Technology Mapping for Lookup-Table Based Field-Programmable Gate Arrays

by

Robert J. Francis

A thesis
submitted in conformity with the requirements
for the degree of
Doctor of Philosophy

Graduate Department of Electrical Engineering
Computer Group
University of Toronto
Toronto, Ontario
Canada

©Robert J. Francis 1993

Abstract

Field Programmable Gate Arrays (FPGAs) provide a new approach to Application Specific Integrated Circuit (ASIC) implementation that features both large scale integration and user programmability. The logic capacity of these devices is large enough to make automated synthesis essential for the efficient design of FPGA circuits. This thesis focuses on the class of FPGAs that use lookup tables (LUTs) to implement combinational logic. A K-input lookup table is a digital memory that can implement any Boolean function of K variables. Lookup table circuits present new challenges for logic synthesis, particularly technology mapping, which is the phase of logic synthesis directly concerned with the selection of the circuit elements to implement the final circuit. Conventional library-based technology mapping has difficulty with lookup table circuits because each lookup table can implement a large number of different functions.

This thesis presents two new technology mapping algorithms that construct circuits of K-input lookup tables from networks of ANDs, ORs and NOTs. The first algorithm, referred to as the area algorithm, minimizes the number of LUTs in the final circuit, and the second algorithm, referred to as the delay algorithm, minimizes the number of levels of LUTs. The key feature of both algorithms is the application of bin packing to the decomposition of nodes in the original network. The original network is first partitioned into a forest of trees, each of which is mapped separately. For each tree, the circuit constructed by the area algorithm is shown to be an optimal tree of LUTs for values of $K \leq 5$ and the circuit constructed by the delay algorithm is an optimal tree of LUTs for values of $K \leq 6$. Both algorithms also include optimizations that exploit reconvergent paths, and the replication of logic at fanout nodes to further optimize the final circuit. The algorithms described in this thesis are implemented in a software program called Chortle and experimental results for a set of benchmark networks demonstrate the effectiveness of the algorithms.

Acknowledgements

First and foremost, I would like to thank Dr. Jonathan Rose and Dr. Zvonko Vranesic. This thesis could not have been completed without their advice and guidance. I owe a unique debt to Jonathan for introducing me to the *new and exciting* world of FPGAs. I would also like to thank Dr. David Lewis for encouraging me to return to graduate school.

Rejoining the community of students in the Sandford Fleming and Pratt labs has been one of the greatest rewards of the last five years. I thank you all for providing inspiration and friendship.

I can only begin to express my gratitude to Ming-Wen Wu for her constant understanding and encouragement. Thank you Ming. I would also like to thank my parents and Ann, Jyrki, Bill, and Cathy, for their wholehearted support, and Emily, Laura, Naomi, and Gillian. for the joy and wonder that helped me keep it all in perspective. A special thanks goes to Ya-Ya for sharing her strength and warmth when she herself faced adversity.

Finally, I would like to thank the Natural Sciences and Engineering Research Council, the Information Technology Research Centre, the University of Toronto, and my parents for the financial support that permitted me to pursue a full-time degree.

This thesis is dedicated to the memory of Gladys Clara Adams.

Contents

1 Introduction **1**

2 Background **6**

2.1 Logic Optimization 7

2.2 Technology Mapping 9

2.2.1 Rule-based Technology Mapping 9

2.2.2 Library-based Technology Mapping 11

2.2.3 Cell Generator Technology Mapping 13

2.3 Lookup Table Technology Mapping 15

2.3.1 Xilinx 3000 CLBs 16

2.3.2 Mis-pga 16

2.3.3 Asyl 19

2.3.4 Hydra 20

2.3.5 Xmap 21

2.3.6 VISMAP 23

2.3.7 DAG-Map 24

2.4 Summary 25

3 The Area Algorithm **26**

3.1 Mapping Each Tree 29

3.1.1 Constructing the Decomposition Tree 33

3.1.2 Optimality 36

3.2 Exploiting Reconvergent Fanout 38

3.3 Replication of Logic at Fanout Nodes 43
3.4 Summary 47

4 The Delay Algorithm 48
4.1 Mapping Each Tree 50
4.1.1 Constructing the Decomposition Tree 52
4.1.2 Optimality 55
4.2 Exploiting Reconvergent Paths 56
4.3 Replication of Logic at Fanout Nodes 56
4.4 Reducing the Area Penalty 57
4.4.1 Avoiding Unnecessary Replication 58
4.4.2 Merging LUTs into their Fanout LUTs 59
4.4.3 Mapping for Area on Non-Critical Paths 60
4.5 Summary 62

5 Area Optimality 63
5.1 Outline for Proof of Lemma 5.2 64
5.1.1 Notation for the Circuit $\mathcal{A}$ 65
5.1.2 Transforming the Circuit $\mathcal{B}$ 66
5.1.3 Proof of Lemma 5.2 70
5.2 Circuits with Fanout 73
5.3 Summary 73

6 Delay Optimality 75
6.1 Outline for Proof of Lemma 6.2 76
6.1.1 Notation for the Circuit $\mathcal{A}$ 77
6.1.2 Transforming the Circuit $\mathcal{B}$ 79
6.1.3 Proof of Lemma 6.3 82
6.1.4 Proof of Lemma 6.2 83
6.2 Summary 83

7 Experimental Evaluation **84**

7.1 Results for the Area Algorithm 84

7.1.1 Circuits of 5-input LUTs 84

7.1.2 Circuits of Xilinx 3000 CLBs 88

7.1.3 Chortle vs. Mis-pga 93

7.1.4 Chortle vs. Xmap 95

7.1.5 Chortle vs. Hydra 95

7.1.6 Chortle vs. VISMAP 98

7.2 Results for the Delay Algorithm 99

7.2.1 Circuits of 5-input LUTs 99

7.2.2 Reducing the Area Penalty 103

7.2.3 Chortle vs. Mis-pga 107

7.2.4 Chortle vs. DAG-Map 108

7.3 Summary 109

8 Conclusions **111**

8.1 Future Work 112

A Optimality of the First Fit Decreasing Algorithm **114**

A.1 Bin Packing 114

A.2 Outline of Proof 116

A.3 Notation 118

A.4 Deriving a Complete Set of Cases 121

A.4.1 Eliminating Holes Greater than H 121

A.5 How to Prove a Case 124

A.5.1 An Example 125

A.5.2 The General Case 128

A.5.3 Finding the Required Linear Combination 130

A.6 Reducing the Number of Cases 131

A.7 Presentation of the Cases 132

A.8 $K = 2$ 133

A.9 $K = 3$ 136
A.10 $K = 4$ 140
A.11 $K = 5$ 147
A.12 $K = 6$ 161
A.13 Counter Examples 171
A.14 Summary 171

Bibliography **172**

Chapter 1

Introduction

Field-Programmable Gate Arrays (FPGAs) provide a new approach to Application Specific Integrated Circuit (ASIC) implementation that features both large scale integration and user programmability. An FPGA consists of a regular array of logic blocks that implement combinational and sequential logic functions and a user-programmable routing network that provides connections between the logic blocks. In conventional ASIC implementation technologies, such as Mask Programmed Gate Arrays and Standard Cells, the connections between logic blocks are implemented by metalization at a fabrication facility. In an FPGA the connections are implemented in the field using the user-programmable routing network. This reduces manufacturing turn-around times from weeks to minutes and reduces prototype costs from tens of thousands of dollars to hundreds of dollars [Rose90].

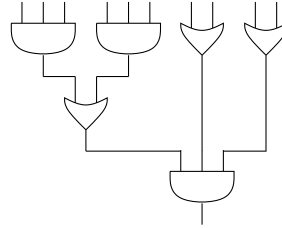
There are, however, density and performance penalties associated with user-programmable routing. The programmable connections, which consist of metal wire segments connected by programmable switches, occupy greater area and incur greater delay than simple metal wires. To reduce the density penalty, FPGA architectures employ highly functional logic blocks, such as lookup tables, that reduce the total number of logic blocks and hence the number of programmable connections needed to implement a given application. These complex logic blocks also reduce the performance penalty by reducing the number of logic blocks and programmable connections on the critical paths in the circuit.

The high functionality of FPGA logic blocks presents new challenges for logic synthesis. This thesis focuses on new approaches to technology mapping for FPGAs that use lookup tables to implement combinational logic. A K-input lookup table (LUT) is a digital memory that can implement any Boolean function of K variables. The K inputs are used to address a 2^K by 1-bit memory that stores the truth table of the Boolean function. Lookup table-based FPGAs account for a significant portion of the commercial FPGA market [Rohl91] and recent studies on FPGA logic block architectures have suggested that lookup tables are an area-efficient method of implementing combinational functions [Rose89], [Rose90], [Koul92] and that the delays of LUT-based FPGAs are at least comparable to the delays of FPGAs using other types of logic blocks [Sing91], [Sing92].

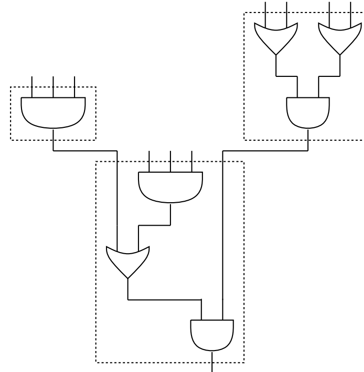
The level of integration available in FPGAs is large enough to make manual circuit design impractical and therefore automated logic synthesis is essential for the efficient design of FPGA circuits. In addition, the development of FPGA architectures requires effective logic synthesis to evaluate the benefits of alternative logic blocks. Logic synthesis, in general, takes a functional description of the desired circuit, and using the set of circuit elements available in the ASIC implementation technology, produces an optimized circuit. For an FPGA the set of available circuit elements consists of the array of logic blocks.

Technology mapping is the logic synthesis task that is directly concerned with selecting the circuit elements used to implement the optimized circuit. Previous approaches to technology mapping have focused on using circuit elements from a limited set of simple gates. However, such approaches are inappropriate for complex logic blocks where each logic block can implement a large number of functions. A K-input lookup table can implement 2^{2^K} different functions. For values of K greater than 3 the number of different functions becomes too large for conventional technology mapping. Therefore, new approaches to technology mapping are required for LUT-based FPGAs.

This thesis presents two technology mapping algorithms that are among the earliest work specifically addressing technology mapping for LUT circuits. Both of these



a) Boolean network



b) Lookup table circuit

Figure 1.1: Network and Circuit

algorithms implement a Boolean network as a circuit of K-input LUTs. For example, consider the network shown in Figure 1.1a. This network can be implemented by the circuit of 5-input lookup tables shown in Figure 1.1b. In this figure, dotted boundaries indicate the function implemented by each lookup table.

The original motivation for this thesis was an architectural investigation into LUT-based FPGA architectures [Rose89], [Rose90]. To support this investigation, a simple LUT technology mapping program called *chortle* [1] was developed. This program used a greedy algorithm to decompose large functions into smaller functions and pack these into K-input LUTs. Experience with chortle lead to the development of a more effective program called *Chortle-x* [Fran90]. The key feature of this program was the decomposition of nodes in the network to reduce the number of LUTs in the final circuit. The network was first partitioned into a forest of trees and dynamic

[1]This program was based on the parsing and data structures of the *Tortle* [Lewi91] functional simulator, and used a *chewing* action to decompose functions too large to fit into a single LUT, hence the name *chortle*.

programming was used to find the minimum number of K-input LUTs required to implement each tree. The optimal decomposition of each node in the tree was found by an exhaustive search.

This thesis focuses on the algorithms used in two subsequent programs, *Chortle-crf* [Fran91a] and *Chortle-d* [Fran91b]. The goal of Chortle-crf is to minimize the number of K-input LUTs required to implement a network. This increases the size of designs that can be realized with the fixed number of lookup tables available in a given LUT-based FPGA. The major innovation in Chortle-crf is the application of a bin packing approximation algorithm to the construction of effective decompositions. For values of K less than or equal to 5, this bin packing approach constructs an optimal tree of K-input LUTs implementing a network that is a fanout-free tree. The bin packing approach is much faster than the exhaustive search used in Chortle-x. This increase in speed makes it practical to consider optimizations exploiting reconvergent paths and the replication of logic at fanout nodes to further reduce the number of lookup tables. The algorithm used in Chortle-crf will be referred to as the *area algorithm*.

The goal of Chortle-d is to minimize the number of *levels* of K-input LUTs in the final circuit. This can improve circuit performance by reducing the contribution of logic block delays to the total delay on the critical paths in the circuit. Chortle-d uses bin packing to construct effective decompositions and is able to construct an optimal depth tree of LUTs implementing a network that is a fanout-free tree, for values of K less than or equal to 6. It also exploits reconvergent paths and the replication of logic at fanout nodes to reduce the number of levels. The algorithm used in Chortle-d will be referred to as the *delay algorithm*.

Chortle and the program Mis-pga [Murg90], which was developed concurrently with Chortle, represent the first research to specifically address technology mapping for LUT circuits. Several other LUT technology mappers have been developed subsequently, and the synthesis of LUT circuits is currently an active area of research.

This dissertation is organized as follows: Chapter 2 presents background material on logic synthesis and technology mapping in general, and discusses other approaches to technology mapping for LUT circuits that have been developed concurrently with,

or subsequent to the work presented in this thesis. Chapters 3 and 4 describe the details of the area and delay algorithms. Chapters 5 and 6 present proofs that the area and delay algorithms are optimal for restricted classes of networks, and Chapter 7 presents experimental results for both algorithms including comparisons to other LUT technology mapping algorithms. Finally, Chapter 8 draws some conclusions and suggests directions for future research.

Chapter 2

Background

This chapter presents background material on logic synthesis. It begins with a brief description of logic optimization and then describes conventional approaches to technology mapping. The limitations of these previous approaches when applied to LUT circuits provide the motivation for the LUT technology mapping algorithms presented in this thesis. The chapter concludes by discussing other research that also addresses technology mapping for LUT circuits.

The goal of logic synthesis is to produce a minimum cost circuit that implements a desired combinational function. The cost of the circuit is typically a measure of its area or delay, or a function of both. The combinational function can be represented by a Directed Acyclic Graph (DAG) known as a Boolean network. Nodes in this network represent Boolean functions, and each node has an associated variable and local function. The *support* of this local function is the set of variables corresponding to the node's predecessors in the DAG. The global function represented by the node is determined by applying the local function to the global functions represented by its support. Examples of local functions include ANDs, ORs, sum-of-products expressions. In the network shown in Figure 2.1 the local functions are sum-of-products. The support of node z is $\{y, d, e\}$, the local function is $z = yd + e$ and the global function is $z = (a + bc)d + e$.

The net-list for the final circuit can also be represented by a Boolean network. In this case, each node corresponds to one circuit element and each edge corresponds

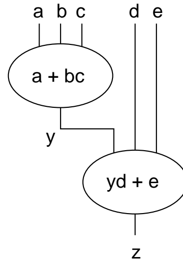


Figure 2.1: Boolean Network

to a wire. The function implemented by a circuit element is specified by the local function of its corresponding node. In this thesis the term *network* will be used to refer to a Boolean network representing a combinational function and the term *circuit* will be used to refer to a Boolean network representing a circuit net-list.

Logic synthesis, as illustrated in Figure 2.2, can be conceptually divided into technology-independent logic optimization and technology-dependent optimization which is known as technology mapping [Detj87], [Rude89]. Logic optimization takes the network describing the desired combinational function and produces a functionally equivalent network optimized for some cost function. Technology mapping then constructs an optimized circuit that realizes the optimized network using the restricted set of circuit elements available in the implementation technology.

2.1 Logic Optimization

In many logic synthesis systems such as misII [Bray87] and BOLD [Bost87], the original network is first restructured to reduce a cost function that is calculated directly from the network itself. The intention is to improve the final circuit by reducing the complexity of the network. However, this technology-independent logic optimization does not consider which circuit elements will implement the circuit. The modifications applied to the network typically include redundancy removal and common sub-expression elimination. Logic optimization may also exploit *don't cares* in the specification of the desired combinational function to simplify the network.

In the misII logic synthesis system the complexity of a network is measured by

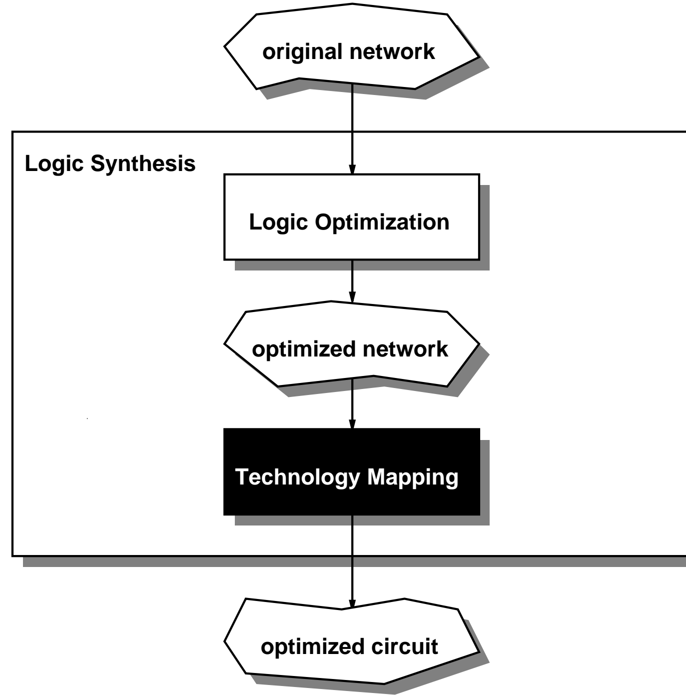


Figure 2.2: Logic Synthesis

counting the number of *literals* in the local function for each node. Each local function is a sum-of products expression and each instance of a variable in this expression counts as one literal. For example, the following 4-input, 2-output network has 11 literals

$$f = ac + ad + bc + bd$$

$$g = a + b + c$$

The complexity of this network can be reduced by the following modifications. The expression $(a + b)$ can be *factored* out of the equations for nodes f and g, and a new node e, implementing the function $a + b$ created. The variable e is *resubstituted* into the equations for nodes f and g, resulting in the following 7-literal network:

$$e = a + b$$

$$f = e(c + d)$$

$$g = e + c$$

2.2 Technology Mapping

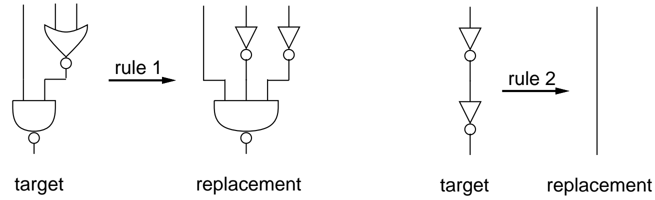
After logic optimization has produced the optimized network, technology mapping selects circuit elements to implement sub-functions within this network. When wired together, these circuit elements form a circuit implementing the entire network. This circuit is optimized to reduce a cost function that typically incorporates area and delay. Conventional approaches to technology mapping can be categorized as rule-based, library-based and cell generator approaches. The following sections briefly describe each of these approaches.

2.2.1 Rule-based Technology Mapping

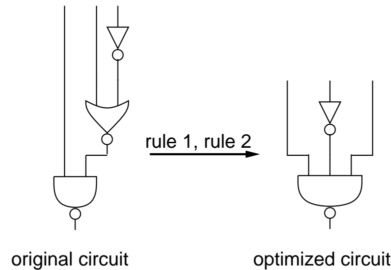
In early logic synthesis systems, such as SOCRATES [Greg86] and LSS [Joyn86], technology mapping is performed by a series of local transformations to a circuit net-list. The net-list is initially constructed by implementing each node in the original network by a single circuit element. The area and delay of the circuit are then optimized by selecting the appropriate sequence of transformations.

In SOCRATES, a rule-based expert system is used to select the sequence of local transformations. Each transformation is expressed as a rule that consists of a *target* configuration and a functionally equivalent *replacement* configuration. Figure 2.3a illustrates the target and replacement configurations of two rules. Figure 2.3b illustrates an original circuit and the optimized circuit that results from applying these two rules in sequence.

The first step in applying a rule to the circuit consists of finding a sub-circuit that matches the target configuration. A complete match consists of a series of partial matches that proceed from the output to the inputs of the target configuration. The matching algorithm uses backtracking to recover from failed partial matches. When a matching sub-circuit is found, the cost benefit of substituting the target configuration with the replacement configuration is calculated using area and delay estimates extracted from the resulting circuit. The decision to apply the rule is then based on this incremental cost.



a) Two rules



b) Applying the rules sequentially

Figure 2.3: Rule-based Technology Mapping, from [Greg86]

The key to optimizing the circuit is the selection of the next rule to apply. One strategy is to consider the rules in a fixed order and to always apply the first beneficial rule. This approach is effective for area optimization, because the effect of each rule on circuit area can be predicted locally. However, the strategy is less effective for delay optimization, because it is difficult to make a local prediction of the effect of each transformation on the overall delay of the circuit.

A second approach considers short sequences of rules rather than individual rules. The extent of this search is controlled by limiting its depth and breadth. The depth is the number of rules in the short sequence and the breadth is the number of alternative rules to be considered at any point in the sequence. The search is also restricted to sequences of rules that only apply to a limited neighborhood in the circuit to avoid sequences of rules that could be applied independently. This approach improves the quality of the final circuit, but it is computationally expensive. The computational cost is reduced by using meta-rules that modify the depth, breadth, and neighborhood of the search depending upon the current rule in the search sequence.

A major obstacle for rule-based expert systems is the acquisition of the knowledge base. In the SOCRATES system, knowledge acquisition is semi-automated. An

expert user first generates the target and replacement configurations specifying a rule and then the system verifies that the two configurations are equivalent and extracts the characteristics that are used to calculate the incremental cost when the rule is applied.

2.2.2 Library-based Technology Mapping

An important advance in technology mapping was the formalization introduced by Keutzer in DAGON [Keut87] and used in misII [Detj87]. In this formalization the set of available circuit elements is represented as a library of functions and the construction of the optimized circuit is divided into three sub-problems: decomposition, matching and covering.

In DAGON, the original network is first decomposed into a canonical representation that uses limited-fanin NAND nodes. This decomposition guarantees that there will be no nodes in the network that are too large to be implemented by any library element, provided the library includes NAND gates that reach the fanin limit. Note, however, that there can be many possible NAND decompositions and that the one selected may not be the best decomposition.

After decomposition, the network is partitioned into a forest of trees. The optimal sub-circuit covering each tree is constructed, and finally the circuit covering the entire network is assembled from these sub-circuits. To form the forest of trees, the decomposed network is partitioned at fanout nodes into a set of single-output sub-networks. Each of these sub-networks is either a tree or a leaf-DAG. A leaf-DAG is a multi-input single-output DAG where only the input nodes have fanout greater than one. Each leaf-DAG is converted into a tree by creating a unique instance of every input node for each of its multiple fanout edges.

The optimal circuit implementing each tree is constructed using a dynamic programming traversal that proceeds from the leaf nodes to the root node. For every node in the tree an optimal circuit implementing the sub-tree extending from the node to the leaf nodes is constructed. This circuit consists of a library element that matches a sub-function rooted at the node and previously constructed circuits imple-

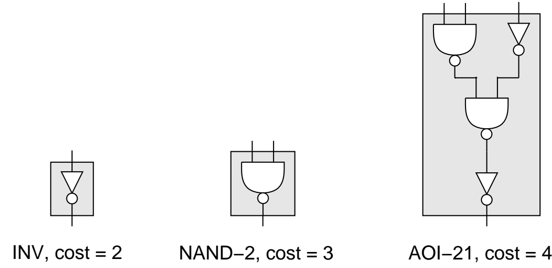
menting its inputs. The cost of the circuit is calculated from the cost of the matched library element and the cost of the circuits implementing its inputs. To find the lowest cost circuit, DAGON first finds all library elements that match sub-functions rooted at the node. The cost of the circuit using each of these candidate library elements is then calculated and the lowest cost circuit is retained. The set of candidate library elements is found by searching through the library and using tree matching [Aho85] to determine if each library element matches a sub-function rooted at the node.

As an example of the above procedure, consider the library shown in Figure 2.4a and the circuit shown in Figure 2.4b. The circuit elements are standard cells and their costs are given in terms of the area of the cells. The cost of the INV, NAND-2 and AOI-21 cells are 2, 3, and 4, respectively. In Figure 2.4b. the only library element matching at node E is the NAND-2 and the cost of the optimal circuit implementing node E is therefore 3. At node C the only matching library element is also the NAND-2. The cost of the NAND-2 is 3 and the cost of the optimal circuits implementing its input E is also 3. Therefore, the cumulative cost of the optimal circuit implementing node C is 6.

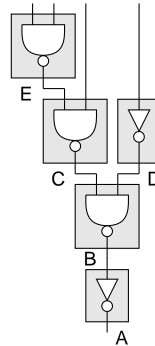
Eventually, the algorithm will reach node A. For node A there are two matching library elements, the INV as used in Figure 2.4b and the AOI-21 as used in Figure 2.4c. The circuit constructed using the INV matching A includes a NAND-2 implementing node B, a NAND-2 implementing node C, an INV implementing node D, and a NAND-2 implementing node E. The cumulative cost of this circuit is 13. The circuit constructed using the AOI-21 matching A includes a NAND-2 implementing node E. The cumulative cost of this circuit is 7. The circuit using the AOI-21 is therefore the optimal circuit implementing node A.

The tree matching algorithm represents each library function using limited-fanin NAND nodes. For some functions, however, there are many possible decompositions into limited-fanin NAND ndoes. The inclusion of all decompositions can significantly increase the size of the library and the computational cost of the matching algorithm.

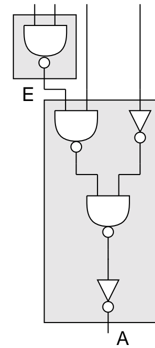
General graph matching was considered as an alternative to tree matching in misII [Detj87], and Ceres [Mail90] used Boolean matching by recursive Shannon decompo-



a) Library



b) INV at node A, cost = 13



c) AOI-21 at node A, cost = 7

Figure 2.4: Dynamic Programming, from [Keut87]

sition. These approaches produced small improvements in the final circuits, but were computationally more expensive than tree matching.

2.2.3 Cell Generator Technology Mapping

In ASIC implementation technologies that use cell generators to create circuit elements, the set of available circuit elements consists of a parameterized family of cells rather than a specific library of functions. This cell family contains all members of a

class of functions, such as AndOrInverts (AOIs), that do not exceed parameters defining the family. Library-based technology mapping is inappropriate for cell generator technologies when the number of cells in the family is too large to be practically expressed in a library. Examples of technology mapping that deals specifically with cell generators are the approaches of Berkelaar and Jess [Berk88] and Liem and Lefebvre [Liem91].

The key to cell generator technology mapping is the completeness of the cell family. This simplifies the matching of network sub-functions to circuit elements. If a sub-function does not exceed the parameters defining the family, it can be implemented by a cell in the family. In addition, simplified matching makes it possible to improve the final circuit by combining decomposition and matching.

Berkelaar addresses technology mapping for a cell generator that creates NMOS or CMOS AndOrInvert gates. The set of available circuit elements includes all AOI gates that meet limits on the maximum number of transistors in series and in parallel. The network is first partitioned into a forest of trees and a circuit implementing each tree is then constructed by traversing the tree proceeding from the root node to the leaf nodes. The decomposition of each AND or OR node in the tree is determined by the parameters defining the cell family. When the in-degree of the node exceeds the limits of the cell family, the node is decomposed into a tree of nodes that match the largest available cell. When the in-degree of the node does not exceed these limits, the node is implemented by a single cell. If this cell is not the largest cell in the family, then the remaining unused *capacity* is passed on to the fanin nodes. In this case, the cell also implements part of the functions of the fanin nodes.

The goal of the ROSSINI technology mapper [Liem91] is the delay minimization of circuits implemented using a cell generator for complex CMOS gates. The family of cells is defined by limits on the number of series n-transistors and series p-transistors as well as the total number of inputs to the cell. The original network is first partitioned into a forest of trees and each tree is decomposed into a minimum-depth binary tree. The circuit implementing each tree is then constructed using a dynamic programming approach similar to the DAGON approach. At each node, the set of

matching circuit elements is constructed using a recursive traversal that is pruned by the limits defining the cell family.

2.3 Lookup Table Technology Mapping

A major obstacle in applying conventional technology mapping approaches to LUT circuits is the large number of different functions that a LUT can implement. A K-input LUT can implement 2^{2^K} different Boolean functions. Rule-based systems lack a systematic method of developing a set of rules that encapsulates the complete functionality of a LUT. For library-based systems, the library representing a K-input LUT need not include all 2^{2^K} different functions. Input permutations, input inversions and output inversions can be used to reduce the number of functions in the library. For example, there are 256 different 3-input functions, but considering input permutations there are 80 different functions and considering input permutations input inversions and output inversions there are 14 different functions. However, the matching algorithms used in library-based technology mappers require the expansion of the library to include all possible decompositions of each function. For values of K greater than 3 the size of a library required to represent a K-input LUT becomes impractically large.

Cell generator technology mapping avoids the problems of large libraries by using matching algorithms that simply test network sub-functions against the parameters defining the cell family. The number of sub-functions that must be considered is reduced by using the network itself to direct the search. However, the cell families used by these approaches do not completely encompass the functionality of a K-input LUT.

The limitations of previous technology mapping approaches provide the motivation for technology mapping that deals specifically with LUT circuits. The first LUT technology mappers were Chortle-x [Fran90] and Mis-pga [Murg90]. Improvements to Chortle-x were incorporated in the program Chortle-crf [Fran91a], and the program Chortle-d [Fran91b] was the first technology mapper to optimize the delay perfor-

mance of LUT circuits by minimizing the number of levels of LUT in the final circuit. The next two chapters of this dissertation present the details of the area and delay algorithms used in Chortle-crf and Chortle-d.

Subsequent to Chortle and Mis-pga, several other LUT technology mappers have been reported [Abou90], [Filo91], [Karp91], [Woo91], [Cong92]. The remainder of this chapter briefly describes these LUT technology mappers and Mis-pga. A common feature of these programs is the ability to map Boolean networks into circuits of Xilinx 3000 series Configurable Logic Blocks (CLBs). The Xilinx 3000 series FPGAs are examples of commercial LUT-based FPGAs. The following section briefly describes the Xilinx 3000 series CLB.

2.3.1 Xilinx 3000 CLBs

An important motivation for LUT technology mapping has been the commercial success of the Xilinx 3000 series FPGAs. The Configurable Logic Blocks in these devices use LUTs to implement combinational logic. Each CLB can implement a single function of up to 5 inputs or two separate functions of up to 4 inputs that together have at most 5 distinct inputs.

A network can be mapped into a circuit of CLBs in two steps. The first step uses LUT technology mapping to map the original Boolean network into functions using at most 5 inputs. The second step then assigns these functions to CLBs. Each CLB in the final circuit will either implement a single 5-input function or two 4-input functions that together have at most 5 distinct inputs. The programs described in the following sections use various methods to maximize the number of CLBs implementing two functions and thereby minimize the total number of CLBs in the final circuit.

2.3.2 Mis-pga

The Mis-pga technology [Murg90] mapper minimizes the number of K-input LUTs required to implement a Boolean network in two phases. The first phase decomposes the original network to ensure that every node can be implemented by a single K-

input LUT and the second phase solves a covering problem to reduce the number of LUTs in the final circuit. Nodes that can be implemented by a single LUT are referred to as *feasible* nodes and a network consisting entirely of feasible nodes is referred to as a feasible network.

In the first phase, two approaches are used to decompose infeasible nodes in the original network into feasible nodes. The first approach is based on Roth-Karp decomposition [Roth62] and the second approach is based on kernel extraction. The first approach searches for a disjoint decomposition using the necessary and sufficient conditions provided by Roth and Karp. To avoid the computational expense of the complete search required to find the best decomposition, Mis-pga accepts the first bound set that meets the Roth-Karp conditions.

The second approach decomposes an infeasible node by extracting its kernels [Bray82] and calculating a cost for each kernel. The lowest cost kernel determines the decomposition of the infeasible node. If a kernel and its residue are both feasible then its cost is the number of variables shared by the kernel and its residue. This provides an estimate of the number of new edges added to the network by the decomposition using this kernel. The number of edges in the network is used as a measure of the routing complexity of the final circuit. If either the kernel or its residue are infeasible, then they are recursively decomposed using kernel extraction. If kernels cannot be extracted, then a decomposition into 2-input ANDs and ORs is used.

The kernel-based decomposition also includes a greedy heuristic to collapse nodes into their fanouts. All single-fanout nodes are collapsed if the resulting node is feasible. For multiple fanout nodes each fanout is considered in turn and the node is collapsed if the resulting node is feasible. The order in which the fanouts are considered is determined by an estimate of the number of edges that will be added to the network.

The second phase of Mis-pga minimizes the number of nodes in the feasible network produced by the first phase. A cluster of nodes, referred to as a *supernode*, that has at most K inputs can be collapsed into a single feasible node. Reducing the number of supernodes required to cover the network is expressed as a binate covering

problem [Rude89]. For small networks an exact solution to this covering problem is used, however, for larger networks the computational cost of the exact solution is prohibitive and a heuristic solution is used.

Mis-pga addresses technology mapping for Xilinx 3000 CLBs by first mapping the Boolean network into 5-input functions and then assigning these functions to CLBs. Each CLB can implement one 5-input function or two functions of up to 4 inputs that together have no more than 5 inputs. Maximizing the number of CLBs that implement two functions, and thereby minimizing the total number of CLBs, is restated and solved as a Maximum Cardinality Matching problem.

A newer version of Mis-pga [Murg91a] includes two additional approaches to decomposition. The first adapts a bin packing approach introduced in Chortle-crf [Fran91b], and described in this dissertation, and the second is based on Shannon cofactoring. The first decomposition approach decomposes an infeasible function into a tree of feasible nodes. The cubes of the function are treated as items and the nodes in the tree are treated as bins. The size of each item is the number of variables in the cube and the capacity of every bin is K. Minimizing the number of nodes in the tree is expressed as a bin packing problem and solved using the Best Fit Decreasing heuristic.

The second decomposition approach uses Shannon cofactoring to decompose infeasible functions. An infeasible function $f(x_1, ..., x_n)$ is decomposed into three functions: f_{x_1}, $f_{\overline{x_1}}$ and $f = x_1 f_{x_1} + \overline{x_1} f_{\overline{x_1}}$. The function f now depends on three variables x_1, f_{x_1}, and $f_{\overline{x_1}}$ and is therefore feasible for values of K greater than or equal to 3. The functions f_{x_1} and $f_{\overline{x_1}}$ each depend on at most $n-1$ variables and if $n-1$ equals K then they are feasible. If $n-1$ is greater than K then the functions f_{x_1} and $f_{\overline{x_1}}$ are recursively decomposed.

The new version of Mis-pga also includes optimizations that improve performance by reducing the number of levels of LUTs in the final circuit [Murg91b]. The original network is first decomposed into a depth-reduced network of 2-input nodes [Sing88] and then the critical paths are traversed from the primary inputs to the primary outputs. A critical node at depth d is collapsed into its fanout nodes, at depth $d+1$,

whenever the resulting node is feasible, or can be redecomposed with a reduction in depth.

2.3.3 Asyl

The Asyl logic synthesis system incorporates technology mapping for Xilinx 3000 CLBs [Abou90]. The technology mapping phase of Asyl depends upon a reference ordering of the primary input variables that is determined by the logic optimization phase. The Boolean network produced by the logic optimization phase is a lexicographical factorization. If this network is collapsed into a sum-of-products expression, then the order of variables within the product terms defines the reference ordering. The technology mapping phase of Asyl consists of two steps. The first step uses the reference ordering to decompose the Boolean network into 4 and 5-input functions and the second step assigns these functions to CLBs.

The first step considers slices of 4 variables within the reference ordering, beginning with the last variable and proceeding slice by slice toward the first variable. Within each slice, cut points are introduced to produce sub-functions of 4 or 5 inputs in the following order: first, sub-functions of 4 variables from the current slice, next, sub-functions of 4 variables where three variables are from the current slice and one variable is from a preceding slice, and finally sub-functions of 5 variables beginning with those having the maximum number of variables from the current slice.

The second step in the Asyl technology mapping phase assigns the functions produced by the first step to CLBs. First, each 5-input function is assigned to a single CLB. Next, a greedy heuristic is used to maximize the number of CLBs implementing two 4-input functions and thereby reduce the total number of CLBs. This heuristic sorts the 4-input functions into a list and iteratively assigns pairs of functions to CLBs. Two functions can be paired if together they have at most 5 distinct inputs. The list is sorted by the number of potential partners each function has. The first function in this list, which has the least number of partners, is assigned to a CLB along with its first available partner in the remainder of the list. If a partner cannot be found then the function is assigned to a CLB without a partner.

2.3.4 Hydra

The Hydra technology mapper [Filo91] addresses two-output RAM-based logic blocks such as the Xilinx 3000 series CLBs. The two-phase strategy employed by Hydra to minimize the number of CLBs in the final circuit emphasizes the use of both CLB outputs. The first phase decomposes nodes in the original network to ensure that every node can be implemented by a single CLB and the second phase then finds pairs of nodes that can be implemented by two-output CLBs.

The principal decomposition technique used in the first phase searches for disjoint decompositions that increase the number of functions that can be paired into single CLBs by the second phase. The first phase begins with an AND-OR decomposition that limits the in-degree of nodes in the network and thereby reduces the computational cost of the search for disjoint decompositions. After the disjoint decompositions are found another AND-OR decomposition limits the in-degree of every node to 5, thereby ensuring that every node can be implemented by a single CLB.

The search for simple disjoint decompositions considers pairs of nodes beginning with the pair having the greatest number of shared inputs and proceeding to the pair with the least number of shared inputs. For each pair, an exhaustive search is used to find disjoint decompositions. Once a disjoint decomposition is found, the decision to accept it is based on the number of shared inputs and the total number of inputs in the extracted functions.

