that connect to this vertex which is $\frac{1}{2}\sum_j$

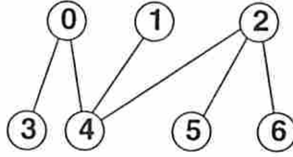


Figure 3.10: A connected acyclic bipartite graph

$C_j^{degree(v_i)}$ where j runs from 0 to $degree(v_i) - 1$. The summation over all vertices in the graph gives the number of possible ways to partition it. □

Example 3.4.6 Given a connected acyclic bipartite graph in Figure 3.10, the number of possible two-way partition is 7.

We use a heuristic algorithm to find a two-way partition on the large connected acyclic bipartite graph (larger than the bin size constraints). This algorithm is called only when the uni-code graph-based algorithm (steps 1, 2 and 8 in *bipartition_dependency_graph*) fails to find a solution.

The basic idea of this heuristic algorithm is to search through graph G and find the connected subgraphs which are oversized and then separate the oversized subgraphs into two connected subgraphs. This is in turn performed by splitting each vertex in the subgraph into two vertices with about equal number of incident edges. If the resulting partition fulfills the size constraints, then we proceed to the next oversized subgraph, otherwise we look for another vertex to split. If every vertex splitting fails to produce the desired property, then we exit the procedure with a FALSE flag. This algorithm can be performed in time linear in the number of edges.

3.4.4 Extending Graph-Based Shared Subfunction Encoding

The graph-based shared subfunction encoding scheme solved the problem of finding shared subfunctions between two functions. This scheme can be extended to find

shared subfunctions among functions (more than two functions). In general, extending the two-dimensional bin-packing problem into N-dimensional bin-packing problem can solve this problem.

Given column_vectors $\mathcal{V}^{f_1}$, $\mathcal{V}^{f_2}$, ..., $\mathcal{V}^{f_N}$ of functions f_1, f_2, ..., f_N, the corresponding $N-1$ code dependency graphs, $G_{1,2}$, $G_{2,3}$, ..., $G_{N-1,N}$, can be builded. One single graph can be builded by sharing the vertex of f_2 to f_{N-1} among each of the code dependency graphs. This single graph is called N-connected dependency graph. In this graph, each connected subgraph has N size associate with it. Packing these connected subgraphs (N-dimensional items) into two-bins with N-dimensional size constraints is exactly the N-dimensional bin-packing problem. The following example illustrates the case of finding shared subfunctions among three functions, f_1, f_2 and f_3.

Example 3.4.7 Given column_vectors, $\mathcal{V}^{f_1} = [\ 6\ 5\ 5\ 5\ 5\ 4\ 4\ 4\ 3\ 3\ 4\ 3\ 2\ 1\ 0\ 0\ ]$, $\mathcal{V}^{f_2} = [\ 7\ 6\ 6\ 7\ 6\ 5\ 4\ 2\ 1\ 3\ 2\ 1\ 0\ 0\ 0\ 0\ ]$ and $\mathcal{V}^{f_3} = [\ 6\ 6\ 6\ 6\ 5\ 4\ 4\ 3\ 1\ 1\ 2\ 1\ 0\ 0\ 0\ 0\ ]$ of functions f_1, f_2, f_3, a 3-connected dependency graph G is builded (Figure 3.11). For each connected subgraph in G, three sizes are associated with it. For example, the subgraph sg_1 has size of (2, 2, 2). These subgraphs can be packed into two bins that $Bin_0 = \{sg_1, sg_3\}$ and $Bin_1 = \{sg_2, sg_4\}$. This resulting code $= [\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 1\ 0\ 1\ 1\ 1\ 1\ ]$ is shared among $\mathcal{V}^{f_1}$, $\mathcal{V}^{f_2}$ and $\mathcal{V}^{f_3}$. □

3.4.5 Output Partitioning for Graph-based Encoding

Given N-outputs function, if we directly build N-connected dependency graph when N is large, then usually no solution can be found in this N-dimensional bin-packing problem. Partitioning the outputs into groups before applying the graph-based encoding is thus necessary in order to find a solution. This partitioning problem, which is similar to the output grouping problem for column_encoding described in Section 3.1.1, is formulated as follows.

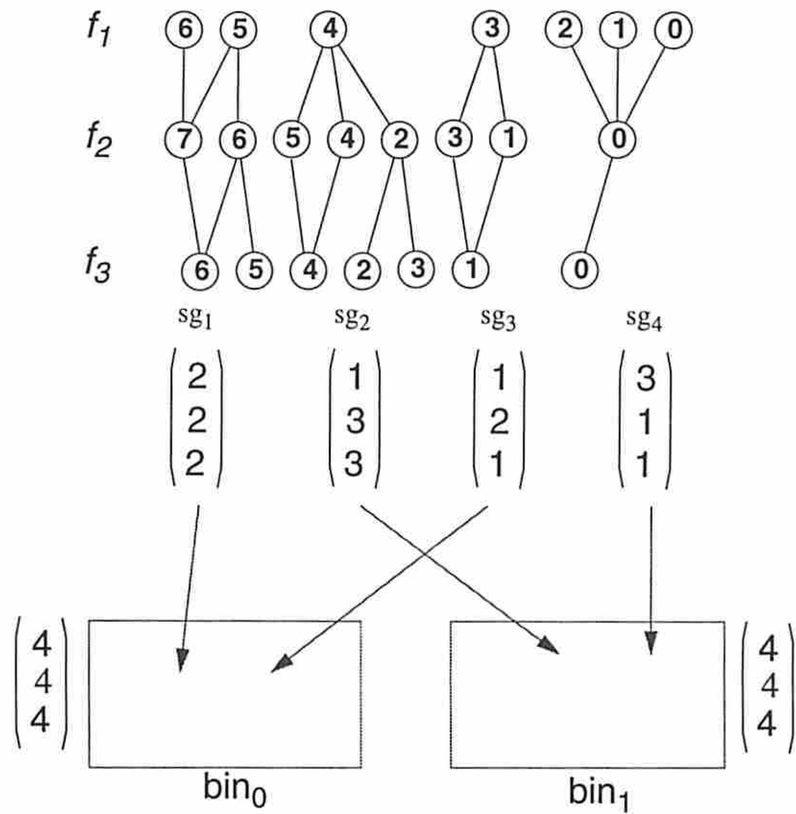


Figure 3.11: N-dimensional bin packing problem (N = 3)

Definition 3.4.5 Given a set of column_vectors $\mathcal{V}_{i,j_k}^{f_k}$'s with respect to m Boolean functions $F = \langle f_0, \ldots, f_{m-1} \rangle$ and bound set B, partition this set into $P_0, \ldots, P_{\ell-1}$ such that there is a graph-based encoding solution for each set P_i and l is minimized.

We use the following greedy algorithm to solve this problem.

Algorithm *output_partition_for_graph_based_encoding*:

1. Order $\mathcal{V}_{i,j_k}^{f_k}$ in increasing order of *residue* of $\mathcal{V}_{i,j_k}^{f_k}$, $R(\mathcal{V}_{i,j_k}^{f_k})$.

2. Starting from the first element of the above list, put as many column_vectors $\mathcal{V}_{i,j_k}^{f_k}$'s as possible that dependency graph for these column_vectors still can find shared subfunctions. As soon as dependency graph can not result in finding shared subfunction, initiate a new group of outputs. Repeat until all column_vectors are processed.

The above algorithm is based on the observation that a $\mathcal{V}^{f_k}$ with larger residue, $R(\mathcal{V}^{f_k})$ has larger number of column_ids that multi-code assignment is allowed, so it has higher chance finding shared subfunction code with other column_vectors. Thus choose the column_vector with smaller residue first can lead to fewer number of groups that maximize the sharing.

Example 3.4.8 Given a set of column_vectors as following:

$\mathcal{V}^{f_1} = [\,7\;7\;6\;6\;5\;5\;4\;4\;3\;3\;2\;2\;1\;1\;0\;0\,]$,

$\mathcal{V}^{f_2} = [\,7\;7\;7\;6\;6\;5\;5\;4\;4\;3\;3\;2\;2\;1\;1\;0\,]$,

$\mathcal{V}^{f_3} = [\,6\;6\;6\;6\;5\;5\;4\;4\;3\;3\;2\;1\;1\;0\;0\;0\,]$, and

$\mathcal{V}^{f_4} = [\,4\;4\;4\;4\;3\;2\;2\;1\;1\;1\;1\;0\;0\;0\;0\;0\,]$.

If we build dependency graph on all column_vectors without using output_partition algorithm, then the resulting graph is a single connected graph and can not be partitioned. There is no shared code can be found.

If we apply output_partition algorithm, then two groups will be produced and the shared *pg*-code can be found.

group 1:

$$
\begin{aligned}
\mathcal{V}^{f_1} &= [\,7\;7\;6\;6\;5\;5\;4\;4\;3\;3\;2\;2\;1\;1\;0\;0\,] \\
\mathcal{V}^{f_3} &= [\,6\;6\;6\;6\;5\;5\;4\;4\;3\;3\;2\;1\;1\;0\;0\;0\,] \\
\textit{shared pg}\text{-code} &= [\,1\;1\;1\;1\;1\;1\;1\;1\;0\;0\;0\;0\;0\;0\;0\;0\,]
\end{aligned}
$$

group 2:

$$
\begin{aligned}
\mathcal{V}^{f_2} &= [\,7\;7\;7\;6\;6\;5\;5\;4\;4\;3\;3\;2\;2\;1\;1\;0\,] \\
\mathcal{V}^{f_4} &= [\,4\;4\;4\;4\;3\;2\;2\;1\;1\;1\;1\;0\;0\;0\;0\;0\,] \\
\textit{shared pg}\text{-code} &= [\,1\;1\;1\;1\;1\;0\;0\;1\;1\;1\;1\;0\;0\;0\;0\;0\,]
\end{aligned}
$$

□

3.5 Summary

In this chapter, we described four methods to extract common subfunctions from multiple Boolean functions. The first method (*column encoding*) is based on the stacking of the decomposition charts of the individual outputs; the second method (*uni-code shared subfunction encoding*) are based on the examination of all possible g-functions that can be generated; the third method (*multi-code shared subfunction encoding*) is the extension of the second method to allow the multi-code assignment to each g-function; the four method (*graph-based shared subfunction encoding*) is based on graph bipartitioning technique and can handle decomposition with respect to large column_set size.

In next chapter, we will apply these extraction techniques on different objective functions (e.g, minimum delay, minimum energy).

Chapter 4

Other Objective Functions

4.1 Decomposition for Minimum Delay

The problem of technology mapping of FPGA for minimum delay has been studied by [47, 26, 17, 19, 20]. These include mis-pga-d by Murgai *et al.* which combines the technology mapping with layout synthesis [47], Chortle-d by Francis *et al.* which minimizes the depth increase at each bin package step [26], and FlowMap by Cong *et al.* which is based on a topological labeling algorithm [20].

Our FGSyn_d [49, 50] is based on function decomposition for achieving minimum network depth as described next.

4.1.1 Minimum Delay Decomposition Using Unit Delay Model

We assume each node in the mapping result contributes to a constant delay (*unit delay model*) and each node can be direct mapped to a look-up-table (LUT) based logic block of FPGA. Thus, delay is determined by the maximum number of nodes (LUTs) on the path from primary inputs to primary outputs and is referred to

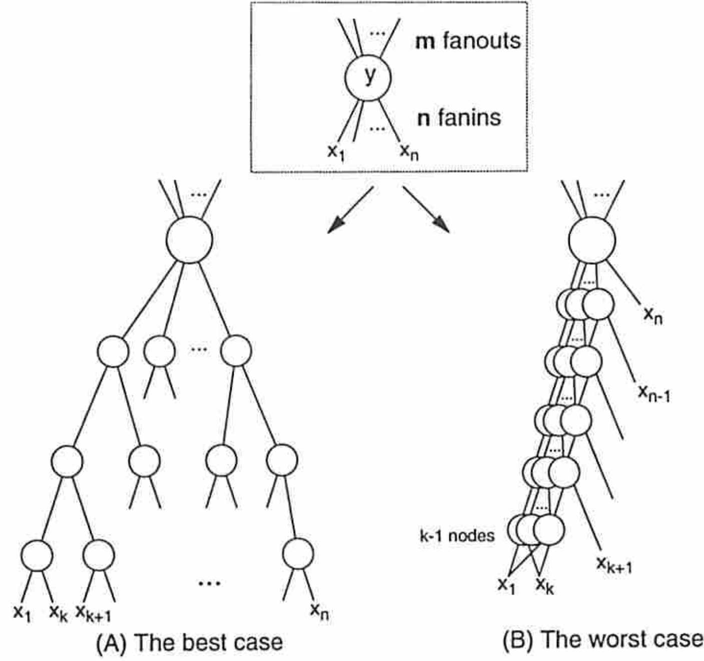


Figure 4.1: Upper and lower bound delay estimation

as LUT delay. Our delay minimization algorithm FGSyn_d is based on the Huffman algorithm for constructing minimum average code length [33].

Definition 4.1.1 Given a Boolean network N and a node $y \in N$, if y is a primary input, then the unit delay of y is given by:

$$u_delay(y) = 0 \tag{4.1}$$

If y is an internal node or a primary output with support size $\leq K$ (K is the number of inputs of LUT), then the delay of y is given by:

$$u_delay(y) = max\{u_delay(x_i) \mid x_i \in fanin(y)\} + d_{unit} \tag{4.2}$$

where d_{unit} which represents the intrinsic delay parameter is know for the given FPGA device.

For each g-function g_i, the the cost function for delay is:

$$pg_cost(g_i) = u_delay(g_i) \tag{4.3}$$

The delay calculation becomes more involved at the output of node f which is being decomposed since the new node f' (cf. Definition 2.1.1) may have an immediate fanins size $\geq K$ ($fi_cnt(f) \geq K$). This is discussed next.

Lemma 4.1.1 Given a Boolean network N, the lower-bound and upper-bound delay on the combinational delay at the output of node $y \in N$ when $fi_cnt(y) > K$ are given by (Figure 4.1):

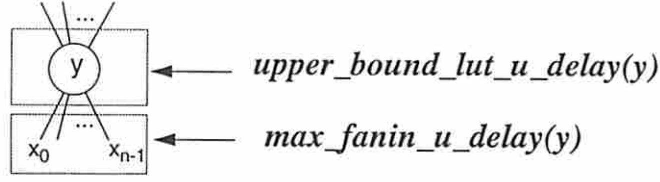
$$lower_bound_lut_u_delay(y) = \lceil \log_K \mid fi_cnt(y) \mid \rceil \times d_{unit} \tag{4.4}$$

$$upper_bound_lut_u_delay(y) = max(1, fi_cnt(y) - K + 1) \times d_{unit} \tag{4.5}$$

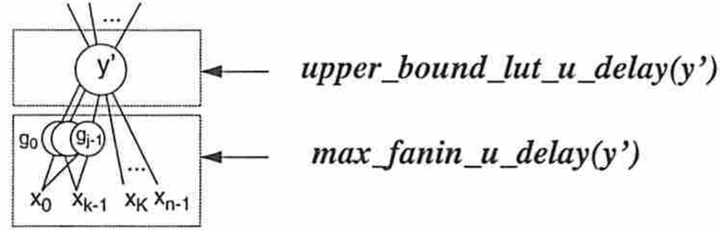
where K is the number of inputs of LUT and d_{unit} is a constant. Note that the lower bound and upper bound delays are equal to d_{unit} when $fi_cnt(y) \leq K$.

Proof: A lower bound on the delay at the output of y is obtained by assuming maximum reduction in variable support of y for each decomposition (that is, $K-1$ variables are eliminated after each decomposition). This leads to a balanced K-ary tree decomposition of y as shown in Figure 4.1.A. It is easy to see that Equation 4.4 hold.

An upper-bound on the delay of node y is obtained by assuming minimum reduction in variable support of y per decomposition (that is, 1 variable reduction) as shown in Figure 4.1.B. Note that at least one variable must be eliminated per decomposition, otherwise we cannot call the transformation a decomposition (cf.



(A) Before decomposition



(B) After decomposition

Figure 4.2: Delay estimation

Definition 2.1.1). This leads to a chain-like K-ary tree decomposition as shown in Figure 4.1.B. It is easy to see that Equation 4.5 hold. □

Definition 4.1.2 Given a Boolean network N and a variable y, if y is an internal node or a primary output, then the $u_delay(y)$ is estimated as the maximum delay of immediate fanins, plus the estimated LUT delay of y (the upper-bound LUT delay is used as estimated LUT delay). The maximum delay of immediate fanins of y is:

$$max_fanin_u_delay(y) = max\{u_delay(x_i) \mid x_i \in fanin(y)\} \tag{4.6}$$

The delay of y is given by (Figure 4.2.A):

$$u_delay(y) = upper_bound_lut_u_delay(y) + max_fanin_u_delay(y) \tag{4.7}$$

Consider a function $f(x_0, \ldots, x_{n-1})$ that is decomposable under bound set B $= \{x_0, \ldots, x_{K-1}\}$ and can be transformed to $f'(g_0(x_0, \ldots, x_{K-1}), \ldots, g_{j-1}(x_0, \ldots, x_{K-1}), x_K, \ldots, x_{n-1})$, one can see that (Figure 4.2.B):

$$\begin{aligned} u_delay(f, B) &= u_delay(y') \\ &= upper_bound_lut_u_delay(y') + max_fanin_u_delay(y') \end{aligned} \tag{4.8}$$

where the upper-bound delay is:

$$upper_bound_lut_u_delay(y') = max(1, fi_cnt(y') - K + 1) \times d_{unit} \tag{4.9}$$

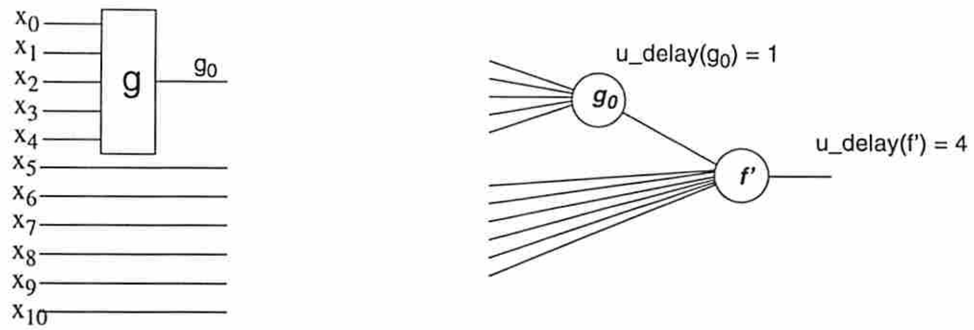
where $fi_cnt(y') = fi_cnt(y) - K + j$. The maximum delay of immediate fanins of y' is:

$$\begin{aligned} max_fanin_u_delay(y') = \; max(& \max_{i \in 0,\ldots,j-1} \{u_delay(g_i)\}, \\ & \max_{i \in K,\ldots,n-1} \{u_delay(x_i)\}) \end{aligned} \tag{4.10}$$

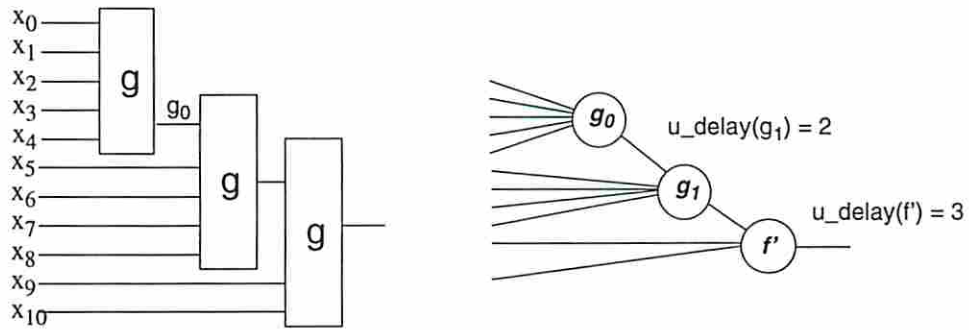
Example 4.1.1 Given a partially decomposed network (Figure 4.3.A), some of the possible bound sets for next decomposition are:

$B_0 = \{x_5, x_6, x_7, x_8, x_9\}$ with column_set $\mathcal{S}_{5,2}$ and $u_delay(f, B_0) = 2$
$B_1 = \{g_0, x_5, x_6, x_7, x_8\}$ with column_set $\mathcal{S}_{5,1}$ and $u_delay(f, B_1) = 3$
$B_2 = \{x_6, x_7, x_8, x_9, x_{10}\}$ with column_set $\mathcal{S}_{5,4}$ and $u_delay(f, B_2) = 2$
$B_3 = \{x_5, x_7, x_8, x_9, x_{10}\}$ with column_set $\mathcal{S}_{5,3}$ and $u_delay(f, B_3) = 2$
$B_4 = \{g_0, x_5, x_6, x_7, x_9\}$ with column_set $\mathcal{S}_{5,2}$ and $u_delay(f, B_4) = 3$

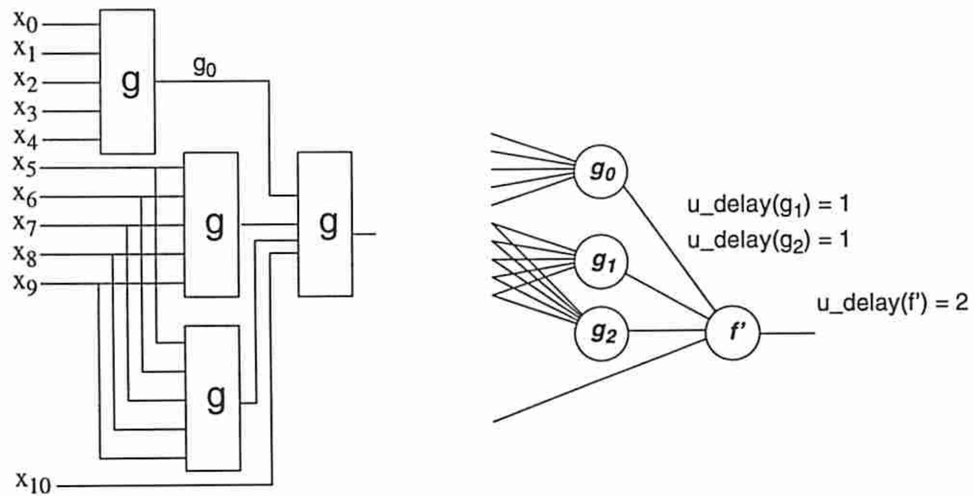
B_1 results in minimum area but a network delay of three (Figure 4.3.B) whilst B_0 leads to minimum delay and then maximum variable support reduction (Figure 4.3.C). □



(A) Partial decomposed network



(B) Area minimized decomposition



(C) Delay minimized decomposition

Figure 4.3: Delay minimization decomposition

Definition 4.1.3 Given a multiple-output function $F = \langle f_0, \ldots, f_{n-1} \rangle$ and bound set B, $\mid B \mid = K$, the average support reduction is defined as:

$$supp_red(F, B) = \frac{1}{n} \sum_{i=0}^{n-1} supp_red(f_i, B). \quad (4.11)$$

and the maximum u_delay of function F decomposed under bound set B is defined as:

$$delay_cost(F, B) = max\{u_delay(f_i, B) \mid f_i \in F\}. \quad (4.12)$$

In the case of multiple-output Boolean functions, we use the following heuristic algorithm to find the best bound set for next decomposition.

Algorithm *minimum_delay_bound_set_selection*:

Given a collection of bound sets $\mathcal{B}$ for function $F = \langle f_0, \ldots, f_{n-1} \rangle$:

1. Create $K - 1$ empty lists as follows:

   For i = 1 to $K - 1$ do

   {

   List i (denoted by L_i) for $K - i - 1 < supp_red(F, B) \leq K - i$

   }

2. For each B in $\mathcal{B}$, put B in the appropriate list.

3. Sort each list in the increasing order of $delay_cost(F, B)$; If there is a tie, then sort it again in the decreasing order of $supp_red(F, B)$.

4. Return the first bound set from the non-empty list in the following order L_1, L_2, ..., L_{k-1}.

This minimum delay decomposition scheme has been incorporated into our system as FGSyn_d. Our results show an average 8% reduction in the network delay over the FlowMap-r [20] results.

4.1.2 Minimum Delay Decomposition Using Unit Fanout Delay Model

Delay minimization has been an important optimization problem in FPGA technology mapping. Most of delay minimization mapping algorithms mis-pga(delay) [47], FlowMap [20], FlowSyn [18] and including FGSyn_d [49] use depth of the mapping results as a measure of delay. This is based on the unit delay model However, the delay calculated using the unit delay model is very inaccurate. In LUT-based FPGA designs, although the delay of each LUT is constant, the interconnect delay of each net may vary considerably. Experiments in [59] have shown that the interconnect delay in an FPGA is closely related to the number of fanouts of the net.

In this section, delay optimum decomposition under a *unit fanout delay model* is considered. Note that this model is accurate for small to medium number of fanouts. For large fanout counts, *unit fanout delay* becomes a sub-linear function of number of fanouts. This is not considered in here, it can be easily handled by our method.

Definition 4.1.4 Given a Boolean network N and a node $y \in N$, if y is a primary input, then the unit fanout delay of y is given by:

$$uf_delay(y) = fo_cnt(y) \times d_{unit_fanout} \tag{4.13}$$

where d_{unit_fanout} which represents the extrinsic delay parameter is know for the given FPGA device and $fo_cnt(y)$ is the number of fanouts of y.

If y is an internal node or a primary output with support size $\leq K$, then the unit fanout delay of y is given by:

$$uf_delay(y) = max\{uf_delay(x_i) \mid x_i \in fanin(y)\} + d_{unit} + fo_cnt(y) \times d_{unit_fanout} \tag{4.14}$$

where d_{unit} which represents the intrinsic delay parameter is also know for the given FPGA device.

For each g-function g_i, the the cost function for delay is:

$$pg_cost(g_i) = uf_delay(g_i) \tag{4.15}$$

Lemma 4.1.2 Given a Boolean network N, the lower-bound and upper-bound delay on the combinational delay at the output of node $y \in N$ when $fi_cnt(y) > K$ are given by:

$$\begin{aligned} lower_bound_lut_uf_delay(y) &= \lceil \log_K \mid fi_cnt(y) \mid \rceil \times (d_{unit} + d_{unit_fanout}) \\ &\quad +(fo_cnt(y) - 1) \times d_{unit_fanout} \end{aligned} \tag{4.16}$$

$$\begin{aligned} upper_bound_lut_uf_delay(y) &= max[1, (fi_cnt(y) - K + 1)] \\ &\quad \times(d_{unit} + (K - 1) \times d_{unit_fanout}) \\ &\quad +(fo_cnt(y) - K + 1) \times d_{unit_fanout} \end{aligned} \tag{4.17}$$

where K is the number of inputs of LUT, $fi_cnt(y)$ is the number of immediate fanins of y and $fo_cnt(y)$ is the number of fanouts of y. Note that the lower bound and upper bound delays are equal to $d_{unit} + fo_cnt(y) \times d_{unit_fanout}$ when $fi_cnt(y) \leq K$.

Proof: Same as Lemma 4.1.1.

Definition 4.1.5 Given a Boolean network N and a variable y, if y is an internal node or a primary output, then the $uf_delay(y)$ is given by (the LUT delay of y is estimated as the upper-bound delay):

$$uf_delay(y) = upper_bound_lut_uf_delay(y) + max\{uf_delay(x_i) \mid x_i \in fanin(y)\} \tag{4.18}$$

Theorem 4.1.1 Consider a function $f(x_0, \ldots, x_{n-1})$ that is decomposable under bound set B = $\{x_0, \ldots, x_{K-1}\}$ and can be transformed to $f'(g_0(x_0, \ldots, x_{K-1}), \ldots, g_{j-1}(x_0, \ldots, x_{K-1}), x_K, \ldots, x_{n-1})$, then the upper-bound delay at the output of f, is given by:

$$\begin{aligned} uf_delay(f, B) \;&=\; upper_bound_lut_uf_delay(f') + max[\max_{i \in 0,\ldots,j-1}\{uf_delay(g_i)\}, \\ &\quad \max_{i \in K,\ldots,n-1}\{uf_delay(x_i)\}] \end{aligned} \tag{4.19}$$

Proof: Follows from delay calculation equation and definition of upper bound delay (see Figure 4.2).

There are two selection decisions which affect the fanout count of a mapped node. The first decision variable is the g-function encoding. The objective function used for g-function encoding in Section 3.2 and Section 3.3 is generally not suitable for delay-minimal decomposition under the unit fanout delay model as shown by the following example.

Example 4.1.2 Given a function f that is decomposable under bound set $B = \{a, b, c, d, e\}$, two of the possible g-function encodings for this decomposition are shown in Figure 4.4. Assume that c is a late arriving signal compared to a, b, d, and e. Both encodings result in three g-functions. Encoding I has a total of 11 fanins for g_1, g_2, and g_3 whilst encoding II has a total of 14 fanins. Encoding I is better than encoding II under unit delay model, but under the unit fanout delay

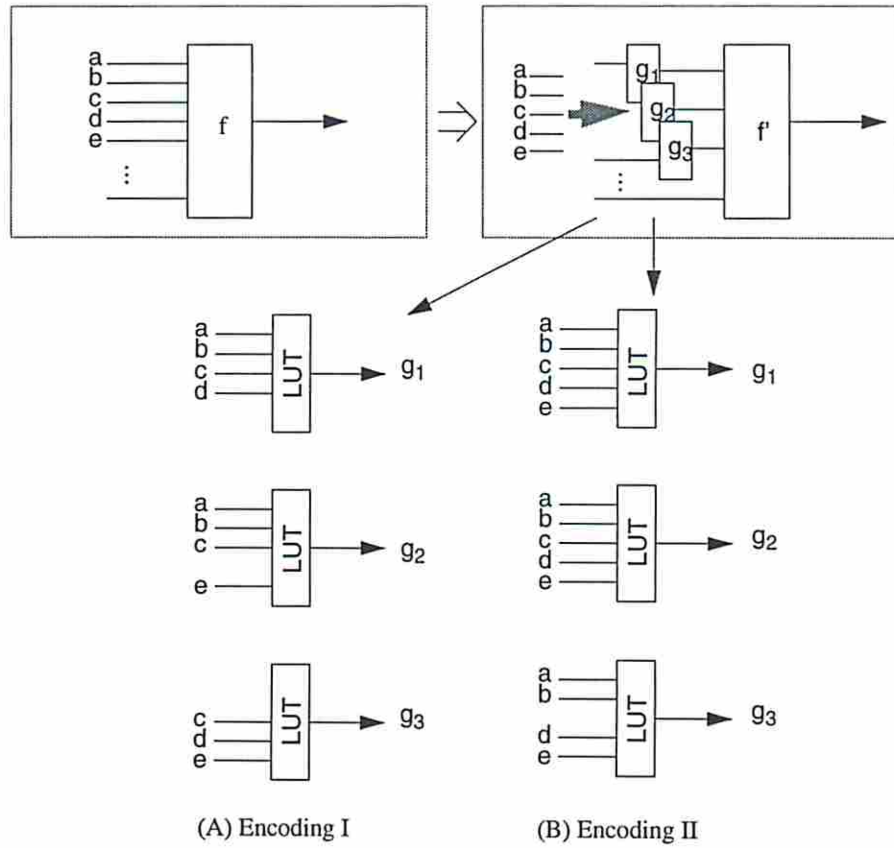


Figure 4.4: Encoding effects node fanout

model, encoding II is a better choice because it leads to fewer fanouts for signal c. □

The second decision variable is the bound set selection. The objective of bound set selection for minimum delay under the unit delay model may not be suitable for minimum delay under the unit fanout delay model as shown by the following example.

Example 4.1.3 Given a multiple output, partially decomposed network $F = \langle f_1, f_2 \rangle$ (Figure 4.5.A), two of the possible bound sets for next decomposition are:
bound set $B_0 = \{x_4, x_5, x_6, x_7, x_8\}$ with $u_delay(f, B_0) = 1$, $uf_delay(f, B_0) = 2.8$
bound set $B_1 = \{x_5, x_6, x_7, x_8, x_9\}$ with $u_delay(f, B_1) = 1$, $uf_delay(f, B_1) = 2.6$

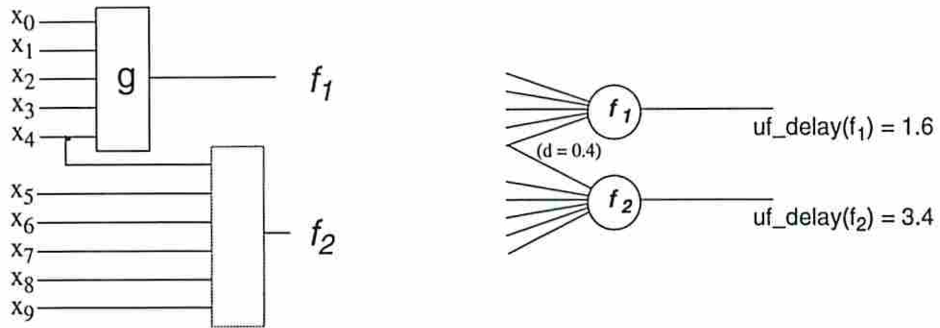
Since both bound sets B_0 and B_1 results in the same delay under unit delay model, assume bound set B_0 is used for minimum delay decomposition under unit delay model (Figure 4.5.B) while use B_1 as next decomposition leads to minimum delay under unit fanout delay model (Figure 4.5.C). □

We modify Definition 4.1.3 as following and use it to select the best bound set.

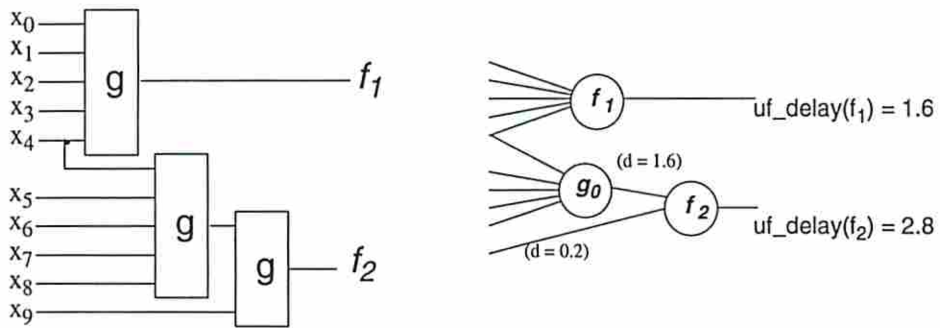
Definition 4.1.6 Given a multiple-output function $F = \langle f_0, \ldots, f_{n-1} \rangle$ and bound set B, the delay_cost is defined as:

$$delay_cost(F, B) = max\{uf_delay(f_i, B) \mid f_i \in F\}. \quad (4.20)$$

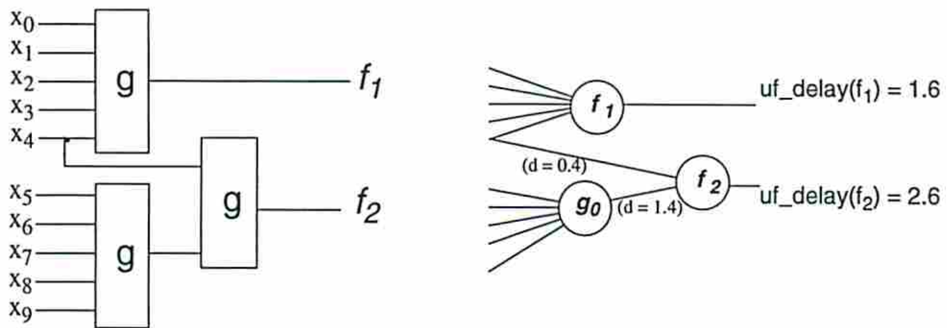
If there is a tie between two bound set B_1 and B_2 in terms of their *delay_cost*, then we pick the bound set B^* that maximizes the *supp_red*(F, B^*). This choice results in an algorithm that first minimizes the network delay and then maximizes the LUT utilizations. This minimum delay decomposition scheme using unit fanout delay model has been incorporated into our system as FGSyn_ufd.



(A) Partial decomposed network



(B) Decomposition under unit delay model



(C) Decomposition under unit fanout delay model

Figure 4.5: Delay minimization under unit fanout delay models

4.2 Decomposition for Minimum Energy Dissipation

Low energy VLSI design can be achieved at various levels. Once these system level, architectural and technological choices are made, it is the switching activity of the logic (weighted by the capacitive loading) that determines the energy consumption of a circuit. FPGA devices are often used for prototyping and designed to implement any functionality, therefore by design, they are not low energy devices. It is however still noteworthy to minimize the energy dissipation in FPGA design once a decision has been made to use these devices (for reasons of field programmability, fast turnaround time and cost). In fact, in some cases, designers are using FPGAs to replace low-end ASICs as FPGA speed and density approach those of low-end ASICs. Under this scenario, low energy consumption becomes an important design consideration. Even in cases where FPGAs are used as fast prototyping devices, excessive energy consumption can pose serious problems in terms of increasing circuit delays, reducing circuit lifetime and increasing packaging/cooling cost.

A decomposition procedure for minimizing the switching activity in general at the outputs of decomposed blocks or in specific at the LUT outputs of FPGA device will be described next.

4.2.1 Energy Consumption Model

The energy consumption can be separated into static and dynamic components. The dominant source is dynamic consumption which refers to the energy involved in the charging/discharging of circuit node capacitances. The dynamic energy consumption in FPGAs can be divided into logic, clock, and I/O pads. This paper focuses an energy consumption in the combinational logic blocks in an FPGA device.

The average energy consumption of a look-up-table (LUT) in an FPGA device is calculated by the following equation:

$$E^{LUT}_{ext.} = \frac{1}{2} \times V^2_{dd} \times C^{LUT}_{ext.} \times sw(\text{LUT}) \tag{4.21}$$

where $C^{LUT}_{ext.}$ is the load capacitance seen at the output of the LUT, V_{dd} is the supply voltage and $sw(\text{LUT})$ is the average number of transitions per clock cycle at the output of the LUT. Here, we have ignored the energy consumption within the LUT itself. Indeed, any time a new input pattern is applied to an LUT, the values on exactly two lines within the LUT change, and so

$$E^{LUT}_{int.} = V^2_{dd} \times C_{int.} \tag{4.22}$$

where $C_{int.}$ is the internal capacitance seen by any line within the LUT. Therefore as long as there is some change at the inputs of the LUT, the same amount of internal energy is dissipated independent of how big or small the change is. Here we assume $C_{int.} << C^{LUT}_{ext.}$ and hence $E^{LUT}_{int.} << E^{LUT}_{ext.}$. Under fixed clock cycle time, power minimize and energy minimize problems coincide. This is precisely the problem we are considering in this section.

We also adopt a non-glitch delay model where signal transitions due to hazards/glitches are ignored. Primary inputs are assumed to be uncorrelated, but spatial correlations among internal lines (due to re-convergent fanouts) are taken into account.

4.2.2 Energy Consumption and Function Decomposition

There are many different g-function encoding schemes that can be used to decompose a function, each resulting in a different set of f' and g_i functions with different energy costs. The following example illustrates this point.

Example 4.2.1 Consider a function $f = x_1x_3x_5x_6 + x_1x_3\bar{x}_5x_7 + x_1\bar{x}_3x_4x_8 + x_1\bar{x}_3\bar{x}_4x_9 + \bar{x}_1x_2x_4x_8 + \bar{x}_1x_2\bar{x}_4x_9 + \bar{x}_1\bar{x}_2x_{10}$. Let bound set $B = \{x_1, x_2, x_3, x_4, x_5\}$.

Encoding I: Function f can be decomposed to functions f' and g_0, g_1, g_2. The resulting functions are (Figure 2.6.B):

$$f' = g_0g_1g_2x_6 + g_0g_1\bar{g}_2x_7 + g_0\bar{g}_1g_2x_8 + g_0\bar{g}_1\bar{g}_2x_9 + \bar{g}_0x_{10}$$

$$g_0 = x_1 + x_2$$

$$g_1 = x_1x_3 + \bar{x}_1\bar{x}_2$$

$$g_2 = x_1x_3x_5 + x_1\bar{x}_3x_4 + \bar{x}_1x_2x_4 + \bar{x}_1\bar{x}_2$$

Encoding II: Function f can also be decomposed into different f' and g_0, g_1, g_2 (Figure 4.6 whose boolean equations are given below:

$$f' = g_0x_6 + \bar{g}_0g_1g_2x_7 + \bar{g}_0g_1\bar{g}_2x_8 + \bar{g}_0\bar{g}_1g_2x_9 + \bar{g}_0\bar{g}_1\bar{g}_2x_{10}$$

$$g_0 = x_1x_3x_5$$

$$g_1 = x_1x_3\bar{x}_5 + x_1\bar{x}_3x_4 + \bar{x}_1x_2x_4$$

$$g_2 = x_1x_3\bar{x}_5 + x_1\bar{x}_3\bar{x}_4 + \bar{x}_1x_2\bar{x}_4.$$

Without loss of generality, let us assume that circuit inputs are uncorrelated and each has a signal and transition probability of 0.5 (corresponding to pseudo random input statistics).

