Architecture and Synthesis of Field-Programmable Gate Arrays with Hard-wired Connections

by

Kevin Charles Kenton Chung

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

Graduate Department of Electrical and Computer Engineering
Computer Group
University of Toronto
Toronto, Ontario
CANADA

“Architecture and Synthesis of Field-Programmable Gate Arrays with Hard-wired Connections”, by Kevin C. K. Chung, Ph. D. 1994, Department of Electrical and Computer Engineering, University of Toronto, Canada.

Abstract

Current Field-Programmable Gate Arrays (FPGAs) are roughly three times slower and ten times less dense than Mask Programmed Gate Arrays (MPGAs) in the same VLSI technology. This speed and density difference arises mainly because of the slow and large programmable connections between FPGA logic blocks.

One way to improve the speed and density of an FPGA is to substitute fast and small fixed metal connections, which we call *hard-wired connections*, between some of the primitive gates or *basic blocks* of an FPGA. We use hard-wired connections in FPGAs with *hard-wired logic blocks* (HLBs), where an HLB consists of several basic blocks connected by hard-wired connections.

This dissertation describes algorithms for mapping basic block circuits to HLB circuits optimized for speed or area. HLB mapping is done in two steps: First, a covering algorithm generates a set of HLB *fragments* to implement the input circuit. Second the covering fragments are packed together to minimize the number of HLBs in the final HLB netlist. We prove that the fragment covering algorithm, when optimizing delay, generates an HLB netlist with minimal number of programmable connections along critical paths. We also prove sufficient conditions for the fragment packing algorithm to generate a minimal number of HLBs and show that all two-level HLB topologies satisfy these conditions.

This dissertation explores a wide selection of LUT-based HLB FPGAs empirically. A suite of benchmark circuits is implemented in each HLB architecture and each circuit’s area and delay is measured. The goal is to find the HLB architectures that will yield fast FPGA circuits with reasonable density, and conversely, dense FPGA circuits with good speed. Since an HLB architecture is defined by its LUT size and its topology, the specific research questions are as follows:

i) Which LUT size should be used to build an HLB-based FPGA that will yield the fastest (densest) circuits with reasonable density (speed)?

ii) Which topologies should be used to build an HLB-based FPGA that will yield the fastest (densest) circuits with reasonable density (speed)?

The results of the empirical study show that 6-input LUTs should be used in HLB-based FPGAs for the fastest circuits with reasonable area and that 5-input LUTs should be used for the smallest circuits with reasonable speed. The topologies that led to the fastest circuits had nodes with a high fan-in of hard-wired connections, while the topologies that gave the densest circuits had all nodes with two or more non-hard-wired inputs.

Acknowledgements

I would like to thank my parents, Joyce and Donald, for providing a loving, nurturing and supportive environment throughout my life. Many thanks to my sister, Donna, for being there when I needed her, especially when mom and dad were not available.

I would like to thank my supervisor, Professor Jonathan Rose, for his unwavering technical, moral and financial support during my thesis. I am a beneficiary of Jonathan's enthusiastic and dedicated attitude towards all of his students and peers, and I hope to provide a similar positive environment for all who work with me.

Thanks to my examiners, Dr. Nam-Sung Woo, Professor David Lewis, Professor Safwat Zaky, Professor Zvonko Vranesic, Professor Derek Corneil and Professor Jonathan Rose for their helpful comments and suggestions for improving my thesis.

Thanks to Bob Francis, Keith Farkas, Steve Wilton and Mike Hutton for proof-reading various parts of my thesis and related papers. Thanks to the students in the computer group, in particular, Bob, Keith, Steve, John C., John N., Heather, Yaska, Mike vdP, Aris, Chris and Lawrence for making my time in school so much fun. Maybe next year, we'll have a winning softball team!

Thanks to the badminton and tennis groupies in Agincourt for the on-court and off-court activities. You have made keeping fit so much fun!

In closing, I'd like thank all the "teachers" in my life (my parents, sister, relatives, friends and professional instructors) for sharing their wisdom with me.

Wisdom... is more precious than rubies;

and all the things you may desire cannot compare with her.

Length of days is in her right hand,

and in her left hand riches and honour.

Her ways are ways of pleasantness,

and all her paths are peace.

——— Proverbs 3:13-17

Table of Contents

List of Tables

List of Figures

Glossary

basic block

the smallest combinational logic unit or primitive gate of the FPGA.

connection box

part of the routing architecture that allows connections between a logic block pin and a routing channel.

covering fragments

a set of HLB fragments that implement the basic block network with minimized area or delay.

D_R

the average delay of a programmable routing connection.

envelope set

the set of HLB architecture (area, speed) points such that no point outside the envelope set has both higher speed and lower area than some point within the envelope set. Conversely, every point outside the envelope set has higher area and lower speed than some point within the envelope set. The envelope set represents the "best" HLB architectures among a given group of HLB architectures.

FFD

first fit decreasing.

flexibility

refers to the number of choices in making a routing connection.

fragment

see **HLB fragment**.

fragment covering

the first phase of the HLB mapping algorithm that generates the covering fragments.

fragment packing

the second phase of the HLB mapping algorithm that packs the set of covering fragments into a minimized number of packed HLBs.

fragment pattern

represents an HLB fragment in the pattern library used during fragment covering.

hard-wired connection (or hard-wired link)

a fixed connection (usually a simple metal wire) between two basic blocks.

hard-wired logic block

an FPGA logic block consisting of several basic blocks connected by hard-wired connections.

HLB

hard-wired logic block.

HLB fragment

a connected subset of the basic blocks of an HLB.

HLB architecture

defined by the granularity of the basic block and the connection topology of the HLB.

HLB-based FPGA

an FPGA whose logic blocks are HLBs.

HLB mapping (or HLB technology mapping)

the phase of logic synthesis that transforms an input basic block network into an output HLB netlist.

HLB template

the graph that describes the HLB.

HLB topology

how the basic blocks of the HLB are connected.

logic block

the part of the FPGA that is used to implement the combinational and sequential logic of a circuit.

logic synthesis

the synthesis step that converts a Boolean description into a netlist of FPGA logic blocks.

lookup-table

a programmable gate that can implement any Boolean function of its inputs.

LUT

see **lookup table**.

LUT size

the number of inputs to the LUT or lookup table.

maximal packing set

the largest possible packing sets.

packed HLB

an HLB during or after the packing algorithm.

packing set

a set of fragments that can be legally packed within the same HLB.

R_B

the area of each routing bit in the connection boxes and switch boxes.

speed versus area curve

the curve made by connecting together the points in the envelope set.

switch box

part of the routing architecture that allows connections between horizontal and vertical routing channels.

Xilinx 4000 CLB

a commercial FPGA with hard-wired LUT basic blocks.

Chapter 1 Introduction

Field-Programmable Gate Arrays (FPGAs) are the newest and currently most popular media for designing *new* digital Application Specific Integrated Circuits (ASICs) [1]. An FPGA consists of an array of user-programmable combinational and sequential logic elements (called logic blocks), which implement the functionality of a circuit, and a set of user-programmable routing resources, which connect the logic blocks [2]. Like Programmable Logic Devices, the designer "manufactures" the ASIC in the office within minutes by programming the logic elements and connections[1]. FPGAs have speed and density within an order of magnitude of Mask-Programmed Gate Arrays (MPGAs), the previous most popular choice of ASIC designers. Since the non-recurring engineering costs of FPGAs are much lower than MPGAs, FPGAs are cheaper when manufactured in small quantities and thus pose less of a financial risk. Another advantage is that FPGAs have the properties of a commodity chip, such as a random access memory: the FPGA chips are all the same and, because of large volumes, can be produced more economically. This commodity property makes FPGAs a more attractive product than MPGAs for silicon foundries, which have become expensive capital ventures.