The AND-OR decomposition is used to ensure that the in-degree of every node is less than or equal to a specified limit. If the in-degree of a node exceeds this limit, then it is factored into a tree of AND and OR nodes. This tree is traversed from the leaves to the root and at each node if the combined support of the predecessor nodes exceeds the limit, then a group of predecessor nodes with combined support less than or equal to the limit is replaced by a new node. The group is selected by a heuristic that considers the size of the support for the group, the maximum support of any node in the group, and the number of nodes in the group.

A local optimization at the end of the first phase reduces the number of nodes by collapsing nodes into their successors if the resulting node can still be implemented

by a single CLB. For nodes with only one successor, the elimination is allowed if the resulting node has in-degree less than or equal to 5. For nodes with more than one successor, the elimination is allowed if the resulting node has in-degree less than or equal to 4. This preserves opportunities for the second phase to pair functions into two-output CLBs.

The second phase of Hydra uses a greedy heuristic to find pairs of functions that can be implemented by two-output CLBs. Two functions can be paired if they each have no more than 4-inputs and if together they have at most 5-inputs. The benefit of pairing two functions is calculated as $\alpha N_{shared} + \beta N_{total}$, where N_{shared} is the number of shared inputs and N_{total} is the total number of inputs. The values of α and β are tuned to control the tradeoff between input sharing and input utilization. The heuristic iteratively assigns pairs of functions to CLBs. Each iteration begins by assigning the function with the greatest in-degree to the first output of a CLB. If this function can be paired with any of the remaining functions then the partner resulting in the greatest benefit is assigned to the second output of the CLB.

2.3.5 Xmap

The Xmap technology mapper [Karp91] uses two passes to minimize the number of K-input LUTs required to implement a Boolean network and a third pass to produce a circuit of Xilinx 3000 CLBs. The first pass decomposes nodes in the network to ensure that all nodes have in-degree less than or equal to K, and the second pass marks nodes to be implemented by single-output LUTs. The third pass assigns 5-input functions produced by the first two passes to CLBs.

The first pass decomposes the original network into an *if-then-else* DAG. Each node in the if-then-else DAG is a 2 to 1 selector. For example, the node with inputs a, b, and c implements the Boolean function (*if* a *then* b *else* c). Every sum-of-products node in the original network is decomposed into the following if-then-else expression:

$$\textit{if}\,(E_d)\ \textit{then}\ (\text{TRUE})\ \textit{else}\ (\textit{if}\,(v)\ \textit{then}\ (E_1)\ \textit{else}\ (E_0))$$

The variable v is the variable that appears most often in the product terms and the expressions E_1 and E_0 consist of the product terms that contain v and $\overline{v}$ respectively. The expression E_d consists of the product terms that use neither v nor $\overline{v}$. If the expressions E_1, E_0 and E_d depend upon more than one variable, then they are recursively decomposed using the same procedure.

For values of K greater than 2, every node in the if-then-else DAG can be implemented by a single K-input LUT. When K is equal to 2, every 3-input node ($z =$ *if* a *then* b *else* c) is replaced with the three 2-input nodes $x = ab$, $y = \overline{a}c$, and $z = x + y$.

The second pass in Xmap marks nodes in the decomposed network that are implemented as outputs of K-input LUTs in the final circuit. Initially, only the primary outputs are marked. The network is then traversed from the primary inputs to the primary outputs. At each node the set of previously marked nodes and primary inputs required to compute the node is referred to as the node's *signal set*. If necessary, additional nodes are marked to ensure that the size of the signal set is less than or equal to K. These newly marked nodes reduce the size of the signal set by hiding previously marked nodes. When the size of the signal set for the current node is greater than K, the first additional nodes that are marked are preceding nodes with high fanout. If further reduction of the signal set is required, then the support nodes of the current node are marked in decreasing order of their own signal set sizes.

The final pass of Xmap minimizes the number of CLBs required to implement the 5-input functions produced by the marking pass by iteratively assigning pairs of functions to two-output CLBs. Each iteration assigns the function with the greatest number of inputs to the first output of a CLB. The second output can be assigned to another function provided that both functions have at most 4 inputs and that together they have at most 5 inputs. From the remaining functions that satisfy these conditions, the function with the greatest number of inputs is assigned to the second output of the CLB.

2.3.6 VISMAP

The VISMAP technology mapper [Woo91] focuses on the covering problem in LUT technology mapping. It assumes that the original network has been previously decomposed to ensure that every node has in-degree less than or equal to K. Therefore, every node can be directly implemented by a K-input LUT.

Reducing the number of LUTs required to cover the network is addressed by labelling every edge in the network as either *visible* or *invisible*. A visible edge is implemented by a wire in the final circuit. Its source is the output of one LUT, and its destination is the input of a different LUT. For an invisible edge both the source and destination node are implemented in the same LUT. In this case, the network can be simplified by merging the source node into the destination node. If the resulting node has in-degree no greater than K, then it can still be implemented by a single K-input LUT.

The assignment of visibility labels to edges is performed by first dividing the original network into a collection of subgraphs that each contain at most m edges. Within each subgraph, every edge can be labelled as either visible or invisible. The optimal assignment is found by exhaustively searching all possible combinations of edge labels. For a subgraph containing m edges there are 2^m different combinations. For each combination, a simplified subgraph is formed by merging the source and destination nodes of invisible edges. When the simplification of an invisible edge results in a node with in-degree greater than K, the combination is rejected because the node cannot be implemented by a K-input LUT. Otherwise, the combination resulting in the simplified subgraph having the fewest nodes is retained as the optimal label assignment for the subgraph.

The computational cost of the search is controlled by the limit on the number of edges in each subgraph. In addition, the order in which the combinations are considered allows the search to be pruned by skipping over some combinations whenever the simplification of an invisible edge results in a node with in-degree greater than K.

VISMAP can map a Boolean network into a circuit of two-output LUT-based

logic blocks by first mapping the network into K-input functions and then iteratively assigning pairs of these functions to logic blocks. It is assumed that each logic block can implement two functions that each have at most 5 inputs. Each iteration begins by assigning the function with the least number of potential partners to a logic block. The function assigned to the second output of the logic block, from the potential partners of the first function, is the function that has the fewest potential partners.

2.3.7 DAG-Map

The DAG-Map [Cong92] technology mapper addresses the delay optimization of LUT circuits. The delay of each LUT is modeled as a unit delay, and DAG-Map minimizes the delay of the circuit by minimizing the number of levels of LUTs in the final circuit. DAG-Map also addresses the minimization of the total number of LUTs as a secondary objective. The original network, which is a graph of AND, OR and INVERT nodes, is mapped into a circuit of K-input LUTs in four phases: decomposition, labelling, covering, and area optimization.

The first phase decomposes nodes in the original network into trees of 2-input nodes. This phase proceeds from the primary inputs to the primary outputs, and at each node, given the depth of its immediate fanin nodes, decomposes the current node into a minimum-depth tree of 2-input nodes.

The second phase labels nodes in the 2-input network to determine the level of the LUT implementing each node. The label at primary inputs is set to 1, and the label for any other node is calculated from the labels of its immediate fanin nodes. If p is the maximum label at any of the fanin nodes, then the label at the current node will be either p or $p + 1$. If the total number of distinct inputs to the current node and the set of preceding nodes with label p is less than or equal to K, then the label of the current node is also set to p. Otherwise, the label is $p + 1$.

The third phase covers the labelled network with K-input LUTs. Initially, a LUT is created to implement each of the primary outputs. This LUT implements all preceding nodes that have the same label as the LUT's output node. Additional LUTs are then created to implement the inputs to each LUT. This process continues until the

primary inputs have been reached. Note that this covering phase will replicate logic at fanout nodes if necessary to achieve the labeled depth. The labelling and covering algorithms produce an optimal depth circuit provided the network is a fanout-free tree, and that the maximum fanin at any node in the network is K.

The final step attempts to reduce the total number of LUTs in the circuit, without increasing the number of levels, by using two local optimizations. The first optimization searches for a pair of LUTs that generate inputs for a third LUT that implements an associative function such as AND and OR. The associative function can be decomposed to produce an intermediate node implementing the same associative operation for these two inputs. If these two LUTs together have at most K distinct inputs, then they can be combined into one LUT that also implements the intermediate node. The one output of this merged LUT replaces the two inputs to the third LUT, without increasing the number of levels in the circuit. The second optimization searches for a LUT that can be combined with a LUT that uses its output as an input. If the two LUTs together have at most K distinct inputs, then the LUTs can be combined into a single LUT without an increase in the number of levels.

2.4 Summary

This chapter has presented a brief introduction to logic synthesis, and discussed the motivation for technology mapping algorithms that deal specifically with LUT cirucits. The following chapters present the details of the area and delay algorithms used in Chortle.

Chapter 3

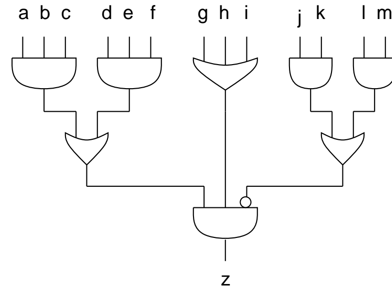
The Area Algorithm

A circuit can be implemented by a given FPGA only if the number of logic blocks in the circuit does not exceed the available number of logic blocks and the required connections between the logic blocks do not exceed the capacity of the routing network. This chapter describes a technology mapping algorithm, referred to as the *area algorithm*, that minimizes the total number of K-input LUTs in the circuit implementing a given network. Minimizing the number of LUTs in the circuit allows larger networks to be implemented by the fixed number of logic blocks available in a given LUT-based FPGA.

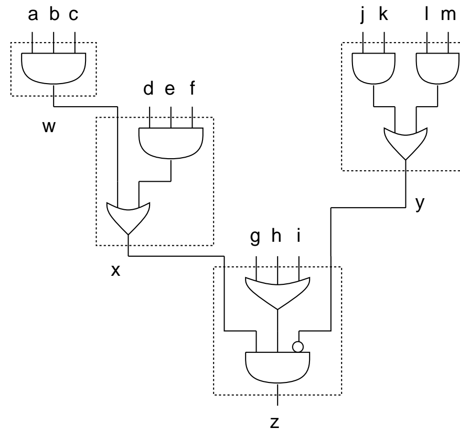
The algorithm takes the original network and produces a circuit of K-input LUTs implementing the network. The nodes in the original network represent AND or OR functions and inversion is represented by labeling edges. For example, in Figure 3.1a, nodes a to m are the primary inputs of the network, and node z is the primary output. In this figure, inverted edges are represented by a circle at the destination of the edge. The function specified for the primary output z is

$$z = (abc + def)(g + h + i)\overline{(jk + lm)}$$

Figure 3.1b illustrates a circuit of 5-input LUTs implementing the network shown in Figure 3.1a. The dotted boundaries indicate the functions implemented by each LUT, and each LUT is referred to by the name of the node it implements. LUT y implements the local function $y = jk + lm$ and LUT z implements the local function



a) Boolean network



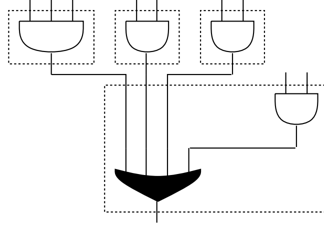
b) Circuit of 5-input LUTs

Figure 3.1: Mapping a Network

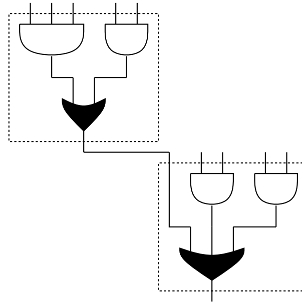
$z = x(g + h + i)\overline{y}$. Note that the LUT y uses only 4 of the 5 available inputs. All examples in the remainder of this chapter will assume that the value of K is equal to 5.

The overall strategy employed by the area algorithm is similar to the library-based approach introduced by DAGON [Keut87]. The original network is first partitioned into a forest of trees and then each tree is separately mapped into a circuit of K-input LUTs. The final circuit is then assembled from the circuits implementing the trees.

The major innovation of the area algorithm is that it simultaneously addresses the decomposition and matching problems using a bin-packing approximation algorithm. The correct decomposition of network nodes can reduce the number of LUTs required to implement the network. For example, consider the circuit of 5-input LUTs shown in Figure 3.2a. The shaded OR node is not decomposed and four 5-input LUTs are required to implement the network. However, if the OR node is decomposed into the



a) Without decomposition, 4 LUTs



b) With decomposition, 2 LUTs

Figure 3.2: Decomposition of a Node

two nodes shown in Figure 3.2b, then only two LUTs are required. The challenge is to find the decomposition of every node in the network that minimizes the number of LUTs in the final circuit.

The original network is partitioned into a forest of trees by dividing it at fanout nodes. The resulting sub-networks are either trees or *leaf-DAGs*. A leaf-DAG is a multi-input single-output DAG where the only nodes with fanout greater than one are the input nodes [Deva91]. The leaf-DAGs are converted into trees by creating a unique instance of every input node for each of its fanout edges.

The following section describes how dynamic programming and bin packing are used to construct the circuit of K-input LUTs implementing each tree. Later sections will consider local optimizations at fanout nodes that further reduce the number of LUTs in the circuit by exploiting reconvergent paths and the replication of logic at fanout nodes.

3.1 Mapping Each Tree

After the original network has been partitioned into a forest of trees, each tree is separately mapped into a circuit of K-input LUTs. Before each tree is mapped a preprocessor applies DeMorgan's Theorem and AND-OR associativity rules to ensure that the only inverted edges in the tree originate from leaf nodes and that there are no consecutive AND nodes and no consecutive OR nodes in the tree. The presence of consecutive AND or consecutive OR nodes would restrict the decompositions that the area algorithm could use. Provided the above conditions are satisfied, Chapter 5 proves that the area algorithm constructs an optimal tree of K-input LUTs implementing the tree for values of K less than or equal to 5.

After preprocessing, each tree is mapped using the dynamic programming approach outlined as pseudo-code in Figure 3.3. The tree is traversed in a postorder depth-first fashion and at each node a circuit of LUTs implementing the sub-tree extending to the leaf nodes is constructed. For leaf nodes, this circuit is simply a single LUT implementing a buffer function. At non-leaf nodes, the circuit is constructed from the circuits implementing the node's immediate fanin nodes. The order of the traversal ensures that these fanin circuits have been previously constructed.

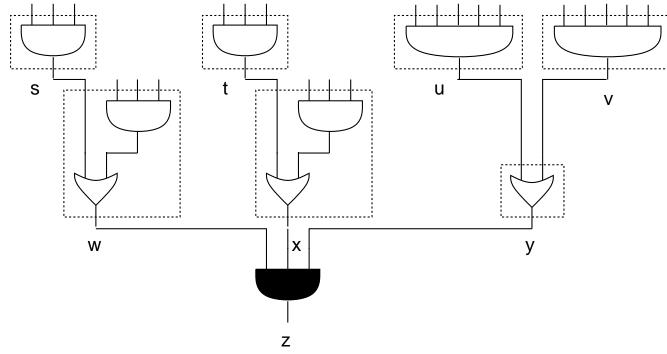
The circuit implementing a non-leaf node consists of two parts. The first part, referred to as the *decomposition tree*, is a tree of LUTs that implements the functions of the root LUTs of the fanin circuits and a decomposition of the non-leaf node. The second part is the non-root LUTs of the fanin circuits. For example, Figure 3.4a illustrates the circuits implementing the three fanin nodes of node z. The root LUTs of the fanin circuits are referred to as the *fanin LUTs*. In this example, the LUTs w, x, and y are the fanin LUTs and the LUTs s, t, u, and v, are the non-root LUTs of the fanin circuits. Figure 3.4b illustrates the circuit implementing node z that is constructed from the fanin circuits. It includes the non-root LUTs s, t, u, and v, and the decomposition tree consisting of LUTs w, $z.1$, and z. Note that the node z has been decomposed and that the node $z.1$ has been introduced.

The essence of the dynamic programming approach is to construct the optimal

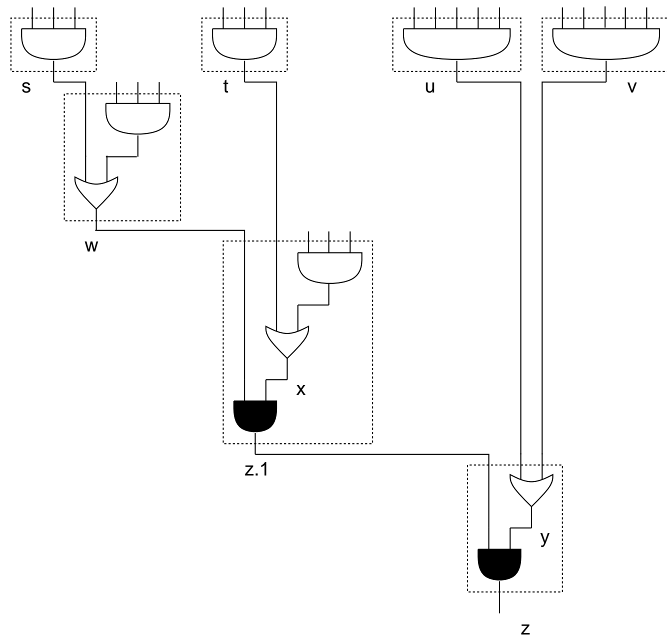
```
MapTree (tree)
/* construct circuit implementing tree */
    {
    traverse tree from leaves to root, at each node
        {
        /* construct circuit implementing node */
        if node is a leaf
            circuit ⇐ single LUT buffering node
        else
            circuit ⇐ MapNode (node)
        }

    /* return circuit implementing root node */
    return (circuit)
    }


MapNode (node)
/* construct circuit implementing sub-tree rooted at node */
    {
    /* separate fanin LUTs */
    faninLUTs ⇐ root LUTs of circuits for all fanin nodes
    precedingLUTs ⇐ non-root LUTs of circuits for all fanin nodes

    /* construct decomposition tree */
    decompositionTree ⇐ DecomposeArea (node, faninLUTs)

    /* join decomposition tree and preceding LUTs */
    circuit ⇐ decompositionTree ∪ precedingLUTs

    return (circuit)
    }
```

Figure 3.3: Pseudo-code for Mapping a Tree

circuit implementing each non-leaf node using the optimal circuit implementing its fanin nodes. The key to the area algorithm is the definition of the optimal circuit. The principal optimization goal is to minimize the number of LUTs in the circuit, and the secondary optimization goal is to minimize the number of inputs the circuit's root LUT uses. This secondary optimization goal is the key to ensuring that the op-



a) Fanin circuits

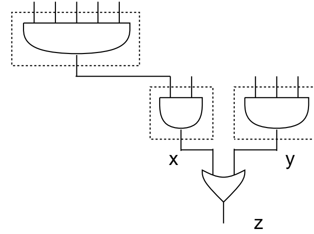


b) Circuit implementing node z

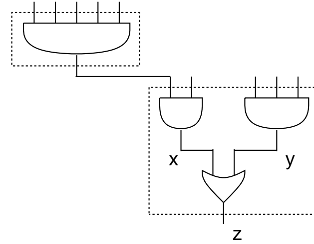
Figure 3.4: Mapping a Node

timal circuit implementing the non-leaf node is constructed from the optimal circuits implementing its fanin nodes. The following example illustrates the importance of the secondary optimization goal.

The number of LUTs in the circuit implementing the non-leaf node is the sum of the number of LUTs in the decomposition tree and the number of non-root LUTs in the fanin circuits. Given that the fanin circuits each contain the minimum number of LUTs, minimizing the number of LUTs in the decomposition tree minimizes the number of LUTs in the circuit implementing the non-leaf node. The secondary optimization goal is the key to minimizing the number of LUTs in the decomposition tree.



a) Fanin circuits



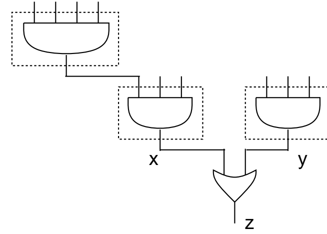
b) Circuit implementing node z

Figure 3.5: Fanin Circuit Satisfying Both Optimization Goals

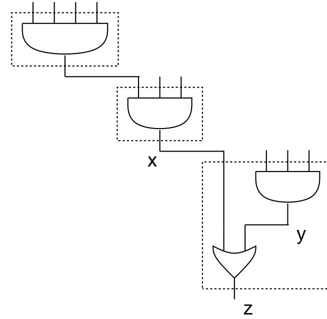
Consider the fanin circuits shown in Figure 3.5a. These fanin circuits satisfy both optimization goals. The fanin circuit at node x contains 2 LUTs and its root LUT uses 2 inputs. The fanin circuit at node y consists of one LUT using 3 inputs. Figure 3.5b shows the best circuit implementing node z that can be constructed using these fanin circuits. The circuit contains two LUTs and the decomposition tree consists of the one LUT z.

Figure 3.6a shows an alternative fanin circuit at node x. This fanin circuit also contains 2 LUTs, but its root LUT now uses 3 inputs. The best circuit implementing node z that can be constructed using this fanin circuit, shown in Figure 3.6b, contains 3 LUTs. The decomposition tree in this circuit consists of the LUTs z and x.

The success of the dynamic programming approach requires that the circuits constructed at every node satisfy both of the optimization goals. Given that the fanin circuits satisfy both goals, the circuit constructed at the non-leaf node will satisfy both optimization goals provided that the decomposition tree contains the minimum number of LUTs and that its root LUT uses as few inputs as possible. The following section describes how the decomposition tree is constructed.



a) Fanin circuit



b) Circuit implementing node z

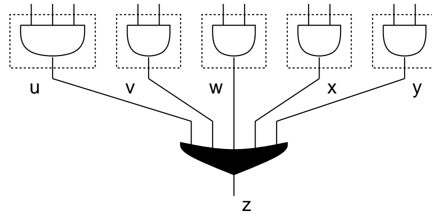
Figure 3.6: Fanin Circuit Satisfying Only Primary Optimization Goals

3.1.1 Constructing the Decomposition Tree

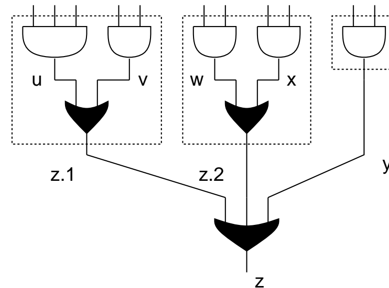
At each node, the decomposition tree implementing the fanin LUTs and a decomposition of the node is constructed in two steps. The first step packs the fanin LUTs into what are called *second-level* LUTs. The second step connects these LUTs to form the complete decomposition tree.

Consider the node z and its fanin LUTs shown in Figure 3.7a. Note that in this example each fanin LUT implements a single AND gate; however, in general the fanin LUTs can implement more complicated functions. Figure 3.7b shows the second-level LUTs constructed by the first step and Figure 3.7c shows the complete decomposition tree. The second-level LUTs specify a two-level decomposition of the node z. Each second-level LUT implements some subset of the fanin LUTs and the corresponding decomposition of the node z. In Figure 3.7b the LUT $z.1$ implements the functions of the fanin LUTs u and v. In Figure 3.7c the output of LUT $z.1$ has been connected to an input of LUT $z.2$ and the output of LUT $z.2$ has been connected to an input of LUT z to form the complete decomposition tree.

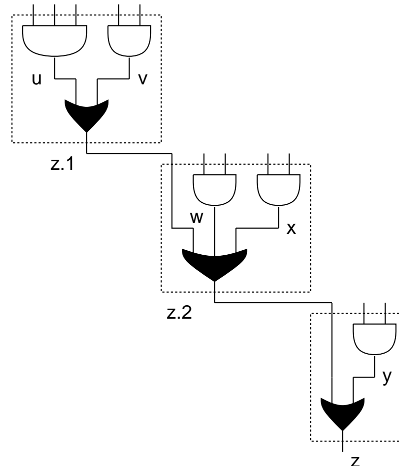
For a given set of fanin LUTs, the optimal decomposition tree contains the mini-



a) Fanin LUTs



b) Two-level decomposition



c) Multi-level decomposition

Figure 3.7: Constructing the Decomposition Tree

mum number of LUTs, and its root LUT uses the minimum number of inputs. The key to the construction of the optimal decomposition tree is to find the two-level decomposition that contains the minimum number of second-level LUTs. The major innovation of the area algorithm is to restate this as a bin-packing problem. This approach is based on the observation that the function of a fanin LUT cannot be split across more than one second-level LUT.

In general, the goal of bin packing is to find the minimum number of subsets into which a set of items can be partitioned such that the sum of the sizes of the items

```
FirstFitDecreasing (node, faninLUTs)
/* construct two level decomposition */
    {
    boxList ⇐ faninLUTs sorted by decreasing size
    binList ⇐ ∅

    while (boxList is not ∅)
        {
        boxLUT ⇐ largest lookup table from boxList

        find first binLUT in binList such that size (binLUT) + size (boxLUT) ≤ K

        if such a binLUT does not exist
            {
            binLUT ⇐ a new lookup table
            add binLUT to end of binList
            }

        pack boxLUT into binLUT,
        /* implies decomposition of node */
        }

    return (binList)
    }
```

Figure 3.8: Pseudo-code for Two-Level Decomposition

in every subset is less than or equal to a constant C. Each subset can be viewed as a set of boxes packed into a bin of capacity C. In the construction of the two-level decomposition, the boxes are the fanin LUTs, and the bins are the second-level LUTs. The size of each box is its number of used inputs and the capacity of each bin is K. For example, in Figure 3.7a the boxes have sizes 3, 2, 2, 2, and 2. In Figure 3.7b the final packed bins have filled capacities of 5, 4, and 2.

Bin packing is known to be an NP-hard problem [Gare79], but there exist several effective approximation algorithms. The procedure used to construct the two-level decomposition, outlined as pseudo-code in Figure 3.8, is based on the First Fit Decreasing algorithm. The fanin LUTs are referred to as boxes and the second-level

LUTs are referred to as bins. The procedure begins with an empty list of bins. The boxes are first sorted by size, and then packed into bins one at a time, beginning with the largest box and proceeding in order to the smallest box. Each box is packed into the first bin in the list having an unused capacity greater than or equal to the size of the box. If no such bin exists, then a new bin is added to the end of the bin list and the box is packed into this new bin. Note that packing more than one box into a bin requires the introduction of a decomposition node. For example, in Figure 3.7b, when boxes u and v are packed into one bin this requires the introduction of the decomposition node $z.1$.

The procedure used to convert the two-level decomposition into the multi-level decomposition is outlined as pseudo-code in Figure 3.9. The second-level LUTs are first sorted by their size. Then, while there is more than one second-level LUT remaining, the output of the LUT with the greatest number of used inputs is connected to the first available unused input in the remaining LUTs. If no unused inputs remain then an extra LUT is added to the decomposition tree. Note that the decomposition node in the destination LUT is altered, and now implements part of the first level node. For example, in Figure 3.7c, when LUT $z.1$ is connected to LUT $z.2$, the decomposition node $z.2$ is altered. This procedure constructs an optimal decomposition tree provided that the two-level decomposition contains the minimum number of LUTs, and that its least filled LUT is as small as possible. Appendix A presents a proof that the First Fit Decreasing bin-packing algorithm constructs a two-level decomposition satisfying both of these requirements when the box and bin sizes are restricted to integers less than or equal to 5.

3.1.2 Optimality

The goal of the area algorithm is to reduce the number of K-input LUTs required to implement the original network. The original network is first partitioned into a forest of trees and each of these is mapped separately into a tree of LUTs. The final circuit implementing the original network is assembled from the circuits implementing the trees. Chapter 5 proves that the circuit constructed for each tree is an optimal tree

```
DecomposeArea (node, faninLUTs)
/* construct tree of LUTs implementing decomposition of node and fanin LUTs */
    {
    /* construct two level decomposition */
    packedLUTs = FirstFitDecreasing (node, faninLUTs)

    lookList ⇐ packedLUTs sorted by decreasing size

    while (lookList contains more than one lookup table)
        {
        sourceLUT ⇐ largest lookup table from lookList

        find first destinationLUT in lookList such that size (destinationLUT) +1 ≤ K

        if such a destinationLUT does not exist
            {
            destinationLUT ⇐ a new lookup table
            add destinationLUT to end of lookList
            }

        connect sourceLUT output to destinationLUT input,
        /* implies decomposition of node */
        }

    return (lookList)
    }
```

Figure 3.9: Pseudo-code for Multi-Level Decomposition

of LUTs implementing that tree, provided that the value of K is less than or equal to 5. For these values of K, the FFD bin-packing algorithm results in the two-level decomposition with the minimum number of LUTs and the smallest possible least filled LUT. This two-level decomposition leads to an optimal decomposition tree, which in turn leads to an optimal circuit implementing each non-leaf node including the root node of the tree being mapped.

If f is the number of fanin edges at a given node in the tree, and bucket sorts are used in the implementation of the above algorithm, then the time taken to construct the decomposition tree is bounded by Kf. The dynamic programming traversal visits

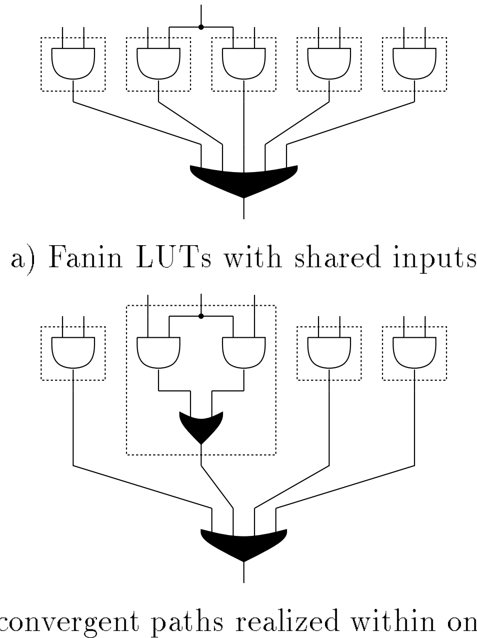
each node in the tree once. If there are n nodes in the tree, and the maximum fanin at any node is F, then the time taken to map the entire tree is bounded by nKF.

Even though the tree of LUTs implementing each tree in the forest is optimal, the final circuit implementing the entire network that is assembled from these circuits is not necessarily optimal. Partitioning the original network into a forest of fanout-free trees precludes LUTs that realize functions containing reconvergent paths and assembling the final circuit from the separate circuits implementing each tree precludes the replication of logic at fanout nodes. The following sections describe local optimizations that exploit reconvergent paths and the replication of logic at fanout nodes to further reduce the number of LUTs in the final circuit.

3.2 Exploiting Reconvergent Fanout

When the original network is partitioned at fanout nodes into single-output sub-networks, the resulting sub-networks are either trees or leaf-DAGs. In a leaf-DAG, a leaf node with out-degree greater than one is the source of reconvergent paths that terminate at some other node in the leaf-DAG. This section describes two alternative optimizations that exploit the reconvergent paths to improve the circuit implementing the terminal node. These optimizations replace the FFD algorithm and improve the two-level decomposition used to construct the decomposition tree. The first optimization uses an exhaustive search that repeatedly invokes the FFD algorithm. The second optimization uses a greedy heuristic that simplifies to the FFD algorithm when there are no reconvergent paths.

Both optimizations exploit reconvergent paths that begin at the inputs to the fanin LUTs and that terminate at the node being mapped. In the following description, the fanin LUTs are again referred to as boxes and the second-level LUTs are referred to as bins. Consider the set of boxes shown in Figure 3.10a. Two of the boxes share the same input and so there exists a pair of reconvergent paths terminating at the shaded OR node. Each of these boxes has two inputs, for a total of four inputs. However, when they are packed into the same bin, as in Figure 3.10b, only three inputs are



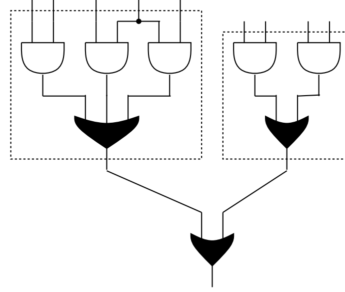
a) Fanin LUTs with shared inputs

b) Reconvergent paths realized within one LUT

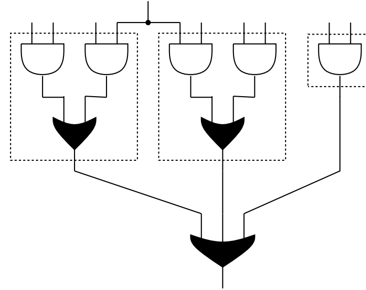
Figure 3.10: Local Reconvergent Paths

needed. The reconvergent paths are realized within the LUT and the total number of inputs used is less than the sum of the sizes of the two boxes. The decrease in the number of bin inputs that are used may allow additional boxes to be packed into the same bin and may therefore improve the final two-level decomposition. Figure 3.11a illustrates the two-level decomposition constructed by applying the FFD bin-packing algorithm after the reconvergent paths have been realized within one LUT. By contrast, Figure 3.11b shows the result if the reconvergent paths are ignored, and the bin-packing algorithm is applied directly to the fanin LUTs. In this case, the two-level decomposition that realizes the reconvergent paths within a LUT contains fewer second-level LUTs.

The reconvergent paths can only be realized within one LUT if the two boxes with the shared input are packed into the same bin. To ensure that the boxes are packed together they can be merged before the FFD bin-packing algorithm constructs the two-level decomposition. However, forcing the two boxes into one bin can interfere with the FFD algorithm and actually produce an inferior two-level decomposition. To find the best two-level decomposition, the bin-packing algorithm is applied both with and without the forced merging of the two boxes and the superior two-level



a) With forced merge, 2 LUTs



b) Without forced merge, 3 LUTs

Figure 3.11: Exploiting Reconvergent Paths

decomposition is retained.

When more than one pair of fanin LUTs share inputs, there are several pairs of reconvergent paths. To determine which pairs of reconvergent paths to realize within LUTs, an exhaustive search is used to find the best two-level decomposition, as outlined as pseudo-code in Figure 3.12. The search begins by finding all pairs of boxes that share inputs. Next, every possible combination of these pairs is considered. For each combination a two-level decomposition is constructed by first merging the respective boxes of the chosen pairs and then proceeding with the FFD bin-packing algorithm. The two-level decomposition with the fewest bins and the smallest least filled bin is retained.

The exhaustive search becomes impractical when there is a large number of pairs of boxes that share inputs. In this case, a heuristic, referred to as the Maximum Share Decreasing (MSD) algorithm, is used to construct the two-level decomposition. This heuristic, outlined as pseudo-code in Figure 3.13, is similar to the FFD algorithm, but it attempts to improve the two-level decomposition by maximizing the sharing of inputs when boxes are packed into bins. The MSD algorithm iteratively packs boxes

```
Reconverge (node, faninLUTs)
/* construct two level decomposition */
/* exploit reconvergent paths */
/* exhuastive search */
    {
    pairList ⇐ all pairs of faninLUTs with shared inputs

    bestLUTs ⇐ ∅

    for all possible chosenPairs from pairList
        {
        mergedLUTs ⇐ copy of faninLUTs with forced merge of chosenPairs

        packedLUTs ⇐ FirstFitDecreasing (node, mergedLUTs)

        if packedLUTs are better than bestLUTs
            bestLUTs ⇐ packedLUTs
        }

    return (bestLUTs)
    }
```

Figure 3.12: Pseudo-code for Exhaustive Reconvergent Search

```
MaxShare (node, faninLUTs)
/* construct two level decomposition */
/* exploit reconvergent paths */
/* greedy heuristic */
    {
    boxList ⇐ faninLUTs
    binList ⇐ ∅

    while (boxList is not ∅)
        {
        boxLUT ⇐ highest priority LUT from boxList
        /* precedence of rules for highest priority boxLUT */
        /* 1) most inputs */
        /* 2) most inputs shared with a bin in binList */
        /* 3) most inputs shared with a box in boxList */

        find binLUT in binList that shares most inputs with boxLook

        if such a binLUT does not exist
            {
            binLUT ⇐ a new LUT
            add binLUT to end of binList
            }

        pack boxLUT into binLUT exploiting shared inputs,
        /* implies decomposition of node */
        }
    return (binList)
    }
```

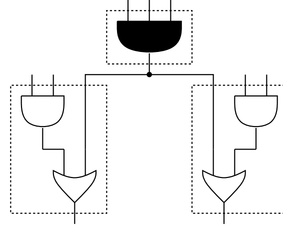
Figure 3.13: Pseudo-code for Maximum Share Decreasing

into bins until all the boxes have been packed. Each iteration begins by choosing the next box to be packed and the bin into which it will be packed. The chosen box satisfies three criteria: first it has the greatest number of inputs, second it shares the greatest number of inputs with any existing bin, and third it shares the greatest number of inputs with any of the remaining boxes. The first criterion ensures that the MSD algorithm simplifies to the FFD algorithm when there are no reconvergent paths. The second and third criteria encourage the sharing of inputs when the box is packed into a bin. The chosen box is packed into the bin with which it shares the most inputs while not exceeding the capacity of the bin. If no such bin exists, then a new bin is created and the chosen box is packed into this new bin. Note that the second and third criteria for choosing the box to be packed only consider combinations of boxes and bins that will not exceed the bin capacity.

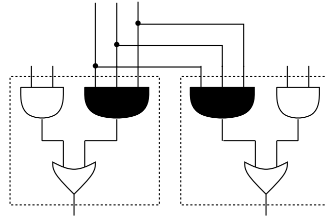
Both reconvergent optimizations only find local reconvergent paths that begin at the inputs of the fanin LUTs. However, when the fanin circuits are constructed, no consideration is given to reconvergent paths that terminate at subsequent nodes. The propagation of these reconvergent paths through the fanin LUTs is therefore dependent upon the network traversal order. This is demonstrated by the experimental results presented in Chapter 7, where some circuits produced with the MSD algorithm contain fewer LUTs than circuits produced using the exhaustive reconvergent search.