The switching activities of g_0, g_1, g_2 are 0.375, 0.500 and 0.469 for the g-functions in the encoding I, but are 0.219, 0.469, and 0.469 for the g-functions in the encoding II. Thus the total switching activity for the 3 g-functions in the second encoding scheme is 14% less than that in the first encoding scheme.

Alternatively, assume the following signal probabilities for the primary inputs:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
0.3	0.3	0.1	0.2	0.1	0.5	0.5	0.5	0.5	0.5

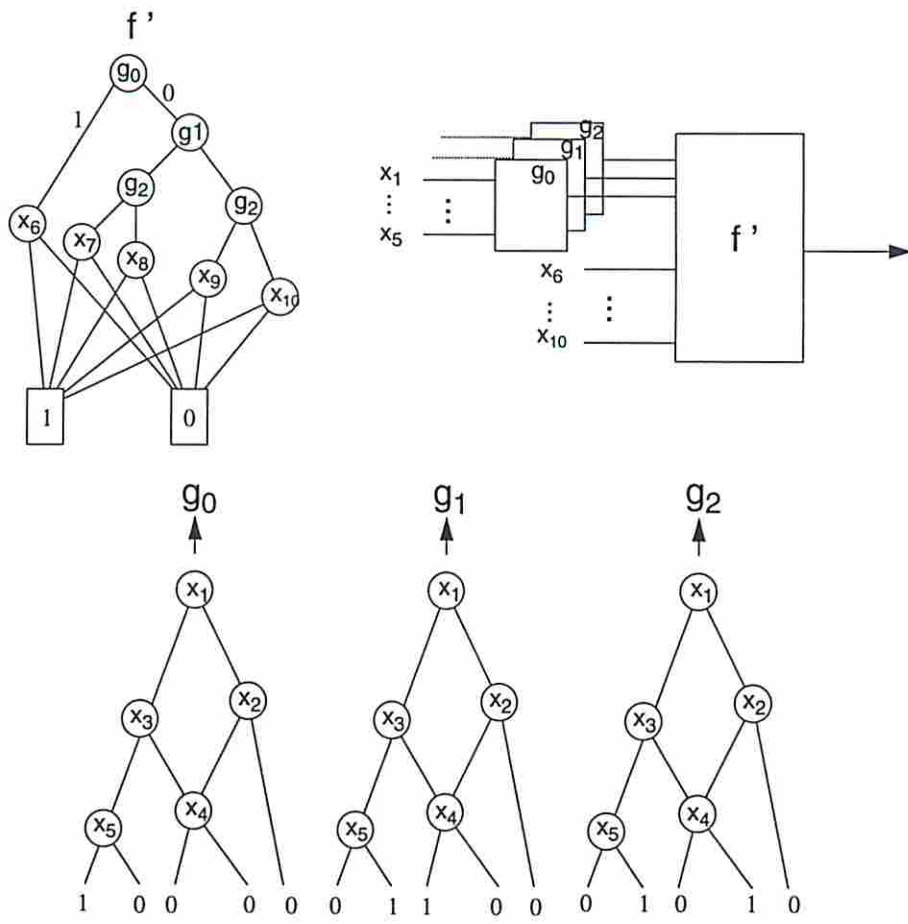


Figure 4.6: OBDD function decompositions with different encoding

The switching activities of g_0, g_1, g_2 are 0.4998, 0.4992 and 0.4842 for the g-functions in the first encoding, but are 0.0060, 0.2157, and 0.4842 for the g-functions in the second encoding. Thus the total switching activity for the 3 g-functions in the second encoding scheme is 52% less than that in the first encoding scheme.

4.2.3 Minimum Energy Decomposition

The input loading of each LUT is assumed to be C_0, and therefore, $C_{ext.}^{LUT} = fo_cnt(\text{LUT}) \times C_0$ where $fo_cnt(\text{LUT})$ is the number of fanouts of the LUT. Assume $V_{dd}^2 \times C_o = 1$, then the energy consumption of each permissible g-function pg contributes to the energy consumption by:

$$\Delta E(pg_i) = sw(pg_i) \times fo_cnt(pg_i) + \sum_{x_i \in support(pg_i)} sw(x_i). \tag{4.23}$$

The number of fanouts of a pg_i is > 1 only when the g-function has been shared among the outputs. At the same time, the larger this number, the smaller the number of g-functions required to produce a valid encoding. Therefore, the energy contribution of each pg_i is divided by its number of fanouts to yield the cost of a pg_i as:

$$pg_cost(pg_i) = \frac{\Delta E(pg_i)}{fo_cnt(pg_i)}. \tag{4.24}$$

Example 4.2.2 Using the function in Example 4.2.1, function f can be decomposed to f' and g_0, g_1, g_2 with respect to the bound set $\{x_1, x_2, x_3, x_4, x_6\}$:

$$f' = g_0 g_1 g_2 x_5 + g_0 g_1 \bar{x}_5 x_7 + g_0 \bar{g}_1 g_2 x_8 + g_0 \bar{g}_1 \bar{g}_2 x_9 + \bar{g}_0 x_{10}$$

$$g_0 = x_1 + \bar{x}_1 x_2$$

$$g_1 = x_1 x_3 + \bar{x}_1 \bar{x}_2$$

$$g_2 = x_1 x_3 x_6 + x_1 \bar{x}_3 x_6 x_4 + \bar{x}_1 x_2 x_4 x_6 + x_1 \bar{x}_3 x_4 \bar{x}_6 + \bar{x}_1 x_2 x_4 \bar{x}_6 + \bar{x}_1 \bar{x}_2.$$

In Example 4.2.1, according to equation 4.24:

$pg_cost(g_0) = 1.3398,$

$pg_cost(g_1) = 1.5192,$

$pg_cost(g_2) = 2.0042$

In Example 4.2.2, according to equation 4.24:

$pg_cost(g_0) = 1.0260,$

$pg_cost(g_1) = 1.3157,$

$pg_cost(g_2) = 2.0042$

□

The *pg_cost* reflects both the energy contribution of the pg_i and its sharing potential. After generating the *pg*-sets for individual Boolean functions, we must select a set G' of *pg*'s with minimum total *pg_cost* such that this set produces a valid, complete encoding of each function. This problem is formulated as follows:

Definition 4.2.1 *Low energy subfunction encoding problem*:
Given a collection PGS of *pg*-sets $\{pgs_1, pgs_2, \ldots, pgs_N\}$ and $G = \cup_{i=1}^{n} pgs_i$, find a set $G' \subseteq G$ with minimum total *pg_cost* such that for each *pg*-set pgs_n with *bit_size* j, there is a subset of G' with size j which induces a j-bit-partition of pgs_n.

This problem is NP-hard and we resort to the following greedy algorithm for solving the minimum energy subfunction covering problem heuristically.

Algorithm *Low_energy_subfunction_encoding* [48]:
Given a collection of *pg*-sets in which each *pg*-set is annotated with a value *count* initialized to its *bit_size* k (this is for indicating when a function has a complete encoding) and a 0-bit partition *bp* initialized to its *column_set*,

1. Find a permissible g-function g that has the lowest *pg_cost*.

2. For each *pg*-set that contains g, decrease its *count* by 1. If $count = 0$, remove this set.

3. For each *pg*-set that contains g, remove any *pg*-code that can not produce the valid encodings with g for corresponding column_vector $\mathcal{V}$.

4. Repeat steps 1-3 until every *pg*-set is removed. Then, return the set of *bp*'s, which defines the encoding for each *pg*-set.

Theorem 4.2.1 Given a multiple-output function $F = \langle f_0, \ldots, f_{n-1} \rangle$ and bound set B, the *energy_cost* which is calculated as the difference of energy consumption before and after decomposition is given by:

$$\begin{aligned} energy_cost(F, B) &= \sum_{g_i} sw(g_i) fo_cnt(g_i) + \sum_{g_i} \sum_{x_j \in supp(g_i)} sw(x_j) \\ &\quad - \sum_{f_k} \sum_{x_j \in [supp(f_k) \cap B]} sw(x_j). \end{aligned} \quad (4.25)$$

Proof: The energy consumption before decomposition is:

$$energy(F) = \sum_{f_k} \Big(\sum_{x_j \in [supp(f_k) \cap B]} sw(x_j) + sw(f_k) \times fo_cnt(f_k) \Big) \quad (4.26)$$

and the energy consumption after decomposition is:

$$energy(F') = \sum_{g_i} sw(g_i) fo_cnt(g_i) + \sum_{g_i} \sum_{x_j \in supp(g_i)} sw(x_j) + \sum_{f_k} sw(f'_k) \times fo_cnt(f'_k) \quad (4.27)$$

But global function and output connections of f_k and f'_k are the same, thus $sw(f'_k) = sw(f_k)$, $fo_cnt(f'_k) = fo_cnt(f_k)$, thus we get Equation 4.25. □

In Example 4.2.1, $energy_cost(F, B) = 3.3432$; In Example 4.2.2, $energy_cost(F, B) = 2.8259$. It is clear to see that the encoding in Example 4.2.2 has produced less $energy_cost(F, B)$ compared to encoding in Example 4.2.1.

We have developed an algorithm for the low-energy decomposition as follows. Our goal is to minimize the total energy consumption in the circuit. Given a multiple-output function $F = \langle f_0, \ldots, f_{M-1} \rangle$, our algorithm is carried out recursively as follows.

Algorithm *fg_synthesis_le*:

1. *For each bound set B of size k do*
2. *Partition F into N groups $\langle H_0, \ldots, H_{N-1} \rangle$ such that each group is decomposable with respect to bound set B and N is as small as possible;*
3. *For each group H_n, generate all its pg functions and store them in pgs_n;*
4. *Extract common pg_i's according to their assigned cost so as to produce a minimum cost valid encoding of all f_m's;*
5. *Compute and store the supp_red and energy_cost of the final encoding;*
6. *Pick the bound set B^* with maximum supp_red and minimum energy_cost;*
7. *Decompose F with respect to B^* and generate F' for next iteration.*

In step 1, we choose the bound set size as the input size k of a LUT so that each g-function can be directly mapped to a single LUT. Step 2 is carried out as described in Section 3.1.1. Steps 3 and 4 are performed as described in Section 3.2. The cost of a pg_i is given by Equation 4.24. In step 5, the *supp_red* of a decomposition is defined in Definition 4.1.3.

In step 6, we pick the bound set with the maximum variable reduction first. If there is a tie, then we use the energy cost as a tie-breaking rule. This scheme is used as we don't want to reduce the energy consumption at the cost of an increase in the LUT count. Step 7 is carried out in a straight forward fashion as described in Section 2.3.1.

Example 4.2.3 Consider two-output Boolean function $F = \langle f_0, f_1 \rangle$ shown in Figure 4.7. Switching activity on each input signal, x_i is 0.5. Furthermore, assume

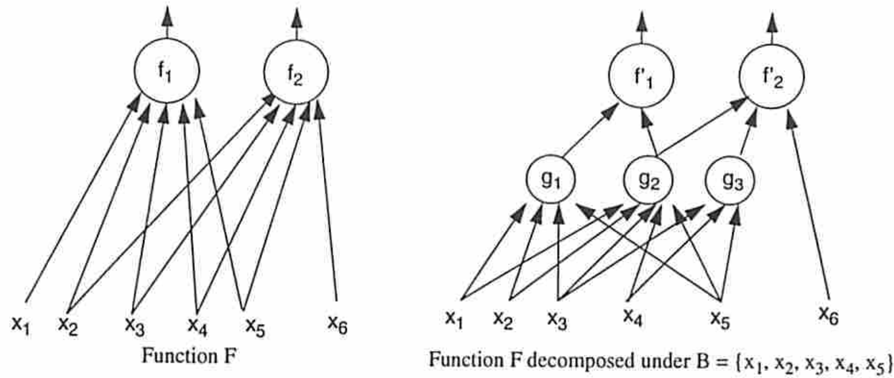


Figure 4.7: Example of multiple-output function decomposition

that $sw(g_1) = 0.32$, $sw(g_2) = 0.18$ and $sw(g_3) = 0.255$. Then,

$$supp_red(F, B) = \tfrac{1}{2}(3 + 2) = 2.5$$

$$energy_cost(F, B) = 0.935 + 6 - 4.5 = 2.435 \quad \text{where } B = \{x_1, x_2, x_3, x_4, x_5\}$$

□

This minimum energy decomposition algorithm have been incorporated into our system as FGSyn_e. The results show 18% reduction over mis-pga(new) in terms of energy consumption.

4.3 Energy-Delay Optimum Decomposition

Logic decomposition changes both the switched capacitance and delay of the overall circuit. The product of C_L and $E(sw)$ which is offen referred to as the *switched capacitance* describes the average capacitance switched during each data period $1/f_{clk}$. Minimizing the switched capacitance may however adversely affect the maximum clock frequency in the circuit, which may or may not be acceptable depending on the design constraints. The key question is therefore what objective

function should be used for low power design. The answer varies from one application domain to next. If extending the battery life is the only concern, then energy (that is, the power-delay product) should be minimized. In this case the battery consumption is minimized even though an operation may take a very long time. On the other hand, if both the battery life and the circuit delay are important, then *action* (that is, the energy-delay product) must be minimized [31]. The energy-delay product allows a designer to find optimizations that provide the largest reduction in energy for the smallest change in performance.

The objective function in many scenarios is to minimize energy and maximize performance. In these cases, one should really minimize the energy-delay product. This is precisely the problem we are considering in this section. We motivate this section with an example.

Example 4.3.1 Given a function $f = x_1\ x_2\ x_3\ x_4\ x_5\ x_6$, we decompose function f using only 2 inputs LUT. Figure 4.8.A shows the delay minimum decomposition result and Figure 4.8.B shows the energy minimum decomposition. The delay minimum decomposition gives minimum delay but produces higher switched capacitance while the energy minimum decomposition gives minimum switched capacitance but produces worse circuit delay. The energy-delay products for delay minimum and energy minimum decompositions are $3 \times 5.243 = 15.729$ and $5 \times 4.771 = 23.855$, respectively. Thus the delay minimum decomposition is a better solution from energy-delay product perspective. The energy-delay product minimum decomposition produce the same result as delay minimum decomposition in this case. □

The objective function of minimum energy-delay decomposition was thus modified as follows.

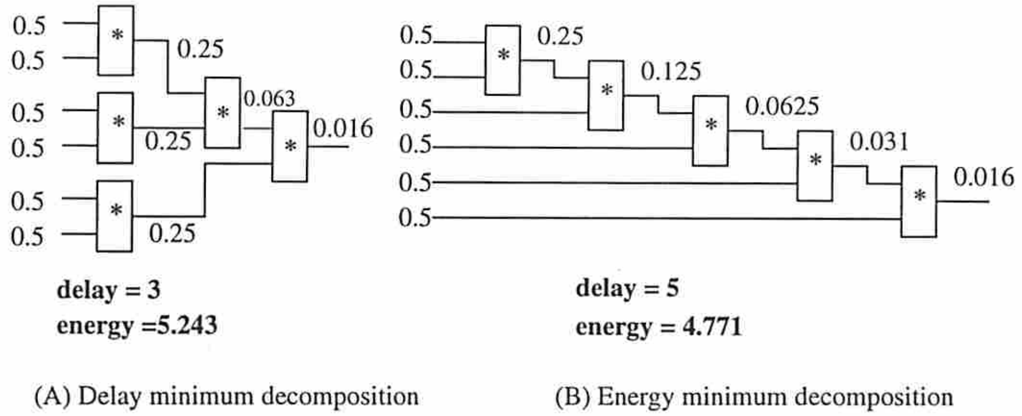


Figure 4.8: Delay minimum vs. energy minimum decompositions

Definition 4.3.1 Given a multiple-output function $F = \langle f_0, \ldots, f_{n-1} \rangle$ and bound set B, the *pg_cost* is calculated as:

$$pg_cost(g_i) = \frac{\Delta E(g_i)}{fo_cnt(g_i)} \times uf_delay(g_i) \quad (4.28)$$

The *energy_delay_cost* is calculated as:

$$energy_delay_cost(F, B) = energy_cost(F, B) \times delay_cost(F, B) \quad (4.29)$$

where $energy_cost(F, B)$ is from Equation 4.25 and $delay_cost(F, B)$ is from Equation 4.20.

This minimum energy-delay product decomposition algorithm have been incorporated into our system as FGSyn_ed. The results show 10.1% reduction over FGSyn_e and 28.8% reduction over mis-pga(new) in terms of energy-delay product.

4.4 Decomposition into Special Classes of Functions

Functions that possess certain properties, such as being symmetric or unate, tend to have smaller size OBDD [15] representation. A decomposition scheme that decomposes a function f into symmetric or unate function f' will be thus useful.

4.4.1 Decomposition and Symmetric Functions

Definition 4.4.1 [27] A function $f(x_1, x_2, \ldots, x_n)$ is symmetric in variables x_i, x_j denoted by $x_i \sim x_j$ if $f(x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n) = f(x_1, \ldots, x_j, \ldots, x_i, \ldots, x_n)$.

Lemma 4.4.1 [27] A function $f(x_1, x_2, \ldots, x_n)$ is symmetric in variables x_i, x_j if and only if $f_{\bar{x_i}, x_j} = f_{x_i, \bar{x_j}}$.

Definition 4.4.2 Given a function $f(x_1, x_2, \ldots, x_m)$, if f can be decomposed into $f'(g_1, g_2, \ldots, g_n, x_{n+1}, \ldots, x_m)$ such that f' is symmetric in variables g_i, g_j for some $i, j \in \{1, 2, \ldots, n\}$, then we say that f admits a *partially symmetric decomposition.* If f' is symmetric in variables g_i, g_j for all $i, j \in \{1, 2, \ldots, n\}$, then we say that f admits a *totally symmetric decomposition.*

Definition 4.4.3 Given column_vector $\mathcal{V} = [v_0, \ldots, v_n]$ and a one bit encoding $pg = [b_0, \ldots, b_n]$ of $\mathcal{V}_{i,j}$, $bit_cover(\mathcal{V}, pg)$ is defined as $\{v_k \mid b_i = 1, 0 \leq i \leq n\}$. Similarly, $bit_cover(\mathcal{V}, \overline{pg}) = \{v_k \mid b_i = 0, 0 \leq i \leq n\}$.

Example 4.4.1 Given column_vector $\mathcal{V}_{4,3} = [\ 3\ 3\ 4\ 4\ 0\ 0\ 4\ 0\ 3\ 3\ 2\ 2\ 3\ 2\ 1\ 0\ ]$ and $pg = [\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ ]$, bit_cover($\mathcal{V}_{4,3}$, pg) $= \{0, 3\}$, while bit_cover($\mathcal{V}_{4,3}$, $\overline{pg}$) $= \{1, 2, 4\}$. □

Definition 4.4.4 bit_cover($\mathcal{V}$, $\langle pg_1, pg_2 \rangle$) = bit_cover($\mathcal{V}$, pg_1) $\cap$ bit_cover($\mathcal{V}$, pg_2), where $\langle pg_1, pg_2 \rangle$ is (pg_1 && pg_2).

Lemma 4.4.2 Given column_vector $\mathcal{V}_{i,j}$ and two one-bit encodings, pg_1 and pg_2, if bit_cover($\mathcal{V}_{i,j}$, $\langle pg_1, \overline{pg_2}\rangle$) = ($\mathcal{V}_{i,j}$, $\langle \overline{pg_1}, pg_2\rangle$), then the decomposed function f' is symmetric in pg_1 and pg_2. Note that such pg_1 and pg_2 can only be found by multi-code assignment because of non-tree structure of OBDD that caused by pg_1 and pg_2.