1.1 Motivation

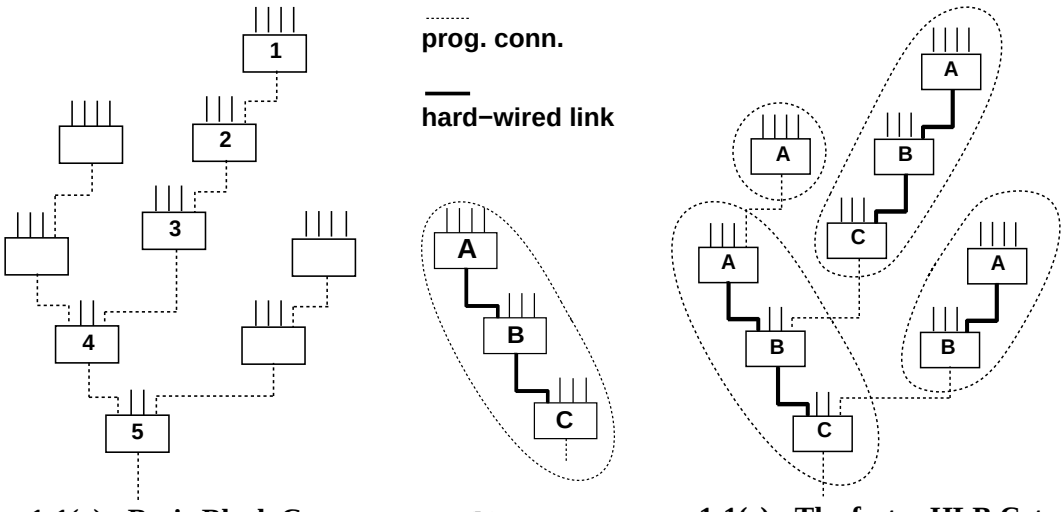
Current FPGAs are roughly three times slower and ten times less dense than MPGAs made in the same fabrication process technology [2]. This disparity is caused mostly by the routing used to connect logic components in each technology. In MPGAs, the logic is connected via mask-pro-

1. The term "field-programmable" means that the ASIC can be manufactured in the designer's office or modified during field operation, without having to send the design to a fabrication plant.

grammed metal wires, whereas in FPGAs, logic block pins are connected via programmable switches. Regardless of the type of programmable switch in the FPGA (whether based on static RAM-controlled pass transistors [3], anti-fuses [4] or floating gate transistors [5]) the capacitance, resistance and size of the switch makes them much slower and larger than a simple metal wire. In addition, for an MPGA, the amount of routing resources used is exactly what is needed to connect the logic. However, to provide good logic block utilization and routability in an FPGA, there must exist a rich and flexible programmable switching structure to provide many alternate paths between logic block pins. Since many of the programmable switches in the routing matrix will be unused, this further reduces FPGA logic density with respect to MPGAs.

One way to improve the speed and density of FPGAs is to replace some of the slow and large programmable connections between the logic blocks with *hard-wired* connections, which are simple metal wires. We explore the use of hard-wired connections in FPGAs by postulating an FPGA architecture based on *hard-wired logic blocks* (HLBs), with each HLB consisting of several identical simple logic blocks connected together by hard-wired links into a coarse-grained logic block. The use of hard-wired logic blocks (HLBs) in an FPGA may reduce the delay and size of circuits.

For example, Figure 1-1 illustrates how HLBs can improve the speed and density of an FPGA circuit. Define a *basic block* to be the primitive gate or the simplest combinational logic unit of the FPGA. Figure 1-1(a) illustrates a network of 4-input basic blocks. Assuming that the longest path is the critical path and that only gate-output programmable connections are counted. Then this network has five slow programmable connections in the routing along the critical path (through blocks **1**, **2**, **3**, **4** and **5**) and nine programmable connections in total. Suppose that three of the basic blocks are hard-wired together to create a hard-wired logic block (Figure 1-1(b)) with a fast three-block path. If this hard-wired logic block (HLB) is used to implement the circuit of Figure 1-1(a), the circuit in Figure 1-1(c) results. This circuit has only two slow programmable links instead of five along the critical path and this represents a sizeable reduction in routing delay.



1-1(a): Basic Block Cct. **1-1(b): An HLB** **1-1(c): The faster HLB Cct.**

Figure 1-1: Using hard-wired logic blocks to speed up a circuit

Also, the total number of programmable connections has been reduced from nine to four and this may lead to a significant reduction in routing area.

The use of hard-wired connections in FPGA logic blocks, however, leads to a reduction in the flexibility of the FPGA compared to an FPGA that has only programmable connections between basic blocks. For example, in the hard-wired logic block of Figure 1-1(b), the hard-wired input of basic block **C** is no longer independent of the output of basic block **B**. Thus if a logic function only requires two of the three basic blocks (say **A** and **B**), then because of the hard-wired connections between the basic blocks, one basic block (**C**) would be wasted. The hard-wired input between blocks **B** and **C** also renders basic block **C** unusable for logic functions that do not depend on the output of basic block **B**, yet require all of the four basic block inputs. This effect may lead to lower FPGA logic density.

To improve an HLB-based FPGA's density, each HLB is assumed to have a *tapping buffer* on the output of each basic block. The tapping buffers allow access to the output of every basic block in the HLB and this can improve both the density and speed of HLB circuits. For example, Figure 1-2 illustrates the tapping buffers for the HLB from Figure 1-1(b).

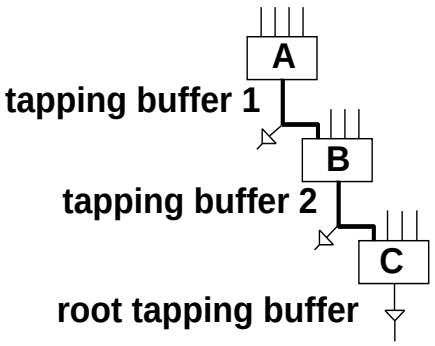


Figure 1-2: HLB tapping buffers

HLBs suffer from reduced logic density because of reduced connection flexibility between basic blocks. Because tapping buffers give access to basic block outputs, one can use subsets of the HLB basic blocks independently. For example, basic blocks **A** and **B** can be used to implement one logic function, while basic block **C** is used to implement another.

The presence of tapping buffers also leads to faster circuits since the output of one basic block can be accessed directly instead of propagating it through another basic block. For example, the output of **A** in Figure 1-2 can be accessed directly through **tapping buffer 1**, whereas without intermediate tapping buffers, the output of A would have to be propagated through **B** and **C**.

This thesis will investigate the speed improvements and density benefits of hard-wired connections in FPGA logic blocks, as well as the logic synthesis algorithms needed to automate the design of such FPGAs.

1.2 Research Scope, Goals and Methodology

The HLB architecture of an FPGA is defined by the choice of basic block and connection topology between the hard-wired basic blocks. Figure 1-1(b) shows an example of an HLB architecture consisting of three 4-input basic blocks connected in a chained topology. There are many other possible HLB connection topologies. For example, Figure 1-3 illustrates another connection topology, a balanced tree, for three 4-input basic blocks. We restricted our hard-wired connection

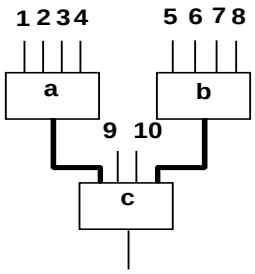


Figure 1-3: Balanced tree topology

lookup tables (LUTs) are a good basic block from both a logic density and speed perspective [6] [8] [29] [30]. Thus, only LUTs are considered as basic blocks for the HLB-based FPGAs in this work.

There are architectural trade-offs between basic blocks and topology of the HLB and the speed and density of the HLB-based FPGA. A more functional basic block, in general, leads to faster circuits because it reduces the number of logic levels. However, when the basic block is too complex, it may be difficult to make efficient use of its functionality, and this may lead to lower logic density. A greater number of hard-wired links in an HLB leads to faster circuits since more of the critical path connections can be implemented by fast hard-wired connections. More hard-wired links in the HLBs also imply more hard-wired connections between the basic blocks of the circuit. This reduces the total number of programmable connections in the circuit and may reduce the routing area. However, as discussed above, the increase in hard-wired links may lead to more wasted logic, and so reduce logic density.

The goal of this research is to explore these trade-offs in basic block functionality and hard-wired connection topologies to find those HLBs that will lead to fast FPGA architectures with good logic density. The results will show that certain FPGAs with hard-wired links are not only faster than FPGAs without hard-wired links, but can also achieve better area-efficiency.

To explore the HLB architectural space, this thesis uses an empirical approach to evaluate the different HLB-based FPGA architectures. Each FPGA architecture is used to implement a set of benchmark circuits and then the speed and area of the resulting circuits are measured using area

and delay models. A comparison of the speed and area of the implemented circuits yields the best architectural alternatives.

When performing such an empirical study, it is preferable to use the best CAD tools available for the circuit implementations. Since the synthesis of HLB-based FPGAs is a new CAD problem, a new HLB mapping tool had to be constructed [9]. This thesis describes the algorithms used in a novel HLB mapper as well as a discussion and statements concerning the optimality of the algorithm. When compared to a commercial CAD tool [11] aimed at mapping a particular hard-wired logic block [13], this HLB mapper is comparable in effectiveness. However, it should be noted that this new tool can be applied to a much broader range of hard-wired logic block structures.

1.3 Thesis Organization

This thesis is organized as follows. Chapter 2 presents the background necessary to understand the rest of the thesis and related work. Chapter 3 describes the CAD algorithms used to map the benchmark circuits to HLB-based FPGA circuits. Chapter 4 discusses the complexity and optimality of the HLB mapping algorithms described in Chapter 3. The effectiveness of the HLB mapping algorithms is evaluated with respect to theoretical bounds and a commercial mapping tool in Chapter 5. Chapter 6 describes the range of HLB architectures investigated, the experimental method used to implement benchmark circuits in the various HLBs and the area and delay models used to calculate the size and speed of the resulting HLB circuits. Chapter 6 also presents the results for various HLB architectures when optimizing for speed and area and discusses the limitations of the experimental method. The final chapter concludes with a summary of the thesis and gives suggestions for future work.