3.3 Replication of Logic at Fanout Nodes

This section describes how the replication of logic at fanout nodes can reduce the number of LUTs required to implement a network. Recall that the original network is partitioned into a forest of trees and that each tree is separately mapped into a circuit of K-input LUTs. When these separate circuits are assembled to form the circuit implementing the entire network, the replication of logic at fanout nodes can reduce the total number of LUTs in the final circuit. For example, in Figure 3.14a, three LUTs are required to implement the network when the fanout node is explicitly



a) Without replicated logic, 3 LUTs



b) With replicated logic, 2 LUTs

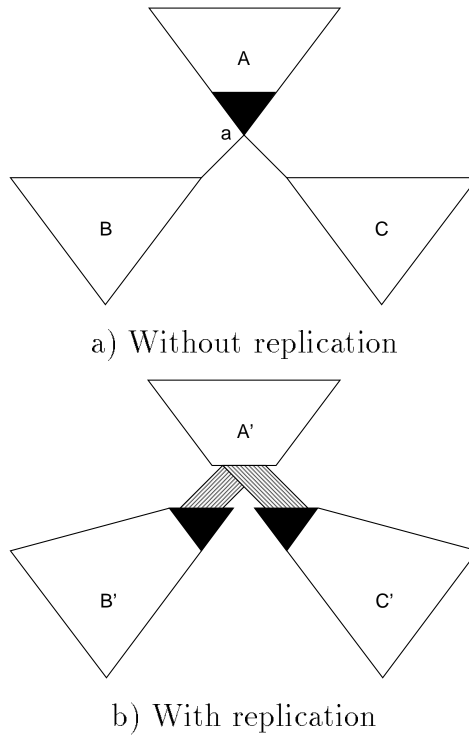
Figure 3.14: Replication of Logic at a Fanout Node

implemented as the output of a LUT. In Figure 3.14b, the AND gate implementing the fanout node is replicated and only two LUTs are required to implement the network.

When the original network is partitioned into a forest of trees, each fanout node is the root of one *source* tree and a leaf of several *destination* trees. For example, in Figure 3.15a the source and destination trees are represented by large triangles. The fanout node a is the root of the source tree A and is a leaf in each of the destination trees B and C.

The replication optimization considers replicating the function of the root LUT of the circuit implementing the source tree. In Figure 3.15a, the small shaded triangle at the root of the source tree represents the root LUT. The root LUT can be eliminated if a replica of its function is added to each of the destination trees, as illustrated in Figure 3.15b. If the total number of LUTs required to implement the destination trees does not increase, then eliminating the root LUT results is an overall reduction in the number of LUTs in the final circuit.

The *Root Replication* procedure, outlined as pseudo-code in Figure 3.16, begins by constructing the circuit implementing the source tree. The destination trees are first mapped without the replication of logic and are then re-mapped with a replica



a) Without replication

b) With replication

Figure 3.15: Replication of the Root LUT

of the function of the source tree's root LUT added to each destination tree. If the total number of LUTs required to implement the destination trees with replication is less than or equal to the number without replication, then the replication is retained and the source tree's root LUT is eliminated.

If the replication at the fanout node is retained, there is a reduction of one LUT in the total number of LUTs. When the original network contains many fanout nodes, the replication optimization is a greedy local optimization that is applied at every fanout node. If the destination tree of one fanout node is the source tree or destination tree of a different fanout node, there can be interactions between the replication of logic at the two fanout nodes. In this case, the replication of logic at the first fanout node can preclude the replication of logic at the second fanout node. The overall success of the replication optimization depends upon the order in which it is applied to the fanout nodes.

In addition to the interaction among the local replication optimizations, there can be interactions between the replication optimization and the reconvergent optimiza-

```
RootRep (sourceTree)
    {
    sourceCircuit = mapTree (sourceTree)
    rootLUT = root LUT of sourceCircuit

    /* find cost without replication */
    noRepTotal = 0

    for all fanout destinationTrees
        {
        noRepCircuit = mapTree (destinationTree)
        noRepTotal = noRepTotal + number of lookup tables in noRepCircuit
        }

    /* find cost with replication */
    repTotal = 0

    for all fanout destinationTrees
        {
        add replica of rootLook to destinationTree

        repCircuit = mapTree (destinationTree)
        repTotal = repTotal + number of lookup tables in repCircuit
        }

    if (repTotal ≤ noRepTotal)
        {
        retain repCircuits
        eliminate rootLUT from sourceCircuit
        }
    else
        {
        retain noRepCircuits
        }
    }
```

Figure 3.16: Pseudo-code for Root-LUT Replication

tion. The replication of logic at fanout nodes can expose reconvergent paths and thereby create additional opportunities for the reconvergent optimization.

3.4 Summary

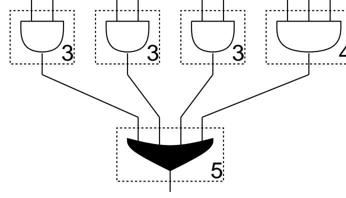
The area algorithm presented in this chapter maps a network into a circuit of K-input LUTs. The algorithm uses bin packing to find decompositions of each node in the network that minimize the number of LUTs in the final circuit. Chapter 5 will prove, for values of $K \leq 5$, that the area algorithm produces an optimal tree of LUTs implementing a network that is a fanout-free tree. General networks are mapped by first partitioning them at fanout nodes into a forest of trees, and then mapping each tree separately. Additional optimizations exploit reconvergent paths, and the replication of logic at fanout nodes, to further reduce the number of LUTs in the final circuit. Chapter 7 will present some experimental results using the area algorithm. The following chapter presents the delay algorithm which minimizes the number of levels of LUTs in the final circuit.

Chapter 4

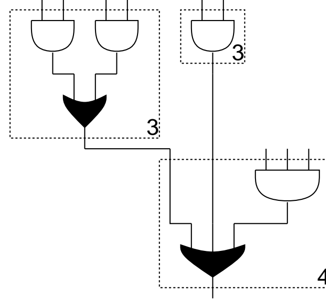
The Delay Algorithm

The FPGA implementation of a typical circuit is roughly three times slower than an MPGA implementation [Brow92]. Since circuit performance is a critical issue in many ASIC applications, this speed disadvantage increases the importance of performance optimization in logic synthesis for FPGAs. One approach to improving the performance of an FPGA circuit is to reduce the number of levels of logic blocks in the circuit. This is one aspect of performance optimization that can be addressed by technology mapping without considering detailed routing. Even though routing delays are a significant portion of total delay in an FPGA circuit, a recent study indicates that minimizing the number of levels of LUTs in a circuit of lookup tables minimizes the total delay of the circuit [Murg91b]. This chapter presents a technology mapping algorithm, referred to as the *delay algorithm*, that minimizes the number of *levels* of K-input lookup tables in the circuit implementing the Boolean network.

The key feature of the delay algorithm is the application of a bin-packing approximation algorithm to the decomposition problem. Unlike the area algorithm, which decomposed nodes to reduce the total number of LUTs, the delay algorithm decomposes nodes to minimize the number of levels in the final circuit. For example, consider the circuit of 5-input LUTs shown in Figure 4.1a. In this figure, the number in the lower right hand corner of a LUT indicates its *depth*, which is the maximum number of LUTs along any path from a primary input to the output of the LUT. The LUTs preceding the AND nodes are not shown in this figure, but they are assumed to



a) Without decomposition, depth = 5



b) With decomposition, depth = 4

Figure 4.1: Decomposition of a Node

contribute to the overall depth as indicated. In Figure 4.1a the shaded OR node is not decomposed, and 5 levels of LUTs are required to implement the network. However, if the OR node is decomposed into the two nodes shown in Figure 4.1b then only 4 levels of LUTs are required. The challenge is to find the decomposition of every node in the network that minimizes the number of levels in the final circuit.

The delay algorithm, like the area algorithm, first partitions the original network into a forest of trees, maps each tree separately into a circuit of K-input LUTs, and then assembles the circuit implementing the entire network from the circuits implementing the trees. The trees are mapped in a breadth-first order proceeding from the primary inputs toward the primary outputs. This ensures that when each tree is mapped that the trees implementing its leaf nodes have already been mapped. The following section describes how dynamic programming and bin packing are used to construct the circuit of K-input LUTs implementing each tree in the forest. Sections 4.2 and 4.3 describe local optimizations at fanout nodes that further reduce the number of levels in the final circuit by exploiting reconvergent paths and the replication of logic at fanout nodes.

The overall strategy employed by the delay algorithm is to minimize the number

of levels of LUTs by minimizing the depth of every path in the final circuit. This can result in a circuit that contains a large number of LUTs. The final section of this chapter discusses methods of reducing this area penalty.

4.1 Mapping Each Tree

Like the area algorithm, the delay algorithm begins mapping each tree in the forest by applying DeMorgan's Theorem and AND-OR associativity rules to ensure that the only inverted edges in the tree originate from the leaf nodes and that there are no consecutive AND nodes and no consecutive OR nodes. The presence of consecutive AND or consecutive OR nodes would restrict the decompositions that the delay algorithm could use. Provided the above conditions are satisfied, Chapter 6 proves that the delay algorithm produces an optimal depth tree of K-input LUTs implementing the tree for values of K less than or equal to 6.

The overall approach taken by the delay algorithm to mapping the tree, outlined as pseudo-code in Figure 4.2, is similar to that used by the area algorithm. Beginning at the leaf nodes and proceeding to the root node, the delay algorithm constructs a circuit at each node that implements the sub-tree extending to the leaf nodes. At leaf nodes, this circuit is simply a single LUT implementing a buffer. Each leaf node is either a primary input or a fanout node that is the root of another tree in the forest. If the leaf node is a primary input the depth of the buffer LUT is one. If the leaf node is the root of another tree, then the depth of the buffer LUT is one greater than the depth of the root LUT of the circuit implementing the leaf node. The order in which the trees in the forest are mapped ensures that the tree rooted at the leaf node has already been mapped.

The circuit implementing a non-leaf node is constructed from the circuits implementing its fanin nodes and consists of two parts. The first part is the decomposition tree implementing the functions of the root LUTs of the fanin circuits and a decomposition of the non-leaf node. The second part is the non-root LUTs of the fanin circuits. The key difference between the delay algorithm and the area algorithm is

```
MapTree (tree)
/* construct circuit implementing tree */
    {
    traverse tree from leaves to root, at each node
        {
        /* construct circuit implementing node */
        if node is a leaf
            circuit ⇐ single LUT buffering node
        else
            circuit ⇐ MapNode (node)
        }

    /* return circuit implementing root */
    return (circuit)
    }


MapNode (node)
/* construct circuit implementing sub-tree rooted at node */
    {
    /* separate fanin LUTs */
    faninLUTs ⇐ root LUTs of circuits for all fanin nodes
    precedingLUTs ⇐ non-root LUTs of circuits for all fanin nodes

    /* construct decomposition tree */
    decompositionTree ⇐ DecomposeDelay (node, faninLUTs)

    /* join decomposition tree and preceding LUTs */
    circuit ⇐ decompositionTree ∪ precedingLUTs

    return (circuit)
    }
```

Figure 4.2: Pseudo-code for Mapping a Tree

the procedure used to construct the decomposition tree at each node.

Once again, there are two optimization goals for the circuit implementing each non-leaf node. The primary goal is to minimize the depth of the root LUT of the circuit, and the secondary goal is to minimize the number of inputs used by the root LUT. The secondary optimization goal is the key to ensuring that the optimal

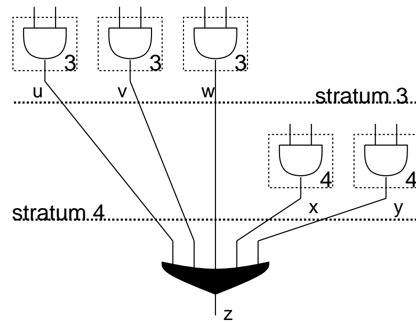
decomposition tree, and therefore the optimal circuit implementing the non-leaf node, is constructed from the optimal fanin circuits. The following section describes how the decomposition tree is constructed.

4.1.1 Constructing the Decomposition Tree

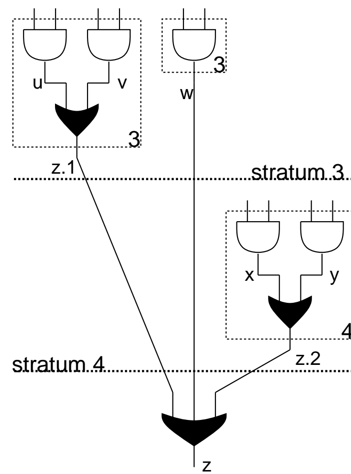
At each non-leaf node, the decomposition tree is constructed in three steps. The first step separates the root LUTs of the fanin circuits according to their depth into *strata*, the second step packs the LUTs within each stratum to minimize the number of LUTs in the stratum, and the final step connects the strata together to form the complete decomposition tree. The remainder of this section will refer to the root LUTs of the fanin circuits as the fanin LUTs.

Consider the node z and its fanin LUTs shown in Figure 4.3a. Initially, stratum 3 contains the fanin LUTs u, v and w, and stratum 4 contains the fanin LUTs x and y. Figure 4.3b shows the result of minimizing the number of LUTs within each stratum, and Figure 4.3c shows the complete decomposition tree. In Figure 4.3b each LUT in a given stratum implements some subset of the fanin LUTs at that stratum's depth and the corresponding decomposition of the node z. For example, the stratum-4 LUT $z.2$ implements the functions of the fanin LUTs x and y. In Figure 4.3c the outputs of the stratum-3 LUTs w and $z.1$ are connected to the existing stratum-4 LUT $z.2$ and the new stratum-4 LUT $z.3$, and the outputs of these stratum-4 LUTs are in turn connected to a new stratum-5 LUT to form the complete decomposition tree.

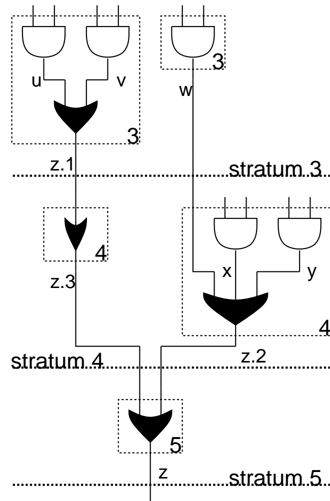
The decomposition tree is optimal if its root LUT is at the minimum depth and uses the minimum number of inputs. Minimizing the number of LUTs within each stratum, using the FFD bin-packing algorithm outlined in Section 3.2 is the key to constructing the optimal decomposition tree. The procedure used to construct the decomposition tree is outlined as pseudo-code in Figure 4.4. This procedure begins by separating the fanin LUTs into strata according to their depth and using the FFD bin-packing algorithm to minimize the number of LUTs within each stratum. Note that packing more than one fanin LUT (box) into a stratum LUT (bin) requires the introduction of a decomposition node. For example in Figure 4.3b when boxes x and



a) Fanin LUTs separated into strata



b) Minimized strata



c) The complete decomposition tree

Figure 4.3: Constructing the Decomposition Tree

```
DecomposeDelay (node, faninLUTs)
/* construct tree of LUTs implementing decomposition of node and faninLUTs */
/* minimize depth of decomposition tree */
    {
    minD ⇐ minimum depth of faninLUTs
    maxD ⇐ maximum depth of faninLUts

    for all d from minD to maxD
        {
        stratumLUTs [d] ⇐ FirstFitDecreasing (node, faninLUTs at depth d)
        }

    lookList ⇐ ∅
    d ⇐ minD

    until (only one LUT in stratumLUTs [d] & d ≥ maxD)
        {
        /* connect LUTs in stratumLUTs [d] to LUTs in stratumLUTs [d + 1] */

        for all sourceLUT in stratumLUTs [d]
            {
            destinationLUT ⇐ first LUT with unused input in stratumLUTs [d + 1]

            if such a destinationLUT does not exist
                {
                destinationLUT ⇐ new LUT
                add destinationLUT to end of stratumLUTs [d + 1]
                }

            connect sourceLUT output to destinationLUT input
            /* implies decommposition of node */

            add sourceLUT to lookList
            }
        d ⇐ d + 1
        }

    add stratumLUT [d] to lookList

    return (lookList)
    }
```

Figure 4.4: Pseudo-code for Constructing Decomposition Tree

y are packed into one bin this requires the introduction of the node $z.2$.

After minimizing the number of LUTs within each stratum the algorithm proceeds from the uppermost stratum to the deepest stratum connecting the outputs of LUTs in stratum D to unused inputs in stratum $D+1$. The decomposition tree is complete when the deepest stratum contains only one LUT. Connecting the output of a LUT in stratum D to an unused input of a LUT in stratum $D+1$ alters the decomposition node in the stratum $D+1$ LUT. For example, in Figure 4.3c when the stratum-3 LUT w is connected to the stratum-4 LUT $z.2$ the decomposition of the OR node is altered. It may be necessary to add extra LUTs to stratum $D+1$ if there are more LUTs in stratum D than unused inputs in stratum $D+1$. For example, in Figure 4.3c the output of the stratum-3 LUT $z.1$ is connected to the new stratum-4 LUT $z.3$. Note that in this example the extra LUT is simply a buffer and it can be eliminated after the decomposition tree is completed. This procedure produces an optimal decomposition tree provided that the FFD bin-packing algorithm packs the fanin LUTs (boxes) within each stratum into the minimum number of stratum LUTs (bins).

4.1.2 Optimality

The goal of the delay algorithm is to minimize the number of levels of K-input LUTs in the circuit implementing the original Boolean network. The network is first partitioned into a forest of trees, each of these is mapped separately and the circuit implementing the entire network is assembled from these separate circuits. Section 5.3 of Chapter 5 proves that the tree of LUTs constructed for each tree is optimal, provided that the value of K is less than or equal to 6.

If f is the number of fanin edges at a given node in the tree, and bucket sorts are used, then the time taken to construct the decomposition tree is bounded by Kf. Each node in the tree is visited once by the dynamic programming traversal. If there are n nodes in the tree, and the maximum fanin at any node is F, then the time taken to map the entire tree is bounded by nKF.

As in the area algorithm, the circuit implementing the entire network that is

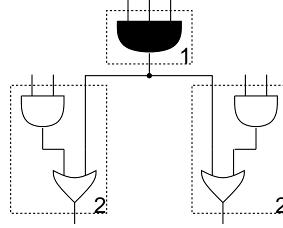
assembled by the delay algorithm from the separate circuits implementing each tree is not necessarily optimal, even if the separate circuits are optimal. Partitioning the original network into a forest of fanout-free trees precludes LUTs that realize functions containing reconvergent paths and assembling the final circuit from the separate circuits implementing each tree precludes the replication of logic at fanout nodes. The following sections describe local optimizations that exploit reconvergent paths and the replication of logic at fanout nodes to further reduce the number of levels in the final circuit.

4.2 Exploiting Reconvergent Paths

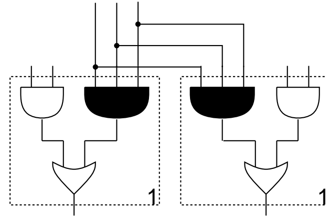
As discussed in Section 3.3 of Chapter 3, shared inputs among the fanin LUTs indicate the presence of local reconvergent paths. The key to a better decomposition tree is minimizing the number of LUTs within each stratum. When two fanin LUTs with a shared input are packed into one stratum LUT, the number of inputs used is less than the sum of the number of inputs used by each LUT. This can allow the number of LUTs in the stratum to be reduced, which in turn can improve the decomposition tree. Either the exhaustive search, or the Maximum Share Decreasing heuristic, described in Section 3.3 of Chapter 3, can be used to replace the FFD bin-packing algorithm in the procedure used to construct the decomposition tree.

4.3 Replication of Logic at Fanout Nodes

This section describes how the replication of logic at fanout nodes can reduce the depth of the final circuit. After the original network has been partitioned into a forest of trees, every fanout node is the root of one source tree and the leaf of several destination trees. As in the area algorithm, the delay algorithm first maps the source tree into a circuit of K-input LUTs and then determines if a replica of the function of the root LUT of this circuit should be added to each of the destination trees. Consider the circuit shown in Figure 4.5a. In this circuit the fanout node is implemented as



a) Without replication, depth = 2



b) With replication, depth = 1

Figure 4.5: Replicating Logic at a Fanout Node

the output of a LUT and the circuit contains two levels of LUTs. In Figure 4.5b the function of the LUT implementing the fanout node has been replicated for each of the fanout edges and the resulting circuit contains only one level.

The delay algorithm adds a replica of the root LUT for every fanout edge of every fanout node. Provided the minimum depth circuit is constructed for each destination tree, this simple replication optimization will either leave the number of levels unchanged, or reduce the number of levels. However, replication of logic can increase the number of LUTs in the final circuit. The following section will discuss a replication optimization that avoids replications that increase area when they do not decrease depth.

4.4 Reducing the Area Penalty

The delay algorithm presented so far is concerned solely with minimizing the number of levels in the final circuit, and does not attempt to minimize the number of LUTs in the circuit. Compared to the area algorithm, the construction of the decomposition tree and the replication optimization in the delay algorithm significantly increase the number of LUTs in the circuit. The following sections describe three optimizations

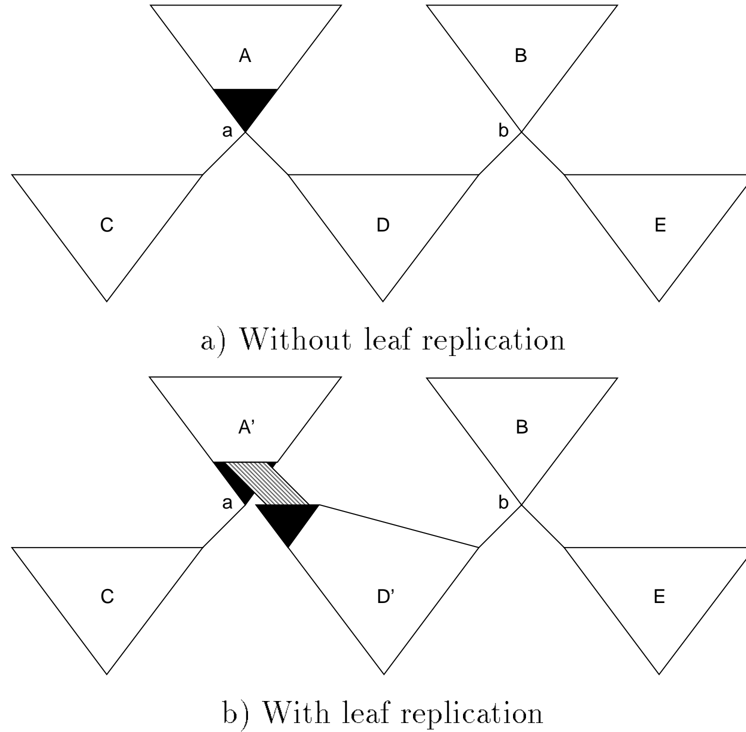


Figure 4.6: Replication at a Leaf Node

that attempt to reduce this area penalty. The first optimization avoids replications that increase the number of LUTs without decreasing the number of levels in the circuit. The second optimization searches for LUTs that can be merged into their fanout LUTs. The third optimization uses the area algorithm decomposition to reduce the number of LUTs on non-critical paths.

4.4.1 Avoiding Unnecessary Replication

At each fanout node, the replication optimization described in Section 4.3 creates a replica of the function of the LUT implementing the fanout node for every fanout edge. For some fanout edges the addition of the replica can increase the number of LUTs in the circuit without decreasing the number of levels. Recall that in the area algorithm, the Root Replication procedure created a replica for either all or none of the fanout edges. This section describes a *Leaf Replication* procedure that determines if a replica should be added to each edge independently.

Consider the forest of trees shown in Figure 4.6a. In this figure each large triangle

represents one tree. For the tree D the two leaf nodes leaf nodes a and b are fanout nodes that are also the roots of trees A and B. When the Leaf replication procedure maps the tree D, the order in which the trees are mapped ensures that the trees A and B have already been mapped. At each leaf node, the root LUT of the circuit implementing the node is referred to as the *fanout LUT* at that node. In figure 4.6a, the fanout LUT at node a is represented by the small shaded triangle. At each leaf node, the procedure determines if a replica of the function of the fanout LUT should replace the leaf node as the source of the fanout edge leading to the tree being mapped. Figure 4.6b shows the result of replicating the fanout LUT at node a for the fanout edge leading to tree D. Note that the decision to add a replica to the other fanout edge from a is made independently when the tree C is mapped.

Since the goal of the delay algorithm is to minimize the number of levels in the final circuit, the Leaf Replication procedure outlined as pseudo-code in Figure 4.7 first maps the tree with replication at all leaf nodes. Next the procedure determines at which nodes the replica can be removed without increasing the depth of the circuit. It considers each leaf node in sequence, removing the replica of the fanout LUT at that leaf node and re-mapping the tree. If the resulting circuit does not increase in depth and contains fewer LUTs, then the new circuit is greedily retained. Otherwise, the replica is restored at the leaf node.

4.4.2 Merging LUTs into their Fanout LUTs

As shown in Figure 4.3c the decomposition tree constructed by the delay algorithm can include LUTs that implement simple buffer functions. These are examples of LUTs that can be merged with their fanout LUTs without increasing the depth of the circuit. Provided that the result is a LUT with at most K-inputs, merging any single-fanout LUT into its fanout LUT eliminates one LUT from the circuit without increasing the depth of the circuit. After the original network has been mapped into a circuit of K-input LUTs, a peephole optimization proceeds from the primary inputs to the primary outputs merging single-fanout LUTs with their fanout LUTs whenever the result is a LUT with at most K-inputs.

```
LeafRep (tree)
    {
    /* begin by replicating at every leaf node */
    for all leaf nodes of tree
        {
        fanoutLUT [leaf] ⇐ root LUT of circuit implementing leaf
        add replica of fanoutLUT [leaf] to tree
        }
    bestCircuit ⇐ mapTree (tree)

    for all leaf nodes of tree
        {
        /* remap tree without the replica at the leaf */
        remove replica of fanoutLUT [leaf] from tree
        circuit ⇐ mapTree (tree)

        if (area of circuit < area of bestCircuit & depth of circuit ≤ depth of bestCircuit)
            {
            /* update bestCircuit */
            bestCircuit ⇐ circuit
            }
        else {
            /* restore replica */
            add replica of fanoutLUT [leaf] to tree
            }
        }
    }
```

Figure 4.7: Pseudo-code for Replication

4.4.3 Mapping for Area on Non-Critical Paths

To minimize the depth of the critical path, the delay algorithm described so far has minimized the depth of all LUTs. This section refers to the deepest LUTs in the circuit as the critical LUTs. In addition, any other LUT is critical if an increase in its depth increases the number of levels in the circuit. Minimizing the depth of non-critical LUTs may unnecessarily increase the number of LUTs in the circuit. This section describes an optimization that reduces the number of LUTs in the circuit without increasing the number of levels. The depth of a non-critical LUT can be increased by

```
mapCritical (network)
    {
    for all nodes in network
        decompositionMode [node] ⇐ DelayDecomposition

    circuit ⇐ mapNetwork (network)

    targetDepth ⇐ depth of circuit

    for all nodes in network
        decompositionMode [node] ⇐ AreaDecomposition

    circuit ⇐ mapNetwork (network)

    while (depth of circuit exceeds targetDepth)
        {
        find super critical LUTs

        for all nodes implemented by super critical LUTs
            decompositionMode [node] ⇐ DelayDecomposition

        circuit ⇐ mapNetwork (network)
        }
    }
```

Figure 4.8: Pseudo-code for Critical

an amount referred to as its *slack* without increasing the depth of subsequent LUTs. This slack represents an opportunity to reduce the number of LUTs in the circuit by locally increasing depth. The overall strategy is to construct the decomposition tree for each node in the network using one of two modes. Nodes implemented by critical LUTs are decomposed using the delay decomposition described in this chapter, and nodes implemented by non-critical LUTs are decomposed using the area decomposition described in the previous chapter.

Before the original network is mapped, it is not readily apparent which nodes will be implemented by critical LUTs. The optimization, outlined as pseudo-code in Figure 4.8, therefore uses an iterative approach to determine in which mode each

node should be decomposed. The procedure first maps the network using the delay decomposition for all nodes. This establishes a target depth for the final circuit. Next the network is re-mapped using the area decomposition for all nodes. This minimizes the number of LUTs, however, the circuit may now contain LUTs that exceed the target depth. These LUTs are referred to as *super-critical* LUTs. In addition, any LUT is super-critical if increasing its depth increases the depth of another super critical LUT. To restore the number of levels in the circuit to the target depth the procedure uses an iterative approach. Each iteration changes the decomposition mode of nodes implemented by super-critical LUTs from the area decomposition to delay decomposition and then re-maps the network. The network is iteratively re-mapped until there are no LUTs in the circuit that exceed the target depth. An iterative approach is required because reducing the depth of existing super-critical LUTs can result in other LUTs becoming super-critical.

4.5 Summary

The delay algorithm presented in this chapter maps Boolean networks into circuits of K-input LUTs. The algorithm uses bin packing to find decompositions of each node in the network that minimize the number of levels in the final circuit. Chapter 6 will prove, for values of $K \leq 6$, that the delay algorithm produces an optimal tree of LUTs implementing a network that is a fanout-free tree. Chapter 7 will present some experimental results produced with the delay algorithm.

Chapter 5

Area Optimality

This chapter presents a proof that the area algorithm constructs an optimal tree of K-input LUTs implementing a network that is a fanout-free tree, for values of $K \leq 5$. In this chapter script letters, such as $\mathcal{A}$, are used to represent circuits of K-input LUTs, and the number of LUTs in the circuit $\mathcal{A}$ is denoted by $|\mathcal{A}|$. If $\mathcal{A}$ is a single-output circuit then the number of inputs used at the root (output) LUT of $\mathcal{A}$ is denoted by $\langle \mathcal{A} \rangle$. The *area-optimal* circuit implementing a single-output network is defined as follows:

> The circuit $\mathcal{A}$ is area optimal if and only if for all circuits $\mathcal{B}$ implementing the same function, $|\mathcal{B}| \geq |\mathcal{A}|$ and $\langle \mathcal{B} \rangle \geq \langle \mathcal{A} \rangle$ whenever $|\mathcal{B}| = |\mathcal{A}|$.

The remainder of this chapter will prove the following theorem.

Theorem 5.1

> Given an original network that is a fanout-free tree, an area algorithm constructs an area-optimal tree of K-input LUTs implementing the network for values of $K \leq 5$.

As described in Chapter 3, the area algorithm traverses the original tree beginning at the leaf nodes and proceeds to the root node. At every node, a circuit implementing the sub-tree rooted at that node is constructed. Section 5.1 will prove the following lemma:

Lemma 5.2

> At each non-leaf node, the area algorithm constructs an area-optimal tree of LUTs implementing that node, for values of $K \leq 5$, if the circuits implementing its fanin nodes are area-optimal.

Theorem 5.1 is proved by induction using Lemma 5.2. The basis of the induction is the circuit constructed by the area algorithm at each leaf node. This circuit consists of a single LUT implementing a buffer. Since no other circuit implementing the leaf node can have fewer LUTs, or use fewer inputs at its root LUT, this buffer LUT is by definition an area-optimal circuit implementing the leaf node.

Since the circuits implementing the leaf nodes are area-optimal, it follows by induction from Lemma 5.2 that the circuit constructed at every node, including the root node of the tree, is area-optimal. Therefore, Theorem 5.1 follows from Lemma 5.2. The remainder of this chapter presents a proof of Lemma 5.2.

5.1 Outline for Proof of Lemma 5.2

To prove Lemma 5.2 the following notation is introduced: Let $\mathcal{A}$ be the circuit constructed by the area algorithm and let $\mathcal{B}$ be an arbitrary tree of LUTs implementing the same function. To prove that $\mathcal{A}$ is area-optimal, it is sufficient to show that $|\mathcal{B}| \geq |\mathcal{A}|$ and $\langle\mathcal{B}\rangle \geq \langle\mathcal{A}\rangle$ whenever $|\mathcal{B}| = |\mathcal{A}|$. Note that the proof only considers circuits that are trees of LUTs.

The proof proceeds by transforming the circuit $\mathcal{B}$ without changing its function, or increasing $|\mathcal{B}|$ or $\langle\mathcal{B}\rangle$, and then showing for the transformed circuit $\mathcal{B}$ that $|\mathcal{B}| \geq |\mathcal{A}|$ and that $\langle\mathcal{B}\rangle \geq \langle\mathcal{A}\rangle$ if $|\mathcal{B}| = |\mathcal{A}|$. The following section introduces notation that describes the circuit $\mathcal{A}$ constructed by the area algorithm, and Section 5.1.2 describes how the circuit $\mathcal{B}$ is transformed without changing its function, or increasing $|\mathcal{B}|$ or $\langle\mathcal{B}\rangle$. Finally, Section 5.1.3 proves Lemma 5.2.

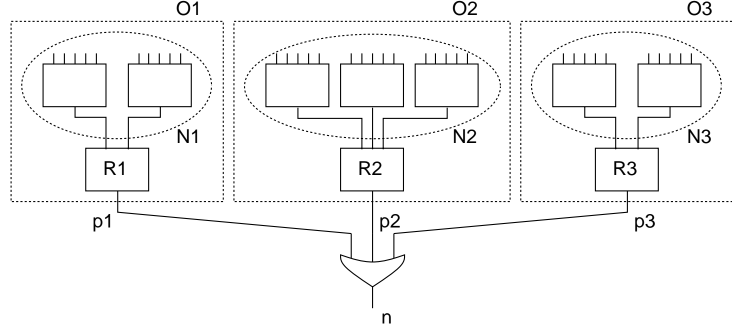


Figure 5.1: The Optimal Fanin Circuits, $\mathcal{O}_i$

5.1.1 Notation for the Circuit $\mathcal{A}$

This section introduces the notation to describe the non-leaf node being mapped, the area-optimal circuits implementing its fanin nodes, and the circuit constructed by the area algorithm.

Let n be the non-leaf node and let p_1 to p_f be its fanin nodes. Without loss of generality this section assumes that the non-leaf node is an OR node and that its fanin nodes are AND nodes or primary inputs. The case where n is an AND node is the dual of the case considered here, and the first step in the area algorithm uses DeMorgan's Law and AND-OR associative rules to ensure that the fanin nodes of an OR node are either AND nodes, or primary inputs.

Let $\mathcal{O}_i$ be an optimal circuit implementing the fanin node p_i. As illustrated in Figure 5.1 each fanin circuit $\mathcal{O}_i$ consists of a root LUT $\mathcal{R}_i$ and the non-root LUTs $\mathcal{N}_i$. In this figure, LUTs are represented by solid rectangles, each optimal fanin circuit, $\mathcal{O}_i$, is bounded by a dotted rectangle and the non-root LUTs ,$\mathcal{N}_i$, are bounded by a dotted ellipse. In this example, $|\mathcal{O}_1| = 3$, $|\mathcal{O}_2| = 4$, $|\mathcal{O}_3| = 3$, $\langle\mathcal{O}_1\rangle = 2$, $\langle\mathcal{O}_2\rangle = 3$, and $\langle\mathcal{O}_3\rangle = 2$. Note that $\langle\mathcal{R}_i\rangle = \langle\mathcal{O}_i\rangle$, and that $|\mathcal{O}_i| = |\mathcal{N}_i| + 1$.

The circuit $\mathcal{A}$, constructed by the area algorithm, is illustrated in Figure 5.2. This circuit consists of the decomposition tree, $\mathcal{D}$, and the non-root LUTs, $\mathcal{N}_i$, for all i from 1 to f. The root LUT of the decomposition tree $\mathcal{D}$ is the root LUT of the circuit $\mathcal{A}$. Therefore, $\langle\mathcal{A}\rangle = \langle\mathcal{D}\rangle$. In the example shown in Figure 5.2, $\langle\mathcal{D}\rangle = 3$.

As described in Chapter 3, the decomposition tree is constructed from the fanin LUTs $\{\mathcal{R}_i\}$ in two steps. The first step uses the FFD bin packing algorithm to pack

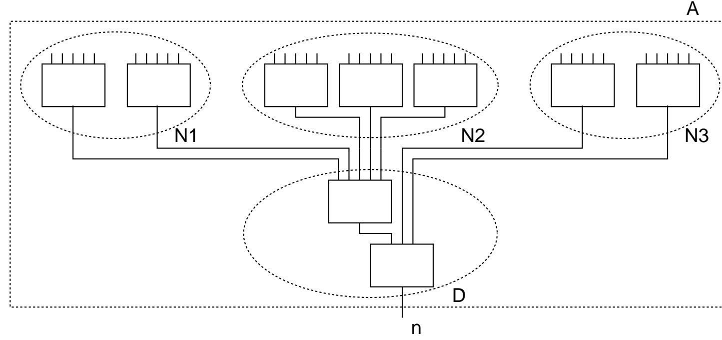


Figure 5.2: The Circuit $\mathcal{A}$

the fanin LUTs into second-level LUTs. The fanin LUTs correspond to boxes of size $\langle \mathcal{R}_i \rangle = \langle \mathcal{O}_i \rangle$ and the second-level LUTs correspond to bins of capacity K. The second step connects the second-level LUTs to form the final decomposition tree. The procedure begins by sorting the second-level LUTs into a list ordered by increasing number of unused inputs. Then, while more than one LUT remains in the list the procedure iteratively removes the first LUT from the list and connects its output to the first available unused input in the list. If there are no unused inputs in the list, then an empty LUT is added to the end of the list and the output of the first LUT is connected to an input of this new LUT.

Note that whenever the first LUT in the list has at least one unused input that all the remaining LUTs in the list must have at least one unused input. In this case, the output of each LUT in the list, except the last LUT, is connected to an input of the next LUTs in the list. No new LUTs are added and the last LUT in the list becomes the root LUT of the decomposition tree.

From the above observation, it follows that a new bin is added only when all of the previously removed bins have no unused inputs, and all of the remaining bins have no unused inputs. Therefore, when a new bin is added all non-root LUTs in the final decomposition tree will have no unused inputs.