Example 4.4.2 Given column_vector $\mathcal{V}^f_{4,3} = [\,3\,3\,4\,4\,0\,0\,4\,0\,3\,3\,2\,2\,3\,2\,1\,0\,]$, if we encode $pg_1 = [\,0\,0\,0\,0\,1\,1\,0\,1\,0\,1\,0\,1\,1\,1\,1\,1\,]$, $pg_2 = [\,1\,1\,0\,0\,1\,1\,0\,1\,1\,0\,1\,0\,0\,0\,1\,1\,]$ and $pg_3 = [\,0\,0\,0\,0\,1\,1\,0\,1\,0\,0\,1\,1\,0\,1\,0\,1\,]$, then bit_cover($\mathcal{V}^f_{4,3}$, $\langle pg_1, \overline{pg_2}\rangle$) = $\{2, 3\}$ = bit_cover($\mathcal{V}^f_{4,3}$, $\langle \overline{pg_1}, pg_2\rangle$). This leads to a partially symmetric decomposition of f into f' where $g_1 \sim g_2$. □

Definition 4.4.5 Given a Pascal triangle (Figure 4.9.c), the kth row of triangle (denoted by Ps_k) is referred as the *Pascal signature* for k symmetric variables.

The symmetric property of any given function f can be easily verified using its OBDD representation as follows. For a given bound set $B = \{g_1, g_2, g_3, g_4\}$, if all 4 variables in B are symmetric, then the multiplicity of the distinct columns of column_vector of f with respect to bound set B will match the corresponding Pascal signature Ps_4. If only a subset of the variables (e.g. g_1, g_2, g_3) are symmetric, then only the column_vectors with respect to these variables, in this case, ($\{g_1\}$, $\{g_1, g_2\}$, $\{g_1, g_2, g_3\}$) will have the corresponding Pascal signatures (Ps_1, Ps_2, Ps_3) (see Figure 4.9).

Based on this observation, we can use the following operators to perform partially or totally symmetric decomposition.

Definition 4.4.6 The *column_groups* $\mathcal{G}^f$ is a set of column groupings of $\{s_0, s_1, \ldots, s_n\}$, such that each set s_i is a collection of $v_i \in \mathcal{V}^f$ with the same index.

Example 4.4.3 Given column_vector $\mathcal{V}^f_{4,3} = [v_{15}, v_{14}, \ldots, v_0] = [\,3\,3\,4\,4\,0\,0\,4\,0\,3\,3\,2\,2\,3\,2\,1\,0\,]$, the corresponding *column_groups* $\mathcal{G}^f$ is $\{\{v_9, v_{12}, v_{13}\}, \{v_3, v_6, v_7, v_{14}, v_{15}\}, \{v_2, v_4, v_5\}, \{v_1\}, \{v_0, v_8, v_{10}, v_{11}\}\}$. □

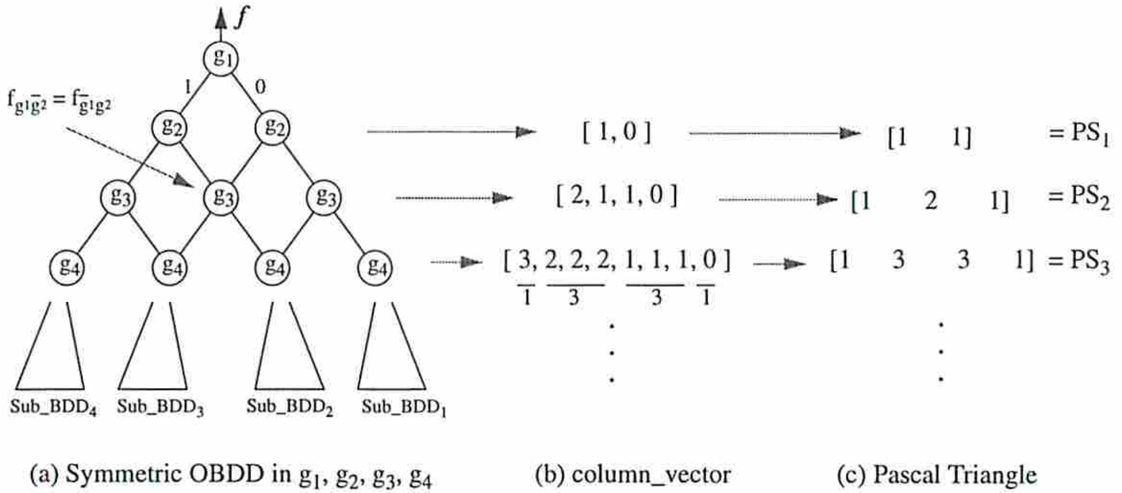


Figure 4.9: Symmetric property in OBDD

Definition 4.4.7 Given column_groups $\mathcal{G}^f$ and Pascal signatures Ps_k, operator *match_psp* (match th Pascal signature property) is defined as follow:
$match_psp(\mathcal{G}, Ps_k) = \{(s_i, p_j) \mid$ For each set $s_i \in \mathcal{G}$ and each $p_j \in Ps_k$, there is a one-to-one correspondence from s_i to p_j such that $\mid s_i \mid \geq p_j \}$.
If this one-to-one correspondence cannot be found, then $match_psp(\mathcal{G}, Ps_k) = \phi$.

Example 4.4.4 From example 4.4.3, $match_psp(\mathcal{G}^f, Ps_3) = \{ (\{v_1, v_0, v_8, v_{10}, v_{11}\}, 1), (\{v_2, v_4, v_5\}, 3), (\{v_9, v_{12}, v_{13}\}, 3), (\{v_3, v_6, v_7, v_{14}, v_{15}\}, 1) \}$; $match_psp(\mathcal{G}^f, Ps_4) = \phi$. □

Lemma 4.4.3 Given a column_vector $\mathcal{V}^f$, the maximum number of symmetric variables (g-functions) that can be produced as a result of decomposition is given by

$$max\{i \mid match_psp(\mathcal{G}^f, Ps_i) \neq \phi\}$$

where Ps_i is the Pascal signature for bound set size i.
If the maximum number of symmetric variables that can be produced for $\mathcal{V}^f$ is equal to $\mid \mathcal{G}^f \mid -1$, then a totally symmetric decomposition can be performed, otherwise, only partially symmetric decomposition can be performed.

Definition 4.4.8 Given a $match_psp(\mathcal{G}^f, Ps_k)$ result (denoted as $\mathcal{P}^f_k$), the operator $divide(\mathcal{P}^f_k)$ is defined as: Sort each $(s_i, b_j) \in \mathcal{P}^f_k$ in increasing order of j; for each $(s_i, b_j) \in \mathcal{P}^f_k$, divide s_i into b_j groups ($\{s_{i1}, s_{i2}, \ldots, s_{ib_j}\}$) and put them into a list G'.

We use following procedure to carry out the symmetric decomposition.

Algorithm *Symmetric_decomposition_procedure*:

Given a column_vector $\mathcal{V}^f$ and column_groups $\mathcal{G}^f$

1. Find $K = max\{i \mid match_psp(\mathcal{G}^f, Ps_i) \neq \phi\}$.

2. Do $\mathcal{P}^f_k = match_psp(\mathcal{G}^f, Ps_K)$ and $G' = divide(\mathcal{P}^f_k)$.

3. Annotate each $s_j \in G'$ with $(b_K, \ldots, b_1)$ where $j = 2^{K-1}b_k + \ldots + 2^0 b_1$ (For example, $s_5 \rightarrow s_{(1,0,1)}$ for K = 3).

4. For j = 1 to K do /* Assign K symmetric g-functions */

   {

   for (each $s_{(b_K,\ldots,b_1)} \in G'$) { assign b_j to each v_i in $s_{(b_K,\ldots,b_1)}$ }

   g_j code = $\mathcal{V}^f = [v_n, \ldots, v_0]$ with each v_i assigned "0" or "1" from G'.

   }

5. Assign rest of non-symmetric g-functions using shared subfunction encodings scheme in Section 3.2.

The symmetric decomposition technique may not be able reduce the total number of g-functions needed in the final solution since totally symmetric decomposition always use more g-functions than the non-symmetric decomposition (except when $\mid \mathcal{S}^f \mid = 3$ decomposition). Partially symmetric decomposition also has the tendency to increase the number of g-functions. However, the f' function is simpler to implement for symmetric decompositions. Thus, this symmetric decomposition technique can be used for special purpose decomposition.

4.4.2 Decomposition and Unate Functions

Definition 4.4.9 [27] A logic function f is positive unate (negative unate) in a variable x_j from 0 to 1 causes all the outputs of f that change, to increase also from 0 to 1 (from 1 to 0). A function that is either positive unate or negative unate in x_j is said to be unate in x_j. A function is unate if it is unate in all its variables.

Lemma 4.4.4 [27] Given a function f, if $f_x \supset f_{\bar{x}}$ (or $f_x + \overline{f_{\bar{x}}} = 1$), then function f is positive unate in variable x; if $f_x \subset f_{\bar{x}}$ (or $\overline{f_x} + f_{\bar{x}} = 1$), then function f is negative unate in variable x.

Definition 4.4.10 Given a function $f(x_1, x_2, \ldots, x_m)$, if we decompose f into $f'(g_1, g_2, \ldots, g_n, x_{n+1}, \ldots, x_m)$ such that f' is unate in variables g_i for some $i \in \{1, 2, \ldots, n\}$, then we say that f admits a *partially unate decomposition.* If f' is unate in variables g_i for all $i \in \{1, 2, \ldots, n\}$, then we say that f admits a *totally unate decomposition.*

Each distinct column in column_vector of function f with respect to bound set B corresponds to a sub_OBDD. To check the unateness of each *pg*-code, we simply take the "or"of every sub_OBDD that corresponds to "1" ("0") in the *pg*-code to produce f_g ($f_{\bar{g}}$). The complexity of checking unateness of each *pg*-code is linear to the number of OBDD nodes. We use the following algorithm to check unateness of each *pg* encoding in column_set.

Algorithm *Unate_decomposition_encoding*:
Given a column_vector $\mathcal{V}^f$ and sub-OBDD set SB-set that corresponds to each distinct column in $\mathcal{V}^f$.

1. Produce the uni-code assignment *pg*-set of $\mathcal{V}^f$.

2. For each pg_k in pg-set do

   {

   Divide sub-OBDD set into B_0 and B_1 according to each bit ("0" or "1") in pg_k.

   Let $B_{\bar{x}}$ equal to the result of "or" operation on each sub-OBDD in B_0;

   Let B_x equal to the result of "or" operation on each sub-OBDD in B_1.

   If $(B_x + \bar{B_{\bar{x}}}) = 1$, then annotated this pg code with positive unate ("1").

   If $(\bar{B_x} + B_{\bar{x}}) = 1$, then annotated this pg code with negative unate ("-1").

   Otherwise annotated this pg code with binate ("0").

   }

Finding a minimum set of pg-sets with maximum number of unate variables (pg) can be formulated as the *minimum subfunction covering problem* as in Section 3.2. We use the *minimum subfunction covering* algorithm with a lexicographic cost function where the first objective minimizes the number of g-functions and the second objective maximizes the unateness of f' with respect to the g-functions.

4.5 Summary

In this chapter, we described the decompositions for minimum delay using two different delay models (unit delay model and unit fanout delay model), minimum energy, minimum energy-delay product, and special classes functions (symmetric and unate functions). In next chapter, we will apply these techniques to look-up table based field programmable gate arrays synthesis.

Chapter 5

Application to LUT-Based FPGA Synthesis

5.1 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are ASICs that can be configured by the user. They combine the logic integration benefits of custom VLSI with the design, production, and time-to-market advantages of standard logic ICs. FPGAs can be viewed as an evolution of PALs where size is increased by an order of magnitude, or a refinement of mask-programmed gate arrays, where the reprogramming time and cost are drastically reduced. Figure 5.1 shows the structure of a FPGA which consist a matrix of logic block with routing resource around the blocks. Each logic block consists sequential (latch) and combinational logic parts. In this thesis, we only use the combinational logic for random logic mapping.

There are mainly two types of FPGA architecture: one is Look-Up-Table (LUT) based FPGAs, and the other is multiplexers based FPGAs. The LUT-based FPGA is a popular architecture used by several FPGA manufactures, including Xilinx and AT&T. In a LUT-based FPGA device, the basic programmable logic block is a K-input lookup table (K-LUT) also called a configurable logic block CLB which can implement any Boolean function of up to K variables.

A K-input lookup table is a digital memory that k inputs are used to address a 2^k by 1-bit digital memory that stores the truth table of the Boolean function.

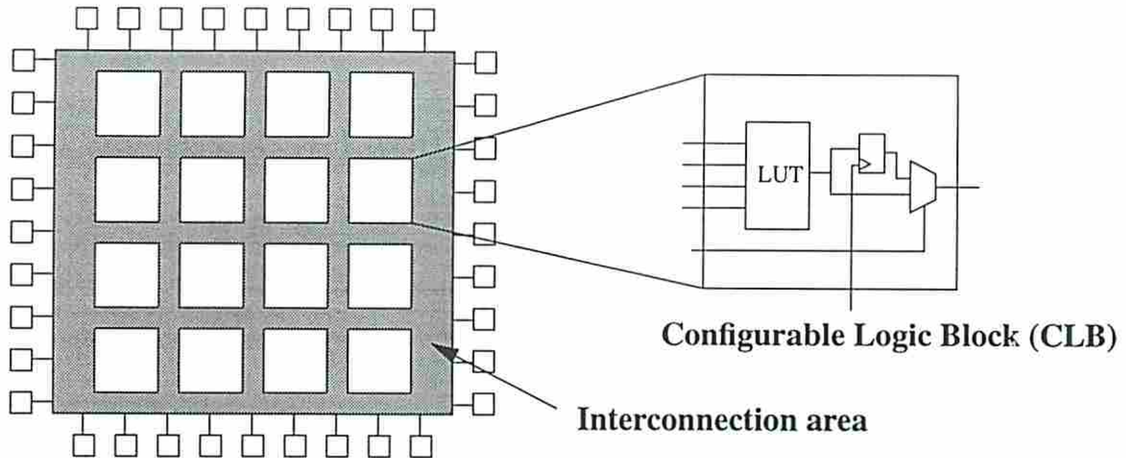


Figure 5.1: The FPGA structure

For example, Figure 5.2 illustrates the function $g_0 = x_0x_1\bar{x_2} + x_0\bar{x_1}x_2 + \bar{x_0}x_1x_2$ Karnaugh map implemented in the 3-input LUT. The Karnaugh map is stored in an 8 by 1-bit memory, and an 8 to 1 multiplexer, controlled by the variables x_0, x_1 and x_2, selects the output value g_0.

In a LUT-based FPGA device (e.g., XC3000 device from Xilinx Inc [35]), the basic programmable logic block is a K-input lookup table (K-LUT) which can implement any Boolean function of up to K variables.

The technology mapping problem for LUT-based FPGA designs is to transform a Boolean network into a functionally equivalent network of K-LUTs. There are several different approaches for solving the FPGA mapping problem [24, 23, 25, 37, 64, 46, 58, 1, 55]. All of these works are based on the algebraic decomposition method. Our works [42, 40, 41, 48, 49, 50] are based on the function decomposition which is Boolean method.

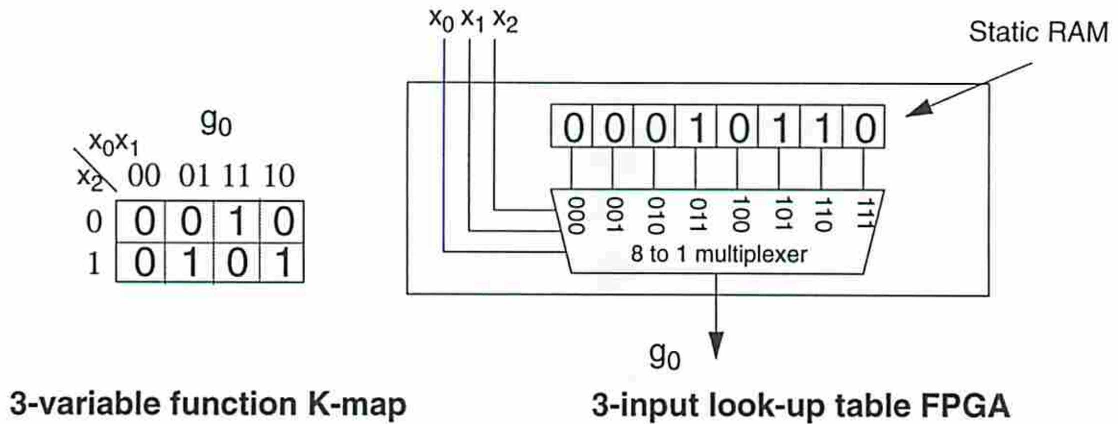


Figure 5.2: The LUT structure

5.2 Overview of FGSyn

Based on the theory presented in the previous chapters, we have developed an OBDD-based decomposition program, called FGSyn, for LUT-based FPGA synthesis. FGSyn is based on the multiple-output function decomposition theory.

Given a multiple-output function $F = \langle f_0, \ldots, f_{M-1} \rangle$, the FGSyn algorithm is carried out recursively as follows.

Algorithm $fg_synthesis$:

1. For each bound set B of size K do

 /* K is the maximum number of inputs of each LUT */

2. Partition F into N groups $\langle H_0, \ldots, H_{N-1} \rangle$ such that each group is decomposable with respect to bound set B and N is as small as possible;

3. For each group H_i,

 if its compatible class size $\mid \mathcal{S} \mid \leq 8$, then generate all its *pg* functions;
 else use the bipartition_dependency_graph algorithm to generate the common *pg* functions;

4. Extract common pg_i's according to their assigned cost function (pg_cost)

so as to produce a minimum cost valid encoding of all f_m's;

5. Compute and store the specified cost function (*decomp_cost*) of the final encoding;

6. Pick the bound set B^* with the best *decomp_cost*;

7. Decompose F with respect to B^* and generate F' for next iteration.

In step 1, we choose the bound set size as the input size k of a LUT so that each g-function can be directly mapped to a single LUT. Step 2 is carried out as described in Section 3.1. Steps 3 and 4 are performed as described in Sections 3.2 and 3.4. Note that *pg_cost* in step 4 and *decomp_cost* in steps 5, 6 are defined according to the objective function that is being minimized during synthesis. For minimization of the number of LUTs, we used:

$$pg_cost(g_i) = supp(g_i) \tag{5.1}$$

$$decomp_cost(F, B) = supp_red(F, B) \tag{5.2}$$

Step 7 is carried out in a straight forward fashion as described in Section 2.3.1.

5.2.1 Node Clustering

An important issue is how to come up with the input $F = \langle f_0, \ldots, f_{m-1} \rangle$ to the *fg_synthesis* algorithm. One option is to use the whole Boolean network as F. This is sometimes infeasible as size of the OBDD representing F becomes too large. In addition, the output partitioning performed in step 3 of *fg_synthesis* algorithm may lead to a large number of output groups, and thus, a lot of logic may be duplicated among various output groups. A better technique is to run the rugged script [57] on the network, and then do some node clustering where each cluster is a complex multi-input, multi-output Boolean function. Each such node then becomes the input to the *fg_synthesis* algorithm.

Definition 5.2.1 *Node clustering problem*: Given a collection of nodes $\mathcal{N} = \{N_0, N_1, \ldots, N_m\}$, find a partition that divide $\mathcal{N}$ into $\langle \mathcal{N}_0, \mathcal{N}_1, \ldots, \mathcal{N}_k \rangle$ such that the total supports size of nodes in each $\mathcal{N}_i$ is minimized and the common supports size among nodes is maximized.

Node clustering is currently performed by the following greedy algorithm:

Algorithm *node_clustering*:

Given a set of nodes $\mathcal{N} = \{N_0, N_1, \ldots, N_m\}$. Set k = 0.

1. Start with a seed node N_s and insert it into the first cluster C^k.

2. Find a new node N_i that maximizes the $\mid supp(N_i) \cap supp(N_s) \mid$ and minimizes $\mid supp(N_i) \cup supp(N_s) \mid$.