Chapter 2 Terminology and Previous Work

This chapter presents the background terminology needed to understand the following chapters plus a survey of related research into FPGA architecture. In this thesis, FPGA architecture alternatives are evaluated using the same basic methods employed in other FPGA empirical studies [6] [7] [8] [25] [27] [28] [29] [30] [31]. This methodology can be briefly described as follows. In order to improve upon the speed and density of the current FPGAs an architectural idea is proposed that gives rise to a new class of FPGA architectures, or an existing class of FPGAs is investigated. For a given variation of FPGA architecture, several benchmark circuits are implemented in that FPGA and then the area and/or speed of the circuit implementations are measured. Note that whenever the proposed FPGA architectures are novel, new CAD synthesis tools are often needed to carry out these empirical studies - for example, the Chortle technology mapper [20] was necessary to perform LUT-based FPGA studies [6] [7] [8]. A comparison of the speed and area of the circuits, when implemented in the different FPGA architectures, yields the best alternatives.

Mapping a benchmark circuit to an FPGA is done in two main steps. The circuit is first mapped into the logic blocks of the FPGA in a step referred to as *logic synthesis*, and then the resulting netlist of logic blocks is placed and routed within the interconnection resources of the FPGA. For example, Figure 2-1 shows a circuit that has been implemented in a generic FPGA. On the left side of Figure 2-1 is a circuit consisting of two logic blocks, each represented by a dotted rectangle. One logic block implements a two-input **AND** function and the other a two-input **OR** function. The right side of Figure 2-1 shows the generic FPGA used to implement the circuit. The logic blocks used to implement the circuit are placed in the upper left corner of the FPGA and the

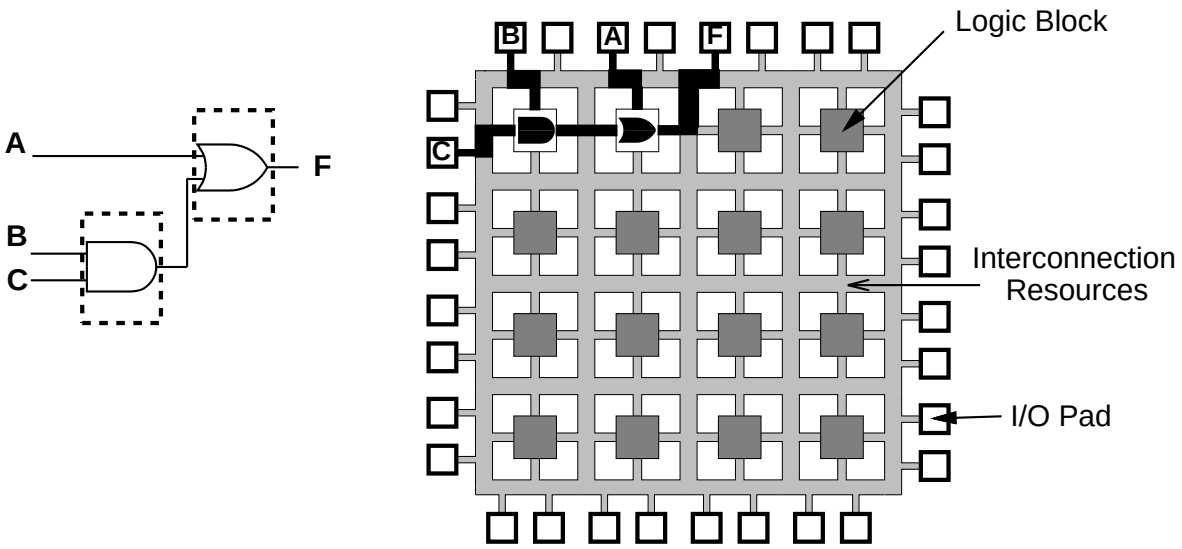


Figure 2-1: Synthesis of a circuit into a Generic FPGA

connections between I/O pads and logic blocks in thick lines. Since the new CAD tool presented in this thesis maps *combinational* circuits to the logic blocks of the HLB-based FPGA, only the combinational logic synthesis phase will be discussed in this chapter. As mentioned in Chapter 1, the logic blocks investigated in this thesis are based on lookup tables (LUTs). This chapter first presents some background information on LUTs and then some of the previous work in logic synthesis and architecture for LUT-based FPGAs.

2.1 Lookup Tables

A K-input lookup table (K-LUT) is a programmable gate that can implement any Boolean function of K or fewer variables. The 2^K memory cells in the LUT contain the truth table for the K-input Boolean function. A 2^K to 1 multiplexer, controlled by the K inputs, is used to select one of the memory cells. For example, Figure 2-2 shows a 3-LUT that implements the Boolean function **`F = a b + c.`** The 8 memory cells and their addresses are on the left side of Figure 2-2 and the 8 to 1 multiplexer is on the right. For example, if **`a = 1`**, **`b = 0`** and **`c = 1`** (memory cell address 101), then the LUT output **`F = 1`**.

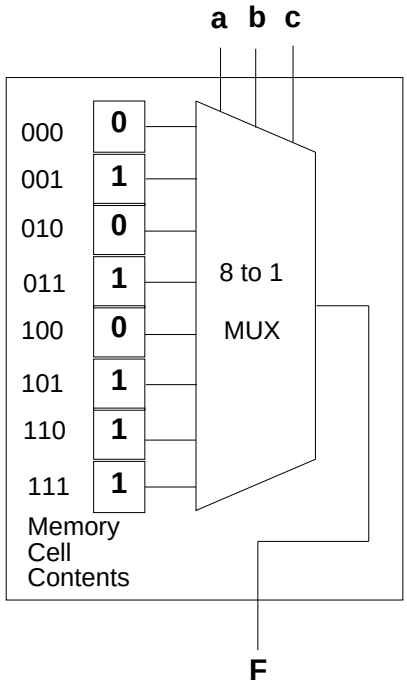


Figure 2-2: 3-LUT implementing F = a b + c

2.2 Logic Synthesis for Lookup-Table Based FPGAs

Logic synthesis takes an input circuit description, often in the form of a Boolean network, and produces a netlist of logic blocks optimized for either speed or area. A Boolean network [38] can be represented as a directed acyclic graph (DAG). Each node in the DAG represents a logic gate, primary input or primary output. In the DAG, there is a directed edge **(i, j)** if the output of gate **i** is an input of gate **j**. The gate functions for the nodes of a Boolean network are usually restricted to implementing a simple function of its inputs, such as an **AND**, **OR** or a sum of products expression. The primary input nodes are those with no incoming edge and primary output nodes are those with no outgoing edge. All other nodes in the Boolean network are termed *internal* nodes. For example, Figure 2-3 shows a Boolean network with primary inputs **a, b, c, d** and primary output **F**. There are two internal nodes in the network, **x** and **y**. Node **x** implements the sum of products expression, **a b + c**, of the primary inputs. Node **y** implements the **AND** function of the output of node **x** and primary input **d**. The primary output node, **F**, is connected to the internal node **y**.

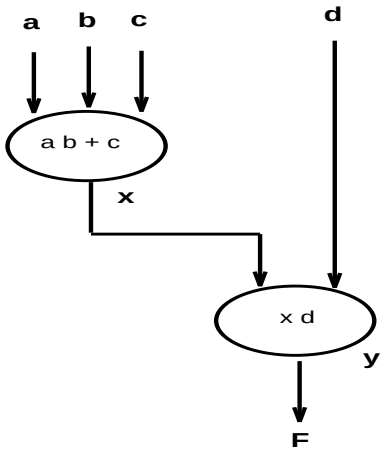


Figure 2-3: A Boolean Network

The output netlist of FPGA logic blocks can also be described by a DAG. In this case, each node represents a logic block and each edge represents a connection between the output of a logic block and the input of another logic block. Associated with each node is a Boolean function that tells how that logic block transforms its inputs to generate an output. For example, Figure 2-4 shows a DAG that represents a netlist of 3-input lookup tables (3-LUTs) after mapping the Boolean network in Figure 2-3. Each of the two 3-LUTs in Figure 2-4 has an associated Boolean function as shown in the figure.

Logic synthesis can be conceptually separated into two phases, technology-independent logic optimization and technology-dependent mapping. In the logic optimization phase, the input

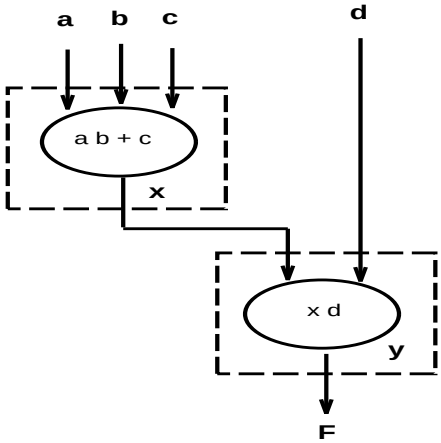


Figure 2-4: A Netlist of 3-LUTs

Boolean description is modified by Boolean operations to minimize some technology-independent cost function that measures area or delay. The structure of the Boolean network can be modified in any manner so long as the output functionality is preserved. In the technology mapping phase, the optimized Boolean description is mapped to a netlist of nodes, each node having a Boolean function that is implemented by an FPGA logic block. Technology mapping operations should preserve the general structure of the optimized Boolean network so that logic optimizations are not undone. Decompositions of nodes and local replication of portions of the Boolean network are the only structure-changing operations usually employed.