5.1.2 Transforming the Circuit $\mathcal{B}$

Recall that $\mathcal{B}$ is a tree of LUTs implementing the same function as the circuit $\mathcal{A}$. This section describes how the arbitrary circuit $\mathcal{B}$ is transformed, without changing

its function, or increasing $|\mathcal{B}|$ or $\langle\mathcal{B}\rangle$. The objective of the transformation is to produce in the circuit $\mathcal{B}$ a sub-circuit that is comparable to the decomposition tree $\mathcal{D}$ in the circuit $\mathcal{A}$. The transformations incorporate in the circuit $\mathcal{B}$ the non-root LUTs $\mathcal{N}_i$, for all i, and connect the $\langle\mathcal{R}_i\rangle$ outputs of the sub-circuit $\mathcal{N}_i$ to one LUT in the circuit $\mathcal{B}$. This is a structure that is similar to the circuit $\mathcal{A}$ where the $\langle\mathcal{R}_i\rangle$ outputs of the non-root LUTs $\mathcal{N}_i$ are connected to the LUT $\mathcal{R}_i$. After the circuit $\mathcal{B}$ has been transformed this observation is used to prove Lemma 5.2. Note that the transformations must not change the function of the circuit $\mathcal{B}$, or increase $|\mathcal{B}|$ or $\langle\mathcal{B}\rangle$.

The circuit $\mathcal{B}$ is assumed to be a tree of LUTs. The key to the transformation that incorporates the non-root LUTs $\mathcal{N}_i$ is finding a sub-tree of LUTs in $\mathcal{B}$ that implements the function

$$p_i + \sum_{j \in \lambda} p_j$$

or its complement, for some subset λ of the fanin nodes. The following argument shows that such a sub-circuit must exist. The notation Φ_i is introduced to represent the set of primary inputs that are the leaf nodes of the sub-tree rooted at the fanin node p_i. Let $\mathcal{L}$ be the root LUT of the smallest sub-tree in the circuit $\mathcal{B}$ that contains all the primary inputs in Φ_i. Let Φ_G be the primary inputs for this sub-tree, and Φ_F be the primary inputs for the remainder of the circuit. Because the circuit $\mathcal{B}$ is a tree, the function implemented by the circuit can be represented by the disjoint decomposition

$$F(G(\Phi_G), \Phi_F)$$

where $G()$ is the function implemented by the sub-tree rooted at the LUT $\mathcal{L}$, and $F()$ is the function implemented by the remainder of the circuit.

The node n, in the network being mapped, is an OR node and all of its fanin nodes are AND nodes. Since the network rooted at n is a tree, it is easy to show that Φ_F cannot include any primary inputs from Φ_j, for $j \neq i$, if Φ_G includes primary inputs from both Φ_i and Φ_j. The proof is a simple application of the necessary and sufficient conditions on disjoint decompositions stated by Kohavi [Koha70]. In informal terms, it is not possible to split the AND node p_j across the OR node n. It is known from

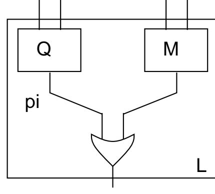


Figure 5.3: Decomposition of $\mathcal{L}$

the definition of $\mathcal{L}$ that Φ_G contains all inputs in Φ_i. Therefore, Φ_G contains either *all* or *none* of the primary inputs in Φ_j, for all $j \neq i$. If λ is the set of j for which Φ_j is contained in Φ_G. then

$$\Phi_G = \Phi_i \cup \bigcup_{j \in \lambda} \Phi_j$$

and the function at the output of $\mathcal{L}$, or its complement, must be

$$\mathcal{L} = p_i + \sum_{j \in \lambda} p_j$$

Note that $\mathcal{L}$ is a LUT in the circuit $\mathcal{B}$, and that p_i and p_j are the fanin nodes of node n in the network being mapped.

From the assumption that $\mathcal{L}$ is the root of the smallest sub-tree that includes all of Φ_i, it follows that at least two of the sub-trees rooted at the inputs of $\mathcal{L}$ include primary inputs from Φ_i. Therefore, any sub-tree rooted at an input of $\mathcal{L}$ that includes any primary inputs from Φ_i must include only primary inputs from Φ_i, or else there would be a splitting of the AND node p_i across the OR node n. Therefore, the LUT $\mathcal{L}$ can be decomposed into the disjoint functions $\mathcal{Q}$ and $\mathcal{M}$ as illustrated in Figure 5.3. The output of $\mathcal{Q}$ is the function p_i, the output of $\mathcal{M}$ is the function $\sum p_j$, for $j \in \lambda$, and the output of $\mathcal{L}$ is the function $\mathcal{Q} + \mathcal{M}$.

Let $\mathcal{P}$ be the sub-tree of LUTs, bounded by the dotted rectangle in Figure 5.4, that contains the LUT $\mathcal{L}$ and all LUTs in the sub-trees rooted at the inputs of the function $\mathcal{Q}$. By setting the inputs to the function $\mathcal{M}$ to values that result in $p_j = 0$, for $j \in \lambda$, this sub-circuit can implement the function of the fanin node p_i. It is given that $\mathcal{O}_i$ is an optimal circuit implementing the fanin node p_i. Therefore, $|\mathcal{P}| \geq |\mathcal{O}_i|$ and if $|\mathcal{P}| = |\mathcal{O}_i|$ then $\langle \mathcal{Q} \rangle \geq \langle \mathcal{O}_i \rangle$ Two different transformations are applied to the circuit $\mathcal{B}$ when $|\mathcal{P}| = |\mathcal{O}_i|$ and when $|\mathcal{P}| > |\mathcal{O}_i|$. The case where $|\mathcal{P}| = |\mathcal{O}_i|$ is

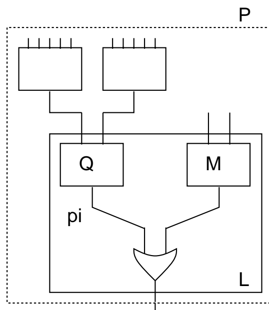


Figure 5.4: The sub-circuit $\mathcal{P}$

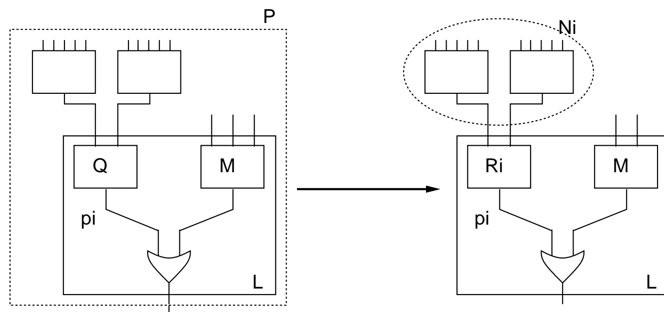


Figure 5.5: Modification when $|\mathcal{L}| = |\mathcal{R}_i|$

considered first.

Case 1: $|\mathcal{P}| = |\mathcal{O}_i|$

If $|\mathcal{P}| = |\mathcal{O}_i|$ then $\langle \mathcal{Q} \rangle \geq \langle \mathcal{O}_i \rangle$. In this case, the function $\mathcal{Q}$ can be replaced with the function $\mathcal{R}_i$, and the non-root LUTs in $\mathcal{P}$ can be replaced with the LUTs $\mathcal{N}_i$, as illustrated in Figure 5.5. Because $|\mathcal{P}| = |\mathcal{O}_i|$ it follows that the number of non-root LUTs in $\mathcal{P}$ is the same as the number of LUTs in $\mathcal{N}_i$, Therefore, this transformation does not increase the number of LUTs in the circuit, and because $\langle \mathcal{Q} \rangle \geq \langle \mathcal{O}_i \rangle$ it does not increase $\langle \mathcal{L} \rangle$.

Case 2: $|\mathcal{P}| > |\mathcal{O}_i|$

If $|\mathcal{P}| > |\mathcal{O}_i|$ then the function $\mathcal{Q}$ can be replaced by a single input connected to the output of the circuit $\mathcal{O}_i$ as illustrated in Figure 5.6. Because $|\mathcal{P}| > |\mathcal{O}_i|$ this transformation does not increase the number of LUTs in the circuit, and because $\langle \mathcal{Q} \rangle \geq 1$ it does not increase $\langle \mathcal{L} \rangle$. Note that the circuit $\mathcal{O}_i$ includes the root LUT $\mathcal{R}_i$

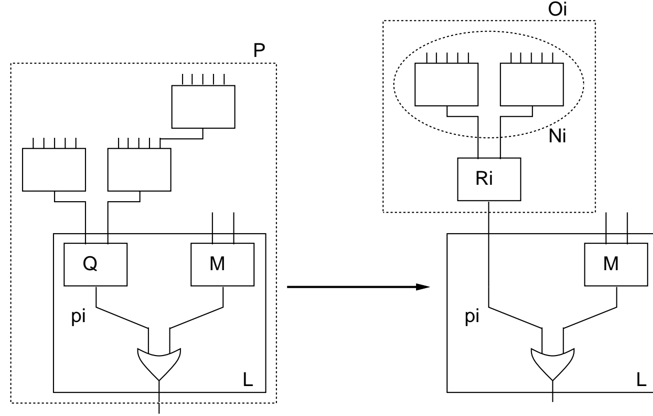


Figure 5.6: Modification when $|\mathcal{L}| > |\mathcal{R}_i|$

and the non-root LUTs $\mathcal{N}_i$.

In either case, the transformation does not change the function of $\mathcal{L}$, and does not increase the number of LUTs in the circuit, or increase $\langle\mathcal{L}\rangle$. Therefore these transformations will not change the function of $\mathcal{B}$, or increase $|\mathcal{B}|$ or $\langle\mathcal{B}\rangle$. The end result of these transformations is that the circuit $\mathcal{B}$ includes the non-root LUTs $\mathcal{N}_i$, for all i, and all $\langle\mathcal{R}_i\rangle$ outputs of $\mathcal{N}_i$ are connected to one LUT.

5.1.3 Proof of Lemma 5.2

This section completes the proof of Lemma 5.2. After the transformations described in Section 5.1.2, the circuits $\mathcal{A}$ and $\mathcal{B}$ both include the non-root LUTs $\mathcal{N}_i$, for all i. These non-root LUTs can be eliminated from the circuits $\mathcal{A}$ and $\mathcal{B}$ without affecting the comparison of the number of LUTs in the circuit. The decomposition tree $\mathcal{D}$ is all that remains of the circuit $\mathcal{A}$, and the LUTs that remain from the circuit $\mathcal{B}$ are referred to as the sub-circuit $\mathcal{E}$. To prove Lemma 5.2 it is sufficient to show, for values fo $K \leq 5$, that $|\mathcal{E}| \geq |\mathcal{D}|$ and that $\langle\mathcal{E}\rangle \geq \langle\mathcal{D}\rangle$ whenever $|\mathcal{E}| = |\mathcal{D}|$.

Two cases are considered, based on the decomposition tree $\mathcal{D}$ described in section 5.1.1. If at least one LUT in $\mathcal{D}$, other than the root LUT, has an unused input, then the decomposition tree is said to be *bin limited*. If none of the non-root LUTs has an unused input, then $\mathcal{D}$ is said to be *pin limited*. Note that if $\mathcal{D}$ contains a single LUT, then it is considered to be pin limited.

The Bin Limited Case

If $\mathcal{D}$ is bin limited, then no extra LUTs were added when the second-level LUTs were connected together to form the decomposition tree. Therefore the LUTs in $\mathcal{D}$ correspond to the bins that the FFD algorithm produces for the set of boxes of size $\langle \mathcal{R}_i \rangle$ for all i. The number of bins is equal to $|\mathcal{D}|$.

For all i, the $\langle \mathcal{R}_i \rangle$ outputs of the non-root LUTs $\mathcal{N}_i$ are connected to one LUT in $\mathcal{E}$. Therefore, the LUTs in $\mathcal{E}$ also correspond to a set of bins containing boxes of size $\langle \mathcal{R}_i \rangle$ for all i, and the number of bins is equal to $|\mathcal{E}|$. The sets of bins corresponding to the LUTs in $\mathcal{D}$ and $\mathcal{E}$ both contain the same set of boxes. Appendix B shows that the FFD algorithm packs any set of integer sized boxes into the minimum number of bins of capacity K for integer values of $K \leq 6$. Therefore for $K \leq 6$, $|\mathcal{E}| \geq |\mathcal{D}|$. All that remains to prove Lemma 5.2 is to show that $\langle \mathcal{E} \rangle \geq \langle \mathcal{D} \rangle$ whenever $|\mathcal{E}| = |\mathcal{D}|$.

The unused capacity of any bin corresponding to a LUT in $\mathcal{D}$ or $\mathcal{E}$ is referred to as a hole. The key to showing that $\langle \mathcal{E} \rangle \geq \langle \mathcal{D} \rangle$, is the comparison of the largest holes in $\mathcal{E}$, and $\mathcal{D}$.

In order to be bin limited, there must be more than one LUT in $\mathcal{D}$. If $|\mathcal{D}| = 1$ then by definition $\mathcal{D}$ would be pin limited. The bin corresponding to the root LUT of $\mathcal{D}$ contains the largest hole of any of the bins, because the bins are ordered by size before they are connected to form the decomposition tree. If the decomposition tree is bin limited, then by definition at least one non-root LUT in $\mathcal{D}$ has an unused input, and when the bins are connected, exactly one input of the root LUT of $\mathcal{D}$ is connected to the output of another LUT in $\mathcal{D}$. This input is not accounted for by the boxes of size $\langle \mathcal{R}_i \rangle$, for all i, and can be considered part of the hole in the bin corresponding to the root LUT. Therefore, the size of the largest hole in any of the bins is $K + 1 - \langle \mathcal{D} \rangle$.

The sub-circuit $\mathcal{E}$ is a tree of LUTs and $|\mathcal{E}| \geq |\mathcal{D}| \geq 2$. Therefore at least one input of the root LUT of $\mathcal{E}$ is connected to the output of another LUT in $\mathcal{E}$. This input can be considered as part of the hole in the bin corresponding to the root LUT. The size of the hole in this bin is therefore at least $K + 1 - \langle \mathcal{E} \rangle$. Appendix B shows, for values of $K \leq 5$, that the set of bins produced by the FFD algorithm

includes a bin with the largest hole possible for any set of bins that contains the same set of boxes in the same number of bin. Therefore if $|\mathcal{E}| = |\mathcal{D}|$, and $K \leq 5$, then $K + 1 - \langle\mathcal{D}\rangle \geq K + 1 - \langle\mathcal{E}\rangle$ This inequality can be simplified to show that $\langle\mathcal{E}\rangle \geq \langle\mathcal{D}\rangle$ whenever $|\mathcal{E}| = |\mathcal{D}|$, Therefore Lemma 5.2 is true if $\mathcal{D}$ is bin limited.

The Pin Limited Case

When $\mathcal{D}$ is pin limited, Lemma 5.2 can be proved by counting the number of used inputs in $\mathcal{D}$ and $\mathcal{E}$. For all i, the decomposition tree $\mathcal{D}$ has $\langle\mathcal{R}_i\rangle$ inputs connected to outputs from the non-root LUTs $\mathcal{N}_i$. In addition, the output of every LUT, except the root LUT, is connected to the input of another LUT in the tree. Therefore the total number of used inputs in $\mathcal{D}$ is $\sum\langle\mathcal{R}_i\rangle + |\mathcal{D}| - 1$. The total number of inputs available is $K|\mathcal{D}|$. If $\mathcal{D}$ is pin limited, then by definition all unused inputs in $\mathcal{D}$ are at the root LUT. The number of LUTs in $\mathcal{D}$ is the minimum integer such that the number of inputs available is greater than or equal to the number of inputs used. Therefore

$$K|\mathcal{D}| \geq \sum\langle\mathcal{R}_i\rangle + |\mathcal{D}| - 1$$

Using the notation $\lceil x\rceil$ to indicate the smallest integer greater than or equal to x, the number of LUTs in $\mathcal{D}$ can be expressed as

$$|\mathcal{D}| = \lceil(\sum\langle\mathcal{R}_i\rangle - 1)/(K-1)\rceil$$

The sub-circuit $\mathcal{E}$ is also a tree of K-input LUTs, and for all i it also has $\langle\mathcal{R}_i\rangle$ inputs connected to outputs from the non-root LUTs $\mathcal{N}_i$. Using the argument that the output of every LUT in $\mathcal{E}$, except the root LUT, is connected to the input of another LUT in $\mathcal{E}$, it follows that

$$|\mathcal{E}| \geq \lceil(\sum\langle\mathcal{R}_i\rangle - 1)/(K-1)\rceil$$

Therefore $|\mathcal{E}| \geq |\mathcal{D}|$, and all that remains to prove Lemma 5.2 is to show that $\langle\mathcal{E}\rangle \geq \langle\mathcal{D}\rangle$ whenever $|\mathcal{E}| = |\mathcal{D}|$.

If $|\mathcal{E}| = |\mathcal{D}|$, then the total number of inputs available in $\mathcal{E}$, is the same as the number of inputs in $\mathcal{D}$. Counting the inputs connected to the LUTs N_i, for all i, and

the inputs connected to the outputs of other LUTs in each tree it follows that the number of used inputs in $\mathcal{E}$ is greater than or equal to the number of used inputs in $\mathcal{D}$. Therefore the number of unused inputs in $\mathcal{E}$ is no greater than the number of unused inputs in $\mathcal{D}$. The number of used inputs at the root LUT of $\mathcal{E}$ cannot be smaller than K less the total number of unused inputs in $\mathcal{E}$, and by definition, all unused inputs in $\mathcal{D}$ are at the root LUT of $\mathcal{D}$. Since the number of unused inputs in $\mathcal{E}$ is no greater than the number of unused inputs in $\mathcal{D}$ it follows that $\langle\mathcal{E}\rangle \geq \langle\mathcal{D}\rangle$ if $|\mathcal{E}| = |\mathcal{D}|$. Therefore, Lemma 5.2 is true if $\mathcal{D}$ is pin limited.

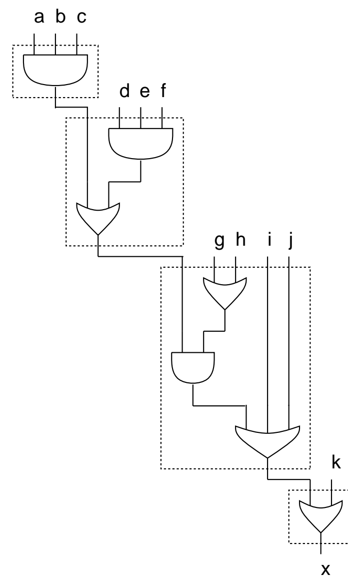
The decomposition tree $\mathcal{D}$ must be either bin limited or pin limited, and in either case Lemma 5.2 is true. It therefore follows by the earlier inductive argument that Theorem 5.1 is true.

5.2 Circuits with Fanout

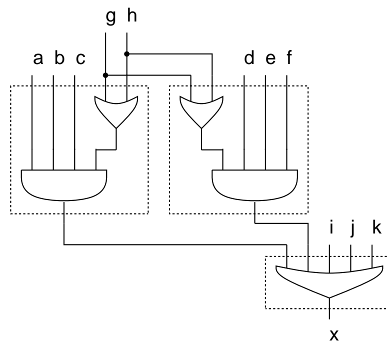
Since technology mapping addresses decomposition and covering, the above proof has assumed that the circuit implementing a fanout-free tree must itself be a tree of LUTs. It is possible, however, that the optimal tree of LUTs is not the smallest circuit implementing the Boolean function represented by the original network. Restructuring the original network to introduce fanout, while implementing the same function, may permit a superior circuit. For example, consider the circuit shown in Figure 5.7a. This 4-LUT circuit is the optimal tree of 5-input LUTs implementing the underlying tree. However, if the underlying tree is restructured, as shown in Figure 5.7b, then the same function can be implemented using 3 LUTs.

5.3 Summary

This chapter has shown that the area algorithm, described in Chapter 3, constructs an optimal tree of LUTs implementing a network that is a tree, for values of $K \leq 5$. The following chapter presents a similar result for the delay algorithm.



a) Optimal Tree of LUTs, 4 LUTs



b) Restructured circuit, 3 LUTs

Figure 5.7: Restructuring a Tree to Introduce Fanout

Chapter 6

Delay Optimality

This chapter presents a proof that the delay algorithm constructs a minimum-depth tree of K-input LUTs implementing a network that is a fanout-free tree if K is less than or equal to 6. The following notation is used in this chapter: Script letters, such as $\mathcal{A}$ are used to represent circuits of K-input LUTs. The number of LUTs in the circuit $\mathcal{A}$ is denoted by $|\mathcal{A}|$ and the maximum depth of any LUT in $\mathcal{A}$ is denoted by $\downarrow\mathcal{A}\downarrow$. If $\mathcal{A}$ is a single-output circuit, then $\langle\mathcal{A}\rangle$ is the the number of inputs used at the root LUT of $\mathcal{A}$. The *depth-optimal* circuit implementing a single-output network is defined as follows:

> The circuit $\mathcal{A}$ is depth-optimal if and only if for all circuits $\mathcal{B}$ implementing the same function, $\downarrow\mathcal{B}\downarrow \geq \downarrow\mathcal{A}\downarrow$ and $\langle\mathcal{B}\rangle \geq \langle\mathcal{A}\rangle$ whenever $\downarrow\mathcal{B}\downarrow = \downarrow\mathcal{A}\downarrow$

The remainder of this chapter will prove the following theorem:

Theorem 6.1

> Given an original network that is a fanout-free tree, a delay algorithm constructs a depth-optimal tree of K-input LUTs implementing the network for values of $K \leq 6$.

As described in Chapter 4, the delay algorithm traverses the original tree beginning at the leaf nodes and proceeding to the root node. At every node, a circuit

implementing the sub-tree rooted at that node is constructed. Section 6.1 will prove the following lemma:

Lemma 6.2

> At each non-leaf node, the delay algorithm constructs the depth-optimal tree of LUTs implementing that node, for values of $K \leq 6$, if the circuits implementing its fanin nodes are depth-optimal.

Theorem 6.1 is proved by induction using Lemma 6.2. The basis of the induction is the circuit constructed by the delay algorithm at each leaf node. This circuit consists of a single LUT implementing a buffer. The depth of this buffer LUT is one greater than the depth of the leaf node, and only a single input of the LUT is used. No other circuit implementing the leaf node can have lower depth or use fewer inputs at its root LUT. Therefore, this buffer LUT is the depth-optimal circuit implementing the leaf node.

Since the circuits implementing the leaf nodes are depth-optimal, it follows by induction from Lemma 6.2 that the circuit constructed at every node, including the root node of the tree, is depth-optimal. Therefore Theorem 6.1 is true. The remainder of this chapter presents a proof of Lemma 6.2.

6.1 Outline for Proof of Lemma 6.2

To prove Lemma 6.2, the following notation is introduced: Let $\mathcal{A}$ be the circuit constructed by the delay algorithm and let $\mathcal{B}$ be an arbitrary tree of LUTs implementing the same function. By definition, the circuit $\mathcal{A}$ is depth-optimal if $\downarrow\mathcal{B}\downarrow \geq \downarrow\mathcal{A}\downarrow$ and $\langle\mathcal{B}\rangle \geq \langle\mathcal{A}\rangle$ whenever $\downarrow\mathcal{B}\downarrow = \downarrow\mathcal{A}\downarrow$.

Both circuits $\mathcal{A}$ and $\mathcal{B}$ consist of a series of strata, where each stratum contains all LUTs at a given depth. The proof proceeds by transforming the circuit $\mathcal{B}$ without changing its function, or increasing $\downarrow\mathcal{B}\downarrow$ or $\langle\mathcal{B}\rangle$. To prove Lemma 6.2 the number of LUTs at a given depth in the transformed circuit $\mathcal{B}$ is compared to the number

of LUTs in the corresponding stratum of the circuit $\mathcal{A}$. Section 6.1.3 will prove the following lemma:

Lemma 6.3

> For values of $K \leq 6$, the number of LUTs at any fixed depth in the transformed circuit $\mathcal{B}$ is greater than or equal to the number of LUTs at the same depth in the circuit $\mathcal{A}$.

If d_{max} is defined as the depth of circuit $\mathcal{A}$, then because the stratum at depth d_{max} in the circuit $\mathcal{A}$ contains exactly one LUT, it follows that the transformed circuit $\mathcal{B}$ contains at least one LUT at depth d_{max}. Therefore $\downarrow\mathcal{B}\downarrow \geq \downarrow\mathcal{A}\downarrow$, and all that remains to prove Lemma 6.2 is to show that $\langle\mathcal{B}\rangle \geq \langle\mathcal{A}\rangle$ if $\downarrow\mathcal{B}\downarrow = \downarrow\mathcal{A}\downarrow$.

The following sections present the details of the proof of Lemma 6.2. Section 6.1.1 introduces notation that describes the circuit $\mathcal{A}$ constructed by the delay algorithm, and Section 6.1.2 describes how the circuit $\mathcal{B}$ is transformed without changing its function, or increasing $\downarrow\mathcal{B}\downarrow$ or $\langle\mathcal{B}\rangle$. Section 6.1.3 proves Lemma 6.3 and then Section 6.1.4 proves Lemma 6.2.

6.1.1 Notation for the Circuit $\mathcal{A}$

This section introduces the notation to describe the non-leaf node being mapped, the depth-optimal circuits implementing its fanin nodes, and the circuit constructed by the delay algorithm. For convenience, the discussion reiterates some of the notation presented in Section 5.1.1 of Chapter 5. Note however, that the decomposition tree constructed by the delay algorithm differs from that constructed by the area algorithm.

Let n be the non-leaf node and let p_1 to p_f be its fanin nodes. Without loss of generality this section assumes that the non-leaf node is an OR node and that its fanin nodes are AND nodes or primary inputs. The case where n is an AND node is the dual of the case considered here, and the first step in the delay algorithm uses DeMorgan's Law and the associative rule to ensure that the fanin nodes of an OR

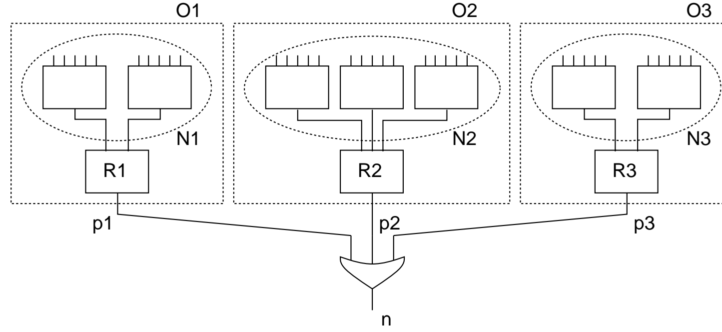


Figure 6.1: The Optimal Fanin Circuits, $\mathcal{O}_i$

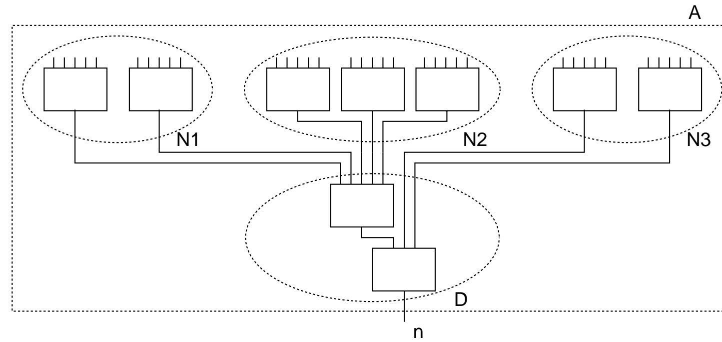


Figure 6.2: The Circuit $\mathcal{A}$

node are either AND nodes, or primary inputs.

Let $\mathcal{O}_i$ be an optimal circuit implementing the fanin node p_i. As illustrated in Figure 6.1 each fanin circuit $\mathcal{O}_i$ consists of a root LUT $\mathcal{R}_i$ and the non-root LUTs $\mathcal{N}_i$. In this figure, LUTs are represented by solid rectangles, each optimal fanin circuit, $\mathcal{O}_i$, is bounded by a dotted rectangle and the non-root LUTs ,$\mathcal{N}_i$, are bounded by a dotted ellipse. In this example, $\downarrow\mathcal{O}_1\downarrow = 2$, $\downarrow\mathcal{O}_2\downarrow = 2$, $\downarrow\mathcal{O}_3\downarrow = 2$, $\langle\mathcal{O}_1\rangle = 2$, $\langle\mathcal{O}_2\rangle = 3$, and $\langle\mathcal{O}_3\rangle = 2$. Note that $\downarrow\mathcal{R}_i\downarrow = \downarrow\mathcal{O}_i\downarrow$, and that $\langle\mathcal{R}_i\rangle = \langle\mathcal{O}_i\rangle$.

The circuit $\mathcal{A}$, constructed by the delay algorithm, is illustrated in Figure 6.2. This circuit consists of the decomposition tree, $\mathcal{D}$, and the non-root LUTs, $\mathcal{N}_i$, for all i from 1 to f. As described in Chapter 4, the decomposition tree is constructed as a series of strata $\{\mathcal{S}_d\}$, for d from d_{min}, the minimum value of $\downarrow\mathcal{R}_i\downarrow$, to $d_{max} = \downarrow\mathcal{A}\downarrow$. The first step in constructing the stratum $\mathcal{S}_d$ uses the FFD bin packing algorithm to pack the fanin LUTs $\{\mathcal{R}_i\}$, for all i where $\downarrow\mathcal{R}_i\downarrow = d$, into K-input LUTs. Each fanin LUT $\mathcal{R}_i$ corresponds to a box of size $\langle\mathcal{R}_i\rangle$, and the LUTs in the stratum correspond

to bins of capacity K. After the first step is completed the LUTs in stratum $\mathcal{S}_d$ can be thought of as the set of bins, produced by the FFD algorithm, containing a box of size $\langle \mathcal{R}_i \rangle$, for all i where $\downarrow\mathcal{R}_i\downarrow = d$.

The second step in constructing the decomposition tree $\mathcal{D}$ proceeds from the uppermost stratum, $\mathcal{S}_{d_{min}}$, to the deepest stratum connecting the outputs of LUTs in stratum $\mathcal{S}_d$ to unused inputs in stratum $\mathcal{S}_{d+1}$. If there are insufficient unused inputs, then new LUTs are added to stratum $\mathcal{S}_{d+1}$. Connecting the output of each LUT in stratum $\mathcal{S}_d$ to an unused input in stratum $\mathcal{S}_{d+1}$ corresponds to packing $|\mathcal{S}_d|$ extra unit boxes into the set of bins in the stratum $\mathcal{S}_{d+1}$. These unit boxes are added on a first fit basis after the other boxes have been packed into stratum $\mathcal{S}_{d+1}$. Therefore, for $d > d_{min}$ the LUTs in the stratum $\mathcal{S}_d$ are the set of bins, produced by the FFD algorithm, containing $|\mathcal{S}_{d-1}|$ unit boxes and a box of size $\langle \mathcal{R}_i \rangle$, for all i where $\downarrow\mathcal{R}_i\downarrow = d$.

The root LUT of the decomposition tree $\mathcal{D}$ is the root LUT of the circuit $\mathcal{A}$. Therefore, $\downarrow\mathcal{A}\downarrow = \downarrow\mathcal{D}\downarrow$, and $\langle \mathcal{A} \rangle = \langle \mathcal{D} \rangle$. In the example shown in Figure 6.2, $\downarrow\mathcal{D}\downarrow = 3$, and $\langle \mathcal{D} \rangle = 3$.

6.1.2 Transforming the Circuit $\mathcal{B}$

This section describes how the circuit $\mathcal{B}$ is transformed, without changing its function, or increasing $\downarrow\mathcal{B}\downarrow$ or $\langle \mathcal{B} \rangle$. The transformations are similar to those described in Section 5.1.2 of Chapter 5, but they preserve $\downarrow\mathcal{B}\downarrow$ rather than $|\mathcal{B}|$. The next section proves that the end result is a circuit $\mathcal{B}$ where every stratum in $\mathcal{B}$ has at least as many LUTs as the corresponding stratum in the circuit $\mathcal{A}$.

The objective of the transformations is to incorporate in the circuit $\mathcal{B}$ the non-root LUTs $\mathcal{N}_i$, for all i, and to connect the $\langle \mathcal{R}_i \rangle$ outputs of the sub-circuit $\mathcal{N}_i$ to one LUT at depth $\downarrow\mathcal{R}_i\downarrow$. The transformations must not change the function of the circuit $\mathcal{B}$, or increase $\downarrow\mathcal{B}\downarrow$ or $\langle \mathcal{B} \rangle$. Note that in the circuit $\mathcal{A}$ the $\langle \mathcal{R}_i \rangle$ outputs of the non-root LUTs $\mathcal{N}_i$ are also connected to the LUT $\mathcal{R}_i$ at depth $\downarrow\mathcal{R}_i\downarrow$. This observation is the key to the proof of Lemma 6.3.

The node n is an OR node, and its fanin nodes are AND nodes. The circuit $\mathcal{B}$ is

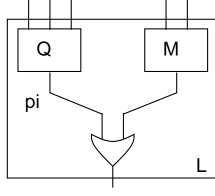


Figure 6.3: Decomposition of $\mathcal{L}$

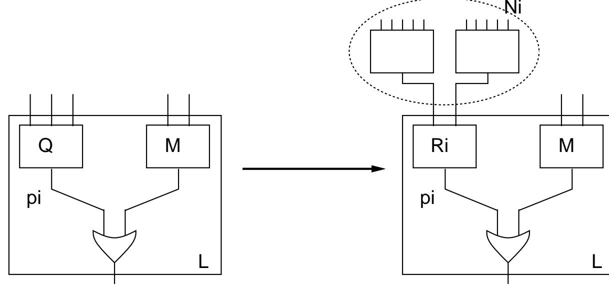


Figure 6.4: Modification when $\downarrow\mathcal{L}\downarrow = \downarrow\mathcal{R}_i\downarrow$

assumed to be a tree of LUTs, and using the argument described in Section 5.1.2 of Chapter 5 there must be a sub-tree of LUTs in $\mathcal{B}$ that implements the function

$$p_i + \sum_{j \in \lambda} p_j$$

or its complement, for some subset λ of the fanin nodes. In addition, the root LUT, $\mathcal{L}$, of this tree can be decomposed into the disjoint functions $\mathcal{Q}$ and $\mathcal{M}$ as illustrated in Figure 6.3. The output of $\mathcal{Q}$ is the function p_i, the output of $\mathcal{M}$ is the function $\sum p_j$, for $j \in \lambda$, and the output of $\mathcal{L}$ is the function $\mathcal{Q} + \mathcal{M}$.

It is given that $\mathcal{O}_i$ is an optimal circuit implementing the fanin node p_i. Therefore, $\downarrow\mathcal{L}\downarrow \geq \downarrow\mathcal{O}_i\downarrow$ and if $\downarrow\mathcal{L}\downarrow = \downarrow\mathcal{O}_i\downarrow$ then $\langle\mathcal{Q}\rangle \geq \langle\mathcal{O}_i\rangle$. The case where $\downarrow\mathcal{L}\downarrow = \downarrow\mathcal{O}_i\downarrow$ is considered first, followed by the case where $\downarrow\mathcal{L}\downarrow > \downarrow\mathcal{O}_i\downarrow$.

Case 1: $\downarrow\mathcal{L}\downarrow = \downarrow\mathcal{O}_i\downarrow$

If $\downarrow\mathcal{L}\downarrow = \downarrow\mathcal{O}_i\downarrow$ then $\langle\mathcal{Q}\rangle \geq \langle\mathcal{O}_i\rangle$, and the function $\mathcal{Q}$ can be replaced with the function $\mathcal{R}_i$, as illustrated in Figure 6.4, without increasing $\downarrow\mathcal{L}\downarrow$, or $\langle\mathcal{L}\rangle$. To generate the inputs for $\mathcal{R}_i$ the non-root LUTs $\mathcal{N}_i$ are added to $\mathcal{B}$.

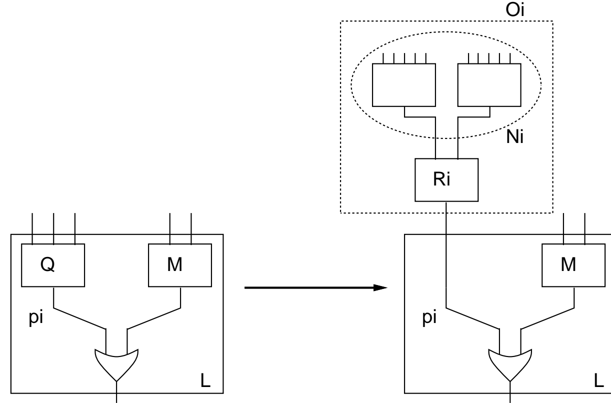


Figure 6.5: Modification when $\downarrow\mathcal{L}\downarrow > \downarrow\mathcal{R}_i\downarrow$

Case 2: $\downarrow\mathcal{L}\downarrow > \downarrow\mathcal{O}_i\downarrow$

If $\downarrow\mathcal{L}\downarrow > \downarrow\mathcal{O}_i\downarrow$ then the function $\mathcal{Q}$ can be replaced by a single input connected to the output of the circuit $\mathcal{O}_i$ as illustrated in Figure 6.5, without increasing $\downarrow\mathcal{L}\downarrow$ or $\langle\mathcal{L}\rangle$. Note that the circuit $\mathcal{O}_i$ includes the root LUT $\mathcal{R}_i$ and the non-root LUTs $\mathcal{N}_i$.

In either case, the transformation does not change the function of $\mathcal{L}$, and does not increase $\downarrow\mathcal{L}\downarrow$ or $\langle\mathcal{L}\rangle$. Therefore these transformations will not change the function of $\mathcal{B}$, or increase $\downarrow\mathcal{B}\downarrow$ or $\langle\mathcal{B}\rangle$. The end result of these transformations is that the circuit $\mathcal{B}$ includes the non-root LUTs $\mathcal{N}_i$, for all i, and all $\langle\mathcal{R}_i\rangle$ outputs of $\mathcal{N}_i$ are connected to one LUT at depth $\downarrow\mathcal{R}_i\downarrow$.