3. If $\mid supp(N_i) \cup supp(C^k) \mid \leq \alpha$ and $1+ \mid C^k \mid \leq \beta$ for some specified parameters α, β, then merge N_i with C^0; otherwise pick a new seed node and increase k by 1 and initialize C^k;

4. Repeat the above until all nodes are assigned to some node cluster.

In step 1, a random seed node is chosen from $\mathcal{N}$ into cluster C^k. In steps 2 and 3, nodes that maximizes the common supports of nodes in C^k and minimizes the total supports of C^k are chosen. Some constraints, α, β are set to limit the total support of C^k and the size of C^k. Instead of a single seed node, one may start with multiple seed nodes and grow them into clusters simultaneously.

Example 5.2.1 Given a Boolean network

$f_1 = abc + c\bar{d} + \bar{a}ex;$

$f_2 = a\bar{g}h + kn + \bar{m}z;$

$f_3 = bc + \bar{b}dg + c\bar{x};$

$f_4 = \bar{b}g + hmn + \bar{h}\bar{y};$

$x = apq + rs;$

$y = dq + rs + tu;$

$z = \bar{b}\bar{g}h + mn;$

If f_1 is selected in step 1, then in step 2

$$\begin{array}{llll}
\mid supp(f_2) \cap supp(f_1) \mid & = 1 & \mid supp(f_2) \cup supp(f_1) \mid & = 11 \\
\mid supp(f_3) \cap supp(f_1) \mid & = 4 & \mid supp(f_3) \cup supp(f_1) \mid & = 7 \\
\mid supp(f_4) \cap supp(f_1) \mid & = 1 & \mid supp(f_4) \cup supp(f_1) \mid & = 11 \\
\mid supp(x) \cap supp(f_1) \mid & = 1 & \mid supp(x) \cup supp(f_1) \mid & = 10 \\
\mid supp(y) \cap supp(f_1) \mid & = 1 & \mid supp(y) \cup supp(f_1) \mid & = 11 \\
\mid supp(z) \cap supp(f_1) \mid & = 1 & \mid supp(z) \cup supp(f_1) \mid & = 11
\end{array}$$

node f_3 will be picked into cluster C^0. In step 3 for given $\alpha = 10, \beta = 5$, no more node can be put in cluster C^0 after $C^0 = \{f_1, f_3\}$.

The node_clustering algorithm leads to the following clusters:

$C^0 = \{f_1, f_3\}$,

$C^1 = \{f_2, f_4, z\}$ and

$C^2 = \{x, y\}$. □

5.2.2 Generating the Subject Network

We use standard scripts, script.rugged, script.algebraic [10, 57], to generate the initial network which then becomes the input to FGSyn. These scripts extract all possible common sub-expressions from the given Boolean equations to reduce the number of literals since the literal count is directly related to the gate area after technology mapping using standard cell libraries. Literal count however does not correlate to the LUT count in FPGA mapping. For example, extracting common sub-expressions from the Boolean equations that have lees than or equal k variable supports will only increase the number of FPGA logic blocks (LUTs) used in the mapping result. Thus partial collapsing on some of the nodes become necessary. The following example illustrates this point.

Example 5.2.2 Given a Boolean function $F = \langle f_1, f_2 \rangle$

$$f_1 = x_1x_2x_3x_4 + x_1x_2x_4x_5 + x_1\bar{x_2}x_3\bar{x_4} + x_1\bar{x_2}\bar{x_4}x_5 + \bar{x_1}x_2\bar{x_3}\bar{x_4} + \bar{x_1}x_2\bar{x_4}\bar{x_5} + \bar{x_1}\bar{x_2}\bar{x_3}x_4 + \bar{x_1}\bar{x_2}x_4\bar{x_5} + x_2x_3x_4x_5 + x_2\bar{x_3}\bar{x_4}\bar{x_5} + \bar{x_2}x_3\bar{x_4}x_5 + \bar{x_2}\bar{x_3}x_4\bar{x_5}$$

$$f_2 = x_1x_3x_5 + x_1\bar{x_3}\bar{x_5} + \bar{x_1}x_3\bar{x_5} + \bar{x_1}\bar{x_3}x_5$$

f_1 and f_2 have variable supports $\{x_1, x_2, x_3, x_4, x_5\}$ and $\{x_1, x_3, x_5\}$, respectively. If we implement function F using 5-inputs LUT, only two LUT's are needed. After running script.rugged, function F becomes:

$$f_1 = x_1\bar{f_2}n_2 + \bar{x_1}f_2\bar{n_2} + x_3x_5n_2 + \bar{x_3}\bar{x_5}\bar{n_2}$$

$$f_2 = x_5n_3 + \bar{x_5}\bar{n_3}$$

$$n_2 = x_2x_4 + \bar{x_2}\bar{x_4}$$

$$n_3 = x_1x_3 + \bar{x_1}\bar{x_3}$$

The script extracts two internal nodes (n_2, n_3) from function F. If we implement this new set of functions using 5-inputs LUTs, we will need four LUT's instead. □

We use following rules to partially collapse nodes into their fanout nodes. These rules will adjust the script result to better fit the LUT-based FPGA mapping.

Rule 1 *K-bounded_support_collapse*:
Given an internal node v and its immediate fanout nodes $o_0, \ldots, o_n$, if all $o_0, \ldots, o_n$ nodes have less than or equal to K supporting variables and if after collapsing node v into its fanout nodes, $o_0, \ldots, o_n$ will result in less than or equal to K variable support, then collapse v into its fanout nodes.

Rule 2 *Support_reduction_collapse*:
Given an internal node v and its immediate fanout nodes $o_0, \ldots, o_n$, if collapsing node v into its fanout node o_i, reduces the support of node o_i, then collapse v into

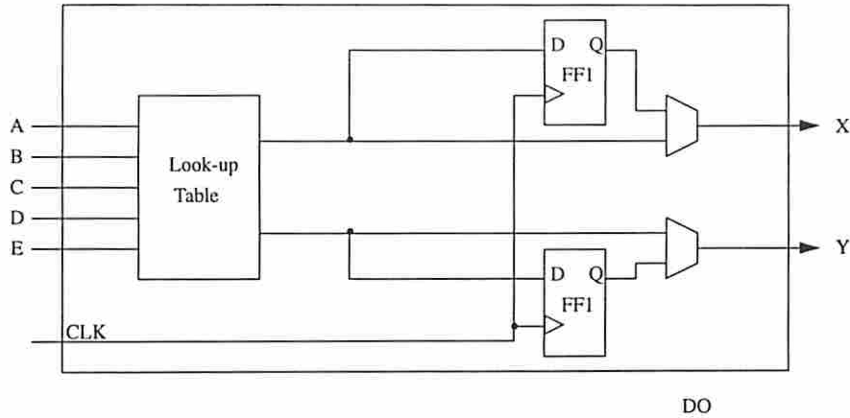


Figure 5.3: The Xilinx XC3000 CLB

o_i.

Rule 3 OBDD_*size_reduction_collapse*:
Given an internal node v and its immediate fanout nodes o_0, ..., o_n, if collapsing node v into its fanout node o_i, preserves the support size of o_i but reduces the number of OBDD nodes needed to represent o_i, then collapse v into o_i.

Example 5.2.3 From the example 5.2.2, after applying these three rules with K = 5, function F is transformed back to its original representation. □

5.3 Mapping for XC3000 Device

A typical LUT-based FPGA device is Xilinx XC3000. The configurable logic block (CLB) in the XC3000 series FPGA is shown in Figure 5.3 which consist of the two flip-flops and a look-up table. Since we are targeting only combinational logic in this thesis, we can ignore the flip-flops. This CLB has a maximum of 5 inputs and can implement any 5-input function or two 4-input functions f_1 and f_2 with $\mid supp(f_1) \cup supp(f_2) \mid \leq 5$.

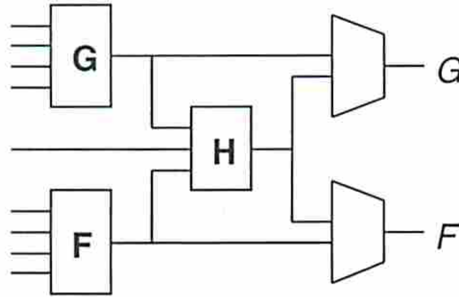


Figure 5.4: The Xilinx XC4000 CLB.

5.3.1 Look-up Table Merge

Because of this special property of XC3000 that allow 4-input functions to merge under the total support constraint. The 4-input LUTs (functions) are packed into CLBs by a greedy heuristic that merges pairs of LUTs with combined support of K or less inputs by trying out all such pairs. Ties are broken by merging the pair that maximizes input utilization in each LUT.

5.4 Mapping for XC4000 Device

The new generation of the Xilinx FPGA devices, i.e., XC4000, contains a number of architectural and technological improvements that allows densities up to 20K equivalent gates and support clock rates up to 60MHz. Among the important architectural improvements that contribute to the XC4000 family's increased logic density and performance is a more powerful and flexible configurable logic block (CLB). A simplified block diagram of the combinational logic part of this CLB is shown in Figure 5.4. One key issue in synthesis for XC4000 device is to obtain maximal utilization of the CLBs provided on the device.

Nine different patterns of XC4000 device are recognized for mapping to different types of functions (Figure 5.5). Among these patterns, the first two patterns are the most interesting and cost effective. Note that the part enclosed by dotted

box in the second pattern of Figure 5.6 can be interpreted as an instance of non-disjunctive decomposition. To have such an interpretation, consider the following decompositions:

$$\begin{aligned} f(X_f, X_g, x_h, \ldots) &= f_1(F(X_f), G(X_g), x_h, \ldots) \\ &= f_1(x_f, x_g, x_h, \ldots) \\ &= f_2(H(x_f, x_g, x_h), x_f, \ldots) \end{aligned}$$

In the first decomposition (f to f_1), variables X_f and X_g are bound variables with respect to the functions F and G. In the second line of the above equation, we replace $F(X_f)$ and $G(X_g)$ by variables x_f and x_g. Then, in the second decomposition (f_1 to f_2), variable x_f is both a bound variable and free variable.

5.4.1 Direct Decomposition

We show how to use the techniques introduced in this paper to map Boolean functions to the first two patterns of Figure 5.5. Given an OBDD v representing $f(x_0, \ldots, x_{n-1})$, two sets of variables X_f and X_g each containing at most 4 variables, and a variable x_h, the algorithm $match_pattern(\mathrm{v}, X_f, X_g, x_h)$ returns 1 if $\{X_f, X_g, x_h\}$ can be mapped to the pattern in Figure 5.5 (a); returns 2 if it can be mapped to the pattern in Figure 5.5 (b); otherwise it returns 0.

Figure 5.5: XC4000 patterns.

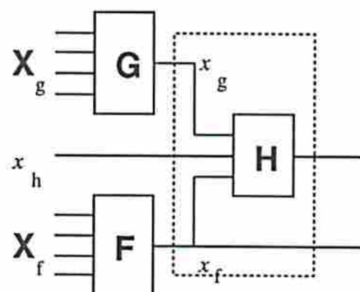


Figure 5.6: the non-disjunctive decomposition view of pattern (b).

```
match_pattern(v, X_f, X_g, x_h)
{
1     for (i = 0; i <| X_f |; i++)
2         v = rotate(v, X_f_i);
3     cset = cut_set(v, | X_f | -1);
4     if (| cset |> 2) return 0;
5     v = decomp_f(cset, {x_f});
6     for (i = 0; i <| X_g |; i++)
7         v = rotate(v, X_g_i);
8     cset = cut_set(v, | X_g | -1);
9     if (| cset |> 2) return 0;
10    v = decomp_f(cset, {x_g});
11    v = rotate(v, x_h);
12    cset = cut_set(v, 2);
13    if (| cset |> 4) return 0;
14    if (| cset |≤ 2) return 1;
15    if (| cut_set_nd(v, 2, 2, 0) |≤ 2 &&  | cut_set_nd(v, 2, 2, 1) |≤ 2)
16        return 2;
17    else return 0;
}
```

The algorithm *match_pattern* [41] is straightforward. The first stage is to move the variables X_f to the top and compute the *cut_set* with respect to X_f. If the *cut_set* size is greater than 2, X_f cannot be mapped to a single LUT. The second stage is the same as the first stage except the variables in X_g is used. The third stage is simple because x_h is a single variable. It computes the *cut_set* with respect to variables x_h, x_g and x_f. If the *cut_set* size is greater than 4, then it requires more than two outputs. Neither patterns can be mapped. On the other hand, if the *cut_set* size is less than or equal to 2, the first pattern is detected. Finally, if

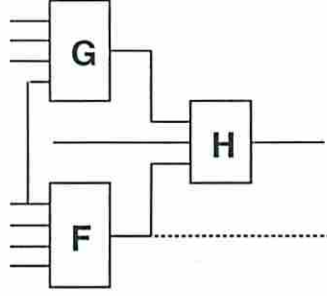


Figure 5.7: Non-disjunctive mapping for the XC4000 device.

the conditions imposed in line 15 for non-disjunctive decomposition are satisfied, then the second pattern is detected; Otherwise no pattern has been found.

To cover the case of non-disjunctive decomposition such as the pattern shown in Figure 5.7, lines 3-5 in *pattern* are modified as follows:

$3'$ $cset[0] = cut_set_nd(\text{v}, \mid X_f \mid -1, \mid X_f \mid -1, 0);$

$cset[1] = cut_set_nd(\text{v}, \mid X_f \mid -1, \mid X_f \mid -1, 1);$

$4'$ if $(\mid cset[0] \mid > 2 \; \mid\mid \; \mid cset[1] \mid > 2)$ return 0;

$5'$ $\text{v} = nd_decomp_f(cset, \{x_f\});$

Note that line 3' assumes that there is only one variable (the last one in X_f) that is both in the bound set and the free set. Modifications to allow more than one shared variable pose no difficulty.

Instead of the above direct decomposition, one could perform a device-specific decomposition that recursively finds matches to the XC4000 pattern as shown next.

5.4.2 Two-Layer Decomposition

Definition 5.4.1 Given a function $f(X, Y, Z)$ and a decomposition $f'(g(X), Y, Z)$, if f' is simply decomposable under bound set $\{g(X), Y\}$ and free set $\{g(X), Z\}$ (e.g., $f' = f''(h(g(X), Y), g(X), Z)$), then f is *type I two-layer decomposable.*

The graphical representation of type I two-layer decomposition is shown in Figure 5.8 and is also the pattern (d) in Figure 5.5.

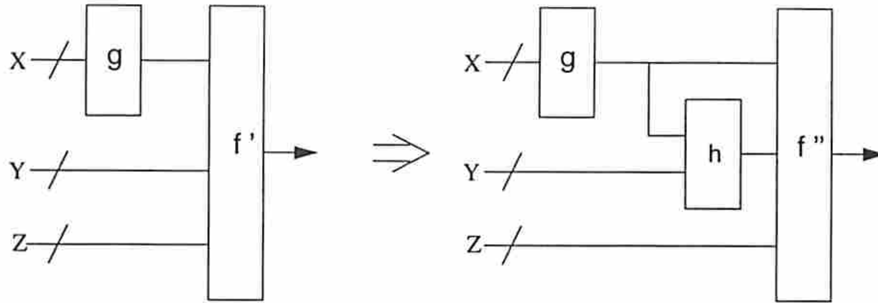


Figure 5.8: Graphical representation of type I two-layer decomposition.

Lemma 5.4.1 Given a function f represented by OBDD $\mathbf{v_f}$ and two bound sets X and Y, the necessary and sufficient conditions for f to be type I two-layer decomposable with respect to X and Y are:

1. $|\ cut_set(\mathbf{v_f}, X)\ | \leq 2$ (let $cut_set(\mathbf{v_f}, X) = \{\mathbf{u}, \mathbf{v}\}$), and
2. $|\ cut_set(\mathbf{u}, Y)\ | \leq 2$ and $|\ cut_set(\mathbf{v}, Y)\ | \leq 2$.

Proof: Sufficiency: If both conditions are satisfied, then we have the case shown in Figure 5.9 (a). With the encoding adopted in Figure 5.9 (a), we have the two g-functions g and h shown in Figure 5.9 (b). After reduction, the reduced OBDD g is shown in Figure 5.9 (c). Thus, the supporting variables of g and h are X and $X \cup Y$, respectively.
Necessity: If $|\ cut_set(\mathbf{u}, Y)\ | > 2$ or $|\ cut_set(\mathbf{v}, Y)\ | > 2$, then there is no encoding such that either g or h can be reduced to the one shown in Figure 5.9 (c). □

Definition 5.4.2 Given a function $f(X, Y, Z)$ and a decomposition $f'(g_0(X), \ldots, g_{i-1}(X), h_0(Y), \ldots, h_{j-1}(Y), Z)$, if f' is simply decomposable under bound set $\{g_k(X), h_l(Y)\}$, $0 \leq k < i$, $0 \leq l < j$, and free set consisting of Z and all g and h functions except for g_k and h_l, then f is *type II two-layer decomposable.*

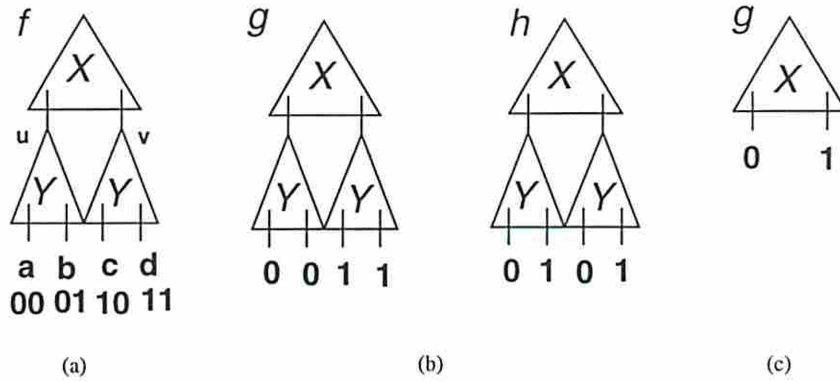


Figure 5.9: The proof of type I two-layer decomposition.

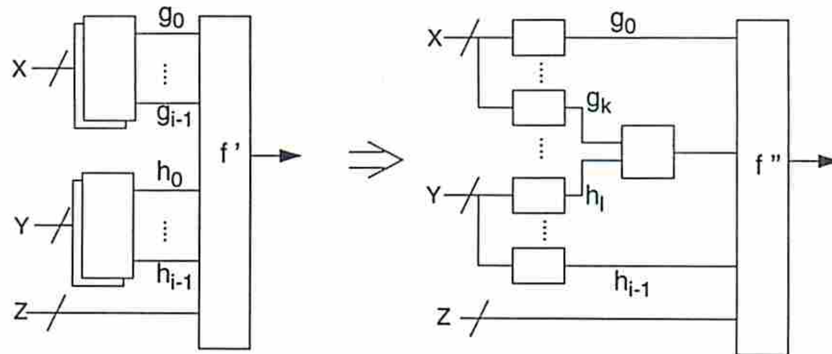


Figure 5.10: Graphical representation of type II two-layer decomposition.

The graphical representation of type II two-layer decomposition is shown in Figure 5.10 and is also the pattern (c) in Figure 5.5.

One way to see if a function $\mathbf{v_f}$ is type II two-layer decomposable under the bound set $X \cup Y$ is the following: If $\lceil \log_2 \mid cut_set(\mathbf{v_f}, X) \mid \rceil + \lceil \log_2 \mid cut_set(\mathbf{v_f}, Y) \mid \rceil > \lceil \log_2 \mid cut_set(\mathbf{v_f}, X \cup Y) \mid \rceil$ and there exists an encoding of $cut_set(\mathbf{v_f}, X \cup Y)$ such that each g-function g satisfies the following conditions:

1. g is a function of variables X, or

2. g is a function of variables Y, or

3. g is simply decomposable under bound sets X and Y.