2.2.1 Technology-Independent Logic Optimization

The goal of the logic optimization phase is to improve the input Boolean network so that the subsequent technology mapping to logic blocks can be more effective. This phase is described as technology-independent because it does not use knowledge of the implementation technology to guide the restructuring of the Boolean network. For example, the total number of literals in the Boolean description is one technology-independent measure of area and the depth of the Boolean network is one measure of delay. The goal of the logic optimization algorithms is to minimize the cost of the Boolean network by applying Boolean operations to the network, such as two-level minimization or factoring. One benefit of technology-independent logic optimization is that the logic optimizer may be applied to the implementation of circuits in many target technologies. However, the main drawback is that the technology-independent measures of area and delay may not be accurate for a particular target technology. This may lead to an inferior result after technology mapping has been performed.

The MIS system [17] is an example of a logic optimization system. The MIS logic optimization system uses a count of the number of literals in the gate functions of the Boolean network as a measure of area. MIS optimization operations include (1) algebraic factoring to extract logic expressions that appear in several parts of the network, (2) node function simplification using techniques similar to Karnaugh-map minimization and (3) decomposition of

large node functions to simplify and improve technology-dependent mapping. An illustration of some of the MIS optimization algorithms is now presented using the following Boolean network:

$$x = a\ b\ c\ d + a\ b\ c\ e \quad (4.1)$$

$$y = a\ b\ c\ f + a\ b\ c\ g \quad (4.2)$$

$$z = \bar{a}\ h + \bar{b}\ h + \bar{c}\ h + a\ b\ c \quad (4.3)$$

The initial Boolean network has 25 literals in the sum-of-product form of the gate functions. Suppose the algebraic factoring operation extracts the factor $a\ b\ c$. Note that $\overline{a\ b\ c}$ is equivalent to $\bar{a} + \bar{b} + \bar{c}$. If a new node, t, with the function $a\ b\ c$ is introduced and substituted in the above network, then the modified Boolean network is:

$$t = a\ b\ c \quad (4.4)$$

$$x = t\ d + t\ e \quad (4.5)$$

$$y = t\ f + t\ g \quad (4.6)$$

$$z = \bar{t}\ h + t \quad (4.7)$$

Simplifying the function for node z, results in the following equation:

$$z = h + t \quad (4.8)$$

The optimized form of the Boolean network now has only 13 sum-of-product literals compared to the original literal count of 25. This significant reduction in the complexity of the Boolean network will likely lead to a smaller circuit implementation when the equations are mapped to logic blocks. Before technology-dependent mapping, the network nodes x and y may be decomposed to yield the following factored equations, which may be easier to map:

$$x = t\ (d + e) \quad (4.9)$$

$$y = t\ (f + g) \quad (4.10)$$

The optimized set of equations for t, x, y and z has a total of 11 literals in the factored form.

2.2.2 Technology-Dependent Mapping to Lookup-Tables

The technology mapping step takes the optimized Boolean network and finds an optimized netlist of K-LUTs that *covers* or implements the network. To determine if a K-LUT covers a portion of the network, one simply counts the number of input edges to the sub-network. If the number of inputs is less than or equal to K, then it can be covered by a K-LUT. The goal of

technology mapping is to minimize the area or delay (or some combination of both) of the K-LUT cover. Area minimization refers to using the smallest possible number of K-LUTs. A delay-optimized K-LUT circuit has the minimum depth (in terms of K-LUTs) along any of the longest paths between primary inputs and a primary output[1].

Sometimes logic optimization produces nodes in the optimized Boolean network with more than K inputs. These are referred to as *infeasible* nodes. An infeasible node has to be *decomposed* into nodes with fewer than or equal to K inputs (called *feasible* nodes) to enable covering by a K-LUT. Decomposition of feasible nodes can also improve the quality of the cover by increasing the number of alternatives available to the covering operations. During the construction of the cover, the process of checking to see if a sub-network rooted at a node can be implemented by a gate is referred to as *matching*. Covering, matching and decomposition operations are common to all technology mapping algorithms.

Library-based Technology Mapping for LUTs

This subsection describes a general approach to technology mapping, called library-based mapping, that can be applied to many ASIC technologies, including LUTs. Library-based technology mapping using graph-covering and dynamic programming was first introduced in the DAGON technology mapper [19]. The HLB-based FPGA mapping algorithm in this dissertation uses a library-based mapping algorithm in one of its stages.

The first step in a general ASIC library-based mapping algorithm is to use decomposition operations to convert the input Boolean network, also called the *subject* network, to a canonical network of gates. Often the canonical gates are **`INVERTER`** or 2-input **`NAND`**, **`NOR`**, **`AND`** or **`OR`** gates. Similar decomposition operations are applied to each gate function in the library to make a set of *pattern* graphs that are used to match against nodes in the subject DAG. Since the gates in

1. This optimizes the longest path delay but may not optimize the critical path delay. In the absence of exact timing information, it is assumed that the longest path is the same as the critical path.

the pattern library and the subject network are of the same type, this reduces the mapping problem to one of covering a directed acyclic graph (DAG) with a set of graphs. An additional constraint is that each input of a gate in the cover must be produced by the output of another gate or a primary input.

The mapping algorithm traverses the input Boolean network from inputs towards the outputs and uses dynamic programming to select the best pattern (and hence best gate) to implement each node in the network. At each node, *every* gate pattern in the library is matched against the network to see if it can cover the network at that node. The matched sub-network includes the node plus a portion of the network feeding the node. The cost of using the matching gate at that node is the sum of the gate cost plus the costs of the nodes that fan-in to the sub-network covered by the gate. Primary inputs are assigned a cost of zero. The matching gate that leads to the lowest cost implementation of the node is selected. The lowest cost (and the matching gate) is retained so that the cost can be used to determine the lowest cost matches for succeeding nodes.

For example, suppose the library consists of the five gates in Figure 2-5 and this library is used to implement the Boolean network shown in Figure 2-6. Figure 2-6(a) shows the mapping of nodes **A** and **C**. The mappings of **A** and **C** are trivial because there is only one possible gate that can be used at each node. The cost of node **A** is 4, which is the cost of an **OR** gate. The cost of node **C** is 1. Figure 2-6(b) shows the mapping of node **B**, which has two possible matching gates: an **AND** gate or an **OA21** gate. If the **AND** gate is used to implement node **B**, then the total cost for implementing **B** would be 7, which is the sum of the cost of the **AND** gate and the cost of fan-in

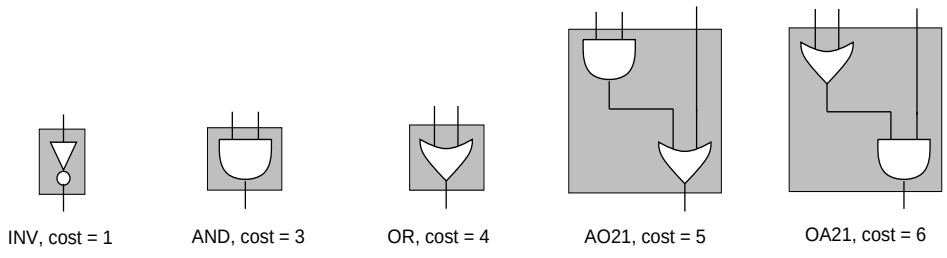
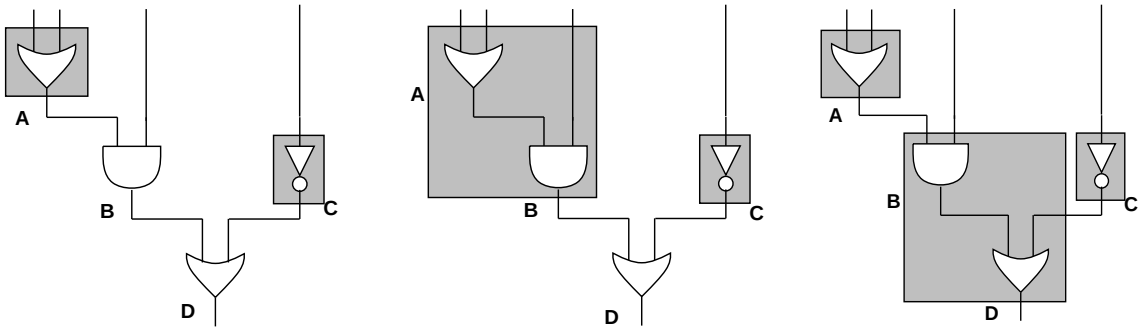


Figure 2-5: Library of gates



2-6(a): Mapping nodes A **2-6(b): Mapping node B** **2-6(c): Mapping node D**

Figure 2-6: Mapping a Boolean network

node **A**. However, if the **OA21** gate is used to implement node **B**, then the cost is 6, and so, the **OA21** gate would be chosen for implementing node **B** (Figure 2-6(b)). Figure 2-6(c) shows the mapping of the root node **D**. At **D** there are two possible matching gates, an **OR** gate or an **AO21** gate. The lowest cost of **D** (a cost of 10) occurs when using the **A021** gate. The best mapping for the entire network is shown in Figure 2-6(c).