One final transformation to the circuit $\mathcal{B}$ is required before the following section can deduce that the number of LUTs at any fixed depth in $\mathcal{B}$ is greater than or equal to the number of LUTs at the same depth in the circuit $\mathcal{A}$. Whenever the output of a LUT at depth d_{src} is connected to the input of a LUT at depth $d_{dst} > d_{src+1}$ a chain of $(d_{dst} - d_{src} - 1)$ buffer LUTs is introduced between the output of the source LUT and the input of the destination LUT. This transformation ensures that if the output of a LUT is connected to the input of another LUT, then the depth of the destination LUT is exactly one greater than the depth of the source LUT. Note that this does not change the function $\mathcal{B}$, or increase $\downarrow\mathcal{B}\downarrow$, or $\langle\mathcal{B}\rangle$.

6.1.3 Proof of Lemma 6.3

This section shows that, for values of $K \leq 6$, the number of LUTs at any fixed depth in the transformed circuit $\mathcal{B}$ is greater than or equal to the number of LUTs at the same depth in the circuit $\mathcal{A}$. From this observation it is possible to deduce that $\mathcal{A}$ is a depth-optimal tree of LUTs implementing the node n.

After the transformations described in Section 6.1.2, the circuits $\mathcal{A}$ and $\mathcal{B}$ both include the non-root LUTs $\mathcal{N}_i$, for all i. The first step in the proof of Lemma 6.3 eliminates these non-root LUTs from the circuits $\mathcal{A}$ and $\mathcal{B}$. Note that this does not affect the comparison of the number of LUTs at any given depth. After eliminating the non-root LUTs from the circuit $\mathcal{A}$, what remains is the decomposition tree $\mathcal{D}$. The LUTs that remain from the circuit $\mathcal{B}$ are referred to as the circuit $\mathcal{E}$.

Recall that the decomposition tree $\mathcal{D}$ consists of a series of strata $\{\mathcal{S}_d\}$ for all d from d_{min} to $d_{max} = \downarrow\!\mathcal{A}\!\downarrow$. The notation $\mathcal{T}_d$ is introduced to represent the LUTs at depth d in the circuit $\mathcal{E}$.

The LUTs in the stratum $\mathcal{S}_{d_{min}}$ can be thought of as a set of bins containing a box of size $\langle \mathcal{R}_i \rangle$ for all i where $\downarrow\!\mathcal{R}_i\!\downarrow\; = d_{min}$. The LUTs in $\mathcal{T}_{d_{min}}$ correspond to a set of bins containing the same boxes. The bins in the stratum $\mathcal{S}_{d_{min}}$ are produced by the FFD algorithm, and Appendix B shows that the FFD bin packing algorithm packs any set of integer sized boxes into the minimum number of bins of capacity K, for integer values of $K \leq 6$. Therefore, for $K \leq 6$, $|\mathcal{T}_{d_{min}}| \geq |\mathcal{S}_{d_{min}}|$.

For $d > d_{min}$ the LUTs in stratum $\mathcal{S}_d$ in $\mathcal{D}$ correspond to a set of bins containing $|\mathcal{S}_{d-1}|$ unit boxes, and a box of size $\langle \mathcal{R}_i \rangle$, for all i where $\downarrow\!\mathcal{R}_i\!\downarrow\; = d$. The inputs to a LUT in stratum $\mathcal{T}_d$ are connected to either outputs from $\mathcal{N}_i$, or the outputs of LUTs in stratum $\mathcal{T}_{d-1}$. Therefore, the LUTs in stratum $\mathcal{T}_d$ correspond to a set of bins containing $|\mathcal{T}_{d-1}|$ unit boxes, and a box $\langle \mathcal{R}_i \rangle$, for all i where $\downarrow\!\mathcal{R}_i\!\downarrow\; = d$. If $|\mathcal{T}_{d-1}| \geq |\mathcal{S}_{d-1}|$ then the boxes in the stratum $\mathcal{S}_d$ are a subset of boxes in stratum $\mathcal{T}_d$. Since the set of bins in stratum $\mathcal{S}_d$ are produced by the FFD algorithm, and the FFD algorithm is optimal for values of $K \leq 6$, it can be deduced that $|\mathcal{T}_d| \geq |\mathcal{S}_d|$ if $|\mathcal{T}_{d-1}| \geq |\mathcal{S}_{d-1}|$. It has already been shown that $|\mathcal{T}_{d_{min}}| \geq |\mathcal{S}_{d_{min}}|$, and therefore by induction, for all $d \geq d_{min}$, $|\mathcal{T}_d| > |\mathcal{S}_d|$. Therefore Lemma 6.3 is true.

6.1.4 Proof of Lemma 6.2

This section uses Lemma 6.3 to prove that the circuit $\mathcal{A}$ is depth-optimal. In the delay algorithm, the procedure that connects the strata $\{\mathcal{S}_d\}$ to form the decomposition tree $\mathcal{D}$ terminates when $|\mathcal{S}_{d_{max}}| = 1$, and therefore $|\mathcal{T}_{d_{max}}| \geq 1$. From Lemma 6.3 it follows that $\mathcal{B}$ contains at least one LUT at depth d_{max}, and therefore $\downarrow\mathcal{B}\downarrow \geq \downarrow\mathcal{A}\downarrow$

To prove that $\mathcal{A}$ is optimal, all that remains is to show that $\langle\mathcal{B}\rangle \geq \langle\mathcal{A}\rangle$ whenever $\downarrow\mathcal{B}\downarrow = \downarrow\mathcal{A}\downarrow$. The LUTs in stratum $\mathcal{S}_{d_{max}}$ correspond to bins containing $|\mathcal{S}_{d_{max}-1}|$ unit boxes, and a box of size $\langle\mathcal{R}_i\rangle$ for all i where $\downarrow\mathcal{R}_i\downarrow = d_{max}$. Since the stratum $\mathcal{S}_{d_{max}}$ consists of a single LUT it follows that $\langle\mathcal{S}_{d_{max}}\rangle = |\mathcal{S}_{d_{max}-1}| + \sum\langle\mathcal{R}_i\rangle$ for all i where $\downarrow\mathcal{R}_i\downarrow = d_{max}$.

The LUTs in stratum $\mathcal{T}_{d_{max}}$ correspond to bins containing $|\mathcal{T}_{d_{max}-1}|$ unit boxes, and a box of size $\langle\mathcal{R}_i\rangle$, for all i where $\downarrow\mathcal{R}_i\downarrow = d_{max}$. If $\downarrow\mathcal{B}\downarrow = \downarrow\mathcal{A}\downarrow$ then the stratum $\mathcal{T}_{d_{max}}$ consists of a single LUT, and $\langle\mathcal{T}_{d_{max}}\rangle = |\mathcal{T}_{d_{max}-1}| + \sum\langle\mathcal{R}_i\rangle$, for all i where $\downarrow\mathcal{R}_i\downarrow = d_{max}$. It is known that $|\mathcal{T}_{d_{max}-1}| \geq |\mathcal{S}_{d_{max}-1}|$, and therefore $\langle\mathcal{T}_{d_{max}}\rangle \geq \langle\mathcal{S}_{d_{max}}\rangle$ if $\downarrow\mathcal{B}\downarrow = \downarrow\mathcal{A}\downarrow$. Since $\langle\mathcal{A}\rangle = \langle\mathcal{S}_{d_{max}}\rangle$ and $\langle\mathcal{B}\rangle = \langle\mathcal{T}_{d_{max}}\rangle$, it follows that $\langle\mathcal{B}\rangle \geq \langle\mathcal{A}\rangle$ if $\downarrow\mathcal{B}\downarrow = \downarrow\mathcal{A}\downarrow$.

Thus, for values of $K \leq 6$, $\downarrow\mathcal{B}\downarrow \geq \downarrow\mathcal{A}\downarrow$, and $\langle\mathcal{B}\rangle \geq \langle\mathcal{A}\rangle$ if $\downarrow\mathcal{B}\downarrow = \downarrow\mathcal{A}\downarrow$. Therefore, the delay algorithm constructs a depth-optimal tree of LUTs implementing a non-leaf node, if the circuits implementing its fanin nodes are depth-optimal. Therefore Lemma 6.2 is true, and Theorem 6 follows by the induction described at the beginning of this chapter.

6.2 Summary

The delay algorithm, described in Chapter 4, maps a general network into a circuit of K-input LUTs by first partitioning the network into a forest of trees, and then mapping each tree separately. This chapter has shown that the algorithm used to map each tree, produces a depth-optimal tree of LUTs implementing that tree, for values of $K \leq 6$. The following chapter presents experimental results for the both the area and delay algorithms.

Chapter 7

Experimental Evaluation

The area and delay algorithms described in Chapters 3 and 4 have been implemented in a program called Chortle. This chapter presents experimental results for Chortle. The purpose of these experiments is to investigate the effectiveness of the divide and conquer strategy which partitions the original network into a forest of trees and then separately maps each tree. These experiments also evaluate the effectiveness of the optimizations that exploit reconvergent paths and the replication of logic at fanout nodes. In addition, these results are compared with other LUT technology mappers. The following section presents the experimental results for the area algorithm, and Section 7.2 presents results for the delay algorithm.

7.1 Results for the Area Algorithm

7.1.1 Circuits of 5-input LUTs

This section presents the results for a series of experiments where the area algorithm maps 29 networks from the MCNC two-level and multi-level logic synthesis benchmark suite [Yang91] into circuits of 5-input LUTs. The goal for these experiments is to reduce the number of LUTs in the final circuits implementing each network.

The first step in the experimental procedure is logic optimization using the misII logic synthesis system [Bray87]. The misII script shown in Figure 7.1 is used to

```
rl BLIF

sw
el 5
sim1 *
asb

gkx -abt 30
asb; sw
gcx -bt 30
asb; sw

gkx -abt 10
asb; sw
gcx -bt 10
asb; sw

gkx -ab
asb; sw
gcx -b
asb; sw

el 0
gd *

wl OPT

rl OPT
simplify
sweep
we EQN
```

Figure 7.1: misII script

optimize the original network "BLIF" into the optimized network "EQN". This script is the standard misII script [Bray87] with the addition of the commands "simplify" and "sweep" and a format change at the end of the script. The intermediate step of writing to and reading from the file "OPT" alters the network order, and is retained for historical reasons.

After logic optimization by misII, the optimized networks are mapped into circuits of 5-input LUTs using Chortle. Five sets of experiments were performed, using the following optimization options:

(-A) basic area algorithm

(-Ar) area algorithm with exhaustive reconvergent search

(-Af) area algorithm with *Root Replication*

(-Arf) with exhaustive reconvergent search, and Root Replication

(-Asf) with Maximum Share Decreasing (MSD) and Root Replication

The number of LUTs in the circuit produced by each option and the execution time on a SparcStation-IPC are recorded in Table 7.1.

Separately, the exhaustive reconvergent (-Ar) and Root Replication (-Af) optimizations never produce circuits containing more LUTs than the basic area algorithm (-A). In total, the (-Ar) circuits contain 3.7% fewer LUTs than the (-A) circuits and the (-Af) circuits contain 3.5% fewer LUTs than the (-A) circuits. Note that there is only a small increase in execution time for the (-Ar) circuits. This indicates that each tree (leaf-DAG) resulting from the partitioning of the original circuit into a forest of trees contains a limited number of reconvergent paths. The total execution time for the (-Af) circuits is an order of magnitude greater than the execution time for the (-A) circuits. This is a result of the Root Replication optimization repeatedly mapping trees to determine when to replicate logic at a fanout node.

The circuits produced by combining the exhaustive reconvergent search and the Root Replication optimizations (-Arf) are never worse than the (-Ar) circuits and the (-Af) circuits. In total, the (-Arf) circuits contain 15% fewer LUTs than the

network	-A		-Ar		-Af		-Arf		-Asf	
	LUTs	sec. [1]	LUTs	sec. [1]	LUTs	sec. [1]	LUTs	sec. [1]	LUTs	sec. [1]
5xp1	34	0.2	31	0.2	34	1.2	27	1.4	27	1.2
9sym	69	0.3	65	0.4	67	21.2	62	27.6	60	22.5
9symml	63	0.3	59	0.3	62	20.6	54	28.3	55	21.8
C499	166	0.7	164	0.9	158	3.3	70	5.8	70	5.1
C5315	666	3.5	611	3.8	653	39.4	513	54.2	512	43.4
C880	115	0.6	110	0.7	112	4.7	86	5.7	86	5.3
alu2	131	0.6	121	0.7	125	23.6	116	27.6	116	25.2
alu4	238	1.1	218	1.3	227	38.6	191	51.6	194	41.9
apex2	123	0.6	123	0.6	121	18.3	120	19.3	120	19.2
apex4	602	2.3	599	2.7	578	47.9	561	254.8	562	63.0
apex6	232	1.2	219	1.5	230	5.4	212	8.5	212	5.7
apex7	72	0.4	71	0.4	71	1.2	63	1.3	63	1.2
b9	39	0.1	39	0.2	38	0.4	38	0.5	38	0.4
bw	65	0.2	62	0.3	61	1.1	54	1.7	54	1.3
clip	41	0.2	37	0.2	38	1.8	33	2.5	33	1.8
count	47	0.2	45	0.2	40	0.6	31	0.8	31	0.7
des	1073	5.6	1060	6.1	1050	68.1	945	83.1	947	75.2
duke2	138	0.5	136	0.5	127	2.8	122	3.2	122	3.1
e64	95	0.3	95	0.3	80	0.6	80	0.6	80	0.6
f51m	41	0.2	39	0.2	40	1.8	35	2.0	35	1.8
misex1	20	0.1	20	0.1	19	0.3	19	0.3	19	0.3
misex2	35	0.1	35	0.1	31	0.2	30	0.3	30	0.3
misex3	160	0.7	156	0.8	157	10.6	142	14.7	139	11.5
rd73	42	0.2	35	0.2	38	3.1	30	4.3	30	3.3
rd84	76	0.3	76	0.3	73	8.1	70	8.5	70	8.4
rot	215	1.0	203	1.1	206	5.1	189	5.6	189	5.6
sao2	48	0.3	42	0.2	45	2.2	38	2.6	38	2.3
vg2	24	0.1	24	0.1	23	0.2	22	0.2	22	0.2
z4ml	9	0.0	9	0.1	9	0.2	5	0.3	5	0.2
total	4679	21.9	4504	24.5	4513	332.6	3958	617.3	3959	372.5

[1] execution time on a SparcStation-IPC (13.5 Specmarks)

Table 7.1: Area Algorithm 5-input LUT Results

(-A) circuits. This reduction exceeds the sum of the reductions realized for the (-Ar) circuits and (-Af) circuits. This indicates that the replication of logic at fanout nodes exposes additional reconvergent paths that can be exploited by the exhaustive reconvergent search. The opportunity to exploit these additional reconvergent paths can improve the final circuit, but it also increases the computational cost of the exhaustive reconvergent search. The total execution time for the (-Arf) circuits is almost twice that of the (-Af) circuits.

The computational cost of exploiting reconvergent paths can be reduced by using the MSD algorithm instead of the exhaustive reconvergent search. The results in Table 7.1 indicate that the (-AsF) results are similar to the (-Arf) results. For 4 of the 29 networks, the (-Asf) circuits contain more LUTs than the (-Arf) circuits, but for 3 networks they contain fewer LUTs. In total, the (-Asf) circuits contain only 1 more LUT than the (-Arf) circuits. The MSD algorithm is able to occasionally outperform the exhaustive reconvergent search because the exhaustive search only finds a locally optimal solution. For some networks, the local decisions made by the MSD algorithm can lead to a superior global solution. The total execution time for the (-Asf) circuits is similar to the execution time for (-Af) circuits, indicating the efficiency of the MSD algorithm.

7.1.2 Circuits of Xilinx 3000 CLBs

This section presents experimental results for Chortle mapping networks into Xilinx 3000 CLBs. As described in Chapter 2, each Xilinx CLB can implement any single function of 5 variables, or any two 4-input functions that together have at most 5 distinct inputs. To realize a circuit of CLBs, a network can be first mapped into a circuit of 5-input LUTs, and then the Maximum Cardinality Matching (MCM) strategy described in Section 2.3.2 of Chapter 2, can be used to pair single-output functions into two-output CLBs.

Five sets of experiments were performed on the 29 networks from the previous section, using the options (-A), (-Ar), (-Af), (-Arf), and (-Asf) to optimize the LUT circuit. Table 7.2 presents the number of CLBs in the circuits produced by these

network	-A		-Ar		-Af		-Arf		-Asf	
	CLBs	sec. [1]	CLBs	sec. [1]	CLBs	sec. [1]	CLBs	sec. [1]	CLBs	sec. [1]
5xp1	23	0.2	22	0.2	23	1.2	21	1.5	21	1.2
9sym	52	0.4	53	0.4	54	20.9	52	27.3	52	22.4
9symml	50	0.3	49	0.4	49	20.3	47	28.5	47	21.5
C499	84	2.0	84	2.2	85	4.3	50	6.0	50	5.1
C5315	413	61.1	401	53.1	416	71.2	389	87.8	387	73.7
C880	74	0.9	76	0.9	76	4.9	75	5.8	75	5.2
alu2	94	1.0	94	0.9	96	24.2	93	28.0	93	25.1
alu4	166	2.9	157	2.8	165	40.2	152	52.6	153	42.1
apex2	94	0.8	94	0.9	96	18.1	95	19.9	95	19.2
apex4	456	30.6	454	29.0	463	70.0	449	275.0	448	83.3
apex6	162	2.4	159	2.4	176	6.6	173	9.4	173	6.5
apex7	48	0.5	48	0.5	49	1.2	45	1.3	45	1.2
b9	26	0.2	27	0.2	26	0.5	26	0.5	26	0.5
bw	39	0.3	39	0.3	39	1.2	38	1.7	38	1.3
clip	27	0.2	25	0.2	28	1.7	25	2.5	24	1.8
count	32	0.2	31	0.3	32	0.6	27	0.8	27	0.7
des	714	154.5	713	116.4	806	195.0	758	175.2	758	187.2
duke2	88	1.0	88	1.0	92	3.2	92	3.4	92	3.3
e64	48	0.7	48	0.6	54	0.7	54	0.7	54	0.7
f51m	29	0.2	29	0.2	28	1.8	26	2.0	26	1.8
misex1	13	0.1	13	0.1	13	0.3	13	0.3	13	0.3
misex2	22	0.1	22	0.1	25	0.2	25	0.3	25	0.3
misex3	120	1.3	121	1.3	123	11.1	119	15.0	117	11.9
rd73	29	0.2	27	0.2	30	3.2	24	4.3	23	3.3
rd84	52	0.4	52	0.4	54	8.0	50	8.6	50	8.5
rot	135	2.8	131	2.4	141	6.3	130	6.6	130	6.3
sao2	37	0.2	34	0.3	37	2.2	33	2.6	33	2.3
vg2	20	0.1	20	0.1	19	0.2	19	0.2	19	0.2
z4ml	5	0.0	5	0.1	5	0.2	4	0.2	4	0.2
total	3152	265.6	3116	217.9	3300	519.5	3104	768	3098	537.1

[1] execution time on a SparcStation-IPC (13.5 Specmarks)

Table 7.2: Area Algorithm Xilinx 3000 CLBs Results

experiments, and the execution time on a SparcStation-IPC. Note that the execution time for each circuit includes the time to map the network into a circuit of 5-input LUTs and the time to pair these LUTs into CLBs.

The first observation from Table 7.2 is that optimizations that reduced the number of 5-input LUTs in the previous section do not always reduce the number of two-output CLBs. In the previous section the (-Ar) circuits never contained more LUTs than the (-A) circuits. In Table 7.2, 12 of the (-Ar) circuits contain fewer CLBs than the (-A) circuits, but for 4 networks the (-Ar) circuit contains more CLBs. In total, the (-Ar) circuits uses 1.1% fewer CLBs than the (-A) circuits. Note also that the total execution time for the (-Ar) circuits is less than the execution time for the (-A) circuits. The execution time for CLB circuits is dominated by the time taken by the MCM algorithm. Even though it takes more time to produce the (-Ar) LUT circuits than the (-A) LUT circuits, using these smaller (-Ar) circuits reduces the size of the MCM problem and reduces the overall execution time.

The next observation is that the Root Replication optimization generally increases the number of CLBs in the final circuit, even though it reduces the number of LUTs. For 18 of the 29 networks, the (-Af) circuit contains more CLBs than the (-A) circuit, and for 4 of the networks the (-Af) circuit contains fewer CLBs. In total, the (-Af) circuits use 4.7% more CLBs than the (-A) circuits. The replication of logic at fanout nodes generally increases the number of inputs used by the LUTs containing the replicated logic, and makes it more difficult to pair these LUTs into a two-output CLB. The availability of the second CLB output can often eliminate the need for the replication of logic. If a CLB contains a single LUT that implements replicated logic, then the second output of this CLB can often be used to explicitly implement the function of the replicated logic with no additional cost. This second output can be used as an input to all other CLBs that replicate logic at the fanout, and thereby eliminate the need for the replication of logic in these CLBs.

Combining the exhaustive reconvergent search and the Root Replication optimization often reduces the number of CLBs in the final circuit. For 19 of the 29 networks, the (-Arf) circuit contains fewer CLBs than the (-A) circuit, and for 7 of the networks

the (-Arf) circuit contains more CLBs. In total, the (-Arf) circuits use 1.5% fewer CLBs than the (-A) circuits. The reduction in the number of LUTs resulting from the combination of the exhaustive reconvergent search and Root Replication optimization can overcome the disadvantage of the replication of logic at fanout nodes when pairing LUTs into CLBs. Note that the (-Arf) circuits are never worse than the (-Af) circuits. However, the (-Arf) circuits are worse than the (-Ar) circuits for 6 of the 29 networks.

As discussed in the previous section, the computational cost can be reduced by using the MSD algorithm instead of exhaustive reconvergent search. For all networks, the number of CLBs in the (-Asf) circuits differs from the (-Arf) circuits by at most 2 CLBs, and in total, the (-Asf) circuits actually use 6 fewer CLBs than the (-Arf) circuits. The total execution time for (-Asf) circuits is 70% that of the (-Arf) circuits. This reduction in execution time is not as significant as the reduction for LUT circuits presented in the previous sections, because the execution time for CLB circuits is dominated by the time taken by the MCM algorithm.

The above results indicate that it is advantageous to use both outputs of a CLB. To increase the opportunities to pair two functions into one CLB the original network can be first mapped into a circuit of 4-input LUTs and then the MCM strategy can be used to pair these functions into two-output CLBs. The 4-input LUT circuit may contain more LUTs than the 5-input LUT circuit, but may result in an increase in the number of pairs of functions that can be implemented by two-output CLBs, and a net reduction in the number of CLBs in the final circuit. Table 7.3 gives the number of 5-input and 4-input LUTs in the circuits produced using the (-Arf) options, and the number of CLBs in the circuits derived from these intermediate circuits. Comparing the number of CLBs in the circuits derived from 4-input LUT circuits and 5-input LUT circuits, there is no consistent pattern. The circuits derived from the 4-input LUTs contain fewer CLBs than the circuits derived from the 5-input LUTs for 13 of the 29 networks, and for 12 of the networks they contain more CLBs. In total, the results derived from the 4-input LUT circuits contain 0.4% fewer CLBs. The circuits derived from the 4-input LUT circuits are smaller if the increase in the number of

network	5-input, -Arf			4-input, -Arf		
	LUTs	CLBs	sec. [1]	LUTs	CLBs	sec. [1]
5xp1	27	21	1.5	35	19	1.4
9sym	62	52	27.3	80	56	22.8
9symml	54	47	28.5	71	47	24.0
C499	70	50	6.0	91	55	5.3
C5315	513	389	87.8	645	410	370.3
C880	86	75	5.8	110	76	5.6
alu2	116	93	28.0	142	92	27.0
alu4	191	152	52.6	236	154	48.6
apex2	120	95	19.9	148	100	19.3
apex4	561	449	275.0	682	466	105.2
apex6	212	173	9.4	241	161	7.7
apex7	63	45	1.3	73	47	1.3
b9	38	26	0.5	45	25	0.5
bw	54	38	1.7	66	34	1.6
clip	33	25	2.5	41	26	2.1
count	31	27	0.8	39	23	0.7
des	945	758	175.2	1149	724	286.7
duke2	122	92	3.4	144	82	3.4
e64	80	54	0.7	94	47	0.8
f51m	35	26	2.0	40	25	2.0
misex1	19	13	0.3	22	14	0.4
misex2	30	25	0.3	36	22	0.3
misex3	142	119	15.0	186	131	14.9
rd73	30	24	4.3	41	24	3.7
rd84	70	50	8.6	84	50	8.2
rot	189	130	6.6	223	127	7.4
sao2	38	33	2.6	53	34	2.5
vg2	22	19	0.2	27	19	0.2
z4ml	5	4	0.2	6	3	0.2
total	3958	3104	768	4850	3093	974.1

[1] execution time on a SparcStation-IPC (13.5 Specmarks)

Table 7.3: Intermediate Circuits of 4-input and 5-input LUTs

LUTs, compared to the 5-input LUT circuits, is offset by a larger increase in the number of LUTs that can be paired into two-output CLBs. The total execution time to derive CLB circuits from 4-LUT circuits is larger than that from 5-input LUT circuits. The intermediate 4-input LUT circuits are larger than the intermediate 5-input LUT circuits, and therefore the MCM problem is larger. This increases the size of the MCM problem, and since the time taken to solve this problem dominates the execution time, the 4-input LUT approach is slower.

7.1.3 Chortle vs. Mis-pga

This section presents a comparison of the Chortle area algorithm and the two versions of the Mis-pga technology mapper described in Section 2.3.2 of Chapter 2. The original version is referred to as Mis-pga(1) [Murg90], and the improved version is referred to as Mis-pga(2) [Murg91a]. Table 7.4 reports experimental results published in [Murg91a]. In these experiments 27 networks from the MCNC logic synthesis benchmark suite were mapped into circuits of 5-input LUTs. The Chortle results were produced using the Chortle-crf [Fran91a] program which implemented the basic area algorithm, the exhaustive reconvergent search and the Root Replication optimization. To permit a direct comparison of the technology mappers, the Chortle-crf, Mis-pga(1) and Mis-pga(2) experiments all started with the same optimized networks. Chortle-crf was run with the exhaustive reconvergent optimization, and the Root Replication optimizations (-Arf).

For 20 of the 27 networks, the circuits produced by Mis-pga(1) contain more LUTs that the circuits produced by Chortle-crf, and for 3 of the networks the Mis-pga(1) circuits contain fewer LUTs. In total, the Mis-pga(1) circuits contain 20% more LUTs than the Chortle-crf circuits.

Mis-pga(2) incorporates a bin packing strategy for decomposition similar to the strategy introduced in Chortle-crf [Fran91a]. For 20 of the 27 networks, the circuit produced by Mis-pga(2) contains fewer LUTs that the circuit produced by Chortle-crf, and for 2 of the networks the Mis-pga(2) circuit contains more LUTs. In total, the Mis-pga(2) circuits contain 14% fewer LUTs than the Chortle-crf circuits. Note that

network	-Arf		Mis-pga(1)	Mis-pga(2)	
	LUTs	sec. [1]	LUTs	LUTs	sec. [1]
5xp1	28	0.4	31	18	22.4
9sym	59	12.8	72	7	339.7
9symml	44	6.4	56	7	127.2
C499	89	2.6	66	68	1074.4
C880	88	2.2	103	82	546.8
alu2	116	7.1	129	109	773.8
alu4	70	2.5	235	55	887.5
apex2	64	2.9	80	67	388.5
apex4	579	98.9	765	412	198.7
apex6	198	2.9	243	182	243.9
apex7	60	0.6	64	60	18.7
b9	41	0.4	42	39	27.6
bw	39	0.3	39	28	17.3
clip	31	0.7	28	28	58.4
count	31	0.3	31	31	5.8
des	927	35.4	1016	904	3186.3
duke2	111	1.7	128	110	203.7
e64	80	0.3	80	80	14.7
f51m	27	0.4	19	17	14.4
misex1	11	0.1	11	11	2.7
misxe2	28	0.1	35	28	3.4
rd73	16	0.3	19	6	24.0
rd84	35	1.3	40	10	73.7
rot	188	2.7	200	181	282.1
sao2	27	0.5	37	28	41.9
vg2	21	0.1	30	20	7.4
z4ml	7	0.1	8	5	5.0
total	3015	184	3607	2593	8590

[1] execution time on a DEC-5500 (27.3 Specmarks)

Table 7.4: Chortle vs. Mis-pga(1) and Mis-pga(2)

Mis-pga(2) produces significantly smaller circuits than Chortle-crf for networks such as "9sym" that have a large number of reconvergent paths, where Shannon decomposition is particularly effective. Both the Chortle-crf and Mis-pga(2) experiments were run on a DEC-5500, and in total, Mis-pga(2) was 47 times slower than Chortle-crf.

Compared to the (-Arf) results presented in Section 7.1 the Chortle-crf circuits recorded in Table 7.4 use fewer LUTs for 17 of the 27 networks, more LUTs for 6 of the networks, and in total 8.7% fewer LUTs. The optimized networks used as input to Chortle-crf in these experiments differ from those described in Section 7.1, and are the source of this improvement.

7.1.4 Chortle vs. Xmap

This section presents an experimental comparison of the Chortle-crf area algorithm and the Xmap [Karp91] technology mapper described in Section 2.3.5 of Chapter 2. Table 7.4 reports experimental results published in [Murg91a]. In these experiments 27 networks from the MCNC logic synthesis benchmark suite are mapped into circuits of 5-input LUTs. Both the Chortle-crf and Xmap experiments use the optimized networks described in the previous section. Chortle-crf was run with the exhaustive reconvergent optimization, and the Root Replication optimizations (-Arf).

For 22 of the 27 networks, the circuit produced by Xmap contains more LUTs that the circuit produced by Chortle-crf, and for 1 of the networks the Xmap circuit contains fewer LUTs. In total, the Xmap circuits contain 14% more LUTs than the Chortle-crf circuits. The Chortle-crf experiments were run on a DEC5500 and the Xmap experiments were run on a SUN-4/370. Taking the relative performance of these machines into account Xmap was 16 times faster than Chortle-crf.

7.1.5 Chortle vs. Hydra

This section presents an experimental comparison of the Chortle area algorithm and the Hydra [Filo91] technology mapper described in Section 2.3.4 of Chapter 2. In these experiments 19 networks from the MCNC logic synthesis benchmark suite were

network	-Arf		Xmap	
	LUTs	sec. [1]	LUTs	sec. [2]
5xp1	28	0.4	31	0.3
9sym	59	12.8	73	0.5
9symml	44	6.4	55	0.4
C499	89	2.6	75	0.5
C880	88	2.2	103	0.8
alu2	116	7.1	126	0.9
alu4	70	2.5	98	0.7
apex2	64	2.9	81	0.7
apex4	579	98.9	664	6.4
apex6	198	2.9	231	1.6
apex7	60	0.6	65	0.5
b9	41	0.4	48	0.4
bw	39	0.3	43	0.3
clip	31	0.7	38	0.3
count	31	0.3	31	0.2
des	927	35.4	1042	6.8
duke2	111	1.7	127	0.8
e64	80	0.3	80	0.5
f51m	27	0.4	33	0.3
misex1	11	0.1	11	0.2
misxe2	28	0.1	28	0.2
rd73	16	0.3	21	0.1
rd84	35	1.3	36	0.4
rot	188	2.7	212	1.4
sao2	27	0.5	37	0.4
vg2	21	0.1	24	0.2
z4ml	7	0.1	9	0.2
total	3015	184	3422	26

[1] execution time on a DEC-5500 (27.3 Specmarks)
[2] execution time on a SUN-4/370 (12 Specmarks est.)

Table 7.5: Chortle vs. Xmap

network	-Arf		Hydra	
	CLBs	sec. [1]	CLBs	sec. [2]
5xp1	21	1.5	21	0.5
9sym	52	27.3	57	2.9
9symml	47	28.5	33	1.1
C499	50	6.0	51	1.8
C5315	389	87.8	229	23.0
C880	75	5.8	71	9.0
alu2	93	28.0	94	2.9
alu4	152	52.6	105	4.8
apex2	95	19.9	67	3.5
apex6	173	9.4	131	35.9
apex7	45	1.3	43	0.9
count	27	0.8	26	0.5
duke2	92	3.4	79	1.9
e64	54	0.7	47	0.7
misex1	13	0.3	8	0.2
rd84	50	8.6	27	0.6
rot	130	6.6	134	6.4
vg2	19	0.2	20	0.3
z4ml	4	0.2	4	0.4
total	1581	288.9	1247	97.3

[1] execution time on a SparcStation-IPC (13.5 Specmarks)
[2] execution time on a DEC-3100 (11.3 Specmarks)

Table 7.6: Chortle vs. Hydra

mapped into circuits of Xilinx 3000 CLBs.

Table 7.6 displays the number of logic blocks in the circuits produced by the Chortle area algorithm and the Hydra results published in [Filo91]. For Hydra, logic optimization preceding technology mapping was performed by misII using the standard script. The Chortle results are the (-Arf) results presented in Section 7.1.2.

The circuits produced by Hydra contain fewer logic blocks than the Chortle circuits for 12 of the 19 networks, and in total contained 21% fewer CLBs. Table 7.6 also shows the execution times for Hydra and Chortle. The Hydra experiments were run on a DECstation-3100 and the Chortle experiments were run on a SparcStation-IPC. Taking the relative performance of these machines into account Hydra is 3 times faster than Chortle.

Hydra achieves better results than Chortle when mapping into CLBs because the decomposition strategy used in Hydra anticipates the pairing of single-output functions into two-output logic blocks, and selects decompositions that encourage the sharing of inputs in addition to reducing the number of single-output functions.

network	-Arf		VISMAP	
	blocks	sec. [1]	blocks	sec. [2]
5xp1	16	1.5	28	1.0
9sym	48	27.5	63	0.9
9symml	42	28.6	52	0.7
C499	46	5.9	50	2.0
C880	68	5.9	78	2.6
apex6	164	10.0	155	12.7
apex7	43	1.4	54	1.5
b9	25	0.5	31	0.6
misex1	12	0.3	14	0.4
misex2	20	0.3	35	0.6
misex3	113	15.3	156	9.5
rd73	20	4.4	26	10.8
z4ml	3	0.2	18	0.6
total	620	101.8	760	43.9

[1] execution time of a SparcStation-IPC (13.5 Specmarks)
[2] execution time of a SparcStation-2 (25.0 Specmarks)

Table 7.7: Chortle vs. VISMAP

7.1.6 Chortle vs. VISMAP

This section presents an experimental comparison of the Chortle area algorithm and the VISMAP [Woo91] technology mapper described in Section 2.3.6 of Chapter 2. In these experiments 13 networks from the MCNC logic synthesis benchmark suite were mapped into circuits of two-output logic blocks. Each of these logic blocks can implement any two 5-input functions that together have at most 5 distinct inputs. Note that these two-output logic blocks are not equivalent to Xilinx 3000 CLBs.

Table 7.7 reports the number logic blocks in the circuits produced by the Chortle area algorithm and the VISMAP results published in [Woo91]. For the VISMAP experiments, logic optimization and decomposition preceding VISMAP were performed by Mis-pga(1). The Chortle experiments started with the optimized networks described in Section 7.1. The optimized networks were mapped into circuits of 5-input LUTs using the exhaustive reconvergent and the Root Replication optimizations. These single-output functions were then paired into two-output logic blocks using the MCM strategy described in Chapter 3.

The effectiveness of the covering algorithm implemented by VISMAP is limited by the decomposition chosen by Mis-pga(1) preceding VISMAP. The circuits produced by VISMAP contain more logic blocks than the Chortle circuits for 12 of the 13

networks, and in total contain 23% more logic blocks. Table 7.7 also shows the execution times for VISMAP and Chortle. The VISMAP experiments were run on a SparcStation-2 and the Chortle experiments were run on a SparcStation-IPC. Taking the relative performance of these machines into account VISMAP is 1.25 times faster than Chortle.

7.2 Results for the Delay Algorithm

7.2.1 Circuits of 5-input LUTs

This section presents experimental results for the delay algorithm described in Chapter 4. These experiments are intended to evaluate the effectiveness of the basic delay algorithm, the exhaustive reconvergent search and the replication of logic at every fanout node.

The goal of these experiments is the minimization of the number of levels of LUTs in the final circuits. The experimental procedure begins with logic optimization to reduce the depth of the network. The networks described in Section 7.1 are further optimized using the misII "speed_up -m unit" command [Sing88] before technology mapping by Chortle. The networks are mapped into circuits of 5-input LUTs using Chortle with the following options

(-Arf) area algorithm with exhaustive reconvergent search
and Root Replication

(-D) basic delay algorithm

(-Dr) with exhaustive reconvergent search

(-DF) with replication at every fanout node

(-DrF) with both exhaustive reconvergent and replication

Table 7.8 displays the number of LUTs and the number of levels of LUTs in the circuits produced by these options. The (-Arf) results provide a basis for evaluating the ability of the basic delay algorithm and the various options to reduce the number

of levels of LUTs in the final circuit. Note that the (-Arf) results recorded in this table differ from the results presented in Section 7.1 because these experiments started with networks that were optimized to reduce delay. Table 7.9 shows the execution time for these experiments on a SparcStation-IPC. To limit the execution time when mapping the network "alu4" with the options (-Arf) any node with more than 12 pairs of reconvergent paths was optimized using the MSD algorithm rather than the exhaustive reconvergent search.

The basic delay algorithm (-D), and the delay algorithm with the exhaustive reconvergent search (-Dr) provide minor reduction in the number of levels in the mapped circuits. For 12 of the 29 networks, the (-D) circuit has fewer levels than the (-Arf) circuit, but for 9 of the networks the (-D) circuit has more levels. In total, the (-D) circuits have 3.2% fewer levels than the -(Arf) circuits, but contain 65% more LUTs.

For 14 of the 29 networks, the (-Dr) circuit has fewer levels than the (-Arf) circuit, but for 9 of the networks the (-Dr) circuit has more levels. In total, the (-Dr) circuits have 6% fewer levels than the -(Arf) circuits, but contain 62% more LUTs.