Therefore, to detect type II two-layer decomposition under bound sets X and Y, one must first compute:

$$\begin{array}{lcl} C_X & = & cut_set(\mathrm{v_f}, X) = \{\mathrm{u_0}, \ldots, \mathrm{u_{k-1}}\}, \\ V_i & = & cut_vector(\mathrm{u_i}, Y) = \langle \mathrm{v_{i,0}}, \ldots, \mathrm{v_{i,2^{|Y|}-1}} \rangle, \mathrm{u_i} \in C_X, \\ C_Y & = & column_encode(V_0, \ldots, V_{k-1}), \text{ and} \\ C_{XY} & = & cut_set(\mathrm{v_f}, X \cup Y) = \{\mathrm{w_0}, \ldots, \mathrm{w_{l-1}}\}. \end{array}$$

If $\lceil \log_2 \mid C_X \mid \rceil + bit_size(C_Y) > \lceil \log_2 \mid C_{XY} \mid \rceil$, then type II two-layer decomposition is possible. We then compute a set of compatible bit-partitions of C_{XY} such that for each bit-partition $S^1 = \{S_0, S_1\}$ and associated permissible g-function g satisfies one of the following conditions:

Let $W_i = \langle b_{i,0}, \ldots, b_{i,2^{|Y|}-1} \rangle$ where

$$b_{i,j} = 0 \quad \text{if } \mathrm{v_{i,j}} \in S_0 \text{ and}$$
$$b_{i,j} = 1 \quad \text{if } \mathrm{v_{i,j}} \in S_1,$$

1. $W_i = \langle 0, \ldots, 0 \rangle$ or $W_i = \langle 1, \ldots, 1 \rangle$, $0 \leq i < k$. Then, g is a function of variables X (Figure 5.11 (a)).

2. $W_i = W_j$, $0 \leq i, j < k$. Then, g is a function of variables Y (Figure 5.11 (b)).

3. There exist only two distinct W's, say W_i and W_j, and $bit_size(coding(W_i, W_j)) = 1$. Then, g is a function of variables $X \cup Y$ and is simple decomposable under bound sets X and Y (Figure 5.11 (c)). The former condition ensures that it is simple decomposable under X and the latter condition ensures that it is simple decomposable under Y.

If f is type I two-layer decomposable under $g_k(B_1)$, $h_l(B_2)$, and $V_f = \phi$, then $g_k(B_1)$ and $h_l(B_2)$ can be mapped to pattern (c). If f is type I two-layer decomposable under $g_k(B_1)$, $h_l(B_2)$, and $V_f = \{x\}$, then $g_k(B_1)$, $h_l(B_2)$ and x can be mapped to pattern (a). If f is type II two-layer decomposable under $g_k(B_1)$ and

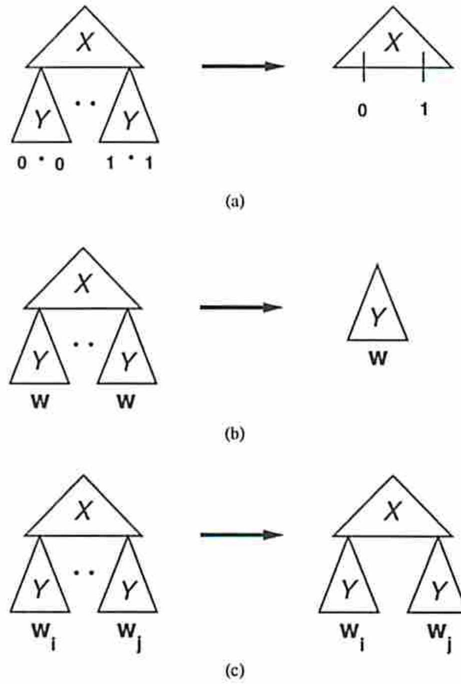


Figure 5.11: Conditions for type II two-layer decomposition.

V_f, and $| \, supp(g_k(B_1)) \, | + | \, V_f \, | \leq 4$, then $g_k(B_1)$ and V_f can be mapped to an LUT.

This process may be viewed in two ways. First, it is viewed as the size of bound set being $| \, B_1 \, | + | \, V_f \, |$ and the number of g-functions remaining i. Second, it is viewed as a case of non-disjunctive decomposition, that is, supporting variables in $g_k(B_1)$ are both in the bound set B_1 and in the free set.

A straightforward way to detect type II two-layer decomposability under two bound sets X and Y is the following: We start with the decomposition of f given by $f = f'(g_0(X), \ldots, g_{i-1}(X), h_0(Y), \ldots h_{j-1}(Y), \ldots) = f'(y_0, \ldots, y_{i-1}, z_0, \ldots, z_{j-1}, \ldots)$. We then test if f' satisfies simple decompositions under bound set $\{y_k, z_l, V_f\}$, for $0 \leq k < i$ and $0 \leq l < j$. The result of this test depends on the binary encodings for y_k and z_l. Using a wrong encoding causes the test to fail when indeed type II two-layer decomposition was possible. Trying all possible encodings is clearly

nonviable. On the other hand, using an arbitrary encoding may cause too many false failures.

Given a function f and a bound set $| B | \leq 4$, we can use $| decomp_g(f, B) |$ LUTs to map the g-functions. However, this may not be the best results we can achieve. For example, let $| decomp_g(f, B) | = 2$, under one encoding we may have two g-functions g_0 and g_1 such that the true support of each function is 4 and 2. In this case, it is possible that we can map g_1 and two other free variables to a single LUT. Furthermore, the new LUT may be combined with the one of g_0 to match the first two patterns. The effect of this process has two different interpretations. First, it is viewed as the size of bound set is 6 and the number of g-functions for this bound set is 2 or 1. Second, it is viewed as a case of non-disjunctive decomposition: supporting variables in g_1 are both in bound set B and in free set.

5.5 Mapping for XC5000 Device

The latest generation of LUT-based FPGA, Xilinx XC5000, is similar to XC4000. Both families use CMOS SRAM technology. Both families use 4-input look-up tables with unshared inputs. There are however some differences between the two families. XC5000 CLBs are roughly equivalent to two XC4000 CLBs. Each XC5000 CLB contains four 4-input function generators and four registers, which are configured as four independent Logic Cells (LCs). The architecture of logic cell is shown in Figure 5.12. It uses a 4-input LUT as the basic cell and allows two basic cells to form one 5-input LUT or four basic cells to form one 6-input LUT (cf. Figure 5.13).

5.5.1 Mapping Using Fixed Input Size LUTs

Technology mapping for XC5000 type of FPGA is more complex and challenging due to the possibility of using different sizes of LUTs. The problem becomes problem

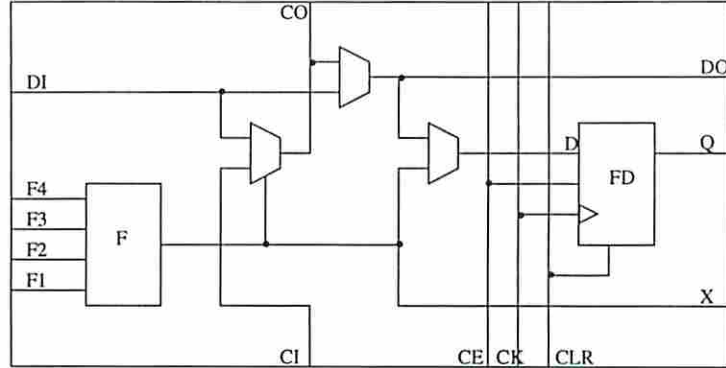


Figure 5.12: The Xilinx XC5000 logic cell

whether one should decompose the circuit into large LUT's or into smaller ones in order to minimize the number of CLBs used.

5.5.2 Mapping Using Variable Input Size LUTs

We modified step 1 of the FGSyn algorithm in Section 5.2 to enumerate all bound sets of size $K = 4$, 5 and 6 and used the following cost function to determine which of the input variable sizes gives the best results.

Given a g-function g_i, the pg_cost of g_i is defined as:

$$pg_cost(g_i) = \begin{cases} 0.25 & \text{if } supp(g_i) \leq 4 \\ 0.50 & \text{if } supp(g_i) = 5 \\ 1.00 & \text{if } supp(g_i) = 6 \end{cases}$$

In this equation, if the g-function supports size is less than or equal to 4, then this function can fit into $\frac{1}{4}$ of CLB (Figure 5.13.A); if the g-function supports size is equal to 5, then this function can fit into $\frac{1}{2}$ of CLB (Figure 5.13.B); if the g-function supports size is equal to 6, then this function can fit into one CLB (Figure 5.13.C);

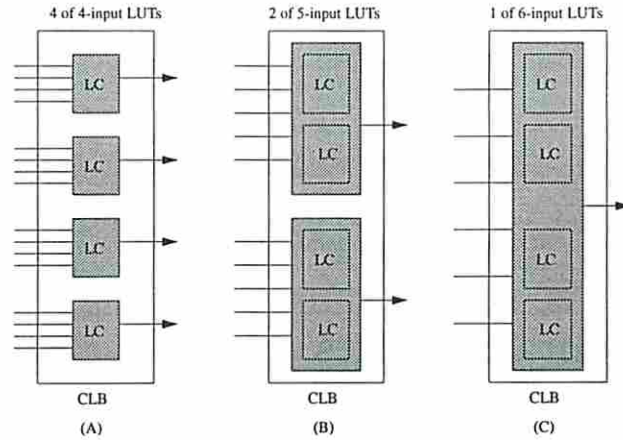


Figure 5.13: The different configurations of XC5000 CLB

$$decomp_cost(F, B) = \frac{supp_red(F, B)}{\sum_{min_cover_g_is} pg_cost(g_i)} \tag{5.3}$$

where the $min_cover_g_i$'s refers to the minimum cover of the *pg*-sets for the given bound set B. This decomp_cost is simply the support reduction per g-function cost.

5.6 Summary

In this chapter, we described the synthesis for different FPGA architectures, XC3000, XC4000, and XC5000 using our decomposition techniques. In next chapter, we will show the experimental results for all the techniques we described.

Chapter 6

Experimental Results

The LUT-based FPGA synthesis algorithm FGSyn has been implemented in C and incorporated into the SIS environment. We used FGSyn to synthesize and map a number of benchmark circuits to the various LUT-based FPGA devices.

6.1 Description of Benchmarks

The benchmark circuits used in this thesis are from the 1991 MCNC logic synthesis benchmark set [65]. Table 6.1 provides some information about these benchmarks. In this table, "PI" is the number of primary inputs; "PO" is the number of primary outputs; "nodes" is the number of internal nodes in the optimized network; "lits(fac)" is the number of literals in factored form in the optimized network.

name	PI	PO	nodes	lits(fac)
5xp1	7	10	10	163
9sym	9	1	1	283
9symml	9	1	1	285
alu2	10	6	6	683
alu4	14	8	69	296
apex2	39	3	75	285
apex6	135	99	210	854
apex7	49	37	45	253
b9	41	21	46	140
bw	5	28	28	296
C499	41	32	90	610
C880	60	26	116	473
C1355	41	32	162	552
C1908	33	25	124	548
C2670	233	140	100	872
C5315	178	123	271	2002
C7552	207	108	348	2430
c8	28	18	29	141
cm162a	14	5	12	48
cm42a	4	10	13	34
comp	32	3	12	107
clip	9	5	5	264
count	35	16	23	151
decod	5	16	18	68
des	256	245	410	3621
duke2	22	29	115	429
e64	65	65	84	274
f51m	8	8	8	169
frg2	143	139	172	934
misex1	8	7	7	49
misex2	25	18	26	103
mux	21	1	6	92
pm1	16	13	17	49
rd73	7	3	3	247
rd84	8	4	4	482
rot	135	107	124	694
sao2	10	4	9	139
vda	17	39	165	615
vg2	25	8	8	92
z4ml	7	4	4	77

Table 6.1: Description of benchmark circuits

name	cube-based (sec.)	OBDD-based (sec.)
5xp1	29.9	2.7
9sym	541.3	3.7
9symml	529.8	3.8
alu2	446.7	224.4
alu4	138.7	16.6
apex2	321.4	35.0
apex7	72.6	11.8
b9	10.3	2.0
bw	19.8	3.3
C880	250.8	38.5
C1908	466.8	108.3
cm162a	7.0	0.7
clip	255.1	14.1
count	48.6	4.2
duke2	303.0	32.5
f51m	33.7	2.7
misex1	11.8	1.2
misex2	26.1	3.0
rd73	116.7	2.5
rd84	612.1	6.2
rot	>1500	469.4
sao2	334.6	19.4
vg2	92.0	8.1
z4ml	27.0	1.5
Total	4695.8	546.2
speed-up	–	8.6

Table 6.2: Comparison between cube-based and OBDD-based decompositions

6.2 OBDD-based Function Decomposition

The OBDD-based decomposition procedure described in this thesis has been implemented and compared with the Roth_Karp decomposition algorithm implemented in SIS [60]. In particular, we used "xl_k_decomp -n 4 -e -d -f 100" which for every node in the Boolean network finds the best bound set of size ≤ 4 that reduces the node's variable support after decomposition, and then decomposes the node and modifies the network to reflect the change.

Results are shown in Table 6.2. Our OBDD-based decomposition approach obtains significant speed-up over Roth_Karp approach by an average factor of 8.6.

In Table 6.3, we present the results (in CLB count of XC3000 FPGA) obtained by using different options of FGSyn [40]: column encoding with output grouping (-c), shared subfunction encoding (-s), and minimum g-function support encoding (-n). The best results are obtained with the -csn option. The LUTs are packed into CLBs by a greedy heuristic that described in Section 5.3.1. In Table 6.4, we compare the run time (CPU seconds) of each options from Table 6.3. The "bx -csn" option is about 3 time slower than "bx".

name	bx	bx-c		bx-cs		bx-csn	
	CLB	CLB	% Red.	CLB	% Red.	CLB	% Red.
5xp1	16	10	37.5	9	43.8	9	43.8
9symml	6	6	0.0	7	-16.7	7	-16.7
alu2	65	59	9.2	55	15.4	55	15.4
alu4	59	59	0.0	56	5.1	56	5.1
apex2	60	60	0.0	59	1.7	60	0.0
apex6	189	182	3.7	182	3.7	181	4.2
apex7	55	47	14.5	44	20.0	43	21.8
b9	28	28	0.0	28	0.0	28	0.0
bw	27	27	0.0	27	0.0	27	0.0
C499	54	54	0.0	54	0.0	54	0.0
C880	93	91	2.2	87	6.5	87	6.5
C1908	75	74	1.3	74	1.3	73	2.7
C2670	136	128	5.9	127	6.6	122	10.3
C5315	364	335	8.0	328	9.9	316	13.2
C7552	348	346	0.6	331	4.9	317	8.9
clip	23	20	13.0	18	21.7	18	21.7
cm162a	9	9	0.0	9	0.0	9	0.0
count	29	29	0.0	24	17.2	23	20.7
duke2	87	86	1.1	85	2.3	85	2.3
e64	44	44	0.0	44	0.0	44	0.0
f51m	12	9	25.0	8	33.3	8	33.3
misex1	9	10	-11.1	10	-11.1	8	11.1
misex2	22	22	0.0	22	0.0	22	0.0
rd73	7	6	14.3	5	28.6	5	28.6
rd84	12	9	25.0	8	33.3	8	33.3
rot	150	161	-7.3	142	5.3	136	9.3
sao2	26	33	-26.9	21	19.2	25	4.0
vg2	27	23	14.8	22	18.5	17	37.0
z4ml	5	4	20.0	4	20.0	4	20.0
Total	2037	1971	–	1890	–	1847	–
Average	–	–	5.2	–	10.0	–	11.6

Table 6.3: Experimental results of FGSyn for XC3000 device

name	bx	bx-c		bx-cs		bx-csn	
	Time	Time	% Inc.	Time	% Inc.	Time	% Inc.
5xp1	3.8	3.8	0.0	12.8	236.8	13.0	242.1
9symml	6.4	6.3	-1.6	24.7	285.9	25.0	290.6
alu2	90.0	96.2	6.9	140.4	56.0	123.3	37.0
alu4	29.2	30.1	3.1	108.5	271.6	108.1	270.2
apex2	42.3	41.8	-1.2	166.0	292.4	167.9	296.9
apex6	170.7	212.4	24.4	508.8	198.1	509.7	198.6
apex7	21.1	27.9	32.2	64.7	206.6	65.5	210.4
b9	1.9	1.9	0.0	2.0	5.3	2.0	5.3
bw	2.2	2.1	-4.5	2.1	-4.5	2.1	-4.5
C499	3.9	3.7	-5.1	3.8	-2.6	3.8	-2.6
C880	47.8	53.1	11.1	148.9	211.5	149.6	213.0
C1908	17.9	17.9	0.0	33.8	88.8	31.2	74.3
C2670	132.0	149.5	13.3	396.3	200.2	394.0	198.5
C5315	341.1	488.4	43.2	971.4	184.8	977.1	186.5
C7552	450.5	523.5	16.2	1209.2	168.4	1263.1	180.4
clip	39.6	46.5	17.4	182.3	360.4	180.7	356.3
cm162a	1.0	0.9	-10.0	1.5	50.0	1.5	50.0
count	4.8	6.9	43.8	15.1	214.6	14.6	204.2
duke2	48.2	49.1	1.9	159.8	231.5	162.7	237.6
e64	2.7	2.6	-3.7	2.7	0.0	2.6	-3.7
f51m	4.8	5.7	18.8	26.9	460.4	26.7	456.2
misex1	1.6	7.1	343.8	20.4	1175.0	21.9	1268.7
misex2	4.0	4.2	5.0	6.5	62.5	6.4	60.0
rd73	2.5	2.3	-8.0	2.3	-8.0	2.4	-4.0
rd84	7.1	6.4	-9.9	14.7	107.0	15.0	111.3
rot	187.7	227.9	21.4	675.6	259.9	689.3	267.2
sao2	40.2	62.2	54.7	207.2	415.4	263.5	555.5
vg2	19.0	19.1	0.5	74.5	292.1	83.4	338.9
z4ml	2.0	1.7	-15.0	4.7	135.0	4.7	135.0
Total	1726.0	2101.2	–	5187.6	–	5310.8	–
Average	–	–	20.6	–	212.3	–	221.7

Table 6.4: Runtime Comparison of FGSyn for XC3000 device

name	FGSyn	FGSyn_lcs	
	CLB	CLB	% Red.
C499	54	51	5.6
C880	87	73	13.8
C1908	73	66	9.6
C2670	104	103	1.0
C5315	298	300	-0.7
C7552	317	317	0.0
alu2	49	41	16.3
alu4	56	55	1.8
apex2	60	61	-1.6
apex6	178	157	11.8
apex7	42	41	2.4
b9	28	25	10.7
clip	15	10	33.0
count	23	23	0.0
des	643	643	0.0
e64	44	43	2.3
frg2	188	183	2.7
misex2	22	22	0.0
sao2	25	18	28.0
vg2	16	16	0.0
Total	2322	2248	–
Average	–	–	7.4

Table 6.5: Large size benchmarks results for XC3000 device.

Our algorithm FGSyn_lcs ("lcs" stands for large common subfunctions which described in Section 3.4), has been implemented in C and incorporated into FGSyn. We ran FGSyn_lcs on a number of benchmarks for XC3000 and XC5000 devices. In Table 6.5, we compared FGSyn_lcs with FGSyn on the large benchmarks. For FGSyn_lcs, we decomposed the circuit into 8-input nodes and then re-decomposed to 5-input LUTs; for FGSyn, we directly decomposed the circuit to 5-input LUTs. FGSyn_lcs does 7.4% better than FGSyn. Obviously, the two-step decomposition approach, which is made possible by FGSyn_lcs, captures more of the logic sharing in the circuit.

name	Chortle-crf	ASYL	mis-pga(new)		FGSyn_lcs				
				Run		Run	% Reduction		
	CLB	CLB	CLB	Time	CLB	Time	Chortle	ASYL	mis-pga
5xp1	20	13	17	16.6	9	13.0	55.0	30.8	47.1
9sym	42	8	7	331.6	7	25.3	83.3	12.5	0.0
9symml	41	8	7	207.6	7	25.0	82.9	12.5	0.0
C499	50	–	66	738.0	51	84.0	-0.2	–	22.7
C880	69	–	78	648.8	73	149.6	-5.8	–	6.4
C1908	–	–	85	264.8	66	31.2	–	–	22.4
C2670	–	–	111	796.9	103	394.0	–	–	7.2
C5315	–	–	306	1285.7	300	977.1	–	–	2.0
C7552	–	–	340	2288.0	317	1263.1	–	–	6.8
alu2	83	60	84	304.1	41	123.3	50.6	31.7	51.2
alu4	138	254	149	2381.8	55	108.1	60.1	78.3	63.1
apex2	93	69	54	491.1	61	167.9	34.4	11.6	-13.0
apex6	161	156	147	144.3	157	509.7	2.5	-0.6	-6.8
apex7	42	44	43	16.0	41	65.5	2.4	6.8	4.7
b9	–	18	27	10.5	27	2.0	–	-50.0	0
bw	–	27	27	3.3	27	2.1	–	0	0
clip	–	33	23	86.6	10	180.7	–	69.7	56.5
count	27	28	28	8.5	23	14.6	14.8	17.9	17.9
des	743	–	750	3186.3	643	4789.9	13.5	–	14.3
duke2	89	82	108	325.0	85	162.7	4.5	-3.7	21.3
e64	54	54	55	12.7	43	2.6	20.4	20.4	21.8
f51m	–	14	11	14.5	8	26.7	–	42.9	27.3
misex1	14	13	9	1.7	8	21.9	42.9	38.5	11.1
misex2	–	24	23	3.2	22	6.4	–	8.3	4.3
rd73	–	8	7	19.6	5	2.4	–	37.5	28.6
rd84	53	14	12	119.5	8	15.0	84.9	42.9	33.3
rot	131	–	139	175.6	136	689.3	-3.8	–	2.2
sao2	–	30	28	58.8	18	263.5	–	40.0	35.7
vg2	18	20	20	5.7	16	83.4	11.1	20.0	20.0
z4ml	3	4	6	6.0	4	4.7	-33.3	0.0	33.3
Total	–	–	2921	13952.8	2371	10204.7	–	–	–
Average	–	–	–	–	–	–	28.6	22.2	18.8

Table 6.6: Experimental results for XC3000 device.