The *completeness* of the set of functions that can be implemented by a K-input LUT makes it a difficult target technology for a library-based mapper [18]. For a given value of K, there are 2^{2^K} possible functions that can be implemented by a K-LUT. Thus, for even small values of K, the library becomes very large. For example, K = 4 would require 65536 gates in the library. Performing the matching of such a large number of library gates against each node would be too time-consuming and so more efficient means of mapping to LUTs were created [18] [20] [21] [32] [33] [35] [37].

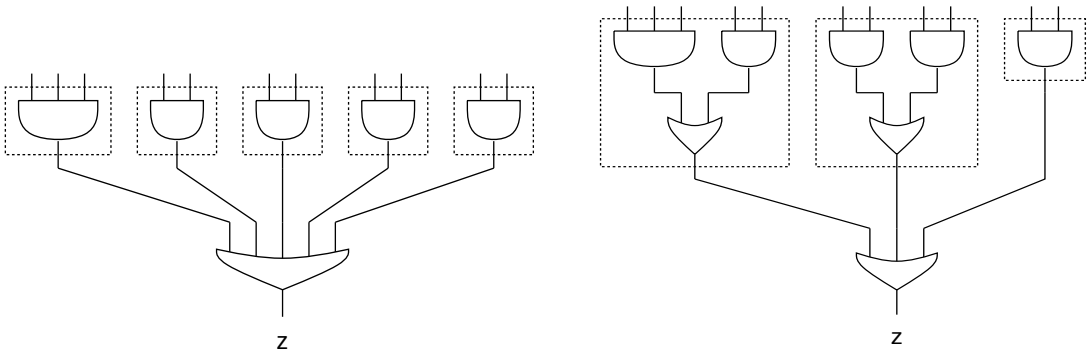
Chortle LUT Technology Mapper

This subsection describes the LUT mapper used for the experiments in this thesis. The Chortle technology mapper for LUT-based FPGAs [20] [21] exploits the completeness of LUTs during decomposition and covering. Similar to the library-based mapping algorithm described above, the Chortle algorithm traverses the input Boolean network from primary inputs to outputs, and at each node finds the best K-LUT circuit to realize the function at that node. The input network consists

of **AND** and **OR** nodes and the decomposition techniques are restricted to **AND-OR** decompositions. At each node in the network, the goal is to find the circuit of K-LUTs rooted at the node with minimum area or delay cost as a primary cost function. When optimizing area, the primary cost function is the total number of K-LUTs. When optimizing delay, the primary cost function is the maximum number of K-LUTs in any path from the node to a primary input. The secondary objective is to minimize the number of inputs to the K-LUT rooted at the node. This is important because the number of inputs to the K-LUT at a node affects the mapping of the fan-out of the node. The rest of this section will describe the Chortle algorithm with respect to area minimization only.

When mapping a given **AND** or **OR** node, the goal is to decompose the node in such a way as to optimize the covering of the node's fan-in K-LUTs and the node itself. Decomposition is also necessary to ensure that all nodes have fan-in less than or equal to K. Because **AND** and **OR** operations are associative and commutative, the decomposition of the **AND** or **OR** node can be formulated as an integer *bin packing* problem. The bin packing problem is as follows. Given a set of boxes of integer size one to K, where the size of the boxes correspond to the number of inputs used by a fan-in LUT, the goal is to pack these boxes into as few as possible bins of size K. Each packed bin corresponds to a LUT. The output of the boxes in each packed bin connect to a common gate of the same type as the decomposed node. In addition, all of the outputs of the packed bins also feed into a common gate of the same type as the decomposed node.

For example, Figure 2-7 illustrates the decomposition of the **OR** node **z** when mapping to 5-input LUTs. The **OR** node in Figure 2-7(a) has five fan-in LUTs with 3, 2, 2, 2 and 2 inputs respectively. Each of the fan-in LUTs in Figure 2-7(a) implement an **AND** function. The best packing of the five fan-in boxes into bins of size 5 has three bins, one with a total of 5 inputs, another with a total of 4 inputs and the last with only 2 inputs. Note that since the two boxes in the left bin of Figure 2-7(b) are packed in the same bins, the outputs of the two boxes fan-in to a common 2-input **OR** gate. We will refer to the LUTs after bin-packing as bin-packed LUTs.



2-7(a): fan-in LUTs for OR node **2-7(b): LUTs after bin packing**

Figure 2-7: OR decomposition guided by bin packing [18]

To complete the mapping of node **z**, the bin-packed LUTs are chained together. The chaining algorithm sorts the bin-packed LUTs in descending order based on the number of used inputs and then links the output of each bin-packed LUT to an unused input of a subsequent bin-packed LUT. If there are no unused inputs in any subsequent bin-packed LUT, a new K-LUT (with K unused inputs) is created. The algorithm terminates when the last bin-packed LUT is encountered. Figure 2-8 illustrates the chaining of the three bin-packed LUTs in Figure 2-7(b).

The Chortle algorithm produces a netlist with the minimal number of K-LUTs when mapping single fan-out[1] **AND-OR** Boolean networks to K-LUTs when K is less than or equal to 5. Optimization across nodes with fan-out greater than one can further reduce the area or delay of

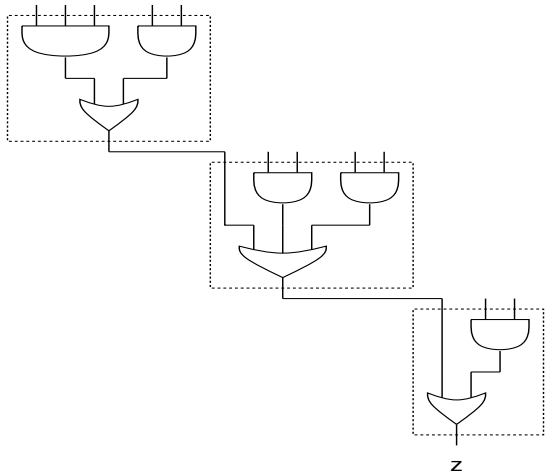


Figure 2-8: Chaining the bins

the K-LUT netlist. For mapping networks with fan-out nodes, Chortle employs heuristics to cover reconvergent paths and replicates logic at fan-out nodes to improve the lookup table netlist.

2.3 Previous FPGA Architectural Studies

The methodology used for investigating hard-wired logic block FPGA architectures in this work is similar to that used for investigating other aspects of FPGA logic block architectures. There have been empirical studies into finding the most area-efficient basic block [8] [15] [30], the non-hard-wired logic block that gives the best speed performance [14] [29] and the minimum levels of interconnection flexibility for good routability [27] [28]. Since this thesis concerns area and speed of LUT-based hard-wired logic blocks, the following subsections will summarize one of the area-efficiency studies [8] and one of the speed studies [14] involved with LUT-based FPGAs. Since hard-wired logic blocks deal in part with routing architecture, this section also describes a study concerning the required flexibility of interconnection structures in FPGAs [27]. This dissertation also uses some of the terminology from [27].

2.3.1 Area-efficiency of LUT-based FPGAs

One of the first empirical studies of FPGA logic block architecture sought to determine the effect of logic block functionality on area-efficiency [8]. This involved the implementation of several benchmark circuits in different LUT-based logic blocks, the measurement of the area of the resulting circuits and then the determination of the best logic blocks using the area measurements. It was observed that the area of the routing was from 3 to 15 times greater than the area devoted to logic and so the best logic blocks were those that minimize routing area. The amount of routing is related to the total number of pins in the logic blocks of the circuit and the total number of connections between the pins. Therefore, logic blocks with high functionality per pin were the most desirable since these kinds of logic blocks would lead to fewer pins and fewer connections for a given amount of circuit functionality. Lookup tables fit this criteria of high

1. that is all non-primary input nodes have fan-out of one.

functionality per pin[1] and the highest densities were achieved for three- and four-input lookup tables.

2.3.2 Speed performance of LUT-based FPGAs

A second empirical study of FPGA logic block architecture [6] [7] investigated the speed performance of FPGAs with different types of basic logic blocks, including mux-based, NAND-based, AND-OR based and LUT-based blocks. In general, the more functional the logic block, the greater the delay per block but the fewer logic block levels between primary inputs and outputs. We refer to the connection between the output pin of a logic block and the input pin of another logic block as a *programmable connection*. Fewer logic block levels means fewer programmable connections along the critical path and this is important because programmable connection delay is often much larger than the combinational logic component of the critical path delay. Thus, the best logic blocks for speed performance will have a combination of high functionality and small delay per logic block. High functionality will minimize the number of logic blocks levels and thus keep the routing delay small, and small delay per logic block will keep the combinational logic portion of the delay small. Lookup table-based logic blocks have this desirable combination of high functionality and small delay. The results of this empirical study showed that 5- and 6-input LUTs were the best for speed performance among the logic blocks investigated.