For those networks where the (-Arf) circuit has fewer levels than the (-D) or (-Dr) circuits, the combination of the reconvergent and replication optimizations has found reconvergent paths that not only reduce the total number of LUTs, but also reduce number of levels in the final circuit.

The replication of logic at all fanout nodes in the (-DF) circuits decreases the number of levels, but substantially increases the number of LUTs in the circuits. For 25 of the 29 networks, the (-DF) circuit has fewer levels than the (-Arf) circuit, and for only 1 of the networks the (-DF) circuit has more levels. In total, the (-DF) circuits have 37% fewer levels than the -(Arf) circuits, but contain 150% more LUTs.

Combining the exhaustive reconvergent search with replication at every fanout node further decreases the number of levels in the circuits, and reduces the area penalty. For 28 of the 29 networks, the (-DrF) circuit has fewer levels than the (-Arf) circuit. In total, the (-DrF) circuits have 43% fewer levels than the -(Arf) circuits, and contain 121% more LUTs. Note that the total execution time for the (-DrF)

network	-Arf		-D		-Dr		-DF		-DrF	
	LUTs	levels	LUTs	levels	LUTs	levels	LUTs	levels	LUTs	levels
5xp1	28	4	56	5	53	5	60	4	31	3
9sym	65	8	98	6	96	6	99	5	70	5
9symml	61	8	93	6	92	6	97	5	57	4
C499	112	9	272	12	270	12	496	9	420	6
C5315	543	13	938	16	915	16	1697	10	1380	9
C880	163	13	286	15	275	14	393	8	367	8
alu2	128	17	214	14	203	13	322	9	271	9
alu4	214 *	19	397	18	370	17	655	11	590	10
apex2	123	9	210	9	205	8	220	6	223	6
apex4	597	11	830	9	828	9	1430	6	1389	6
apex6	232	7	361	7	352	7	496	4	358	4
apex7	75	6	132	7	132	7	147	4	125	4
b9	40	4	62	4	61	4	61	3	58	3
bw	57	4	94	5	94	5	116	3	28	1
clip	34	5	54	5	54	5	69	4	66	4
count	64	5	115	6	112	6	135	3	120	3
des	953	12	1399	10	1394	10	2720	6	2614	6
duke2	150	8	253	8	248	7	287	4	270	4
e64	138	7	267	7	267	7	247	3	247	3
f51m	39	5	66	5	63	5	61	3	38	3
misex1	17	3	36	4	36	4	30	3	19	2
misex2	32	6	61	4	61	4	56	3	52	3
misex3	247	27	455	22	446	22	555	14	510	13
rd73	34	6	55	5	53	5	70	4	36	3
rd84	41	8	77	7	69	6	125	5	81	4
rot	209	12	339	10	332	10	421	7	351	6
sao2	38	7	78	5	74	5	73	4	51	4
vg2	37	5	62	5	60	5	65	4	54	4
z4ml	13	3	30	7	28	6	25	4	21	3
total	4484	251	7390	243	7243	236	11228	158	9897	143

* MSD for some nodes

Table 7.8: Delay Algorithm 5-input LUT Results

circuits is less than the execution time for the (-Arf) circuits. This is a result of the (-Arf) option having to repeatedly map trees to determine when to replicate logic at fanout nodes, whereas the (-DrF) option simply replicates at every fanout node.

network	-Arf		-D		-Dr		-DF		-DrF	
	LUTs	sec. [1]	LUTs	sec. [1]	LUTs	sec. [1]	LUTs	sec. [1]	LUTs	sec. [1]
5xp1	28	1.3	56	0.8	53	0.9	60	1.0	31	2.0
9sym	65	25.8	98	3.1	96	3.3	99	3.6	70	5.4
9symml	61	28.1	93	2.8	92	3.4	97	3.5	57	10.9
C499	112	12.5	272	5.3	270	7.4	496	7.8	420	9.9
C5315	543	82.3	938	19.4	915	22.3	1697	27.0	1380	34.4
C880	163	14.3	286	5.2	275	6.2	393	6.8	367	7.8
alu2	128	17.7	214	3.9	203	4.3	322	5.7	271	6.5
alu4	214 *	48.2	397	8.0	370	9.5	655	12.3	590	14.9
apex2	123	16.0	210	4.1	205	4.5	220	5.2	223	5.6
apex4	597	48.4	830	26.5	828	30.0	1430	74.4	1389	89.3
apex6	232	9.6	361	6.9	352	7.4	496	9.0	358	10.7
apex7	75	2.1	132	1.8	132	2.0	147	2.2	125	2.5
b9	40	0.8	62	0.8	61	0.9	61	0.9	58	1.0
bw	57	1.7	94	1.3	94	1.4	116	1.9	28	25.9
clip	34	2.1	54	1.0	54	1.1	69	1.3	66	1.4
count	64	15.0	115	1.8	112	2.4	135	2.1	120	3.4
des	953	78.9	1399	42.7	1394	45.4	2720	70.9	2614	93.1
duke2	150	5.6	253	3.7	248	4.2	287	5.4	270	6.2
e64	138	1.8	267	2.4	267	2.4	247	3.1	247	3.2
f51m	39	5.2	66	1.2	63	1.3	61	1.5	38	2.1
misex1	17	0.3	36	0.4	36	0.4	30	0.4	19	1.1
misex2	32	0.4	61	0.5	61	0.5	56	0.6	52	0.6
misex3	247	25.7	455	6.1	446	6.5	555	8.7	510	23.4
rd73	34	5.4	55	1.4	53	1.5	70	1.8	36	4.0
rd84	41	3.2	77	1.5	69	1.7	125	2.2	81	3.7
rot	209	7.3	339	5.1	332	5.6	421	6.6	351	7.3
sao2	38	3.0	78	1.4	74	1.5	73	1.6	51	2.1
vg2	37	1.5	62	0.9	60	1.3	65	1.1	54	1.2
z4ml	13	14.7	30	0.4	28	0.6	25	0.5	21	0.8
total	4484	478.9	7390	160.4	7243	179.9	11228	269.1	9897	380.4

[1] execution time on a SparcStation-IPC (13.5 Specmarks)

* MSD for some nodes

Table 7.9: Delay Algorithm Execution Times

7.2.2 Reducing the Area Penalty

The results of the previous section show that the delay algorithm incurs a substantial area penalty while reducing the number of levels. This section presents results for experiments evaluating the optimizations, described in Chapter 4, that reduce the area penalty associated with the delay algorithm. These optimizations include: the single-fanout LUT peephole optimization, the *Leaf Replication* optimization, and the critical path optimization.

The experimental procedure takes the optimized networks used in the previous section and maps them into circuits of 5-input LUTs using Chortle with the following options:

(-DrFp) peephole optimization

(-DrLp) peephole, and Leaf Replication

(-DrLpc) as above, with the critical path optimization

(-DsLpc) as above, with MSD replacing the exhaustive reconvergent search

Table 7.10 presents the number of LUTs, and the number of levels in each of the circuits mapped by the different options. The (-DrF) results presented in the previous section are also included in this table to provide a basis for comparison. Table 7.11 gives the execution times for these experiments on a SparcStation-IPC.

The first observation is that none of the options increased the number of levels or the number of LUTs in any of the circuits. Using the peephole optimization, the (-DrFp) circuits, in total, have 7.9% fewer LUTs than the (-DrF) circuits. Replacing replication at every fanout node with the Leaf Replication algorithm further decreases the number of LUTs. In total, the (-DrLp) circuits have 14% fewer LUTs than the (-DrF) circuits.

The critical path optimization reduces the number of LUTs, but substantially increases the execution time. To limit the execution time for the (-DrLpc) circuits for the networks "f51m" and "z4ml" the exhaustive reconvergent search was replaced by the MSD algorithm, for any nodes where the number of pairs of reconvergent paths

exceeded 12. In total, the (-DrLpc) circuits contain 18% fewer LUTs than the (-DrF) circuits, and need 6.7 times the execution time. The increase in execution time is a result of the critical path optimization repeatedly mapping the network.

The increase in execution time can be reduced by replacing the exhaustive reconvergent search with the MSD algorithm, for all nodes. In total, the (-DsLpc) circuits use half the execution time of the (-DrLpc) circuits. For 9 of the 29 networks, the (-DsLpc) circuits have more LUTs than the (-DrLpc) networks, but for 7 of the networks the (-DsLpc) circuits have fewer LUTs. In total, the (-DsLpc) circuits actually have 8 fewer LUTs than the (-DrLpc) circuits. As described in Section 7.1, the ability of the MSD algorithm to occasionally outperform the exhaustive reconvergent search can be explained by the fact that both approaches are local optimizations.

network	-DrF		-DrFp		-DrLp		-DrLpc		-DsLpc	
	LUTs	levels	LUTs	levels	LUTs	levels	LUTs	levels	LUTs	levels
5xp1	31	3	27	3	27	3	27	3	27	3
9sym	70	5	63	5	63	5	57	5	57	5
9symml	57	4	54	4	54	4	54	4	53	4
C499	420	6	389	6	321	6	305	6	304	6
C5315	1380	9	1275	9	1248	9	1124	9	1128	9
C880	367	8	341	8	318	8	306	8	307	8
alu2	271	9	244	9	239	9	219	9	232	9
alu4	590	10	548	10	517	10	493	10	476	10
apex2	223	6	202	6	200	6	184	6	183	6
apex4	1389	6	1322	6	1078	5	1050	5	1052	5
apex6	358	4	306	4	301	4	303	4	303	4
apex7	125	4	109	4	107	4	99	4	103	4
b9	58	3	52	3	52	3	47	3	48	3
bw	28	1	28	1	28	1	28	1	28	1
clip	66	4	58	4	56	4	43	4	43	4
count	120	3	107	3	107	3	100	3	101	3
des	2614	6	2448	6	2334	6	2243	6	2246	6
duke2	270	4	250	4	230	4	226	4	218	4
e64	247	3	213	3	182	4	175	4	175	4
f51m	38	3	33	3	33	3	31 *	3	31	3
misex1	19	2	17	2	17	2	17	2	17	2
misex2	52	3	38	3	38	3	37	3	37	3
misex3	510	13	465	13	452	13	442	13	432	13
rd73	36	3	31	3	31	3	26	3	25	3
rd84	81	4	75	4	75	4	63	4	63	4
rot	351	6	311	6	300	6	302	6	302	6
sao2	51	4	47	4	47	4	46	4	48	4
vg2	54	4	52	4	52	4	51	4	51	4
z4ml	21	3	15	3	15	3	14 *	3	14	3
total	9897	143	9120	143	8522	143	8112	143	8104	143

* MSD for some nodes

Table 7.10: Reducing the Area Penalty of the Delay Algorithm

network	-DrF		-DrFp		-DrLp		-DrLpc		-DsLpc	
	LUTs	sec. [1]	LUTs	sec. [1]	LUTs	sec. [1]	LUTs	sec. [1]	LUTs	sec. [1]
5xp1	31	2.0	27	2.0	27	2.0	27	2.4	27	1.4
9sym	70	5.4	63	5.4	63	5.4	57	13.1	57	10.0
9symml	57	10.9	54	10.9	54	11.0	54	18.7	53	9.9
C499	420	9.9	389	9.9	321	10.2	305	21.1	304	19.0
C5315	1380	34.4	1275	34.6	1248	34.4	1124	152.7	1128	123.7
C880	367	7.8	341	7.9	318	7.8	306	24.5	307	23.5
alu2	271	6.5	244	6.7	239	6.5	219	18.8	232	20.8
alu4	590	14.9	548	15.0	517	14.9	493	95.8	476	50.5
apex2	223	5.6	202	5.7	200	5.6	184	14.5	183	14.2
apex4	1389	89.3	1322	89.9	1078	159.7	1050	1558.3	1052	364.2
apex6	358	10.7	306	10.6	301	11.2	303	27.2	303	23.4
apex7	125	2.5	109	2.6	107	2.5	99	4.7	103	4.3
b9	58	1.0	52	1.0	52	1.0	47	1.8	48	1.7
bw	28	25.9	28	26.0	28	26.0	28	26.7	28	2.7
clip	66	1.4	58	1.4	56	1.4	43	2.9	43	2.6
count	120	3.4	107	3.4	107	3.4	100	6.5	101	4.4
des	2614	93.1	2448	93.5	2334	98.2	2243	349.1	2246	341.2
duke2	270	6.2	250	6.2	230	6.4	226	16.6	218	11.4
e64	247	3.2	213	3.2	182	3.0	175	5.3	175	5.5
f51m	38	2.1	33	2.1	33	2.1	31 *	63.1	31	3.1
misex1	19	1.1	17	1.0	17	1.1	17	1.8	17	0.9
misex2	52	0.6	38	0.6	38	0.6	37	0.8	37	0.7
misex3	510	23.4	465	23.4	452	23.4	442	69.1	432	31.1
rd73	36	4.0	31	4.1	31	4.1	26	12.3	25	2.8
rd84	81	3.7	75	3.7	75	3.7	63	10.5	63	6.6
rot	351	7.3	311	7.5	300	7.4	302	22.0	302	16.3
sao2	51	2.1	47	2.1	47	2.1	46	3.9	48	3.3
vg2	54	1.2	52	1.2	52	1.1	51	2.3	51	2.0
z4ml	21	0.8	15	0.8	15	0.8	14 *	1.1	14	0.9
total	9897	380.4	9120	382.4	8522	457	8112	2547.6	8104	1102.1

[1] execution time on a SparcStation-IPC (13.5 Specmarks)

* MSD for some nodes

Table 7.11: Execution Times for Reducing the Area Penalty

network	-DrLpc			Mis-pga(3)		
	LUTs	depth	sec. [1]	LUTs	depth	sec. [2]
5xp1	27	3	2.4	21	2	3.5
9sym	57	5	13.1	7	3	15.2
9symml	54	4	18.7	7	3	9.9
C499	305	6	21.1	199	8	58.4
C5315	1124	9	152.7	643	10	282.2
C880	306	8	24.5	259	9	39.0
alu2	219	9	18.8	122	6	42.6
alu4	493	10	95.8	155	11	15.4
apex2	184	6	14.5	116	6	9.8
apex6	303	4	27.2	274	5	60.0
apex7	99	4	4.7	95	4	8.4
b9	47	3	1.8	47	3	2.3
bw	28	1	26.7	28	1	8.3
clip	43	4	2.9	54	4	3.7
count	100	3	6.5	81	4	5.1
des	2243	6	349.1	1397	11	937.8
duke2	226	4	16.6	164	6	16.4
e64	175	4	5.3	212	5	15.7
f51m	31	3	63.1	23	4	5.9
misex1	17	2	1.8	17	2	1.7
misex2	37	3	0.8	37	3	1.4
rd73	26	3	12.3	8	2	4.4
rd84	63	4	10.5	13	3	9.8
rot	302	6	22.0	322	7	50.0
sao2	46	4	3.9	45	5	9.5
vg2	51	4	2.3	39	4	1.7
z4ml	14	3	1.1	10	2	2.1
total	6620	125	920.2	4395	133	1620.2

[1] execution time on a SparcStation-IPC (13.5 Specmarks)
[2] execution time on a DEC-5500 (27.3 Specmarks)

Table 7.12: Chortle vs. Mis-pga(3)

7.2.3 Chortle vs. Mis-pga

This section compares the ability of the latest version of Mis-pga [Murg91b], described in Section 2.3.2 of Chapter 2, and the Chortle delay algorithm to reduce the number of levels of LUTs. This version of Mis-pga is referred to as Mis-pga(3). Table 7.12 shows experimental results for Mis-pga(3) and Chortle mapping 28 networks from the MCNC logic synthesis benchmark suite into circuits of 5-input LUTs. The Mis-pga(3) results were published in [Murg91b], and the Chortle results are the (-DrLpc) results presented in Section 7.2.2. Note that for these experiments, Mis-pga(3) and Chortle start with different optimized networks.

Generally, the Mis-pga(3) circuits contain fewer LUTs, but more levels, than the

network	-DrLpc			DAG-Map	
	LUTs	depth	sec. [1]	LUTs	depth
5xp1	27	3	2.4	28	3
9sym	57	5	13.1	63	5
9symml	54	4	18.7	61	5
C499	305	6	21.1	204	5
C880	306	8	24.5	246	8
alu2	219	9	18.8	199	9
alu4	493	10	95.8	303	10
apex2	184	6	14.5	164	5
apex4	1050	5	1558.3	780	6
apex6	303	4	27.2	284	5
apex7	99	4	4.7	95	4
count	100	3	6.5	87	3
des	2243	6	349.1	1480	6
duke2	226	4	16.6	195	4
e64	175	4	5.3	167	3
misex1	17	2	1.8	17	2
rd84	63	4	10.5	48	4
rot	302	6	22.0	328	6
vg2	51	4	2.3	42	3
z4ml	14	3	1.1	17	3
total	6288	100	2214.3	4808	99

[1] execution time on a SparcStation-IPC (13.5 Specmarks)

Table 7.13: Chortle vs. DAG-Map

Chortle circuits. For 12 of the 28 networks, the Mis-pga(3) circuits contain more levels than the Chortle circuits, and for 7 networks the Mis-pga(3) circuits contain fewer levels. In total, the Mis-pga(3) circuits contain 6.4% more levels and 34% fewer LUTs than the Chortle circuits. In these experiments, Mis-pga(3) was run on a DEC-5500 and the Chortle was run on a SparcStation-IPC. Taking the relative performance of these machines into account, Mis-pga(3) is 3 times slower than Chortle.

7.2.4 Chortle vs. DAG-Map

This section compares the performance of the Chortle delay algorithm and DAG-Map [Cong92] technology mapper, described in Section 2.3.7 of Chapter 2. These two programs were used to map 20 networks from the MCNC logic synthesis benchmark suite into circuits of 5-input LUTs. Table 7.13 displays the DAG-Map results published in [Cong92]. These experiments used the same starting networks as described in Section 7.2. The Chortle results recorded in this table are the Chortle (-DrLpc) results

presented in Section 7.2.2.

DAG-Map and Chortle have nearly identical performance in terms of the number of levels in the final circuit, but DAG-Map uses significantly fewer LUTs. For 4 of the 20 networks, the DAG-Map circuits have one fewer level of LUTs than the Chortle circuits, and for 3 networks, the DAG-Map circuits have one more level. In total, the DAG-Map circuits contain 24% fewer LUTs than the Chortle circuits.

7.3 Summary

This chapter has presented experimental results that investigate the effectiveness of the area and delay algorithms. In addition, these results were used to compare Chortle to other LUT technology mappers. Combinations of the following optimization options were used in the experiments:

(A) basic area algorithm

(D) basic delay algorithm

(r) exhaustive reconvergent search

(s) Maximum Share Decreasing

(f) *Root Replication*

(F) replication at every fanout node

(L) *Leaf Replication*

(p) peephole optimization

(c) critical path optimization

Section 7.1.1 presented results for the area algorithm using the options (-A), (-Ar), (-Af), (-Arf), and (-Asf). Section 7.2.1 presented results for the delay algorithm using the options (-D), (-Dr), (-DF), and (-DrF). The reduction of the area penalty

associated with the delay algorithm was addressed in Section 7.2.2, which presented results using the options (-DrFp), (-DrLp), (-DrLpc), and (-DsLpc).

From the area algorithm experiments it was observed that the reduction in the number of LUTs with the combination of the exhaustive reconvergent search and the root replication optimization exceeds the sum of the separate reductions for these two optimizations. This indicates that the replication of logic at fanout nodes exposes additional reconvergent paths for the exhaustive search to exploit.

The Xilinx 3000 CLB experiments showed that networks can be mapped into circuits of two-output logic blocks by first mapping the networks into circuits of single-output LUTs, and then pairing these LUTs into CLBs. Minimizing the number of LUTs in the intermediate circuit does not, however, necessarily minimize the total number of two-output logic blocks in the final circuit. In particular, the replication of logic at fanout nodes, which can reduce the number of LUTs, is seldom beneficial if the logic block has a second output that can explicitly implement the replicated function at little additional cost.

The delay algorithm experiments showed that a key factor in the reduction of the number of levels of LUTs in the delay algorithm is the replication of logic at fanout nodes. In addition, there is a large area penalty associated with the delay algorithm, and the optimizations intended to reduce this penalty have limited success.

For both the area and delay algorithm, when the exhaustive reconvergent search is replaced by the MSD algorithm there is little change in the number of LUT or the number of levels, but there is a significant reduction in execution time.

When compared to other LUT technology mappers, the area algorithm outperforms Mis-pga(1) and Xmap in terms of the number of LUTs in the final circuit, but Mis-pga(2) outperforms the area algorithm. Hydra produces circuits that contain fewer Xilinx 3000 CLBs than those produced by Chortle, whereas VISMAP produces circuits that contain more two-output logic blocks. In terms of the number of levels in the final circuits, the delay algorithm outperforms Mis-pga(3) and produces results similar to DAG-Map. However, the circuits produced by the delay algorithm contain substantially more LUTs than the Mis-pga(3) and DAG-Map circuits.

Chapter 8

Conclusions

Lookup table-based FGPAs, because of their user-programmability and large scale integration, have become an attractive vehicle for the realization of Application Specific Integrated Circuits (ASICs). These devices present new challenges for logic synthesis, particularly technology mapping, which is the phase of logic synthesis directly concerned with the selection of the circuit elements in the final circuit. This thesis has presented some of the earliest research that addresses technology mapping into lookup-table (LUT) circuits. Two algorithms that map a network of ANDs, ORs and NOTs into a circuit of K-input LUTs were presented. The *area algorithm* minimizes the number of LUTs in the final circuit, and the *delay algorithm* minimizes the number of levels of LUTs.

The overall strategy of both algorithms is to first partition a general network at fanout nodes into a forest of trees, and then to map each tree separately. Each tree is mapped using a dynamic programming strategy similar to conventional library-based technology mapping. The major innovation is the combination of the decomposition of nodes in the network, and the matching of the network to LUTs into one problem that is solved using the First Fit Decreasing bin-packing algorithm. For each tree, the circuit constructed by the area algorithm has been shown to be an optimal tree of LUTs for values of $K \leq 5$. In addition, the circuit constructed by the delay algorithm is an optimal tree of LUTs for values of $K \leq 6$. The area and delay algorithms also include optimizations that exploit reconvergent paths and the replication of logic at

fanout nodes to further improve the final circuit.

The two algorithms were implemented in a technology mapping program called *Chortle*, and their effectiveness was evaluated in a series of experiments that mapped networks from the MCNC logic synthesis benchmark suite into circuits of 5-input LUTs. The MCNC networks were also mapped into circuits of Xilinx 3000 CLBs, by pairing LUTs from the LUT circuits into two-output logic blocks.

8.1 Future Work

This thesis has focused on technology mapping for LUT circuits. In the experimental evaluation, logic optimization preceding technology mapping was performed using existing techniques originally developed for Masked-Programmed Gate Arrays and Standard Cell circuits. Future investigations should determine if logic optimization can be tuned to improve the final LUT circuits produced by technology mapping. In particular, the basic area and delay algorithms presented here are computationally inexpensive, and could provide preliminary LUT circuits to evaluate alternative networks during logic optimization.

The optimizations that exploit reconvergent paths and the replication of logic are both greedy heuristics that consider only local information. There is potential for improvement if more global information is used to determine how reconvergent paths should be covered, and where logic should be replicated.

The decomposition strategy employed in the area algorithm considers only the reduction of the number of LUTs, and the strategy employed in the delay algorithm considers only the reduction of the number of levels. Compared to the circuits produced by the area algorithm, the circuits produced by the delay algorithm incur a substantial increase in the number of LUTs. The decomposition strategies in both algorithms are based on two separate phases that first pack LUTs together and then connect LUT outputs to inputs. Future work should consider how these two phases could be organized to permit a continuous tradeoff between the number of LUTs and the number of levels.

The delay algorithm minimized the number of levels of LUTs in order to reduce delays in the final circuit, by reducing the contribution of logic block delays. In LUT-based FPGAs, the delays incurred in programmable routing account for a substantial portion of total delay. The actual delays in the final FPGA circuit are therefore, dependent upon the placement of logic blocks and the routing of connections between the logic blocks. A recent study has indicated that a correlation exists between the number of levels of LUTs and the actual delays in a LUT-based FPGA circuit [Murg91b]. However, future research could consider placement and routing in conjunction with technology mapping to better address the minimization of delay in LUT-based FPGAs.

The optimality results presented in this thesis are based on the ability of a K-input LUT to implement any function of K variables. The completeness of the set of functions may also lead to additional optimality results for LUT circuits. The area and delay algorithms were able to produce optimal trees of LUTs implementing fanout-free trees. Considering the divide and conquer strategy used to map general networks, two types of sub-networks that merit future investigation for optimality results are leaf-DAGs, and single-output networks.

The experimental results for two-output logic blocks indicate that the minimization of the number of LUTs in the intermediate circuit does not necessarily reduce the number of logic blocks in the final circuit. In some instances, an increase in the number of LUTs in the circuit permits an increase in the number of paired LUTs, and produces a net reduction in the number of two-output logic blocks. The Hydra technology mapper [Filo91] has demonstrated that the optimization goal for the intermediate circuit can be tuned to anticipate the pairing of LUTs into two-output logic blocks. To improve the circuits produced for two-output logic blocks, future research could consider integrating this tuned optimization goal with the decomposition techniques presented here.

Appendix A

Optimality of the First Fit Decreasing Algorithm

This appendix proves two theorems about the First Fit Decreasing bin packing algorithm that are used in the optimality proofs presented in Chapters 5 and 6.

A.1 Bin Packing

The one-dimensional packing problem is a well-known combinatorial optimization problem that can be stated as follows:

> Given a finite set of items, of positive size, and an integer K partition the items into the minimum number of disjoint subsets such that, the sum of the sizes of the items in every subset is less than or equal to K.

The problem is commonly known as the bin packing problem, because each subset can be viewed as a set of boxes packed into a bin of capacity K. In this appendix, the unused capacity of a bin is referred to as a *hole*, and the set of bins containing the boxes is referred to as a *packing* of the boxes.

The bin packing problem is known to be NP-hard [Gare79], however, performance bounds have been presented for several polynomial-time approximation algorithms. The First Fit Decreasing (FFD) algorithm begins with an infinite sequence of empty

bins. The given boxes are first sorted by size, and then packed into bins, one at a time, beginning with a largest box and proceeding in order to a smallest box. Each box is packed into the first bin in the sequence having a hole greater than or equal to the size of the box. The FFD packing consists of the non-empty bins in the sequence after all the boxes have been packed.

It has been shown by Johnson [John74] that if the optimal packing of an arbitrary set of boxes requires n bins, then the number of bins in the FFD packing of the same boxes will be less than or equal to $11 * n/9 + 4$. In general, the box sizes and bin capacity are rational numbers. The remainder of this appendix will consider the bin packing problem where the box sizes and the bin capacity are restricted to integer values. Under these conditions, it will be shown that First Fit Decreasing algorithm is optimal for bins of capacity less than or equal to 6.

Theorem A.1

> For values of K from 2 to 6, the FFD packing of an arbitrary set of boxes into bins of capacity K uses the fewest number of bins possible.

The proof of optimality for the area algorithm in Chapter 5 not only requires that FFD produce the minimum number of bins, but also that it produce a bin with the largest hole size possible. This can shown to be true for bins of capacity less than or equal to 5.

Theorem A.2

> For values of K from 2 to 5, the FFD packing of an arbitrary set of boxes into bins of capacity K includes a bin with the largest hole possible for any packing of the boxes that has the same number of bins as the FFD packing.

A.2 Outline of Proof

For any set of boxes, the existence of a packing with fewer bins than the FFD packing would imply that there exists another packing having the same number of bins as the FFD packing that includes at least one empty bin. Therefore to prove Theorem A.1 for a given value of K it is sufficient to show that

> Given the FFD packing of an arbitrary set of boxes into bins of capacity K there does not exists a re-packing of the same boxes into the same number of bins that includes an empty bin.

Because the bin capacity and the box sizes are restricted to integers, the hole sizes in any packing must be an integer. Therefore to prove Theorem A.2 it is sufficient to prove the following lemma for the values of K and H:

$K = 2,\ H = 0, 1$

$K = 3,\ H = 0, 1, 2$

$K = 4,\ H = 0, 1, 2, 3$

$K = 5,\ H = 0, 1, 2, 3, 4$

Lemma A.3

> If the FFD packing of a set of boxes into bins of capacity K does not include a hole of size greater than H, then there does not exist a re-packing of the same boxes into the same number of bins that includes a hole of size greater than H.

Proving Lemma A.3 for the appropriate values of K and H will also prove Theorem A.1. Note that the hole size of an empty bin is K and that every FFD packing is guaranteed to not include a hole of size greater than $K - 1$. Therefore, to prove Theorem A.1 it is sufficient to prove Lemma A.3 for the following values of K and H.

$K = 2,\ H = 1$

$K = 3,\ H = 2$

$K = 4,\ H = 3$

$K = 5,\ H = 4$

$K = 6,\ H = 5$

To prove Lemma A.3 for given values of K and H all possible FFD packings into bins of capacity K that do not include a hole of size greater than H are categorized into a finite number of cases. Each case is defined by a set of conditions imposed on the FFD packings that are members of the case. The formal definition of a case is given in the following section. The proof of Lemma A.3 consists of a series of separate proofs of the following lemma for each of the cases.

Lemma A.4

> If the FFD packing of a set of boxes into bins of capacity K is a member of the case then there does not exists a re-packing of the same boxes into the same number of bins that includes a hole of size greater than H.

The key to the proof of Lemma A.4 is that the re-packing must have the same number of bins as the FFD packing. This allows a system of equations relating the two packings to be developed. Using this system of equations and the conditions imposed on the FFD packing by membership in the particular case it is possible to deduce that the re-packing cannot include a hole of size greater than H.

The proofs for Theorem A.1 and Theorem A.2 require the proof of Lemma A.4 for a large number of separate cases, however, the derivation of the cases, and the proof of each case can be automated.

The following section introduces the notation required to describe the contents of a bin, a packing, and a case. Section A.4 shows how, for a given value of K and a given value of H, to derive the finite set of cases that includes all possible FFD packings into bins of capacity K that do not include a hole of size greater than H, and Section A.5 shows how to prove Lemma A.4 for one of these cases. Section A.6 shows how to reduce the number of separate cases that must be considered to prove Theorem A.1 and Theorem A.2. Sections A.7 to A.12 present the details of the proofs for the

individual cases required to prove Theorem A.1 and Theorem A.2. Finally, Section A.13 presents counter examples that show that Theorem A.1 cannot be extended to values of K greater than 6 and that Theorem A.2 cannot be extended to values of K greater than 5.

A.3 Notation

A case consists of a possibly infinite set of packings. In order to describe a case, the notation to describe a bin and to describe a packing is first introduced.

Definition: Content Vector

The contents of a bin can be described by a *content vector* $\mathbf{a}$ where:

$\mathbf{a} = (a_0 \ldots a_K)$

for all $i, (1 \leq i \leq K)$

a_i = number of boxes of size i in the bin

and $a_0 = K - \sum_{i=1}^{K} i * a_i$ = hole size

Note that the content vector describing a bin does not depend upon the order of the boxes.

Example: Content Vector

If $K = 5$, then the bin {2, 1, 1} and the bin {1, 2, 1} are both described by the content vector $(1, 2, 1, 0, 0, 0)$. Both bins contain a hole of size 1, two boxes of size 1, and one box of size 2.

Because the only box sizes possible are the integers from 1 to K, the number of combinations of boxes with total size less than or equal to K is also finite. Each of these combinations specifies a distinct vector. Therefore, there is a finite number of distinct content vectors $\mathbf{a}_1$ to $\mathbf{a}_m$.

Example: Content Vectors for $K = 3$

For $K = 3$ there are 7 different content vectors.

content vector	bin
$\mathbf{a}_1 = (0, 0, 0, 1)$	$\{3\}$
$\mathbf{a}_2 = (0, 1, 1, 0)$	$\{2,\ 1\}$
$\mathbf{a}_3 = (1, 0, 1, 0)$	$\{2\}$
$\mathbf{a}_4 = (0, 3, 0, 0)$	$\{1,\ 1,\ 1\}$
$\mathbf{a}_5 = (1, 2, 0, 0)$	$\{1,\ 1\}$
$\mathbf{a}_6 = (2, 1, 0, 0)$	$\{1\}$
$\mathbf{a}_7 = (3, 0, 0, 0)$	$\{\}$

Note that the construction of all different content vectors for a given bin capacity K can be automated.

Definition: Type Vector

An arbitrary set of bins can be described by the *type vector* $\mathbf{y}$ where:

$$\mathbf{y} = (y_1 \ldots y_m)$$

$$\text{for all } i, (1 \leq i \leq m)$$

$$y_i = \text{number of bins in the set with contents described by } \mathbf{a}_i$$

Note that for all i, y_i is an integer and $y_i \geq 0$. In addition, the type vector describing a set of bins does not depend upon the order of the bins.

Example: Type Vector

Using the set of previously shown content vectors for $K = 3$, the set of bins $\{2,\ 1\}$, $\{2,\ 1\}$, $\{1,\ 1,\ 1\}$ is described by the type vector $(0, 2, 0, 1, 0, 0, 0)$.

Definition: Case

Each case is defined by a set of conditions on the type vector, $\mathbf{y}$, describing *any* packing that is a member of the case. Every component of $\mathbf{y}$ must

satisfy a separate condition. Note that the condition applied to one component is independent of the conditions applied to the other components.

These conditions on the components of $\mathbf{y}$ are described by the two Boolean vectors $\mathbf{o}$ and $\mathbf{u}$. For all i, if $o_i = 1$ then y_i may be equal to 1, and if $u_i = 1$ then y_i may be greater than 1. There are four possible conditions for the component, y_i, that are specified as follows:

o_i	u_i	y_i
0	0	$y_i = 0$
1	0	$y_i = 1$
0	1	$y_i > 1$
1	1	$y_i \geq 1$

Example: Case

Using the content vectors $\mathbf{a}_1$ to $\mathbf{a}_7$ shown previously for $K = 3$, consider the case defined by the two Boolean vectors:

$$o = (1, 1, 0, 1, 0, 1, 0)$$
$$u = (1, 1, 0, 1, 0, 0, 0)$$

A set of bins, described by the type vector $\mathbf{y}$, is a member of this case if and only if:

$$y_1 \geq 1$$
$$y_2 \geq 1$$
$$y_3 = 0$$
$$y_4 \geq 1$$
$$y_5 = 0$$
$$y_6 = 1$$
$$y_7 = 0$$

To be a member of this case a set of bins must contain at least one instance of a bin described by the each of the three content vectors $\mathbf{a}_1$,

$\mathbf{a}_2$, and $\mathbf{a}_4$, exactly one bin described by the content vector $\mathbf{a}_6$, and no bins described by the content vectors $\mathbf{a}_3$, $\mathbf{a}_5$, and $\mathbf{a}_7$. The set of bins $\{3\}$, $\{3\}$, $\{2, 1\}$, $\{1, 1, 1\}$, $\{1, 1, 1\}$, $\{1, 1, 1\}$, $\{1\}$ described by the type vector, (2, 1, 0, 3, 0, 1, 0), is a member of this case, but the set of bins $\{3\}$, $\{3\}$, $\{2, 1\}$, $\{2\}$, $\{1, 1, 1\}$, $\{1, 1, 1\}$, $\{1\}$ described by the type vector, (2, 1, 1, 2, 0, 1, 0), is not a member of this case.

A.4 Deriving a Complete Set of Cases

This section shows how to derive, for given values of K and H, the finite set of cases that includes all FFD packings that do not have a hole of size greater than H. First a finite set of cases that includes all possible packings, including non-FFD packings and packings with holes of size greater than H, is derived. This set of cases is then reduced to the desired set of cases by excluding all packings that contain a hole of size greater than H, and all packings that are not FFD packings.

For the given value of K there are m distinct content vectors. The initial set of cases consists of 2^m separate cases. Each case is defined by a set of conditions on the m components of the type vector describing *any* member of the case. For each component y_i there are cases where y_i must equal zero and cases where y_i must be greater than or equal to one. Taking all possible combinations of the two conditions, $y_i = 0$ and $y_i \geq 1$, on the m components results in a set of 2^m cases. Note that this initial set of cases includes all possible packings, including non-FFD packings and packings with a hole of size greater than H.

A.4.1 Eliminating Holes Greater than H

The next step in constructing the desired set of cases is to exclude any packing that contains a hole of size greater than H. For any value of i, where a bin described by the content vector $\mathbf{a}_i$ has a hole of size greater than H, consider any of the initial cases that requires that $y_i \geq 1$. Every packing that is a member of this case must contain at least one bin described by the content vector $\mathbf{a}_i$. Therefore, every member of the

case contains a hole of size greater than H, and the entire case can be eliminated from further consideration.

Eliminating non-FFD Packings

The next step in constructing the desired set of cases is to exclude any packing that is not an FFD packing. In order to identify non-FFD packings the concept of FFD compatibility and the Compatibility Lemma are introduced.

Definition: FFD Compatible

> The contents of two arbitrary bins are FFD compatible if and only if re-packing all the boxes contained in the two bins using the First Fit Decreasing algorithm results in two bins with the same contents as the original bins.

Example: FFD Compatible

> For example, if $K = 4$, the pair of bins $\{2, 2\}$ and $\{1, 1, 1\}$ are FFD compatible and the pair of bins $\{2, 1, 1\}$ and $\{2, 1\}$ are FFD incompatible. Both pairs of bins contain the same boxes, $\{2, 2, 1, 1, 1\}$, however, only the first pair of bins is the FFD packing of this set of boxes.

Compatibility Lemma

> Every bin in an FFD packing is FFD compatible with every other bin in the packing.

Proof of Compatibility Lemma

> From the definition of the First Fit Decreasing algorithm an ordered set of bins is an FFD packing if and only if the size of every box in every bin exceeds the sum of the hole size and the total size of smaller boxes in every preceding bin.

Given an arbitrary FFD packing containing n bins, consider the qth bin and the rth bin, where $1 \leq q < r \leq n$. From the definition of an FFD packing, the size of every box in the rth bin will exceed the sum of the hole size and the total size of smaller boxes in the qth bin. Therefore, the ordered set of bins consisting of the qth bin followed by the rth bin is also an FFD packing. Therefore, the qth bin and the rth bin are FFD compatible.