In Table 6.6, we compared FGSyn results with the mis-pga(new) [46]. As seen in Table 6.6, FGSyn_lcs does 28.6% better than Chortle-crf, 22.2% better than ASYL and 18.8% better than mis-pga (new).

6.3 Other Objective Functions Decomposition

In Table 6.7, we compare FGSyn_d results with FlowMap [20] and mis-pga(delay) [47]. FGSyn_d does 7.9% better than FlowMap and 8.9% better than mis-pga(delay). The number of LUTs used in FGSyn_d is 2.3% smaller than that of the mis-pga(delay) but 6.7% larger than that of FlowMap-r. Total1 reflects the total number of LUTs and depths for results reported by mis-pga(delay) and Total2 reflects the same for those reported by FlowMap.

name	FlowMap-r		mis-pga(delay)		FGSyn_d			
	LUT	Delay	LUT	Delay	LUT	Delay	% Reduction	
							FlowMap-r	mis-pga(d)
5xp1	23	3	21	2	13	2	33	0
9sym	61	5	7	3	7	3	40	0
9symml	58	5	7	3	7	3	40	0
C499	134	5	199	8	76	4	20	50
C880	206	8	259	9	173	9	-13	0
C1355	–	–	–	–	128	6	–	–
C5315	–	–	643	10	550	11	–	-10
alu2	149	8	122	6	70	5	38	17
alu4	253	10	155	11	316	9	10	18
apex2	–	–	116	6	71	9	–	-50
apex6	232	4	274	5	227	5	-25	0
apex7	83	4	95	4	110	4	0	0
b9	–	–	47	3	45	3	–	0
bw	–	–	28	1	28	1	–	0
clip	–	–	54	4	15	2	–	50
count	76	3	81	4	40	3	0	25
des	1109	5	1397	11	1588	5	0	55
duke2	188	4	164	6	124	6	-50	0
e64	–	–	212	5	366	4	–	20
f51m	–	–	23	4	12	3	–	25
misex1	15	2	17	2	17	2	0	0
misex2	–	–	37	3	34	3	–	0
rd73	–	–	8	2	6	2	–	0
rd84	47	4	13	3	9	3	25	0
rot	246	6	322	7	277	7	-17	0
sao2	–	–	45	5	25	3	–	40
vg2	38	4	39	4	27	4	0	0
z4ml	13	3	10	2	5	2	33	0
Total1	–	–	4395	133	4238	117	–	–
Total2	2931	83	–	–	3086	77	–	–
Average	–	–	–	–	–	–	7.9	8.9

Table 6.7: Experimental results of FGSyn_d

In Table 6.8, we compare FGSyn_ufd, which is the minimum delay decomposition algorithm under unit fanout delay model, with FGSyn_d. FGSyn_ufd does 5.9% better than FGSyn_d under unit fanout delay model. *Unit* and *Unit Fanout* columns in table 6.8 are the delay calculated under unit delay model and unit fanout delay model, respectively.

name	FGSyn_d			FGSyn_ufd			
		Delay			Delay		
	CLB	Unit	Unit Fanout	CLB	Unit	Unit Fanout	% Red.
5xp1	14	2	4.6	15	2	4.2	8.7
9sym	7	3	4.2	7	3	4.2	0.0
C880	175	9	19.8	178	9	19.4	2.0
alu2	70	6	14.0	55	6	12.0	14.3
alu4	366	9	30.2	313	9	26.8	11.3
apex6	227	5	24.0	241	6	18.2	24.2
b9	27	3	6.8	28	4	6.6	2.9
clip	23	3	5.8	21	3	5.6	3.4
count	28	4	7.8	23	4	7.4	5.1
f51m	11	3	5.4	8	3	5.2	3.7
rot	200	7	16.8	207	9	16.6	1.2
vg2	21	4	6.4	20	5	6.4	0.0
z4ml	5	2	3.0	5	2	3.0	0.0
Total	1174	60	148.8	1121	65	135.6	–
Average	–	–	–	–	–	–	5.9

Table 6.8: Delay minimum decomposition (unit delay vs. unit fanout delay model)

In Table 6.9, we compare FGSyn_e results with the mis-pga(new). FGSyn_e shows 18.1% energy reduction over mis-pga(new). About half of the energy reduction is due to the reduction in the number of CLBs while the other half is due to the reduction in switching activities of the g-function outputs.

name	mis-pga(new)			FGSyn_e			
	CLB	Energy	Time	CLB	Energy	Time	% Red.
5xp1	17	43.1	9.8	9	23.4	15.7	45.8
9sym	7	16.8	126.2	7	15.5	15.9	7.9
9symml	7	16.8	46.1	7	15.5	15.8	7.9
C499	66	148.0	738.0	54	120.0	2.9	18.9
C880	78	169.0	648.8	87	198.4	116.2	-17.4
C1908	85	171.0	293.1	74	138.7	416.4	18.9
alu2	84	160.9	54.7	55	127.1	67.6	21.0
alu4	149	265.3	17.3	57	136.2	55.1	48.6
apex2	54	112.5	83.9	59	107.5	75.5	4.4
apex7	43	105.2	13.8	41	99.7	26.3	5.2
b9	27	67.1	9.9	28	62.8	3.6	6.6
clip	23	55.0	29.6	13	31.1	81.1	43.4
cm162a	10	19.5	3.7	10	17.5	3.7	10.2
cm42a	5	20.0	3.5	7	15.5	2.6	22.5
comp	23	55.4	11.9	23	48.4	39.2	12.6
count	28	59.9	5.5	23	49.7	13.3	17.0
duke2	108	180.5	62.8	86	139.2	88.6	22.9
e64	55	112.8	18.7	48	97.5	7.7	13.5
f51m	11	26.0	6.3	8	20.7	19.9	20.6
misex1	9	22.6	2.3	9	21.0	8.7	7.3
misex2	23	49.2	3.7	21	39.0	6.8	20.5
pm1	12	27.1	3.2	11	22.2	4.4	18.0
rd73	7	18.0	8.3	5	12.9	4.0	28.0
rd84	12	29.5	24.0	8	19.5	19.9	33.7
rot	139	324.0	125.7	135	306.8	410.3	5.3
sao2	28	62.1	8.9	26	45.7	152.0	26.5
vg2	20	42.9	3.2	18	37.4	47.2	12.8
z4ml	6	13.5	2.6	4	9.5	6.0	24.0
Total	1136	2393.6	–	933	1978.3	–	–
Average	–	–	–	–	–	–	18.1

Table 6.9: Energy comparison between mis-pga(new) and FGSyn_e

No other FPGA synthesis minimize the energy-delay product or even the energy. So it was difficult for us to compare results with other tools. Fortunately, we have the mis-pga(new) program and could generate the energy and delay results with this program using the same set of capacitance values, delay model, and input data activity profile. These results are depicted and compared with our results in Table 6.10, FGSyn_e shows 28.8% energy-delay product reduction over mis-pga(new). This reduction is mainly due to reduction in both energy and delay compared to mis-pga(new). There are however a few cases in which our results are somewhat worse that those of mis-pga(new). This is due to the heuristic nature of the proposed synthesis algorithm where locally optimal solution do not need to be a better solution in the end.

name	mis-pga(new)				FGSyn_ed				
	CLB	Delay	Energy	E-D	CLB	Delay	Energy	E-D	%
5xp1	17	43.1	5.4	232.7	9	23.4	4.2	98.3	57.8
9symml	7	16.8	4.4	73.9	7	15.6	4.4	68.6	7.2
C499	66	148.0	12.4	1835.2	68	124.4	9.4	1169.4	36.3
C880	78	169.0	23.6	3988.4	85	192.8	21.0	4048.8	-1.5
C1908	85	171.0	21.4	3659.4	75	141.1	24.2	3414.6	6.7
alu2	84	160.9	27.4	4408.7	57	131.2	12.8	1679.4	61.9
alu4	149	265.3	36.4	9656.9	57	137.0	29.2	4000.4	58.6
apex2	57	112.5	14.8	1665.0	60	108.9	16.4	1786.0	-7.3
apex6	147	335.1	25.0	8377.5	192	373.5	16.6	6200.1	26.0
clip	23	55.0	13.8	759.0	15	32.1	6.2	199.0	73.8
cm162a	10	19.5	5.8	113.1	9	19.6	5.0	98.0	13.4
comp	24	51.5	13.8	710.7	20	41.9	5.8	243.0	65.8
count	28	59.9	9.8	587.0	29	63.8	7.4	472.1	19.6
duke2	108	180.5	23.6	4259.8	86	141.2	16.0	2259.2	47.0
e64	55	112.8	31.8	3587.0	44	97.5	37.8	3685.5	-2.7
f51m	11	26.0	7.6	197.6	8	20.7	5.2	107.6	45.5
frg2	192	389.8	21.6	8419.7	171	341.9	21.6	7385.0	12.3
misex1	9	22.6	7.2	162.7	9	22.3	4.8	107.0	34.2
misex2	23	49.2	7.4	364.1	22	39.2	9.6	376.3	-3.4
rd73	7	18.0	3.2	57.6	5	12.9	3.0	38.7	32.8
rd84	12	29.5	5.0	147.5	8	19.5	4.6	89.7	39.2
rot	139	324.0	24.0	7776.0	177	394.0	21.2	8352.8	-7.4
sao2	28	62.1	17.0	1055.7	27	48.6	9.6	466.6	55.8
vg2	20	42.9	7.2	308.9	20	40.6	7.8	316.7	-2.5
z4ml	6	13.5	4.2	56.7	4	9.5	3.0	28.5	49.7
Total	1385	2878.5	373.8	62460.8	1264	2593.2	306.8	46691.3	–
Ave	–	–	–	–	–	–	–	–	28.8

Table 6.10: Energy-delay product comparison between mis-pga(new) and FGSyn_ed

In Table 6.11, we compare FGSyn_e with FGSyn_ed which objective is to minimize the energy-delay product ("E-D" column). FGSyn_ed shows 10.1% energy-delay product reduction over FGSyn_e.

name	FGSyn_e				FGSyn_ed				
	CLB	Energy	Delay	E-D	CLB	Energy	Delay	E-D	% Red.
5xp1	9	23.4	4.2	98.3	9	23.4	4.2	98.3	0.0
9symml	7	15.6	4.4	68.6	7	15.6	4.4	68.6	0.0
C499	54	120.0	11.8	1416.0	68	124.4	9.4	1169.4	17.4
C880	85	192.9	21.0	4050.9	85	192.8	21.0	4048.8	0.1
C1908	75	141.1	24.2	3414.6	75	141.1	24.2	3414.6	0.0
alu2	60	140.9	12.6	1775.3	57	131.2	12.8	1679.4	5.4
alu4	56	137.1	29.2	4003.3	57	137.0	29.2	4000.4	0.1
apex2	60	107.9	17.6	1899.0	60	108.9	16.4	1786.0	6.0
clip	23	50.4	8.0	403.2	15	32.1	6.2	199.0	50.6
cm162a	9	19.6	5.0	98.0	9	19.6	5.0	98.0	0.0
comp	23	55.5	10.6	588.3	20	41.9	5.8	243.0	58.7
count	30	57.1	15.2	867.9	29	63.8	7.4	472.1	45.6
duke2	86	141.2	16.0	2259.2	86	141.2	16.0	2259.2	0.0
e64	44	97.5	37.8	3685.5	44	97.5	37.8	3685.5	0.0
f51m	8	20.7	5.2	107.6	8	20.7	5.2	107.6	0.0
frg2	222	400.6	23.0	9213.8	171	341.9	21.6	7385.0	19.8
misex1	9	21.4	6.0	128.4	9	22.3	4.8	107.0	16.7
misex2	22	39.2	9.6	376.3	22	39.2	9.6	376.3	0.0
rd73	5	12.9	3.0	38.7	5	12.9	3.0	38.7	0.0
rd84	8	19.5	4.6	89.7	8	19.5	4.6	89.7	0.0
rot	139	317.5	28.0	8890.0	177	394.0	21.2	8352.8	6.0
sao2	27	48.6	9.6	466.6	27	48.6	9.6	466.6	0.0
vg2	19	40.7	9.2	374.4	20	40.6	7.8	316.7	15.4
z4ml	4	9.5	3.0	28.5	4	9.5	3.0	28.5	0.0
Total	1084	2230.8	318.8	44342.1	1072	2219.7	290.2	40491.2	–
Ave	–	–	–	–	–	–	–	–	10.1

Table 6.11: Comparison of FGSyn_e and FGSyn_ed

In Table 6.12, we compare FGSyn, FGSyn_d and FGSyn_e. The trade-off among area, delay and energy can be clearly seen. FGSyn_d has the least delay but higher number of CLBs and higher energy consumption compared to FGSyn or FGSyn_e while the FGSyn_e has the least energy consumption but higher number of CLBs and delay compared to FGSyn and FGSyn_d. Total1 is the total number of CLBs and total depth for all circuits. Total2 is the total number of CLBs, total depth and total energy consumption for all circuits except for $C5315$ and *des*. Our current switching activity estimation program [62] requires building global OBDD's for all internal nodes of the network. These OBDD's cannot however be build for the above mentioned circuits due to excessive memory requirement.

name	FGSyn			FGSyn_d			FGSyn_e		
	CLB	Delay	Energy	CLB	Delay	Energy	CLB	Delay	Energy
5xp1	9	2	23.4	9	2	23.4	9	2	23.4
9sym	7	3	16.5	7	3	16.5	7	3	15.5
9symml	7	3	16.5	7	3	16.5	7	3	15.5
C499	54	7	120.0	68	4	124.4	54	7	120.0
C880	87	14	200.2	125	9	369.3	87	14	198.4
C5315	298	14	–	550	11	–	–	–	–
alu2	49	5	127.9	50	5	127.0	55	6	127.1
alu4	56	18	137.4	272	9	702.6	57	19	136.2
apex2	60	11	108.5	71	9	130.3	59	11	107.5
apex6	178	10	354.5	174	5	402.4	257	10	347.4
apex7	42	7	104.6	58	5	136.3	41	7	99.7
b9	28	6	62.4	30	3	73.6	28	6	62.8
bw	27	1	69.0	27	1	69.0	27	1	69.0
clip	15	3	37.0	17	2	37.8	13	3	31.1
count	23	9	49.7	32	3	71.8	23	9	49.7
des	643	7	–	1130	5	–	–	–	–
duke2	85	9	140.1	123	6	217.2	86	9	139.2
e64	44	21	97.5	248	4	432.2	48	21	97.5
f51m	8	3	20.7	8	3	20.7	8	3	20.7
misex1	8	5	21.8	9	2	30.3	9	6	21.0
misex2	22	6	39.5	25	3	58.3	21	6	39.1
rd73	5	2	12.9	5	2	12.9	5	2	12.9
rd84	8	3	20.8	8	3	19.8	8	3	19.5
rot	136	13	316.6	172	7	453.2	135	16	306.8
sao2	25	4	46.1	25	3	45.9	26	4	45.7
vg2	16	5	38.9	20	4	46.4	18	6	37.4
z4ml	4	2	9.5	4	2	9.5	4	2	9.5
Total1	1944	193	–	3274	118	–	–	–	–
Total2	1003	172	2192.1	1594	102	3647.3	1092	179	2152.4

Table 6.12: Comparison of FGSyn, FGSyn_d, and FGSyn_e

6.4 Decomposition for Other Architectures of FPGA

In Table 6.13, we present the results obtained by using direct decomposition, type I two-layer decomposition (-T 1) and type II two-layer decomposition (-T 2). In general, type I decomposition produces somewhat better results.

In Table 6.14, we compare FGSyn results on XC4000 with ASYL [1] and PPR [35]. For these benchmarks, we achieved 13.4% CLB reduction over PPR and 12.4% CLB reduction over ASYL.

name	bx4		bx4 -T 1			bx -T 2		
	CLB	Time	CLB	% Red.	Time	CLB	% Red.	Time
5xp1	15	1.5	15	0.0	1.5	15	0.0	1.5
9sym	6	2.3	6	0.0	2.8	6	0.0	2.2
9symml	6	2.3	6	0.0	2.7	6	0.0	2.3
alu2	55	38.3	53	3.6	43.9	53	3.6	38.8
alu4	52	7.2	51	1.9	6.5	51	1.9	6.6
apex2	55	13.8	55	0.0	13.6	53	3.6	13.1
apex6	136	157.9	138	-7.0	150.2	128	0.8	153.2
apex7	39	3.5	37	5.1	3.0	39	0.0	3.5
b9	25	3.0	24	4.0	2.6	24	4.0	2.9
C880	75	13.7	78	-4.0	12.7	72	4.0	12.3
C1355	49	1.6	47	4.1	1.6	47	4.1	1.6
C1908	68	10.6	67	1.5	10.2	67	1.5	10.0
c8	20	1.9	20	0.0	1.7	18	10.0	1.6
clip	26	10.0	24	7.7	10.9	25	3.8	9.7
comp	18	6.9	18	0.0	6.6	17	0.0	6.8
count	21	1.3	19	9.5	1.2	19	9.5	1.3
decod	9	0.2	9	0.0	0.2	9	0.0	0.2
duke2	84	825.6	77	8.3	806.7	76	9.5	825.9
e64	43	0.8	43	0.0	0.9	43	0.0	0.7
f51m	11	1.7	10	9.1	1.7	10	9.1	1.6
misex1	8	0.4	8	0.0	0.5	9	-12.5	0.5
misex2	20	1.4	20	0.0	1.5	19	5.0	1.3
mux	6	0.7	5	16.7	0.7	5	16.7	0.7
rd73	6	1.2	6	0.0	1.2	7	-16.7	1.1
rd84	10	2.9	10	0.0	3.0	10	0.0	2.4
rot	127	205.0	119	6.3	189.5	120	5.5	194.9
sao2	31	15.3	29	6.5	21.4	31	0.0	16.3
vda	121	2785.0	116	4.1	2481.7	111	8.3	2267.7
vg2	16	2.9	15	6.3	3.3	16	0.0	2.7
z4ml	6	0.7	6	0.0	0.7	7	0.0	0.6
Total	1164	–	1131	–	–	1113	–	–
Average	–	–	–	3.3	–	–	2.4	–

Table 6.13: Experimental results of FGSyn options for XC4000 device

name	ASYL	PPR	FGSyn	% Red.	
	CLB	CLB	CLB	ASYL	PPR
5xp1	13	–	15	-15.4	–
9sym	9	–	6	33.3	–
9symml	–	36	6	–	83.3
alu2	51	71	52	-2.0	26.8
alu4	211	–	51	75.8	–
apex6	140	–	128	8.6	–
apex7	38	38	37	2.6	2.6
b9	–	20	23	–	-15.0
C1355	–	91	47	–	48.4
c8	–	17	18	–	-5.9
clip	29	–	23	20.7	–
comp	–	17	17	–	0.0
count	22	21	19	13.6	9.5
decod	–	10	9	–	10.0
duke2	73	–	70	4.1	–
e64	52	–	43	17.3	–
f51m	12	–	10	16.7	–
misex1	9	–	9	0.0	–
misex2	21	–	19	9.5	–
mux	–	5	5	–	0.0
rd73	10	–	7	30.0	–
rd84	14	–	10	28.6	–
sao2	23	–	29	-26.1	–
vda	–	97	109	–	-12.4
vg2	15	–	16	-6.7	–
Total1	742	–	544	–	–
Total2	423	–	342	–	–
Average	–	–	–	12.4	13.4

Table 6.14: Experimental results for XC4000 device

name	LUT				XC5000 CLB			
	4-in	5-in	6-in	mix	4-in	5-in	6-in	mix
5xp1	21	13	14	13	5	5	8	5
9symml	10	7	4	8	3	4	4	3
C499	98	90	90	90	25	25	25	25
C880	148	142	131	134	37	43	47	37
C1908	135	130	131	131	34	35	39	34
C5315	468	469	332	454	109	133	166	119
alu2	112	65	35	35	28	27	23	28
alu4	98	85	82	85	25	27	30	24
apex2	105	93	90	91	27	29	35	26
apex6	296	232	209	208	74	83	99	74
apex7	68	65	50	62	17	20	27	17
b9	54	49	47	49	13	13	14	13
bw	61	28	28	28	16	14	14	14
clip	44	24	12	18	11	11	9	5
cm162a	13	11	9	9	4	4	5	3
count	40	31	30	31	10	10	16	10
e64	85	84	84	84	22	22	22	22
f51m	15	12	10	12	4	4	5	4
frg2	289	305	239	265	71	90	105	70
misex1	18	10	11	10	5	4	6	4
misex2	37	32	29	31	10	11	12	9
rd73	10	6	7	6	3	3	5	3
rd84	14	9	8	9	4	4	6	4
rot	251	204	182	195	63	64	77	58
sao2	40	32	16	25	10	14	13	14
vg2	35	24	20	24	9	8	10	8
z4ml	8	5	6	5	2	2	4	2
Total	2573	2257	1906	2112	641	709	826	635

Table 6.15: Experimental results of various bound sets for XC5000.