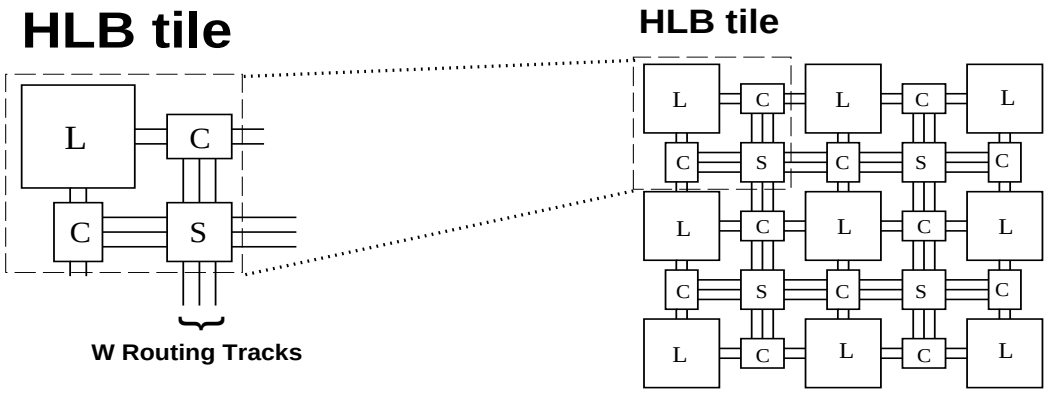
2.3.3 Interconnection flexibility of LUT-based FPGAs

A third empirical FPGA architecture study [27] [28] investigated the interconnection flexibility required to ensure good routability. The greater the interconnection flexibility the greater the routability and the higher the number of programmable switches on the FPGA. Higher numbers of programmable switches impact both the area and delay of the FPGA. More switches require more space on the FPGA. In addition, an increase in the number of switches increases the

1. Recall that a LUT with K input pins can implement 2^{2^K} Boolean functions.

parasitic capacitance and series resistance associated with each routing track in a channel segment. This leads to greater delays for each programmable connection.

The FPGA routing architecture assumed in that work is illustrated in Figure 2-9. The FPGA in Figure 2-9 has both horizontal and vertical routing channels and consists of the HLB tile in Figure 2-9(a) replicated several times. Each HLB tile contains a logic block, labelled “L”, and blocks of programmable switches, labelled “C” and “S”. The pins of the “L” blocks are connected to routing tracks through the “C” blocks. Between connection (“C”) and switch (“S”) blocks are routing channel segments with a fixed number of tracks. Figure 2-10 shows a more detailed view of the HLB tile. In Figure 2-10, there are $W = 3$ routing tracks per channel segment. The “C” block allows logic block pins to connect to a subset of the W routing tracks in each channel segment. The flexibility of the connection block, F_c, is the number of routing tracks to which each logic block pin can connect. F_c ranges from one to W. In Figure 2-10, each logic block pin can connect to two of the three routing tracks and so $F_c = 2$. The “S” block allows connections between horizontal and vertical routing tracks. The flexibility of the switch block, F_s, is the number of tracks on the opposing sides to which an incoming track can connect. F_s ranges from one to $3*W$. Figure 2-10 illustrates a switch block in which each horizontal track can be connected to a track on each of the three opposing sides and thus $F_s = 3$.



2-9(a): HLB tile **2-9(b): Array of HLB tiles**

Figure 2-9: Generic FPGA routing architecture

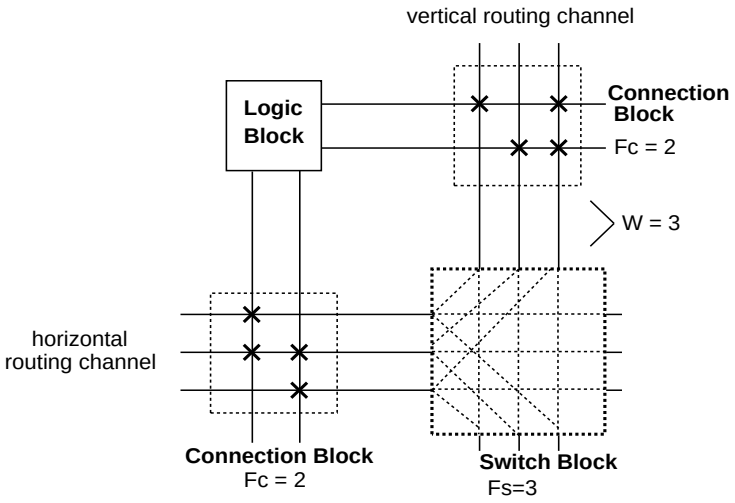


Figure 2-10: Connection and switch blocks of an FPGA tile

The goal of the empirical study in [27] was to determine the effect of connection block and switch block flexibility on the routing completion ratio. The results of the experiments indicate that connection blocks should have high flexibility (F_c between $0.5W$ and W). A high F_c/W ratio is necessary because there is only one "C" block through which a given physical pin can be accessed. A high F_c/W is needed to give a sufficient number of alternative paths to that pin. The experiments also showed that it is sufficient to have a low flexibility (F_s from 3 to 4) in the switch blocks. Ignoring any conflicts with other programmable connection paths, when a connection goes through a switch block, the number of path choices increases by a factor of F_s. Thus, a cascade of switch blocks, between pins on two different logic blocks, provides a number of paths that is exponentially related to base F_s [27]. Therefore, a small F_s should be sufficient to provide enough path choices for good routability.

2.4 Previous work involving hard-wired connections

There has been little previously published work involving hard-wired connections in FPGAs. Currently there exists a commercial FPGA, the Xilinx 4000 [13], with a LUT-based hard-wired logic block. There is also associated software for mapping Boolean networks to Xilinx 4000 circuits [11]. The combinational portion of the Xilinx 4000 Configurable Logic Block (CLB)

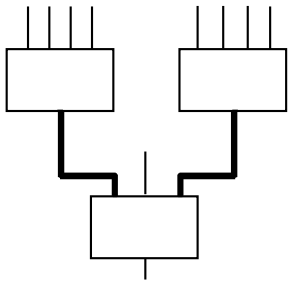


Figure 2-11: Xilinx 4000 CLB

(shown in Figure 2-11) contains two 4-LUTs, whose outputs feed into two of the inputs of a 3-LUT. Note that the Xilinx 4000 CLB only allows two of the three LUT outputs to be accessed simultaneously by the routing. In contrast, the HLBs studied in this dissertation allow all LUT outputs to be accessed simultaneously by the routing.

The idea of using hard-wired connections in LUT-based FPGAs originated in [14]. However, the study in [14] was restricted to determining the potential speedups of a small number of 4-LUT hard-wired logic blocks. The study in [14] was conducted using the novel CAD algorithms presented in this thesis. In comparison, this dissertation will investigate both the speed performance and area-efficiency of a much larger range of hard-wired logic blocks, as well as describe the CAD algorithms used to map to HLB-based FPGAs.

2.5 Conclusion

This chapter has presented the background knowledge needed to understand the technology mapping algorithms for HLB and the HLB-based FPGA architectural studies. The next chapter will describe the HLB technology mapping algorithms.

Chapter 3 Algorithms for Mapping to Hard-wired Logic Blocks

This chapter presents the CAD algorithms used for an empirical evaluation of HLB-based FPGA architectures. An empirical study of HLB-based FPGA architectures, such as the one to be described in Chapter 6, implements benchmark circuits in different HLB-based FPGA architectures and then uses area and delay measurements of the HLB circuits to determine the relative quality of each architecture.

The mapping of any circuit to a netlist of HLBs can occur during either logic or layout synthesis. If the mapping is done in layout synthesis, the circuit would first be mapped to a netlist of basic blocks and then these basic blocks would be placed within HLBs so as to optimize the use of the hard-wired connections. If the mapping is done during logic synthesis, then the entire HLB, basic blocks plus hard-wired links, would simply be considered as a coarse-grained target for technology mapping.

In this dissertation, we chose to do the mapping during logic synthesis because one of the research goals is to explore FPGA logic block architecture without assuming a specific routing architecture. The use of a placement and routing algorithm would require more detailed specification of the routing architecture, whereas mapping to HLBs during the technology mapping phase of logic synthesis can be done without physical layout details. Mapping to HLBs during logic synthesis also allows a tighter focus on the hard-wired logic block itself and its hard-wired connection topology and this may lead to better HLB utilization.

This chapter is organized as follows. Section 3.1 describes the HLB architectures targeted by synthesis. Section 3.2 gives an overview of the logic synthesis methods used to create HLB circuits. The definition of the technology mapping problem addressed by the algorithms in this chapter is given in Section 3.3. Section 3.4 and Section 3.5 describe the details of the two main phases of the HLB technology mapping algorithm. The final section summarizes this chapter.

3.1 Definition of the HLB Architecture

The HLB architecture definition presented in this section is one of many possible choices. We define the architecture here to clarify the target for synthesis.

A hard-wired logic block consists of several identical basic blocks connected together by hard-wired links. The basic blocks are assumed to be hard-wired in tree topologies to simplify the synthesis problem. Each HLB basic block is also assumed to have a tapping buffer that makes the output accessible to the routing. Because tapping buffers make it possible to implement independent functions in different portions of an HLB, they may improve the density of HLB circuits.