Corollary 1

If a bin described by a content vector $\mathbf{a}_s$ and a bin described by a different content vector $\mathbf{a}_t$ are incompatible, then any packing that contains a bin described by the content vector $\mathbf{a}_s$ and another bin described by the content vector $\mathbf{a}_t$, is not an FFD packing.

Corollary 2

If two bins described by the same content vector $\mathbf{a}_s$ are incompatible, then an FFD packing can contain at most one bin described by the content vector $\mathbf{a}_s$.

Using these corollaries, the non-FFD packings can be eliminated from the remaining set of cases. Recall that initially there were 2^m cases that included all possible packings, including non-FFD packings and packings with holes of size greater than H. From this initial set of cases all packings with holes greater than H were eliminated. The next step is the elimination of non-FFD packings from the remaining set of cases.

For any value of i and $j \neq i$, where a bin described by the content vector $\mathbf{a}_i$ is FFD incompatible with a bin described by the content vector $\mathbf{a}_j$, consider any of the remaining cases that requires that $y_i \geq 1$ and $y_j \geq 1$. Every packing that is a member of this case must contain at least one bin described by the content vector $\mathbf{a}_i$, and at least one bin described by the content vector $\mathbf{a}_j$. Therefore, applying Corollary 1, every member of the case is not an FFD packing, and the entire case can be eliminated from further consideration.

For some of the remaining cases, more restrictive conditions on the type vector, $\mathbf{y}$, describing any member of the case can be deduced by applying Corollary 2 of the Compatibility Lemma. For any value of i, where two bins described by the same content vector $\mathbf{a}_i$ are FFD incompatible, consider any of the remaining cases that requires that $y_i \geq 1$. Every packing that is a member of this case must contain at least one bin described by the content vector $\mathbf{a}_i$. However, Corollary 2 of the Compatibility Lemma states that any packing with more than one bin described by the content vector $\mathbf{a}_i$ is not an FFD packing. Therefore, all packings in the case that do not contain exactly one bin described by the content vector $\mathbf{a}_i$ can be eliminated from further consideration. This leaves only those packings where $y_i = 1$. In this case, the condition $y_i \geq 1$ has been replaced with the more restrictive condition $y_i = 1$.

Note that the process of deriving the finite set of cases that includes all FFD packings into bins of capacity K that do not include a hole of size greater than H can be automated.

A.5 How to Prove a Case

The proofs of Theorem A.1 and Theorem A.2 consist of a series of separate proofs of Lemma A.3 for various values of K and H. The previous section showed how for given values of K and H all possible FFD packings into bins of capacity K without a hole of size greater than H are categorized into a finite set of cases. The proof of Lemma A.3 consists of a series of separate proofs of the Lemma A.4 for each of these cases. Each case is defined by the two Boolean vectors $\mathbf{o}$ and $\mathbf{u}$. The case includes all packings that satisfy a set of conditions specified by these vectors.

The key to the proof of Lemma A.4 for each case is a system of equations relating an arbitrary FFD packing that is a member of the case to an arbitrary re-packing of the same boxes into the same number of bins. Let the type vector $\mathbf{y}$ describe the original FFD packing and let the type vector $\mathbf{x}$ describe the re-packing of the same set of boxes into the same number of bins. To show that the re-packing does not include any bins with a hole of size greater than H, it will suffice to show for all i

where the content vector $\mathbf{a}_i$ has a hole of size greater than H that the component x_i must be equal to zero.

The system of equations relating the original FFD packing to the re-packing arises from the following observations. Because both packings contain the same set of boxes, for all i from 1 to K the total number of boxes of size i in each packing is the same, and because the two packings have the same number of bins the total unused capacity in the each of the packings is the same.

Because the original FFD packing is a member of the case defined by the Boolean vectors $\mathbf{o}$ and $\mathbf{u}$, this imposes conditions on the components of $\mathbf{y}$. By taking an appropriate linear combination of the system of equations it is possible to deduce from these conditions on the components of $\mathbf{y}$ that the required components $\mathbf{x}$ are equal to zero.

A.5.1 An Example

This section demonstrates how to develop the system of equations for a given case and how to prove Lemma A.4 using this system of equations. Recall that for $K = 3$ the set of all distinct content vectors consists of

content vector	bin
$\mathbf{a}_1 = (0, 0, 0, 1)$	$\{3\}$
$\mathbf{a}_2 = (0, 1, 1, 0)$	$\{2,\ 1\}$
$\mathbf{a}_3 = (1, 0, 1, 0)$	$\{2\}$
$\mathbf{a}_4 = (0, 3, 0, 0)$	$\{1,\ 1,\ 1\}$
$\mathbf{a}_5 = (1, 2, 0, 0)$	$\{1,\ 1\}$
$\mathbf{a}_6 = (2, 1, 0, 0)$	$\{1\}$
$\mathbf{a}_7 = (3, 0, 0, 0)$	$\{\}$

Consider the original FFD packing described by the type vector $\mathbf{y}$. The component y_1 is the number of bins in the packing described by the content vector $\mathbf{a}_1$. Similarly, for all i from 1 to 7, the component y_i is the number of bins in the packing described by the content vector $\mathbf{a}_i$. Remember that the zeroth component of a content vector specifies the hole size of a bin described by the content vector. In this example, the

content vectors $\mathbf{a}_1$, $\mathbf{a}_2$, and $\mathbf{a}_4$ specify a hole of size 0, the content vectors $\mathbf{a}_3$ and $\mathbf{a}_5$ specify a hole of size 1, the content vector $\mathbf{a}_6$ specifies a hole of size 2, and the content vector $\mathbf{a}_7$ specifies a hole of size 3. Therefore the total unused capacity in the original FFD packing is simply

$$\text{total unused capacity} = y_3 + y_5 + 2y_6 + 3y_7$$

The observation that the original FFD packing and the re-packing, described by the type vector $\mathbf{x}$, both have the same total unused capacity is expressed by the equation.

$$x_3 + x_5 + 2x_6 + 3x_7 = y_3 + y_5 + 2y_6 + 3y_7$$

Similarly, the observation that both packings have the same total number of boxes of size 1 leads to the equation.

$$x_2 + 3x_4 + 2x_5 + x_6 = y_2 + 3y_4 + 2y_5 + y_6$$

Making the same observation for boxes of size 2 and 3 leads to the following equations

$$x_2 + x_3 = y_2 + y_3$$
$$x_1 = y_1$$

Now consider the case defined by the Boolean vectors

$$\mathbf{o} = (1, 1, 0, 1, 0, 1, 0)$$
$$\mathbf{u} = (1, 1, 0, 1, 0, 0, 0)$$

The remainder of this section proves that if the original FFD packing is a member of the case that the re-packing does not include any bins with a hole of size greater than 2. Because the only content vector with a hole size greater than 2 is $\mathbf{a}_7$ it will suffice to show that $x_7 = 0$.

The definition of a case states that if the original FFD packing is a member of this case defined by the Boolean vectors $\mathbf{o}$ and $\mathbf{u}$, then the components of $\mathbf{y}$ must satisfy the following conditions

$$y_1 \geq 1$$

$$y_2 \geq 1$$
$$y_3 = 0$$
$$y_4 \geq 1$$
$$y_5 = 0$$
$$y_6 = 1$$
$$y_7 = 0$$

The key observation is that the exact values of y_3, y_5, y_6 and y_7 are known. However, there are no upper bounds on the components y_1, y_2, and y_4.

Consider the single equation from the system of equations that equates the total unused capacity in the original FFD packing and the re-packing.

$$x_3 + x_5 + 2x_6 + 3x_7 = y_3 + y_5 + 2y_6 + 3y_7$$

An important property of this equation is that it does not include the unbounded components y_1, y_2 and y_4. These components have been zeroed out. Substituting in the known values of components y_3, y_5, y_6 and y_7 results in the equation

$$x_3 + x_5 + 2x_6 + 3x_7 = 2$$

Eliminating the unbounded components of $\mathbf{y}$ is the key to the remainder of the proof. In this case, a single equation from the system of equations zeroed out the unbounded components of $\mathbf{y}$. In general, to zero out the unbounded components of $\mathbf{y}$ requires a linear combination of more than one equation in the system of equations.

Continuing on with the example, because x_3, x_5, x_6 and x_7 are all non-negative it follows that

$$3x_7 \leq 2$$

Obviously, because 3 is greater than 2 it follows that

$$x_7 < 1$$

However, x_7 must be a non-negative integer, therefore

$$x_7 = 0$$

Because $x_7 = 0$, the re-packing cannot have a hole of size greater than 2.

A.5.2 The General Case

This section describes how to prove Lemma A.4 for a general case defined by the Boolean vectors $\mathbf{o}$ and $\mathbf{u}$. Remember that the original FFD packing is described by the type vector $\mathbf{y}$ and that the re-packing is described by the type vector $\mathbf{x}$. The proof will show for all i where the content vector $\mathbf{a}_i$ has a hole of size greater than H, that the component x_i must be equal to zero.

The system of equations relating the original FFD packing to the re-packing will be expressed in matrix notation using the Content Matrix.

Definition: Content Matrix

The Content Matrix $\mathbf{A}$ is defined as

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1 \\ \vdots \\ \mathbf{a}_m \end{bmatrix} = \begin{bmatrix} a_{10} & \dots & a_{1K} \\ \vdots & & \\ a_{m0} & \dots & a_{mK} \end{bmatrix}$$

where a_1 to a_m are all the distinct content vectors for bins of capacity K.

Using the Content Matrix, the observation that the packings described by $\mathbf{x}$ and $\mathbf{y}$ contain the same boxes and have the same total unused capacity can be expressed by the system of equations

$$\mathbf{xA} = \mathbf{yA}$$

Because the original FFD packing is a member of the case defined by the Boolean vectors $\mathbf{o}$ and $\mathbf{u}$ the components y_i must satisfy one of the following conditions as specified by o_i and u_i.

o_i	u_i	y_i
0	0	$y_i = 0$
1	0	$y_i = 1$
1	1	$y_i \geq 1$

The key to the remainder of the proof is the observation that for all i if $u_i = 0$ then $y_i = o_i$, but if $u_i = 1$ then there is no upper bound on y_i.

The unbounded components of $\mathbf{y}$ are zeroed out by taking the appropriate linear combination of the equations in the system of equations. This linear combination is described by the vector $\mathbf{v}$. After taking the linear combination the resulting equation is

$$\mathbf{x}\mathbf{A}\mathbf{v}^T = \mathbf{y}\mathbf{A}\mathbf{v}^T$$

To ensure that the linear combination zeroes out the unbounded components of $\mathbf{y}$ requires that $\mathbf{v}$ satisfies the following condition

$$\begin{array}{l} \text{for all } i, 1 \leq i \leq m \\ \quad \text{if } u_i = 1 \text{ then } (\mathbf{A}\mathbf{v}^T)_i = 0 \end{array}$$

where $(\mathbf{A}\mathbf{v}^T)_i$ is the ith component of the column vector $\mathbf{A}\mathbf{v}^T$. Knowing that $(\mathbf{A}\mathbf{v}^T)_i = 0$ if $u_i = 1$, and observing that $y_i = o_i$ if $u_i = 0$, allows the above equation to be simplified to

$$\mathbf{x}\mathbf{A}\mathbf{v}^T = \mathbf{o}\mathbf{A}\mathbf{v}^T$$

It is assumed that the vector $\mathbf{v}$ satisfies the following conditions

$$\begin{array}{l} \text{for all } i, 1 \leq i \leq m \\ \quad \text{if } u_i = 0 \text{ then } (\mathbf{A}\mathbf{v}^T)_i \geq 0 \end{array}$$

By definition, $\mathbf{x}\mathbf{A}\mathbf{v}^T = \sum_{i=1}^{m} x_i(\mathbf{A}\mathbf{v}^T)_i$. Therefore, for all i, the components $(\mathbf{A}\mathbf{v}^T)_i$ are non-negative, and it follows that

$$x_i(\mathbf{A}\mathbf{v}^T)_i \leq \mathbf{o}\mathbf{A}\mathbf{v}^T$$

In addition, it is assumed that the vector $\mathbf{v}$ satisfies the following conditions

$$\begin{array}{l} \text{for all } i, 1 \leq i \leq m \\ \quad \text{if } a_{i0} > H \text{ then } (\mathbf{A}\mathbf{v}^T)_i \geq \mathbf{o}\mathbf{A}\mathbf{v}^T \end{array}$$

This leads to the deduction, for all i where the content vector $\mathbf{a}_i$ has a hole of size greater than H, that $x_i < 1$ and therefore that $x_i = 0$.

Therefore, to prove the case it is sufficient to present a vector $\mathbf{v}$ satisfying the three conditions

for all $i, 1 \leq i \leq m$

if $u_i = 1$ then $(\mathbf{Av}^T)_i = 0$

if $u_i = 0$ then $(\mathbf{Av}^T)_i \geq 0$

if $a_{i0} > H$ then $(\mathbf{Av}^T)_i \geq \mathbf{oAv}^T$

A.5.3 Finding the Required Linear Combination

Candidates for the linear combination required to prove the case are found by solving for the basis vectors of the nullspace of the matrix $\mathbf{D}$, where $\mathbf{D}$ is defined as

for all $i, (1 \leq i \leq m)$

for all $j, (1 \leq j \leq K)$

if $u_i = 0$ then $d_{ij} = 0$

if $u_i = 1$ then $d_{ij} = a_{ij}$

If the vector $\mathbf{v}$ is a basis vector for the nullspace of $\mathbf{D}$ then

$$\mathbf{Dv}^T = (0 \ldots 0)^T$$

and from the definition of $\mathbf{D}$ it can be deduced that

for all $i, (1 \leq i \leq m)$

if $u_i = 1$ then $(\mathbf{Av}^T)_i = 0$

This is the first of the three conditions that the linear combination must satisfy to prove the case.

Each basis vector of the nullspace of $\mathbf{D}$ is tested in turn to see if it satisfies the two remaining conditions on the linear combination.

for all $i, (1 \leq i \leq m)$

if $u_i = 0$ then $(\mathbf{Av}^T)_i \geq 0$

if $a_{i0} > H$ then $(\mathbf{Av}^T)_i > \mathbf{oAv}^T$

If any basis vector $\mathbf{v}$ of the nullspace of $\mathbf{D}$ satisfies these two conditions then any repacking of the original FFD packing that is a member of the case defined by the Boolean vectors $\mathbf{o}$ and $\mathbf{u}$ does not contain a hole of size greater than H.

Note that solving for the basis vectors of the nullspace of $\mathbf{D}$ and testing for the last two conditions can be automated for each case.

A.6 Reducing the Number of Cases

This section will show how to reduce the number of separate cases that must be proved for a given value of K and a given value of H. It will be shown that the proof of one case can imply the proof of another case. The first case is said to dominate the second case.

Definition: Dominant Case

> A case $\mathcal{A}$, defined by the Boolean vectors $\mathbf{o}_A$ and $\mathbf{u}_A$ is said to *dominate* another case $\mathcal{B}$, defined by the Boolean vectors $\mathbf{o}_B$ and $\mathbf{u}_B$ if and only if
>
> for all $i, (1 \leq i \leq m)$
>
> if $u_{Ai} = 0$ then $u_{Bi} = 0$ and $o_{Ai} \geq o_{Bi}$

Consider a case, $\mathcal{A}$, defined by the Boolean vectors $\mathbf{o}_A$ and $\mathbf{u}_A$, and another case, $\mathcal{B}$, defined by the Boolean vectors $\mathbf{o}_B$ and $\mathbf{u}_B$. To prove case $\mathcal{A}$ requires a vector $\mathbf{v}$ that satisfies the three conditions

for all $i, 1 \leq i \leq m$

if $u_{Ai} = 1$ then $(\mathbf{A}\mathbf{v}^T)_i = 0$

if $u_{Ai} = 0$ then $(\mathbf{A}\mathbf{v}^T)_i \geq 0$

if $a_{i0} > H$ then $(\mathbf{A}\mathbf{v}^T)_i \geq \mathbf{o}_A\mathbf{A}\mathbf{v}^T$

Similarly, to prove case $\mathcal{B}$ requires a vector $\mathbf{v}$ that satisfies the three conditions

for all $i, 1 \leq i \leq m$

if $u_{Bi} = 1$ then $(\mathbf{A}\mathbf{v}^T)_i = 0$

if $u_{Bi} = 0$ then $(\mathbf{A}\mathbf{v}^T)_i \geq 0$

if $a_{i0} > H$ then $(\mathbf{A}\mathbf{v}^T)_i \geq \mathbf{o}_B\mathbf{A}\mathbf{v}^T$

It will be shown that if case $\mathcal{A}$ dominates case $\mathcal{B}$ then the vector $\mathbf{v}$ that satisfies the conditions required to prove case $\mathcal{A}$ also satisfies the conditions required to prove case $\mathcal{B}$. First note that because the vector $\mathbf{v}$ satisfies the conditions required to prove case $\mathcal{A}$ that for all i, $(\mathbf{A}\mathbf{v}^T)_i \geq 0$

Because case $\mathcal{A}$ dominates case $\mathcal{B}$ it follows for all i that if $u_{Ai} = 0$ then $u_{Bi} = 0$. Therefore if $u_{Bi} = 1$ then $u_{Ai} = 1$, and because $\mathbf{v}$ satisfies the conditions required to prove case $\mathcal{A}$ it can be deduced that $(\mathbf{A}\mathbf{v}^T)_i = 0$.

Next note that because $(\mathbf{A}\mathbf{v}^T)_i = 0$ when $u_{Ai} = 1$, $(\mathbf{A}\mathbf{v}^T)_i \geq 1$ when $u_{Ai} = 0$, and $o_{Ai} \geq o_{Bi}$ when $u_{Ai} = 0$ that $\mathbf{o}_A\mathbf{A}\mathbf{v}^T \geq \mathbf{o}_B\mathbf{A}\mathbf{v}^T$. Therefore

for all $i, 1 \leq i \leq m$

if $u_{Bi} = 1$ then $(\mathbf{A}\mathbf{v}^T)_i = 0$

if $u_{Bi} = 0$ then $(\mathbf{A}\mathbf{v}^T)_i \geq 0$

if $a_{i0} > H$ then $(\mathbf{A}\mathbf{v}^T)_i \geq \mathbf{o}_A\mathbf{A}\mathbf{v}^T \geq \mathbf{o}_B\mathbf{A}\mathbf{v}^T$

The vector $\mathbf{v}$ satisfies the conditions required to prove case $\mathcal{B}$. Note that the process of finding dominant cases can be automated.

A.7 Presentation of the Cases

The following sections present the details of the proofs of Theorems 1 and 2. For each value of K, the Content Matrix $\mathbf{A}$ is presented, and a table records the compatibility of the content vectors in $\mathbf{A}$. In this table an entry of 1 at column i of row j indicates that the content vectors $\mathbf{a}_i$ and $\mathbf{a}_j$ are FFD compatible, and an entry of 0 indicates that they are incompatible. For each combination of K and H, the vectors $\mathbf{o}$, and $\mathbf{u}$ defining each of the dominant cases are presented, and for each of these cases Lemma A.4 is proved by presenting a vector $\mathbf{v}$ satisfying the conditions given in Section A.5.2.

A.8 $K = 2$

Content Matrix, A, for $K = 2$

	0	1	2
1	0	0	1
2	0	2	0
3	1	1	0
4	2	0	0

FFD Compatibility for $K = 2$

	$\mathbf{a}_1$	$\mathbf{a}_2$	$\mathbf{a}_3$	$\mathbf{a}_4$
$\mathbf{a}_1$	1			
$\mathbf{a}_2$	1	1		
$\mathbf{a}_3$	1	1	0	
$\mathbf{a}_4$	1	1	1	1

A.8.1 $K = 2,\ H = 1$

Cases for $K = 2,\ H = 1$

	1
	ou
y_1	11
y_2	11
y_3	10
y_4	00

Case 1, $K = 2,\ H = 1$

$o = (1, 1, 1, 0)$
$u = (1, 1, 0, 0)$

$\mathbf{v} = (1, 0, 0)$
$\mathbf{A}\mathbf{v}^T = (0, 0, 1, 2)^T$
$\mathbf{o}\mathbf{A}\mathbf{v}^T = 1$

A.8.2 $K = 2,\ H = 0$

Cases for $K = 2,\ H = 0$

	1
	ou
y_1	11
y_2	11
y_3	00
y_4	00

Case 1, $K = 2,\ H = 0$

$o = (1, 1, 0, 0)$
$u = (1, 1, 0, 0)$

$\mathbf{v} = (1, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 2)^T$
$\mathbf{oAv}^T = 0$

A.9 $K = 3$

Content Matrix, A, for $K = 3$

	0	1	2	3
1	0	0	0	1
2	0	1	1	0
3	1	0	1	0
4	0	3	0	0
5	1	2	0	0
6	2	1	0	0
7	3	0	0	0

FFD Compatibility for $K = 3$

	$\mathbf{a}_1$	$\mathbf{a}_2$	$\mathbf{a}_3$	$\mathbf{a}_4$	$\mathbf{a}_5$	$\mathbf{a}_6$	$\mathbf{a}_7$
$\mathbf{a}_1$	1						
$\mathbf{a}_2$	1	1					
$\mathbf{a}_3$	1	1	1				
$\mathbf{a}_4$	1	1	0	1			
$\mathbf{a}_5$	1	1	0	1	0		
$\mathbf{a}_6$	1	1	0	1	0	0	
$\mathbf{a}_7$	1	1	1	1	1	1	1

A.9.1 $K = 3,\ H = 2$

Cases for $K = 3,\ H = 2$

	1	2	3
	ou	**ou**	**ou**
y_1	11	11	11
y_2	11	11	11
y_3	00	00	11
y_4	11	11	00
y_5	00	10	00
y_6	10	00	00
y_7	00	00	00

Case 1, $K = 3,\ H = 2$

$o = (1, 1, 0, 1, 0, 1, 0)$
$u = (1, 1, 0, 1, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 1, 2, 3)^T$
$\mathbf{oAv}^T = 2$

Case 2, $K = 3,\ H = 2$

$o = (1, 1, 0, 1, 1, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 1, 2, 3)^T$
$\mathbf{oAv}^T = 1$

Case 3, $K = 3,\ H = 2$

$o = (1, 1, 1, 0, 0, 0, 0)$
$u = (1, 1, 1, 0, 0, 0, 0)$

$\mathbf{v} = (1, 1, -1, 0)$
$\mathbf{Av}^T = (0, 0, 0, 3, 3, 3, 3)^T$
$\mathbf{oAv}^T = 0$

A.9.2 $K = 3,\ H = 1$

Cases for $K = 3,\ H = 1$

	1	2
	ou	**ou**
y_1	11	11
y_2	11	11
y_3	00	11
y_4	11	00
y_5	10	00
y_6	00	00
y_7	00	00

Case 1, $K = 3,\ H = 1$

$o = (1, 1, 0, 1, 1, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 1, 2, 3)^T$
$\mathbf{oAv}^T = 1$

Case 2, $K = 3,\ H = 1$

$o = (1, 1, 1, 0, 0, 0, 0)$
$u = (1, 1, 1, 0, 0, 0, 0)$

$\mathbf{v} = (1, 1, -1, 0)$
$\mathbf{Av}^T = (0, 0, 0, 3, 3, 3, 3)^T$
$\mathbf{oAv}^T = 0$

A.9.3 $K = 3,\ H = 0$

Cases for $K = 3,\ H = 0$

	1
	ou
y_1	11
y_2	11
y_3	00
y_4	11
y_5	00
y_6	00
y_7	00

Case 1, $K = 3,\ H = 0$

$o = (1, 1, 0, 1, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 1, 2, 3)^T$
$\mathbf{oAv}^T = 0$

A.10 $K = 4$

Content Matrix, A, for $K = 4$

	0	1	2	3	4
1	0	0	0	0	1
2	0	1	0	1	0
3	1	0	0	1	0
4	0	0	2	0	0
5	0	2	1	0	0
6	1	1	1	0	0
7	2	0	1	0	0
8	0	4	0	0	0
9	1	3	0	0	0
10	2	2	0	0	0
11	3	1	0	0	0
12	4	0	0	0	0

FFD Compatibility for $K = 4$

	$\mathbf{a}_1$	$\mathbf{a}_2$	$\mathbf{a}_3$	$\mathbf{a}_4$	$\mathbf{a}_5$	$\mathbf{a}_6$	$\mathbf{a}_7$	$\mathbf{a}_8$	$\mathbf{a}_9$	$\mathbf{a}_{10}$	$\mathbf{a}_{11}$	$\mathbf{a}_{12}$
$\mathbf{a}_1$	1											
$\mathbf{a}_2$	1	1										
$\mathbf{a}_3$	1	1	1									
$\mathbf{a}_4$	1	1	1	1								
$\mathbf{a}_5$	1	1	0	1	0							
$\mathbf{a}_6$	1	1	0	1	0	0						
$\mathbf{a}_7$	1	1	1	1	0	0	0					
$\mathbf{a}_8$	1	1	0	1	1	0	0	1				
$\mathbf{a}_9$	1	1	0	1	1	0	0	1	0			
$\mathbf{a}_{10}$	1	1	0	1	1	0	0	1	0	0		
$\mathbf{a}_{11}$	1	1	0	1	1	0	0	1	0	0	0	
$\mathbf{a}_{12}$	1	1	1	1	1	1	1	1	1	1	1	1

A.10.1 $K = 4,\ H = 3$

Cases for $K = 4,\ H = 3$

	1	2	3	4	5
	ou	**ou**	**ou**	**ou**	**ou**
y_1	11	11	11	11	11
y_2	11	11	11	11	11
y_3	00	00	00	00	11
y_4	11	11	11	11	11
y_5	00	10	10	10	00
y_6	10	00	00	00	00
y_7	00	00	00	00	10
y_8	00	11	11	11	00
y_9	00	00	00	10	00
y_{10}	00	00	10	00	00
y_{11}	00	10	00	00	00
y_{12}	00	00	00	00	00

Case 1, $K = 4,\ H = 3$

$o = (1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 2, 3, 4)^T$
$\mathbf{oAv}^T = 1$

Case 2, $K = 4,\ H = 3$

$o = (1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 2, 3, 4)^T$
$\mathbf{oAv}^T = 3$

Case 3, $K = 4,\ H = 3$

$o = (1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 2, 3, 4)^T$
$\mathbf{oAv}^T = 2$

Case 4, $K = 4$**,** $H = 3$

$o = (1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 2, 3, 4)^T$
$\mathbf{oAv}^T = 1$

Case 5, $K = 4$**,** $H = 3$

$o = (1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0)$
$u = (1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 1, 0, -1, 0)$
$\mathbf{Av}^T = (0, 0, 0, 0, 2, 2, 2, 4, 4, 4, 4, 4)^T$
$\mathbf{oAv}^T = 2$

A.10.2 $K = 4,\ H = 2$

Cases for $K = 4,\ H = 2$

	1	2	3	4
	ou	**ou**	**ou**	**ou**
y_1	11	11	11	11
y_2	11	11	11	11
y_3	00	00	00	11
y_4	11	11	11	11
y_5	00	10	10	00
y_6	10	00	00	00
y_7	00	00	00	10
y_8	00	11	11	00
y_9	00	00	10	00
y_{10}	00	10	00	00
y_{11}	00	00	00	00
y_{12}	00	00	00	00

Case 1, $K = 4,\ H = 2$

$o = (1,1,0,1,0,1,0,0,0,0,0,0)$
$u = (1,1,0,1,0,0,0,0,0,0,0,0)$

$\mathbf{v} = (1,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,1,2,3,4)^T$
$\mathbf{oAv}^T = 1$

Case 2, $K = 4,\ H = 2$

$o = (1,1,0,1,1,0,0,1,0,1,0,0)$
$u = (1,1,0,1,0,0,0,1,0,0,0,0)$

$\mathbf{v} = (1,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,1,2,3,4)^T$
$\mathbf{oAv}^T = 2$

Case 3, $K = 4,\ H = 2$

$o = (1,1,0,1,1,0,0,1,1,0,0,0)$
$u = (1,1,0,1,0,0,0,1,0,0,0,0)$

$\mathbf{v} = (1,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,1,2,3,4)^T$
$\mathbf{oAv}^T = 1$

Case 4, $K = 4$**,** $H = 2$

$o = (1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0)$
$u = (1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 1, 0, -1, 0)$
$\mathbf{Av}^T = (0, 0, 0, 0, 2, 2, 2, 2, 4, 4, 4, 4, 4)^T$
$\mathbf{oAv}^T = 2$

A.10.3 $K = 4,\ H = 1$

Cases for $K = 4,\ H = 1$

	1	2	3
	ou	**ou**	**ou**
y_1	11	11	11
y_2	11	11	11
y_3	00	00	11
y_4	11	11	11
y_5	00	10	00
y_6	10	00	00
y_7	00	00	00
y_8	00	11	00
y_9	00	10	00
y_{10}	00	00	00
y_{11}	00	00	00
y_{12}	00	00	00

Case 1, $K = 4,\ H = 1$

$o = (1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 2, 3, 4)^T$
$\mathbf{oAv}^T = 1$

Case 2, $K = 4,\ H = 1$

$o = (1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 2, 3, 4)^T$
$\mathbf{oAv}^T = 1$

Case 3, $K = 4,\ H = 1$

$o = (1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 1, 0, -1, 0)$
$\mathbf{Av}^T = (0, 0, 0, 0, 2, 2, 2, 2, 4, 4, 4, 4, 4)^T$
$\mathbf{oAv}^T = 0$

A.10.4 $K = 4$, $H = 0$

Cases for $K = 4$, $H = 0$

	1
	ou
y_1	11
y_2	11
y_3	00
y_4	11
y_5	10
y_6	00
y_7	00
y_8	11
y_9	00
y_{10}	00
y_{11}	00
y_{12}	00

Case 1, $K = 4$, $H = 0$

$o = (1,1,0,1,1,0,0,1,0,0,0,0)$
$u = (1,1,0,1,0,0,0,1,0,0,0,0)$

$\mathbf{v} = (1,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,1,2,3,4)^T$
$\mathbf{oAv}^T = 0$

A.11 $K = 5$

Content Matrix, A, for $K = 5$

	0	1	2	3	4	5
1	0	0	0	0	0	1
2	0	1	0	0	1	0
3	1	0	0	0	1	0
4	0	0	1	1	0	0
5	0	2	0	1	0	0
6	1	1	0	1	0	0
7	2	0	0	1	0	0
8	0	1	2	0	0	0
9	1	0	2	0	0	0
10	0	3	1	0	0	0
11	1	2	1	0	0	0
12	2	1	1	0	0	0
13	3	0	1	0	0	0
14	0	5	0	0	0	0
15	1	4	0	0	0	0
16	2	3	0	0	0	0
17	3	2	0	0	0	0
18	4	1	0	0	0	0
19	5	0	0	0	0	0

FFD Compatibility for $K = 5$

	$\mathbf{a}_1$	$\mathbf{a}_2$	$\mathbf{a}_3$	$\mathbf{a}_4$	$\mathbf{a}_5$	$\mathbf{a}_6$	$\mathbf{a}_7$	$\mathbf{a}_8$	$\mathbf{a}_9$	$\mathbf{a}_{10}$	$\mathbf{a}_{11}$	$\mathbf{a}_{12}$	$\mathbf{a}_{13}$	$\mathbf{a}_{14}$	$\mathbf{a}_{15}$	$\mathbf{a}_{16}$	$\mathbf{a}_{17}$	$\mathbf{a}_{18}$	$\mathbf{a}_{19}$
$\mathbf{a}_1$	1																		
$\mathbf{a}_2$	1	1																	
$\mathbf{a}_3$	1	1	1																
$\mathbf{a}_4$	1	1	1	1															
$\mathbf{a}_5$	1	1	0	1	1														
$\mathbf{a}_6$	1	1	0	1	1	0													
$\mathbf{a}_7$	1	1	1	1	1	1	1												
$\mathbf{a}_8$	1	1	0	1	0	0	0	1											
$\mathbf{a}_9$	1	1	1	1	0	0	0	1	1										
$\mathbf{a}_{10}$	1	1	0	1	0	0	0	1	0	0									
$\mathbf{a}_{11}$	1	1	0	1	0	0	0	1	0	0	0								
$\mathbf{a}_{12}$	1	1	0	1	0	0	0	1	0	0	0	0							
$\mathbf{a}_{13}$	1	1	1	1	0	0	0	1	1	0	0	0	0						
$\mathbf{a}_{14}$	1	1	0	1	1	0	0	1	0	1	0	0	0	1					
$\mathbf{a}_{15}$	1	1	0	1	1	0	0	1	0	1	0	0	0	1	0				
$\mathbf{a}_{16}$	1	1	0	1	1	0	0	1	0	1	0	0	0	1	0	0			
$\mathbf{a}_{17}$	1	1	0	1	1	0	0	1	0	1	0	0	0	1	0	0	0		
$\mathbf{a}_{18}$	1	1	0	1	1	0	0	1	0	1	0	0	0	1	0	0	0	0	
$\mathbf{a}_{19}$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

A.11.1 $K = 5,\ H = 4$

Cases for $K = 5,\ H = 4$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	ou	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**
y_1	11	11	11	11	11	11	11	11	11	11	11	11	11	11
y_2	11	11	11	11	11	11	11	11	11	11	11	11	11	11
y_3	00	00	00	00	00	00	00	00	00	00	00	00	11	11
y_4	11	11	11	11	11	11	11	11	11	11	11	11	11	11
y_5	00	00	00	00	00	00	00	11	11	11	11	11	00	00
y_6	00	00	00	00	00	00	00	00	00	00	00	10	00	00
y_7	00	00	00	00	00	00	00	00	00	00	00	11	00	11
y_8	11	11	11	11	11	11	11	00	00	00	00	00	00	00
y_9	00	00	00	00	00	00	11	00	00	00	00	00	11	00
y_{10}	00	00	10	10	10	10	00	00	00	00	00	00	00	00
y_{11}	00	10	00	00	00	00	00	00	00	00	00	00	00	00
y_{12}	10	00	00	00	00	00	00	00	00	00	00	00	00	00
y_{13}	00	00	00	00	00	00	10	00	00	00	00	00	10	00
y_{14}	00	00	11	11	11	11	00	11	11	11	11	00	00	00
y_{15}	00	00	00	00	00	10	00	00	00	00	10	00	00	00
y_{16}	00	00	00	00	10	00	00	00	00	10	00	00	00	00
y_{17}	00	00	00	10	00	00	00	00	10	00	00	00	00	00
y_{18}	00	00	10	00	00	00	00	10	00	00	00	00	00	00
y_{19}	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Case 1, $K = 5,\ H = 4$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5)^T$
$\mathbf{oAv}^T = 2$

Case 2, $K = 5,\ H = 4$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5)^T$
$\mathbf{oAv}^T = 1$

Case 3, $K = 5$**,** $H = 4$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5)^T$
$\mathbf{oAv}^T = 4$

Case 4, $K = 5$**,** $H = 4$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5)^T$
$\mathbf{oAv}^T = 3$

Case 5, $K = 5$**,** $H = 4$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5)^T$
$\mathbf{oAv}^T = 2$

Case 6, $K = 5$**,** $H = 4$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5)^T$
$\mathbf{oAv}^T = 1$

Case 7, $K = 5$**,** $H = 4$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (2, 2, -1, 1, -2, 0)$
$\mathbf{Av}^T = (0, 0, 0, 0, 5, 5, 5, 5, 0, 0, 5, 5, 5, 5, 10, 10, 10, 10, 10, 10)^T$
$\mathbf{oAv}^T = 5$

Case 8, $K = 5$, $H = 4$

$o = (1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0)$
$u = (1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5)^T$
$\mathbf{oAv}^T = 4$

Case 9, $K = 5$, $H = 4$

$o = (1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0)$
$u = (1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5)^T$
$\mathbf{oAv}^T = 3$

Case 10, $K = 5$, $H = 4$

$o = (1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0)$
$u = (1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5)^T$
$\mathbf{oAv}^T = 2$

Case 11, $K = 5$, $H = 4$

$o = (1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5)^T$
$\mathbf{oAv}^T = 1$

Case 12, $K = 5$, $H = 4$

$o = (1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 1, 2, -2, -1, 0)$
$\mathbf{Av}^T = (0, 0, 0, 0, 0, 0, 0, 0, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5)^T$
$\mathbf{oAv}^T = 0$

Case 13, $K = 5$**,** $H = 4$

$o = (1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (2, 2, -1, 1, -2, 0)$
$\mathbf{Av}^T = (0, 0, 0, 0, 5, 5, 5, 0, 0, 5, 5, 5, 5, 10, 10, 10, 10, 10, 10)^T$
$\mathbf{oAv}^T = 5$

Case 14, $K = 5$**,** $H = 4$

$o = (1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 1, 2, -2, -1, 0)$
$\mathbf{Av}^T = (0, 0, 0, 0, 0, 0, 0, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5)^T$
$\mathbf{oAv}^T = 0$

A.11.2 $K = 5,\ H = 3$

Cases for $K = 5,\ H = 3$

	1	2	3	4	5	6	7	8	9	10	11	12
	ou	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**
y_1	11	11	11	11	11	11	11	11	11	11	11	11
y_2	11	11	11	11	11	11	11	11	11	11	11	11
y_3	00	00	00	00	00	00	00	00	00	00	11	11
y_4	11	11	11	11	11	11	11	11	11	11	11	11
y_5	00	00	00	00	00	00	11	11	11	11	00	00
y_6	00	00	00	00	00	00	00	00	00	10	00	00
y_7	00	00	00	00	00	00	00	00	00	11	00	11
y_8	11	11	11	11	11	11	00	00	00	00	00	00
y_9	00	00	00	00	00	11	00	00	00	00	11	00
y_{10}	00	00	10	10	10	00	00	00	00	00	00	00
y_{11}	00	10	00	00	00	00	00	00	00	00	00	00
y_{12}	10	00	00	00	00	00	00	00	00	00	00	00
y_{13}	00	00	00	00	00	10	00	00	00	00	10	00
y_{14}	00	00	11	11	11	00	11	11	11	00	00	00
y_{15}	00	00	00	00	10	00	00	00	10	00	00	00
y_{16}	00	00	00	10	00	00	00	10	00	00	00	00
y_{17}	00	00	10	00	00	00	10	00	00	00	00	00
y_{18}	00	00	00	00	00	00	00	00	00	00	00	00
y_{19}	00	00	00	00	00	00	00	00	00	00	00	00