In Table 6.15, we show the results of FGSyn_lcs decomposition under bound set size, 4, 5 and 6 on LUT count, CLB count and energy consumption for XC5000 device. Decomposition with a bound set size of 4 (column 4-in) gives the smallest number of CLBs, but has the highest energy consumption. This is because that decomposition with smaller bound set size tends to fit several smaller LUTs instead of one into one CLB, this will increase the number of output lines for each CLB which leads to higher energy consumption.

name	Energy Consumption			
	4-in	5-in	6-in	mix
5xp1	34.3	23.4	27.2	23.4
9symml	17.1	16.6	11.9	15.7
C499	120.9	120.0	120.0	120.0
C880	189.2	200.2	183.5	181.9
C1908	139.1	139.1	142.0	139.1
C5315	680.9	718.7	613.8	679.5
alu2	183.4	131.3	80.3	86.0
alu4	138.3	137.4	137.7	131.0
apex2	111.2	108.5	111.4	105.5
apex6	425.6	397.2	410.2	376.9
apex7	101.5	104.3	100.7	100.5
b9	64.5	62.4	61.4	62.2
bw	94.1	69.0	69.0	69.0
clip	72.7	52.5	28.9	34.1
cm162a	20.7	19.3	19.0	17.4
count	55.3	49.7	49.2	49.7
e64	97.7	97.5	97.5	97.5
f51m	22.8	20.7	19.4	20.7
frg2	352.2	381.3	352.8	340.1
misex1	29.3	21.4	24.0	20.3
misex2	40.9	39.5	38.7	39.1
rd73	17.2	12.9	15.9	13.1
rd84	21.0	19.8	18.5	18.5
rot	329.2	314.4	310.0	298.4
sao2	55.5	52.7	33.1	43.1
vg2	49.5	41.7	38.0	39.2
z4ml	12.6	9.5	12.4	9.5
Total	3476.9	3360.9	3126.5	3131.2

Table 6.16: Energy comparison of various bound sets decomposition for XC5000.

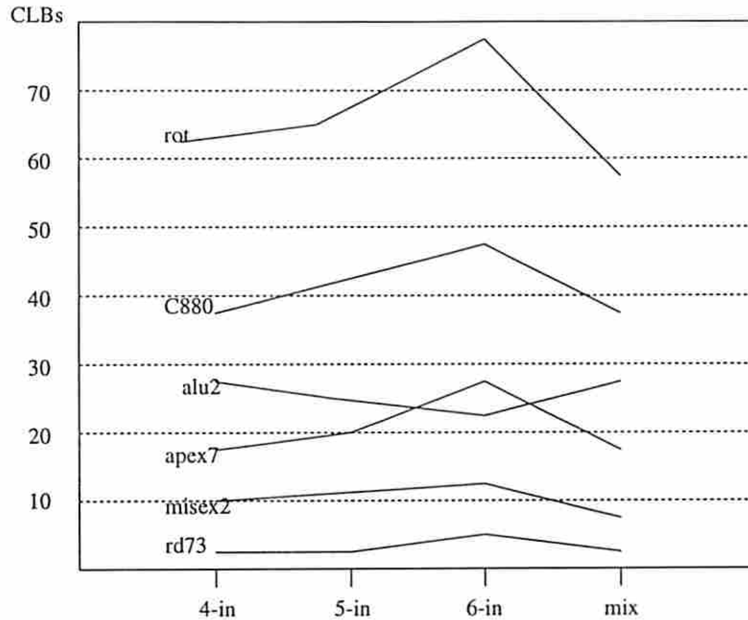


Figure 6.1: Selected benchmarks CLBs count for fix and variable bound set decomposition

In Figure 6.1 we show 6 typical benchmarks over fix bound set size of $\mid B \mid = 4, 5, 6$ and variable bound set sizes decomposition result of CLBs count. The variable bound set decomposition has similar result of fix bound set $\mid B \mid = 4$ but slight better than that. The *alu*2 has smallest CLB count on fix bound set $\mid B \mid = 6$ decomposition due to special logic structure that most of 4-input of 5-input decomposition will have small variables reduction or undecomposable but not the case of 6-input decomposition.

Chapter 7

Conclusion and Future Work

In this thesis, we described techniques for OBDD-based decomposition of multiple-output Boolean functions and presented Boolean methods for extracting common subfunctions from these functions. We also described a heuristic, graph-based approach for extraction of common subfunctions when these subfunctions have large (≥ 5) input variable support. We considered different objective functions, including area, delay, power, and energy-delay product and developed appropriate decomposition scheme for minimizing each objective function. Detailed comparison of results for each objective function was presented.

Application of these methods to the synthesis of LUT-based FPGAs was presented. We showed that the synthesis problem for FPGA architectures is very different from that for the conventional, standard-cell based designs. In same cases, we developed special decomposition schemes (e.g. two-layer decomposition for XC4000 device) to fully explore the intricacies of the FPGA architecture. Results indicate that Boolean techniques produce results that are much better than algebraic techniques which are commonly used, while the efficiency can be maintained by using appropriate techniques.

Techniques presented in this thesis can be further improved. First, don't cares can be used to increase logic sharing among multiple Boolean functions, or to reduce the size of OBDD's [6, 8, 7, 54, 5, 44, 56]. Second, the Boolean mapping

technique presented in this thesis can be also applied to the cell library-based technology mapping. It is yet to be seen whether the increased computational effort is justified in light of the potential for improved mapping results. A related question is to determine what subset of gates in a standard-cell library are most useful in producing high quality mapping results using our Boolean mapping technique. Finally, most of the CPU time used by FGSyn is spent on finding a good bound set for decomposition. A key problem is that of finding a "good" bound set quickly. Ideas taken from the work performed on dynamic OBDD variable ordering [53] seem to be particularly promising here.

Appendix A

FGSyn

The techniques described in this thesis are implemented as command "bx" with options in SIS. The command options are listed as following:

usage: bx [-x #] [-i #] [-e #] [-p #] [-M model] [-w prob] [-bfrlncsgdV] [-v #]

-x 3-10	Specify the bound set size (default = 5)
-i 5-20	Specify the maximum size of the input sets (default = 9)
-b	Build the global BDD if it is smaller than local BDD
-e 0-4	Apply node collapsing rule (default = 0)
	0: Do not apply any rule
	1: Apply rule 1, k-bound support collapse
	2: Apply rule 2, Support reduction collapse
	3: Apply rule 3, BDD size reduction collapse
	4: Apply rules 1-3 above
-f	Do minimum energy-delay product decomposition
-p 1-4	Pick a node clustering strategy (default = 4)
	1: Put all nodes in one cluster
	2: Each node is assigned to a unique cluster
	3: Use greedy clustering with parameters a=1.3, b=10
	4: Use greedy clustering with parameters a=2, b=20
-r	Recompute the best bound set after each decomposition
-l	Do decomposition for variable-size LUT's (XC5000)
-n	Decompose to minimize sum of the g-function supports
	(Note that this may lead to non-disjunctive decomposition)

-c	Use column encoding
-s	Use unit-code shared subfunction encoding
-m	Use multi-code shared subfunction encoding
-g	Use graph based encoding
-d	Do minimum delay decomposition
-M model	Set delay model (default = unit)
	unit: Use unit delay model
	unit-fanout: Use unit fanout delay model
-w prob	Do minimum energy decomposition (default prob = u)
	u: Use uniform 0.5 signal probability for primary inputs
	r: Use pseudo-random signal probability for primary inputs
-V	Verify network after fg synthesis
-v 0-3	Print debugging info (default = 0)

Reference List

[1] P. Abouzeid, B. Babba, M. C. de Paulet, and G. Saucier. Input-driven partitioning methods and application to synthesis on table-lookup-based FPGA's. In IEEE *Transactions on Computer Aided Design of Integrated Circuits and Systems*, pages 913–925, July 1993.

[2] P. Abouzeid, L. Bouchet, K. Sakouti, and G. Saucier. Lexicographical expression of boolean function for multilevel synthesis of high speed circuits. In *Proceedings of SASHIMI*, pages 31–39, October 1990.

[3] R. Ashenhurst. The decomposition of switching functions. In *Proceeding of the International Symposium on Theory of Switching Functions*, pages 74–116, April 1959.

[4] Robert L. Ashenhurst. The decomposition of switching functions. In *Proceedings of International Symposium on Theory of Switching Functions*, April 1959.

[5] K. Bartlett, R. Brayton, G. Hachtel, R. Jacobi, C. Morrison, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Algorithms for multi-level logic minimization using implicit don't-cares. In *ICCD86*, Oct. 1986.

[6] K. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. Multi-level logic minimization using implicit don't cares. In IEEE *Transactions on Computer*

Aided Design of Integrated Circuits and Systems, volume 7, pages 723–740, June 1988.

[7] K.A. Bartlett, R. Brayton, G. Hachtel, R. Jacoby, C. Morrison, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level logic minimization using implicit don't cares. IEEE *Transactions on Computer Aided Design of Integrated Circuits and Systems*, CAD-7(6):723–740, June 1988.

[8] D. Brand. Redundancy and don't cares in logic synthesis. In IEEE *Transactions on Computers*, volume C-32, pages 947–952, October 1983.

[9] R. Brayton, G. Hachtel, and A. Sangiovanni-Vincentelli. Multi-level logic synthesis. *Proceedings of the* IEEE, 1990.

[10] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A multiple-level logic optimization system. IEEE *Transactions on Computer Aided Design of Integrated Circuits and Systems*, CAD-6(6), Nov. 1987.

[11] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Mis: Multiple-level interactive logic optimization system. In *IEEE Transactions on Computer Aided Design*, volume CAD-6, November 1987.

[12] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level logic optimization and the rectangular covering problem. In *Proceedings of the* IEEE *International Conference on Computer Aided Design*, pages 66–69, November 1987.

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

[14] R. K. Brayton and C. McMullen. The decomposition and facorization of Boolean expressions. In *Proceedings of the International Symposium on Circuits and Systems*, pages 49–54, Rome, May 1982.

[15] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. In IEEE *Transactions on Computers*, volume C-35, pages 677–691, August 1986.

[16] S-C. Chang and M. Marek-Sadowska. Technology mapping via transformations of function graph. In *Proceedings of the International Conference on Computer Design*, October 1992.

[17] K. Chaudhary and M. Pedram. A near-optimal algorithm for technology mapping minimizing area under delay constraints. Proceedings of the 29th Design Automation Conference, June 1992.

[18] J. Cong and Y. Ding. Beyond the combinatorial limit in depth minimization for LUT-based FPGA designs. In *Proceedings of the* IEEE *International Conference on Computer Aided Design*, pages 110–114, November 1993.

[19] J. Cong and Y. Ding. On area/depth trade-off in LUT-based FPGA technology mapping. In *Proceedings of the 30th Design Automation Conference*, pages 213–218, June 1993.

[20] J. Cong and Y. Ding. FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA design. In IEEE *Transactions on Computer Aided Design of Integrated Circuits and Systems*, pages 1–12, January 1994.

[21] H. A. Curtis. *A New Approach to the Design of Switching Circuits.* Van Nostrand, Princeton, N.J., 1962.

[22] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Technology mapping in MIS. In *Proceedings of the* IEEE *International Conference on Computer Aided Design*, pages 116–119, November 1987.

[23] D. Filo, J. C. Yang, F. Mailhot, and G. D. Micheli. Technology mapping for a two-output RAM-based FPGAs. In *Proceedings of the European Design Automation Conference*, pages 534–538, February 1991.

[24] R.J. Francis, J. Rose, and Z. Vranesic. Chortle: A technology mapping program for lookup table-based FPGAs. In *Proceedings of the 27th Design Automation Conference*, pages 613–619, June 1990.

[25] R.J. Francis, J. Rose, and Z. Vranesic. Chortle-crf: Fast technology mapping for lookup table-based FPGAs. In *Proceedings of the 28th Design Automation Conference*, pages 227–233, June 1991.

[26] R.J. Francis, J. Rose, and Z. Vranesic. Technology mapping of lookup table-based FPGAs for performance. In *Proceedings of the* IEEE *International Conference on Computer Aided Design*, pages 568–571, November 1991.

[27] A. D. Friedman and P. R. Menon. *Theory and Design of Switching Circuits.* Computer Science Press, 1975.

[28] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.

[29] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W. H. Freeman and Company, 1979.

[30] S. He and M. Torkelson. Disjoint decomposition with partial vertex chart. In *International Workshop on Logic Synthesis*, pages p2a 1–5, May 1993.

[31] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. *IEEE Symposium on Low Power Electronics*, October 1995.

[32] W-J. Hsu and W-Z. Shen. Coalgebraic division for multilevel logic synthesis. In *Proceedings of the 29th Design Automation Conference*, pages 438–442, June 1992.

[33] D. A. Huffman. A method for the construction of minimum redundancy codes. In *Proceedings of the IRE*, volume 40, pages 1098–1101, September 1952.

[34] T-T. Hwang, R. M. Owens, and M. J. Irwin. Efficiently computing communication complexity for multilevel logic synthesis. IEEE *Transactions on Computer Aided Design of Integrated Circuits and Systems*, 11(5):545–554, May 1992.

[35] Xilinx Inc. *The Programmable Logic Data Book.* Xilinx Inc., 2100 Logic Drive, San Jose, CA 95124., 1994.

[36] R. M. Karp. Functional decomposition and switching circuit design. In *J. Soc. Indust. Appl. Math.*, volume 11, pages 291–335, June 1963.

[37] K. Karplus. Xmap: a technology mapper for table-lookup field-programmable gate arrays. In *Proceedings of the 28th Design Automation Conference*, pages 240–243, June 1991.

[38] K. Keutzer. DAGON: Technology mapping and local optimization. In *Proceedings of the Design Automation Conference*, pages 341–347, June 1987.

[39] Y-T. Lai. *Logic Verification and Synthesis using Function Graphs.* PhD thesis, University of Southern California, 1993.

[40] Y-T. Lai, K-R. R. Pan, and M. Pedram. FPGA synthesis using function decomposition. In *Proceedings of the International Conference on Computer Design*, pages 30–35, October 1994.

[41] Y-T. Lai, K-R. R. Pan, and M. Pedram. OBDD-based function decomposition: Algorithms and implementation. *on* IEEE *Transactions on Computer Aided Design of Integrated Circuits and Systems*, February 1996.

[42] Y-T. Lai, K-R. R. Pan, M. Pedram, and Sarma Sastry. FGMap: A technology mapping algorithm for look-up table type FPGAs based on function graphs. In *Proceedings of International Workshop on Logic Synthesis*, May 1993.

[43] Y-T. Lai, M. Pedram, and S. Sastry. BDD based decomposition of logic functions with application to FPGA synthesis. In *Proceedings of the 30th Design Automation Conference*, pages 642–647, June 1993.

[44] F. Mailhot and G.D. Micheli. Technology mapping using boolean matching and don't care sets. In *Proceedings of the European Conference on Design Automation*, pages 212–216, 1990.

[45] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic synthesis for programmable gate arrays. In *Proceedings of the 27th Design Automation Conference*, pages 620–625, June 1990.

[46] R. Murgai, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli. Improved logic synthesis algorithms for table look up architectures. In *Proceedings of the* IEEE *International Conference on Computer Aided Design*, pages 564–567, November 1991.

[47] R. Murgai, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli. Performance directed synthesis for table look up programmable gate arrays. In *Proceedings of the* IEEE *International Conference on Computer Aided Design*, pages 572–575, November 1991.

[48] K-R. R. Pan, Y-T. Lai, and M. Pedram. LUT-based FPGA synthesis for low power. In *International Workshop on Power and Timing Modeling, Optimization and Simulation*, October 1994.

[49] K-R. R. Pan and M. Pedram. FPGA synthesis for minimum area, delay and power using function decomposition. *Poster paper of International Symposium on Field Programmable Gate Arrays*, February 1995.

[50] K-R. R. Pan and M. Pedram. FPGA synthesis for minimum area, delay and power. *Poster paper of European Design and Test Conference*, March 1996.

[51] J.P. Roth and R.M. Karp. Minimization over boolean graphs. In *IBM Journal*, pages 227–238, April 1962.

[52] P. J. Roth and R. M. Karp. Minimization over Boolean graphs. *IBM Journal of Research and Development*, 6, April 1962.

[53] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the Design Automation Conference*, 1993.

[54] Alex Saldanha, Albert Wang, Robert Brayton, and Alberto Sangiovanni-Vincentelli. Multi-level logic simplification using don't cares and filters. In *Proceedings of the Design Automation Conference*, June 1989.

[55] T. Sasao. FPGA design by generalized functional decomposition. In *Logic Synthesis and Optimization*, pages 233–258. Kluwer Academic Publisher, 1993.

[56] H. Savoj, R. K. Brayton, and H. J. Touati. Use of image computation techniques in extracting local don't cares and network optimization. In *International Workshop on Logic Synthesis*, May 1991.

[57] H. Savoj and H. Y. Wang. Improved scripts in mis-ii for logic minimization of combinational circuits. In *International Workshop on Logic Synthesis*, May 1991.

[58] P. Sawkar and D. Thomas. Area and delay mapping for table-look-up based field programmable gate arrays. In *Proceedings of the 29th Design Automation Conference*, pages 368–373, June 1992.

[59] M. Schlag, P. Chan, and J. Kong. Empirical evaluation of multilevel logic minimization tools for a field programmable gate array technology. In *International Workshop on Field Programmable Logic and Applications*, pages 201–213, September 1991.

[60] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovani-Vincentelli. Sequential circuit design using synthesis and optimization. In *Proceedings of the International Conference on Computer Design*, October 1992.

[61] V.Y. Shen and A.C. McKellar. An algorithm for the disjunctive decomposition of switching functions. IEEE *Transactions on Computers*, C-19(3):239–248, March 1970.

[62] C-Y. Tsui, M. Pedram, and A. M. Despain. Efficient estimation of dynamic power dissipation under a real delay model. In *Proceedings of the* IEEE *International Conference on Computer Aided Design*, pages 224–228, November 1993.

[63] J. Vasudevamurthy and J. Rajski. A method for concurrent decomposition and factorization of boolean expressions. In *Proceedings of the* IEEE *International Conference on Computer Aided Design*, pages 510–513, November 1990.

[64] N. Woo. A heuristic method for FPGA technology mapping based on edge visibility. In *Proceedings of the 28th Design Automation Conference*, pages 248–251, June 1991.

[65] Saeyang Yang. Logic synthesis and optimization benchmarks user guide: Version 3.0. Technical report, Microelectronics Center of North Carolina, P.O. Box 12889, Research Triangle Park, NC 27709, January 1991.