3.2 HLB Synthesis Overview

The synthesis of a circuit to a netlist of HLBs takes as input an optimized (in the technology-independent sense) Boolean network circuit description and a description of the hard-wired logic block topology. It is assumed that the basic blocks of the HLB are all the same type of gate, although there may well be reasons to employ different types of gates in the same HLB [13] [46]. The output from the HLB logic synthesis steps is a netlist of HLBs that has been optimized either for delay or area.

The mapping from Boolean network to HLBs is done in two stages. First the optimized Boolean network is mapped to an area- or delay-optimized netlist of basic blocks using one of the many existing basic block technology mappers [20] [21] [33] [34]. Note that the basic blocks

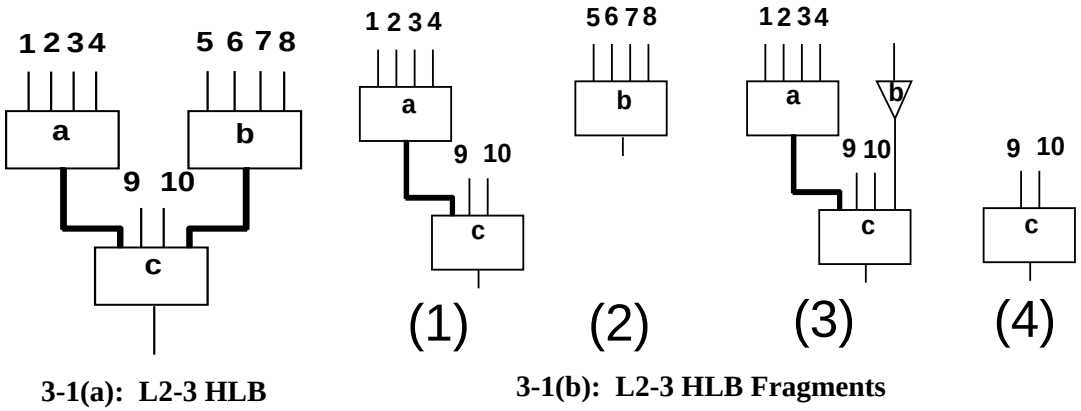
produced by the technology mapper are of the same type as those in the HLB. Second, using the new HLB technology mapping tool described in this chapter, the netlist of basic blocks is mapped to a netlist of HLBs optimized for area or delay.

The reason for dividing the synthesis of a Boolean network to HLBs into two distinct stages is to separate the mapping to basic blocks from the mapping that uses the HLB topology information. By doing so, the CAD algorithms in the second stage can focus on optimizing the use of hard-wired connections and the CAD algorithms used in the first stage can leverage off specialized mappers for different basic blocks. These specialized mappers should provide a good starting point for the second stage mapper, plus the second stage mapper will be useful for HLBs composed of any basic block gate type. Note that the basic block is assumed to be a LUT in the rest of this chapter.

The remainder of this chapter will focus on algorithms for executing the second stage of HLB synthesis, that is, the mapping of the basic block netlist to HLBs. This step will be referred to as technology mapping to HLBs or HLB technology mapping.

3.3 The HLB Technology Mapping Problem

The HLB technology mapping step takes as input a directed acyclic graph (DAG) that describes a netlist of basic blocks and an *HLB template*, which is a tree that describes the HLB topology. The input DAG is also referred to as the *subject DAG*. The output of the technology mapping algorithm is a DAG representation of a netlist of HLBs that implements or covers the subject DAG. The goal of the mapping algorithm is to find a minimum area-cost or delay-cost HLB cover of the input basic block netlist. The area cost is the total number of HLBs needed to implement the subject DAG. The primary delay-cost is the maximum number of programmable connections between any primary input and a primary output. The secondary delay-cost is the number of basic block delays along critical paths. Note that, in the absence of more timing information, this delay-cost assumes that the critical paths are identical to the longest paths. This



3-1(a): L2-3 HLB **3-1(b): L2-3 HLB Fragments**

Figure 3-1: L2-3 HLB and L2-3 HLB Fragments

order of delay-costs assumes that a programmable connection delay is greater than a basic block delay. Note that this assumption is only significant when additional basic blocks may be added along a path in order to reduce the number of programmable connections along the path.

As mentioned in Section 1.1, an important architectural assumption is that each HLB has a tapping buffer on every basic block output. Since every HLB basic block output is accessible, several portions of the same HLB can be used to implement unconnected subgraphs of the subject DAG. The general term HLB *fragment* denotes a connected subset of the basic blocks in the HLB template. An HLB *fragment pattern* is a subtree of the HLB template. Each HLB fragment pattern represents a portion of the HLB that may be used to implement a subtree of the subject DAG. For example Figure 3-1(a) shows the template of the L2-3 HLB[1] consisting of three 4-LUTs (or the L2-3 4-LUT HLB for short) and its four fragment patterns. Note that because we have assumed LUT basic blocks, fragment (1) is equivalent to its any fragment generated by permuting the inputs to basic block **c**. Details of the rules that govern fragment pattern generation will be presented in Section 3.4.3. For a given node in the subject DAG, a *feasible* HLB fragment pattern (or feasible fragment for short) at that subject node is a fragment pattern that matches, and can thus implement, a subtree of the subject DAG rooted at that node.

1. The naming convention for HLBs is given in Section 3.4.2.

The problem of technology mapping to HLBs is solved in two stages. The first stage selects the set of feasible HLB fragments that cover the subject DAG with minimum area or delay cost. The set of feasible fragments chosen by this fragment covering step is referred to as the set of *covering fragments*. The second stage packs these covering fragments together as tightly as possible to reduce the number of HLBs. This division of the algorithm into these two steps occurs naturally because the selection of the covering fragments that minimize delay is independent of the packing stage that puts fragments together into *packed HLBs*.

3.4 Fragment Covering

The selection of the covering fragments is formulated as a DAG covering problem. The inputs are a subject DAG, which represents the basic block network, and the HLB template that describes the HLB topology. Before the actual construction of the cover, the HLB template tree is decomposed to produce the fragment pattern trees in a library of patterns. Informally, the cover is a set of fragment patterns that include all nodes in the subject DAG and has the lowest area or delay cost. A formal definition is given in Section 3.4.1. The solution approach is similar to the tree-matching and dynamic programming methods used in the DAGON [19] and misII [38] technology mappers.

3.4.1 Definitions for the Fragment Covering Algorithm

The graph terminology used to describe the pattern trees and DAGs in the fragment covering algorithms is similar to the one given in [38]. A directed acyclic graph G is a pair $(V(G), E(G))$ consisting of a set of vertices (or nodes) $V(G)$ and a set of directed edges $E(G)$. Each edge in E consists of an ordered pair of vertices (v_i, v_j). Each vertex in V represents a primary input, primary output or a LUT basic block. Each edge represents connectivity between primary inputs and the input of a LUT or between the output of a LUT and the input of another LUT. Following the flow of signals in the basic block network, a source of the DAG is called a primary input node and a sink of the DAG is called a primary output node. The incoming arcs of a node are called the inputs

or fan-ins of the node, and the outgoing arcs are called the fan-outs of the node. For a given node *v*, the fan-in of *v*, *i(v)*, is defined as $i(v) = \{u|(u, v) \in E\}$ and the fan-out of *v*, *o(v)*, is $o(v) = \{u|(v, u) \in E\}$. The in-degree of a node *v* is $|i(v)|$ and the out-degree of *v* is $|o(v)|$. Assume that *H* is a subgraph of *G* and that *H* is defined by the pair *(V(H), E(H))*. The fan-in of *H*, *I(H)*, is defined as $I(H) = \{t|(t, u) \in E(G), t \notin V(H), u \in V(H)\}$. The fan-out of *H*, *O(H)*, is defined as $O(H) = \{t|(u, t) \in E(G), t \notin V(H), u \in V(H)\}$.

An *internal node* is a node that is not a primary input node. Note that primary output nodes are also internal nodes. Each internal node of the subject DAG represents a LUT. Each internal node of a fragment pattern represents a LUT in the HLB template. An edge between two internal input nodes is called an *internal edge*. Thus, an internal edge of an HLB fragment pattern represents a hard-wired link between two of its LUT basic blocks. An edge between a primary input and an internal node is called a *primary input edge*. A *leaf node* is one whose fan-in edges are all primary input edges. Similarly, a *leaf LUT* or *leaf basic block* is one whose fan-in edges are all primary input edges.

Fragment Covering Problem Definition

The inputs to the fragment covering stage are the HLB template, *H*, and the subject DAG network, *S*. The goal of the fragment covering algorithm is to find a set of fragments of *H*, with minimum area or delay cost, that covers *S*. A cover of *S* is defined below.