Case 1, $K = 5,\ H = 3$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5)^T$
$\mathbf{oAv}^T = 2$

Case 2, $K = 5,\ H = 3$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5)^T$
$\mathbf{oAv}^T = 1$

Case 3, $K = 5,\ H = 3$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5)^T$
$\mathbf{oAv}^T = 3$

Case 4, $K = 5,\ H = 3$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5)^T$
$\mathbf{oAv}^T = 2$

Case 5, $K = 5,\ H = 3$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5)^T$
$\mathbf{oAv}^T = 1$

Case 6, $K = 5,\ H = 3$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (2, 2, -1, 1, -2, 0)$
$\mathbf{Av}^T = (0, 0, 0, 0, 5, 5, 5, 0, 0, 5, 5, 5, 5, 10, 10, 10, 10, 10, 10)^T$
$\mathbf{oAv}^T = 5$

Case 7, $K = 5,\ H = 3$

$o = (1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0)$
$u = (1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5)^T$
$\mathbf{oAv}^T = 3$

Case 8, $K = 5,\ H = 3$

$o = (1,1,0,1,1,0,0,0,0,0,0,0,0,1,0,1,0,0,0)$
$u = (1,1,0,1,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0)$

$\mathbf{v} = (1,0,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,1,0,1,2,3,0,1,2,3,4,5)^T$
$\mathbf{oAv}^T = 2$

Case 9, $K = 5,\ H = 3$

$o = (1,1,0,1,1,0,0,0,0,0,0,0,0,1,1,0,0,0,0)$
$u = (1,1,0,1,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0)$

$\mathbf{v} = (1,0,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,1,0,1,2,3,0,1,2,3,4,5)^T$
$\mathbf{oAv}^T = 1$

Case 10, $K = 5,\ H = 3$

$o = (1,1,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0)$
$u = (1,1,0,1,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0)$

$\mathbf{v} = (1,1,2,-2,-1,0)$
$\mathbf{Av}^T = (0,0,0,0,0,0,0,5,5,5,5,5,5,5,5,5,5,5,5)^T$
$\mathbf{oAv}^T = 0$

Case 11, $K = 5,\ H = 3$

$o = (1,1,1,1,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0)$
$u = (1,1,1,1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0)$

$\mathbf{v} = (2,2,-1,1,-2,0)$
$\mathbf{Av}^T = (0,0,0,0,5,5,5,0,0,5,5,5,5,10,10,10,10,10,10)^T$
$\mathbf{oAv}^T = 5$

Case 12, $K = 5,\ H = 3$

$o = (1,1,1,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0)$
$u = (1,1,1,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0)$

$\mathbf{v} = (1,1,2,-2,-1,0)$
$\mathbf{Av}^T = (0,0,0,0,0,0,0,5,5,5,5,5,5,5,5,5,5,5,5)^T$
$\mathbf{oAv}^T = 0$

A.11.3 $K = 5,\ H = 2$

Cases for $K = 5,\ H = 2$

	1	2	3	4	5	6	7	8	9	10
	ou	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**
y_1	11	11	11	11	11	11	11	11	11	11
y_2	11	11	11	11	11	11	11	11	11	11
y_3	00	00	00	00	00	00	00	00	11	11
y_4	11	11	11	11	11	11	11	11	11	11
y_5	00	00	00	00	00	11	11	11	00	00
y_6	00	00	00	00	00	00	00	10	00	00
y_7	00	00	00	00	00	00	00	11	00	11
y_8	11	11	11	11	11	00	00	00	00	00
y_9	00	00	00	00	11	00	00	00	11	00
y_{10}	00	00	10	10	00	00	00	00	00	00
y_{11}	00	10	00	00	00	00	00	00	00	00
y_{12}	10	00	00	00	00	00	00	00	00	00
y_{13}	00	00	00	00	00	00	00	00	00	00
y_{14}	00	00	11	11	00	11	11	00	00	00
y_{15}	00	00	00	10	00	00	10	00	00	00
y_{16}	00	00	10	00	00	10	00	00	00	00
y_{17}	00	00	00	00	00	00	00	00	00	00
y_{18}	00	00	00	00	00	00	00	00	00	00
y_{19}	00	00	00	00	00	00	00	00	00	00

Case 1, $K = 5,\ H = 2$

$$o = (1,1,0,1,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0)$$
$$u = (1,1,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0)$$

$$\mathbf{v} = (1,0,0,0,0,0)$$
$$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,1,0,1,2,3,0,1,2,3,4,5)^T$$
$$\mathbf{oAv}^T = 2$$

Case 2, $K = 5,\ H = 2$

$$o = (1,1,0,1,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0)$$
$$u = (1,1,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0)$$

$$\mathbf{v} = (1,0,0,0,0,0)$$
$$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,1,0,1,2,3,0,1,2,3,4,5)^T$$
$$\mathbf{oAv}^T = 1$$

Case 3, $K = 5,\ H = 2$

$o = (1,1,0,1,0,0,0,1,0,1,0,0,0,1,0,1,0,0,0)$
$u = (1,1,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0)$

$\mathbf{v} = (1,0,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,1,0,1,2,3,0,1,2,3,4,5)^T$
$\mathbf{oAv}^T = 2$

Case 4, $K = 5,\ H = 2$

$o = (1,1,0,1,0,0,0,1,0,1,0,0,0,1,1,0,0,0,0)$
$u = (1,1,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0)$

$\mathbf{v} = (1,0,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,1,0,1,2,3,0,1,2,3,4,5)^T$
$\mathbf{oAv}^T = 1$

Case 5, $K = 5,\ H = 2$

$o = (1,1,0,1,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0)$
$u = (1,1,0,1,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0)$

$\mathbf{v} = (2,2,-1,1,-2,0)$
$\mathbf{Av}^T = (0,0,0,0,5,5,5,0,0,5,5,5,5,10,10,10,10,10,10)^T$
$\mathbf{oAv}^T = 0$

Case 6, $K = 5,\ H = 2$

$o = (1,1,0,1,1,0,0,0,0,0,0,0,0,1,0,1,0,0,0)$
$u = (1,1,0,1,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0)$

$\mathbf{v} = (1,0,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,1,0,1,2,3,0,1,2,3,4,5)^T$
$\mathbf{oAv}^T = 2$

Case 7, $K = 5,\ H = 2$

$o = (1,1,0,1,1,0,0,0,0,0,0,0,0,1,1,0,0,0,0)$
$u = (1,1,0,1,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0)$

$\mathbf{v} = (1,0,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,1,0,1,2,3,0,1,2,3,4,5)^T$
$\mathbf{oAv}^T = 1$

Case 8, $K = 5$, $H = 2$

$o = (1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 1, 2, -2, -1, 0)$
$\mathbf{Av}^T = (0, 0, 0, 0, 0, 0, 0, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5)^T$
$\mathbf{oAv}^T = 0$

Case 9, $K = 5$, $H = 2$

$o = (1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (2, 2, -1, 1, -2, 0)$
$\mathbf{Av}^T = (0, 0, 0, 0, 5, 5, 5, 0, 0, 5, 5, 5, 5, 10, 10, 10, 10, 10, 10)^T$
$\mathbf{oAv}^T = 0$

Case 10, $K = 5$, $H = 2$

$o = (1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 1, 2, -2, -1, 0)$
$\mathbf{Av}^T = (0, 0, 0, 0, 0, 0, 0, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5)^T$
$\mathbf{oAv}^T = 0$

A.11.4 $K = 5,\ H = 1$

Cases for $K = 5,\ H = 1$

	1	2	3	4	5	6
	ou	**ou**	**ou**	**ou**	**ou**	**ou**
y_1	11	11	11	11	11	11
y_2	11	11	11	11	11	11
y_3	00	00	00	00	00	11
y_4	11	11	11	11	11	11
y_5	00	00	00	11	11	00
y_6	00	00	00	00	10	00
y_7	00	00	00	00	00	00
y_8	11	11	11	00	00	00
y_9	00	00	11	00	00	11
y_{10}	00	10	00	00	00	00
y_{11}	10	00	00	00	00	00
y_{12}	00	00	00	00	00	00
y_{13}	00	00	00	00	00	00
y_{14}	00	11	00	11	00	00
y_{15}	00	10	00	10	00	00
y_{16}	00	00	00	00	00	00
y_{17}	00	00	00	00	00	00
y_{18}	00	00	00	00	00	00
y_{19}	00	00	00	00	00	00

Case 1, $K = 5,\ H = 1$

$o = (1,1,0,1,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0)$
$u = (1,1,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0)$

$\mathbf{v} = (1,0,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,1,0,1,2,3,0,1,2,3,4,5)^T$
$\mathbf{oAv}^T = 1$

Case 2, $K = 5,\ H = 1$

$o = (1,1,0,1,0,0,0,1,0,1,0,0,0,1,1,0,0,0,0)$
$u = (1,1,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0)$

$\mathbf{v} = (1,0,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,1,0,1,2,3,0,1,2,3,4,5)^T$
$\mathbf{oAv}^T = 1$

Case 3, $K = 5$, $H = 1$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (2, 2, -1, 1, -2, 0)$
$\mathbf{Av}^T = (0, 0, 0, 0, 5, 5, 5, 0, 0, 5, 5, 5, 5, 10, 10, 10, 10, 10, 10)^T$
$\mathbf{oAv}^T = 0$

Case 4, $K = 5$, $H = 1$

$o = (1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5)^T$
$\mathbf{oAv}^T = 1$

Case 5, $K = 5$, $H = 1$

$o = (1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5)^T$
$\mathbf{oAv}^T = 1$

Case 6, $K = 5$, $H = 1$

$o = (1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (2, 2, -1, 1, -2, 0)$
$\mathbf{Av}^T = (0, 0, 0, 0, 5, 5, 5, 0, 0, 5, 5, 5, 5, 10, 10, 10, 10, 10, 10)^T$
$\mathbf{oAv}^T = 0$

A.11.5 $K = 5,\ H = 0$

Cases for $K = 5,\ H = 0$

	1	2
	ou	**ou**
y_1	11	11
y_2	11	11
y_3	00	00
y_4	11	11
y_5	00	11
y_6	00	00
y_7	00	00
y_8	11	00
y_9	00	00
y_{10}	10	00
y_{11}	00	00
y_{12}	00	00
y_{13}	00	00
y_{14}	11	11
y_{15}	00	00
y_{16}	00	00
y_{17}	00	00
y_{18}	00	00
y_{19}	00	00

Case 1, $K = 5,\ H = 0$

$$o = (1,1,0,1,0,0,0,1,0,1,0,0,0,1,0,0,0,0,0)$$
$$u = (1,1,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0)$$

$$\mathbf{v} = (1,0,0,0,0,0)$$
$$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,1,0,1,2,3,0,1,2,3,4,5)^T$$
$$\mathbf{oAv}^T = 0$$

Case 2, $K = 5,\ H = 0$

$$o = (1,1,0,1,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0)$$
$$u = (1,1,0,1,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0)$$

$$\mathbf{v} = (1,0,0,0,0,0)$$
$$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,1,0,1,2,3,0,1,2,3,4,5)^T$$
$$\mathbf{oAv}^T = 0$$

A.12 $K = 6$

Content Matrix, A, for $K = 6$

	0	1	2	3	4	5	6
1	0	0	0	0	0	0	1
2	0	1	0	0	0	1	0
3	1	0	0	0	0	1	0
4	0	0	1	0	1	0	0
5	0	2	0	0	1	0	0
6	1	1	0	0	1	0	0
7	2	0	0	0	1	0	0
8	0	0	0	2	0	0	0
9	0	1	1	1	0	0	0
10	1	0	1	1	0	0	0
11	0	3	0	1	0	0	0
12	1	2	0	1	0	0	0
13	2	1	0	1	0	0	0
14	3	0	0	1	0	0	0
15	0	0	3	0	0	0	0
16	0	2	2	0	0	0	0
17	1	1	2	0	0	0	0
18	2	0	2	0	0	0	0
19	0	4	1	0	0	0	0
20	1	3	1	0	0	0	0
21	2	2	1	0	0	0	0
22	3	1	1	0	0	0	0
23	4	0	1	0	0	0	0
24	0	6	0	0	0	0	0
25	1	5	0	0	0	0	0
26	2	4	0	0	0	0	0
27	3	3	0	0	0	0	0
28	4	2	0	0	0	0	0
29	5	1	0	0	0	0	0
30	6	0	0	0	0	0	0

FFD Compatibility for $K = 6$

	$\mathbf{a}_1$	$\mathbf{a}_2$	$\mathbf{a}_3$	$\mathbf{a}_4$	$\mathbf{a}_5$	$\mathbf{a}_6$	$\mathbf{a}_7$	$\mathbf{a}_8$	$\mathbf{a}_9$	$\mathbf{a}_{10}$	$\mathbf{a}_{11}$	$\mathbf{a}_{12}$	$\mathbf{a}_{13}$	$\mathbf{a}_{14}$	$\mathbf{a}_{15}$	$\mathbf{a}_{16}$	$\mathbf{a}_{17}$	$\mathbf{a}_{18}$	$\mathbf{a}_{19}$	$\mathbf{a}_{20}$	$\mathbf{a}_{21}$	$\mathbf{a}_{22}$	$\mathbf{a}_{23}$	$\mathbf{a}_{24}$	$\mathbf{a}_{25}$	$\mathbf{a}_{26}$	$\mathbf{a}_{27}$	$\mathbf{a}_{28}$	$\mathbf{a}_{29}$	$\mathbf{a}_{30}$
$\mathbf{a}_1$	1																													
$\mathbf{a}_2$	1	1																												
$\mathbf{a}_3$	1	1	1																											
$\mathbf{a}_4$	1	1	1	1																										
$\mathbf{a}_5$	1	1	0	1	1																									
$\mathbf{a}_6$	1	1	0	1	1	0																								
$\mathbf{a}_7$	1	1	1	1	1	1	1																							
$\mathbf{a}_8$	1	1	1	1	1	1	1	1																						
$\mathbf{a}_9$	1	1	0	1	0	0	0	1	0																					
$\mathbf{a}_{10}$	1	1	1	1	0	0	0	1	0	0																				
$\mathbf{a}_{11}$	1	1	0	1	1	0	0	1	0	0	0																			
$\mathbf{a}_{12}$	1	1	0	1	1	0	0	1	0	0	0	0																		
$\mathbf{a}_{13}$	1	1	0	1	1	0	0	1	0	0	0	0	0																	
$\mathbf{a}_{14}$	1	1	1	1	1	1	1	1	0	0	0	0	0	0																
$\mathbf{a}_{15}$	1	1	1	1	0	0	0	1	1	1	0	0	0	0	1															
$\mathbf{a}_{16}$	1	1	0	1	0	0	0	1	1	0	0	0	0	0	1	0														
$\mathbf{a}_{17}$	1	1	0	1	0	0	0	1	1	0	0	0	0	0	1	0	0													
$\mathbf{a}_{18}$	1	1	1	1	0	0	0	1	1	1	0	0	0	0	1	0	0	0												
$\mathbf{a}_{19}$	1	1	0	1	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0											
$\mathbf{a}_{20}$	1	1	0	1	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0	0										
$\mathbf{a}_{21}$	1	1	0	1	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0									
$\mathbf{a}_{22}$	1	1	0	1	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0								
$\mathbf{a}_{23}$	1	1	1	1	0	0	0	1	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0							
$\mathbf{a}_{24}$	1	1	0	1	1	0	0	1	1	0	1	0	0	0	1	1	0	0	1	0	0	0	0	1						
$\mathbf{a}_{25}$	1	1	0	1	1	0	0	1	1	0	1	0	0	0	1	1	0	0	1	0	0	0	0	1	0					
$\mathbf{a}_{26}$	1	1	0	1	1	0	0	1	1	0	1	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0				
$\mathbf{a}_{27}$	1	1	0	1	1	0	0	1	1	0	1	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0	0			
$\mathbf{a}_{28}$	1	1	0	1	1	0	0	1	1	0	1	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0	0	0		
$\mathbf{a}_{29}$	1	1	0	1	1	0	0	1	1	0	1	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0	0	0	0	
$\mathbf{a}_{30}$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

A.12.1 $K = 6,\ H = 5$

Cases for $K = 6,\ H = 5$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	ou	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**
y_1	11	11	11	11	11	11	11	11	11	11	11	11	11	11
y_2	11	11	11	11	11	11	11	11	11	11	11	11	11	11
y_3	00	00	00	00	00	00	00	00	00	00	00	00	00	00
y_4	11	11	11	11	11	11	11	11	11	11	11	11	11	11
y_5	00	00	00	00	00	00	00	00	00	00	00	00	00	00
y_6	00	00	00	00	00	00	00	00	00	00	00	00	00	00
y_7	00	00	00	00	00	00	00	00	00	00	00	00	00	00
y_8	11	11	11	11	11	11	11	11	11	11	11	11	11	11
y_9	10	10	10	10	10	10	10	10	10	10	10	10	10	10
y_{10}	00	00	00	00	00	00	00	00	00	00	00	00	00	00
y_{11}	00	00	00	00	00	00	00	00	00	00	00	00	00	00
y_{12}	00	00	00	00	00	00	00	00	00	00	00	00	00	00
y_{13}	00	00	00	00	00	00	00	00	00	00	00	00	00	00
y_{14}	00	00	00	00	00	00	00	00	00	00	00	00	00	00
y_{15}	11	11	11	11	11	11	11	11	11	11	11	11	11	11
y_{16}	00	00	00	00	00	00	00	00	00	00	00	10	10	10
y_{17}	00	00	00	00	00	00	00	00	00	00	10	00	00	00
y_{18}	00	00	00	00	00	00	00	00	00	10	00	00	00	00
y_{19}	00	00	00	00	10	10	10	10	10	00	00	00	00	00
y_{20}	00	00	00	10	00	00	00	00	00	00	00	00	00	00
y_{21}	00	00	10	00	00	00	00	00	00	00	00	00	00	00
y_{22}	00	10	00	00	00	00	00	00	00	00	00	00	00	00
y_{23}	10	00	00	00	00	00	00	00	00	00	00	00	00	00
y_{24}	00	00	00	00	11	11	11	11	11	00	00	11	11	11
y_{25}	00	00	00	00	00	00	00	00	10	00	00	00	00	00
y_{26}	00	00	00	00	00	00	00	10	00	00	00	00	00	00
y_{27}	00	00	00	00	00	00	10	00	00	00	00	00	00	10
y_{28}	00	00	00	00	00	10	00	00	00	00	00	00	10	00
y_{29}	00	00	00	00	10	00	00	00	00	00	00	10	00	00
y_{30}	00	00	00	00	00	00	00	00	00	00	00	00	00	00

	15	16	17	18	19	20	21	22	23	24	25	26	27
	ou	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**	**ou**
y_1	11	11	11	11	11	11	11	11	11	11	11	11	11
y_2	11	11	11	11	11	11	11	11	11	11	11	11	11
y_3	00	00	00	00	00	00	00	00	00	00	11	11	11
y_4	11	11	11	11	11	11	11	11	11	11	11	11	11
y_5	00	00	11	11	11	11	11	11	11	11	00	00	00
y_6	00	00	00	00	00	00	00	00	00	10	00	00	00
y_7	00	00	00	00	00	00	00	00	00	11	00	00	11
y_8	11	11	11	11	11	11	11	11	11	11	11	11	11
y_9	10	10	00	00	00	00	00	00	00	00	00	00	00
y_{10}	00	00	00	00	00	00	00	00	00	00	10	10	00
y_{11}	00	00	00	00	10	10	10	10	10	00	00	00	00
y_{12}	00	00	00	10	00	00	00	00	00	00	00	00	00
y_{13}	00	00	10	00	00	00	00	00	00	00	00	00	00
y_{14}	00	00	00	00	00	00	00	00	00	10	00	00	10
y_{15}	11	11	00	00	00	00	00	00	00	00	11	11	00
y_{16}	10	10	00	00	00	00	00	00	00	00	00	00	00
y_{17}	00	00	00	00	00	00	00	00	00	00	00	00	00
y_{18}	00	00	00	00	00	00	00	00	00	00	00	10	00
y_{19}	00	00	00	00	00	00	00	00	00	00	00	00	00
y_{20}	00	00	00	00	00	00	00	00	00	00	00	00	00
y_{21}	00	00	00	00	00	00	00	00	00	00	00	00	00
y_{22}	00	00	00	00	00	00	00	00	00	00	00	00	00
y_{23}	00	00	00	00	00	00	00	00	00	00	10	00	00
y_{24}	11	11	00	00	11	11	11	11	11	00	00	00	00
y_{25}	00	10	00	00	00	00	00	00	10	00	00	00	00
y_{26}	10	00	00	00	00	00	00	10	00	00	00	00	00
y_{27}	00	00	00	00	00	00	10	00	00	00	00	00	00
y_{28}	00	00	00	00	00	10	00	00	00	00	00	00	00
y_{29}	00	00	00	00	10	00	00	00	00	00	00	00	00
y_{30}	00	00	00	00	00	00	00	00	00	00	00	00	00

Case 1, $K = 6$**,** $H = 5$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 0, 1, 0, 1, 2, 3, 0, 0, 1, 2, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 5, 6)^T$
$\mathbf{oAv}^T = 4$

Case 2, $K = 6,\ H = 5$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 0, 1, 0, 1, 2, 3, 0, 0, 1, 2, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 5, 6)^T$
$\mathbf{oAv}^T = 3$

Case 3, $K = 6,\ H = 5$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 0, 1, 0, 1, 2, 3, 0, 0, 1, 2, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 5, 6)^T$
$\mathbf{oAv}^T = 2$

Case 4, $K = 6,\ H = 5$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 0, 1, 0, 1, 2, 3, 0, 0, 1, 2, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 5, 6)^T$
$\mathbf{oAv}^T = 1$

Case 5, $K = 6,\ H = 5$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 0, 1, 0, 1, 2, 3, 0, 0, 1, 2, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 5, 6)^T$
$\mathbf{oAv}^T = 5$

Case 6, $K = 6,\ H = 5$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 0, 1, 0, 1, 2, 3, 0, 0, 1, 2, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 5, 6)^T$
$\mathbf{oAv}^T = 4$

Case 7, $K = 6$**,** $H = 5$

$$o = (1,1,0,1,0,0,0,1,1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,1,0,0,1,0,0,0)$$
$$u = (1,1,0,1,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0)$$

$\mathbf{v} = (1,0,0,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,0,1,0,1,2,3,0,0,1,2,0,1,2,3,4,0,1,2,3,4,5,6)^T$
$\mathbf{oAv}^T = 3$

Case 8, $K = 6$**,** $H = 5$

$$o = (1,1,0,1,0,0,0,1,1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,1,0,1,0,0,0,0)$$
$$u = (1,1,0,1,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0)$$

$\mathbf{v} = (1,0,0,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,0,1,0,1,2,3,0,0,1,2,0,1,2,3,4,0,1,2,3,4,5,6)^T$
$\mathbf{oAv}^T = 2$

Case 9, $K = 6$**,** $H = 5$

$$o = (1,1,0,1,0,0,0,1,1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,1,1,0,0,0,0,0)$$
$$u = (1,1,0,1,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0)$$

$\mathbf{v} = (1,0,0,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,0,1,0,1,2,3,0,0,1,2,0,1,2,3,4,0,1,2,3,4,5,6)^T$
$\mathbf{oAv}^T = 1$

Case 10, $K = 6$**,** $H = 5$

$$o = (1,1,0,1,0,0,0,1,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0)$$
$$u = (1,1,0,1,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)$$

$\mathbf{v} = (1,0,0,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,0,1,0,1,2,3,0,0,1,2,0,1,2,3,4,0,1,2,3,4,5,6)^T$
$\mathbf{oAv}^T = 2$

Case 11, $K = 6$**,** $H = 5$

$$o = (1,1,0,1,0,0,0,1,1,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0)$$
$$u = (1,1,0,1,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)$$

$\mathbf{v} = (1,0,0,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,0,1,0,1,2,3,0,0,1,2,0,1,2,3,4,0,1,2,3,4,5,6)^T$
$\mathbf{oAv}^T = 1$

Case 12, $K = 6$**,** $H = 5$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 0, 1, 0, 1, 2, 3, 0, 0, 1, 2, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 5, 6)^T$
$\mathbf{oAv}^T = 5$

Case 13, $K = 6$**,** $H = 5$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 0, 1, 0, 1, 2, 3, 0, 0, 1, 2, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 5, 6)^T$
$\mathbf{oAv}^T = 4$

Case 14, $K = 6$**,** $H = 5$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 0, 1, 0, 1, 2, 3, 0, 0, 1, 2, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 5, 6)^T$
$\mathbf{oAv}^T = 3$

Case 15, $K = 6$**,** $H = 5$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 0, 1, 0, 1, 2, 3, 0, 0, 1, 2, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 5, 6)^T$
$\mathbf{oAv}^T = 2$

Case 16, $K = 6$**,** $H = 5$

$o = (1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 0, 1, 0, 1, 2, 3, 0, 0, 1, 2, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 5, 6)^T$
$\mathbf{oAv}^T = 1$

Case 17, $K = 6$**,** $H = 5$

$o = (1,1,0,1,1,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)$
$u = (1,1,0,1,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)$

$\mathbf{v} = (1,0,0,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,0,1,0,1,2,3,0,0,1,2,0,1,2,3,4,0,1,2,3,4,5,6)^T$
$\mathbf{oAv}^T = 2$

Case 18, $K = 6$**,** $H = 5$

$o = (1,1,0,1,1,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)$
$u = (1,1,0,1,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)$

$\mathbf{v} = (1,0,0,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,0,1,0,1,2,3,0,0,1,2,0,1,2,3,4,0,1,2,3,4,5,6)^T$
$\mathbf{oAv}^T = 1$

Case 19, $K = 6$**,** $H = 5$

$o = (1,1,0,1,1,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,0)$
$u = (1,1,0,1,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0)$

$\mathbf{v} = (1,0,0,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,0,1,0,1,2,3,0,0,1,2,0,1,2,3,4,0,1,2,3,4,5,6)^T$
$\mathbf{oAv}^T = 5$

Case 20, $K = 6$**,** $H = 5$

$o = (1,1,0,1,1,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0)$
$u = (1,1,0,1,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0)$

$\mathbf{v} = (1,0,0,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,0,1,0,1,2,3,0,0,1,2,0,1,2,3,4,0,1,2,3,4,5,6)^T$
$\mathbf{oAv}^T = 4$

Case 21, $K = 6$**,** $H = 5$

$o = (1,1,0,1,1,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0)$
$u = (1,1,0,1,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0)$

$\mathbf{v} = (1,0,0,0,0,0,0)$
$\mathbf{Av}^T = (0,0,1,0,0,1,2,0,0,1,0,1,2,3,0,0,1,2,0,1,2,3,4,0,1,2,3,4,5,6)^T$
$\mathbf{oAv}^T = 3$

Case 22, $K = 6$**,** $H = 5$

$o = (1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 0, 1, 0, 1, 2, 3, 0, 0, 1, 2, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 5, 6)^T$
$\mathbf{oAv}^T = 2$

Case 23, $K = 6$**,** $H = 5$

$o = (1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 0, 0, 0, 0, 0, 0)$
$\mathbf{Av}^T = (0, 0, 1, 0, 0, 1, 2, 0, 0, 1, 0, 1, 2, 3, 0, 0, 1, 2, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 5, 6)^T$
$\mathbf{oAv}^T = 1$

Case 24, $K = 6$**,** $H = 5$

$o = (1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 1, 2, 0, -2, -1, 0)$
$\mathbf{Av}^T = (0, 0, 0, 0, 0, 0, 0, 0, 3, 3, 3, 3, 3, 3, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6)^T$
$\mathbf{oAv}^T = 3$

Case 25, $K = 6$**,** $H = 5$

$o = (1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 1, 0, 0, 0, -1, 0)$
$\mathbf{Av}^T = (0, 0, 0, 0, 2, 2, 2, 0, 1, 1, 3, 3, 3, 3, 0, 2, 2, 2, 4, 4, 4, 4, 4, 6, 6, 6, 6, 6, 6, 6)^T$
$\mathbf{oAv}^T = 5$

Case 26, $K = 6$**,** $H = 5$

$o = (1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 1, 0, 0, 0, -1, 0)$
$\mathbf{Av}^T = (0, 0, 0, 0, 2, 2, 2, 2, 0, 1, 1, 3, 3, 3, 3, 0, 2, 2, 2, 4, 4, 4, 4, 4, 6, 6, 6, 6, 6, 6, 6)^T$
$\mathbf{oAv}^T = 3$

Case 27, $K = 6$**,** $H = 5$

$o = (1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$
$u = (1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\mathbf{v} = (1, 1, 2, 0, -2, -1, 0)$
$\mathbf{Av}^T = (0, 0, 0, 0, 0, 0, 0, 0, 3, 3, 3, 3, 3, 3, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6)^T$
$\mathbf{oAv}^T = 3$

A.13 Counter Examples

The following counter examples show that Theorem A.1 and Theorem A.2 cannot be extended to all values of K.

Counter Example 1, $K = 6$

The FFD packing of the boxes {3, 2, 2, 2} into bins of capacity 6 consists of the two bins {3, 2}, {2, 2}. The first bin has a hole of size 1 and the second bin has a hole of size 2. The boxes can also be packed into the two capacity 6 bins {3}, {2, 2, 2}. The first bin of this packing has a hole of size 3, which is larger than the biggest hole in the FFD packing.

Counter Example 2, $K = 7$

The FFD packing of the boxes {3, 3, 2, 2, 2, 2} into bins of capacity 7 consists of the three bins {3, 3}, {2, 2, 2}, {2}. The boxes can also be packed into the two capacity 7 bins {3, 2, 2}, {3, 2, 2}. This packing uses fewer bins than the FFD packing.

A.14 Summary

This appendix has presented proofs for two theorems concerning the FFD bin-packing algorithm. These results are used in Chapters 5 and 6 to prove the optimality of the area and delay algorithms. The proofs considered a large number of separate cases, but the derivation of these cases and the proof of each case were automated.

Bibliography

[Abou90] P. Abouzeid, L. Bouchet, K. Sakouti, G. Saucier, P. Sicard, "Lexicographical Expression of Boolean Function for Multilevel Synthesis of high Speed Circuits," Proc. SASHIMI 90, Oct. 1990, pp. 31-39.

[Aho85] A. Aho, M. Ganapathi, "Efficient tree pattern matching: an aid to code generation," 12th ACM Symposium on Principles of Programming Languages, Jan. 1985, pp.334-340.

[Berk88] M. Berkelaar, J. Jess, "Technology Mapping for Standard Cell Generators", Proc. ICCAD-88, Nov 1988, pp. 470-473.

[Bost87] D. Bostick, G. D. Hachtel, R. Jacoby, M. R. Lightner, P. Moceyunas, C. R. Morrison, D. Ravenscroft, "The Boulder Optimal Logic Design System," Proc. ICCAD-87, Nov. 1987, pp. 62-65.

[Bray82] R. K. Brayton, C. McMullen, "The Decomposition and Factorization of Boolean Expressions," Proc. Int. Symp. Circ. Syst. ISCAS, May 1982, pp. 49-54

[Bray87] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, A. Wang, "MIS: a Multiple-Level Logic Optimization System," IEEE Tr. CAD, Vol CAD-6, No. 6, Nov. 1987, pp. 1062-1081.

[Brow92] S. D. Brown, R. J. Francis, J. Rose, Z. G. Vranesic, Field-Programmable Gate Arrays, Kluwer Acedemic Publishers, June 1992.

[Cong92] J. Cong, A. Kahng, P. Trjmar, K. C. Chen, "Graph Based FPGA Technology Mapping For Delay Optimization," Proc. 1st Intl. Workshop on FPGAs, Fep. 1992, pp. 77-82.

[Deva91] S. Devadas, K. Keutzer, S. Malik, "Path Sensitization Conditionts and Delay Computation in Combinational Logic Circuits," MCNC International Workshop on Logic Synthesis, May 1991.

[Detj87] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, A. Wang, "Technology Mapping in MIS", Proc. ICCAD-87, Nov 1987, pp. 116-119.

[Filo91] D. Filo, J. C. Yang, F. Mailhot, G. De Micheli, "Technology Mapping for a Two-Output RAM-based field Programmable Gate Array," Proc. EDAC 91, Feb, 1991, pp. 534-538.

[Fran90] R. J. Francis, J. Rose, K. Chung, "Chortle: A Technology Mapping Program for Lookup Table-Based Field Programmable Gate Arrays," Proc. 27th DAC, June 1990, pp. 613-619.

[Fran91a] R. J. Francis, J. Rose, Z. Vranesic, "Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs," Proc. 28th DAC, June 1991 pp. 227-233.

[Fran91b] R. J. Francis, J, Rose, Z. Vranesic, "Technology Mapping of Lookup Table-Based FPGAs for Performance," Proc. ICCAD-91, Nov, 1991, pp. 568-571.

[Gare79] M. R. Garey, D. S. Johnson, Computers and Intractability, A Guide to the Theory of NP-Completeness, W. H. Freeman and Co., New York, 1979.

[Greg86] D. Gregory, K. Bartlett, A. de Geus, G. Hachtel, "Socrates: a system for automatically synthesizing and optimizing combinational logic," Proc. 23rd DAC, June 1986, pp. 79-85.

[John74] D. S. Johnson, A. Demers, J. F. Ullman, M. R. Garey, R. L. Graham, "Worst-case performance bounds for simple one-dimensional packing algorithms," SIAM Journal of Computing, Vol. 3 1974, pp. 299-325.

[Joyn86] W.H. Joyner Jr., L.H. Trevillyan, D. Brand, T. A. Nix, S.C.Gundersen, "Technology Adaptation in Logic Synthesis," Proc. 23rd DAC, 1986, pp. 94-100.

[Karp91] K. Karplus, "Xmap: a Technology Mapper for Table-lookup Field-Programmable Gate Arrays," Proc, 28th DAC, June 1991, pp. 240-243.

[Keut87] K. Keutzer, "DAGON: Technology Binding and Local Optimization by DAG Matching," Proc. 24th DAC, June 1987, pp. 341-347.

[Koha70] Z. Kohavi, Switching and Finite Automata Theory, McGraw-Hill Inc., 1970.

[Koul92] J. L. Kouloheris, A. El Gamal "FPGA Area versus Cell Granularity - Lookup tables and PLA Cells," FPGA-92 Feb. 1992, pp. 9-14.

[Lewi91] D. M. Lewis, Tortle User's Manual, University of Toronto, Sept. 1991.

[Liem91] C. Liem, M. Lefebvre, "Perfromacne Directed Technology Mapping using Constructive Matching," MCNC International Workshop on Logic Synthesis, May 1991.

[Mail90] F. Mailhot, G. de MIcheli, "Technology Mapping Using Boolean Matching and Don't Care Sets," Proc. EDAC, 1990, pp. 212-216.

[Murg90] R. Murgai, Y, Nishizaki, N. Shenay, R. K. Brayton, A. Sangiovanni-Vincentelli, "Logic Synthesis for Programmable Gate Arrays," Proc. 27th DAC, June 1990, pp. 620-625.

[Murg91a] R. Murgai, N. Shenoy, R.K. Brayton, A. Sangiovanni-Vincentelli, "Improved Logic Synthesis Algorithms for Table Look Up Architectures," Proc. ICCAD, Nov. 1991, pp. 564-567

[Murg91b] R. Murgai, N. Shenoy, R.K. Brayton, "Performance Directed Synthesis for Table Look Up Programmable Gate Arrays," Proc. ICCAD, Nov. 1991, pp. 572-575.

[Rohl91] R. Rohleder, "Marker Overview: User-Programmable Logic," In-Stat Services Research Report, March 1991.

[Rose89] J.S. Rose, R.J. Francis, P. Chow, and D. Lewis, "The Effect of Logic Block Complexity on Area of Programmable Gate Arrays," Proc. CICC, May 1989, pp. 5.3.1-5.3.5.

[Rose90] J. Rose, R. J. Francis, D. Lewis, P. Chow, "Architectures of Field-Programmable Gate Arrays: The effect of Logic Block Functionality of Area Efficiency," IEEE Journal of Solid-State Circuits, Vol. 25, No. 5, Oct. 1990, pp. 1217-1225.

[Roth62] J. P. Roth, R. M. Karp, "Minimization over Boolean Graphs," IBM Journal of Research and Development, vol. 6 no. 2, April 1962, pp. 227-238.

[Rude89] R. Rudell, Logic Synthesis for VLSI Design, Ph.D. Thesis, U.C. Berkeley, Memorandum No. UCB/ERL M89/49, April 1989.

[Sing88] K. J. Singh, A. R. Wang, R. K. Brayton, A. Sangiovanni-Vincentelli, "Timing Optimization of Combinational Logic" Proc. ICCAD-88, Nov 1988, pp.282-285.

[Sing91] S. Singh, J. Rose, D. Lewis, K. Chung, P. Chow "Optimization of Field-Programmable Gate Array Logic Block Architecture for Speed," Proc. CICC, May 1991, pp. 6.1.1 - 6.1.6.

[Sing92] S. Singh, J. Rose, P.Chow, D. Lewis, "The Effect of Logic Block Architecture on FPGA Performance," IEEE Journal of Solid-State Circuits, Vol. 27 No. 3, March 1992, pp. 281-287.

[Woo91] N. Woo, "A Heuristic Method for FPGA Technology Mapping Based on Edge Visibility," Proc. 28th DAC, June 1991 pp. 248-251.

[Yang91] S. Yang, Logic Synthesis and Optimization Benchmarks User Guide, Version 2.0, Microelectronics Center of North Carolina, Jan. 1991.