Definition 3-1 Given a subject DAG *S*, and a set of *n* DAGs, $C = \{C_i\}$, where *C* is the disjoint union of the C_i's, *C* is said to *cover S* iff there exists a surjective mapping, σ: $V(C) \rightarrow V(S)$ such that $\forall\ u, v \in V(C)$,

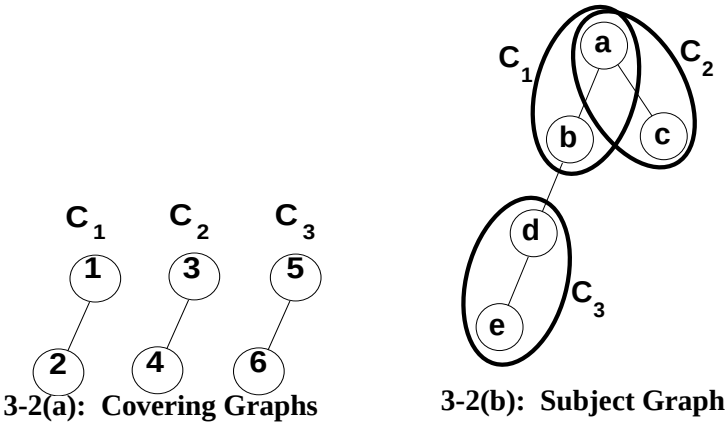
(i) $(u, v) \in E(C) \Rightarrow (\sigma(u), \sigma(v)) \in E(S)$

(ii) $(\sigma(u),\sigma(v)) \in E(S) \Rightarrow (u, v) \in E(C)$ OR *u* is a primary output node of C_i, *v* is a primary input node of C_j, $i \neq j$.

If σ is one-to-one, then *C* is an exact cover of *S*.

The first condition of Definition 3-1 ensures that all edges of the cover, *C*, really do exist in the subject DAG, *S*. Condition (ii) ensures that all edges in *S* are either mapped to edges in one of the C_i or are edges between two distinct covering subgraphs C_i and C_j.

Figure 3-2 shows an example of a cover. The covering graphs, C_1, C_2 and C_3 are shown in Figure 3-2(a) and the subject graph is illustrated in Figure 3-2(b). The covering graphs contain six nodes labelled **1**, **2**, **3**, **4**, **5** and **6**. The subject graph contains five nodes labelled **a**, **b**, **c**, **d** and **e**. Each of the graphs in the three ovals of Figure 3-2(b) corresponds to one of the covering graphs in Figure 3-2(a). The mapping between the nodes of *C* and *S* is shown in Table 3-1. Note that node **a** in the subject graph is covered by two nodes (**1** and **3**) in the covering graphs and so this covering does not have a one-to-one mapping between covering graph nodes and subject graph nodes.



3-2(a): Covering Graphs **3-2(b): Subject Graph**

Figure 3-2: Example of a Cover

C vertex	S vertex
1	a
2	b
3	a
4	c
5	d
6	e

Table 3-1: Mapping between Subject nodes and Covering Graph nodes

3.4.2 Naming Convention for HLBs

The naming convention [41] is unique for each HLB tree topology and is the letter “L” followed by the number of levels (or height) of the HLB followed by “-” followed by a listing of the subtree sizes (that is, the number of LUTs in the subtrees) from a pre-order traversal of the canonical HLB tree. The canonical HLB tree is generated by the canonical labelling algorithm given in Section 3.5.4. Each subtree size is separated by a “.” and leaf inputs and single-LUT subtrees are not listed. Some of the 4-LUT HLB tree topologies are illustrated in Figure 3-3. The circles in Figure 3-3 represent LUT nodes in the HLB tree. The thick lines in Figure 3-3 represent a hard-wired connection edge between two LUTs, while the thin lines represent a primary input edge. For example, the L3-6.3.2 HLB in Figure 3-3 contains six 4-LUTs connected in an asymmetric tree with three levels. The “L3” part of L3-6.3.2 says that the HLB has three levels. The entire L3-6.3.2 tree has six LUTs, hence the 6 in “6.3.2”. The “3” in “6.3.2” represents the number of LUTs in the left subtree. Since the subtrees of the left-most subtree of the root node are either a single

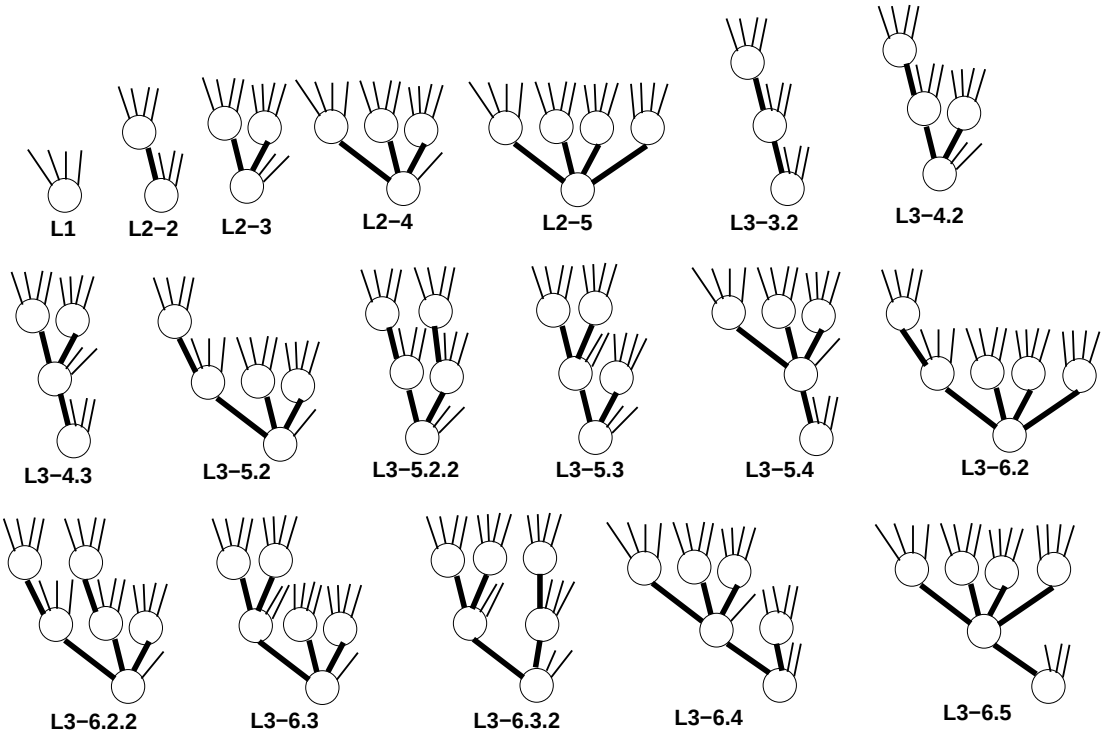


Figure 3-3: Some 4-LUT HLB topologies

LUT or a leaf input their sizes are not listed. Next the second subtree of the root node is traversed. This subtree has 2 nodes and thus the “2” in “6.3.2”. Note that the L1 HLB in Figure 3-3 represents a 4-LUT HLB without hard-wired connections.

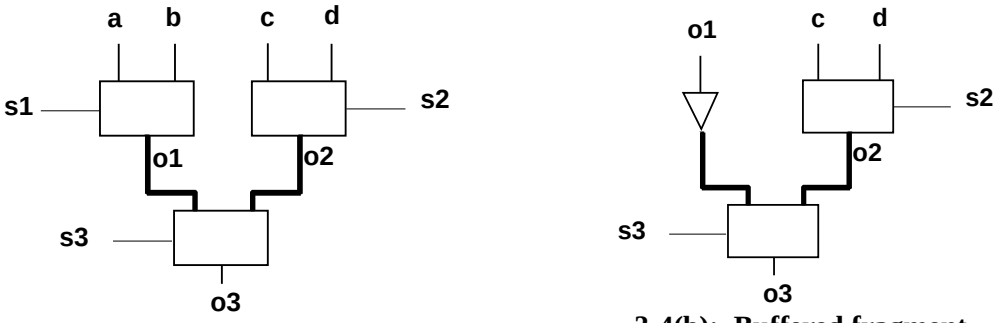
3.4.3 Generation of the Fragment Pattern Library

The fragment pattern library is generated from the HLB template, *H*, using *fragmentation* operations. These fragmentation operations depend upon the properties of the basic block. One property of LUTs that can be exploited during fragmentation is that an input can be ignored or not used. This property is due to the fact that a K-input lookup table can implement any function of K or *fewer* inputs. The fragmentation operation that results from this *ignorable input* property is the deletion of an internal edge of the HLB fragment pattern tree that is connected to a given input. For example, Figure 3-1 shows the L2-3 HLB template and its *delete-edge* fragment patterns. Fragment patterns (1) and (2) in Figure 3-1(b) were generated by the deleting edge **(b, c)** of the HLB template in Figure 3-1(a). Fragment pattern (4) was generated by deleting edge **(a, c)** from fragment pattern (1).

Another property of LUTs is that a LUT can implement the identity function, which we shall call a buffer. The resulting fragmentation operation converts an internal edge connected to a leaf LUT into a primary input edge. The *buffered* fragment pattern, Fragment (3), in Figure 3-1(b) was generated by converting LUT **b** of the HLB template in Figure 3-1(a) into a buffer.

One or both of the above fragmentation operations may be applied to HLBs that consist of non-LUT basic blocks. For example, Figure 3-4 shows a mux-based HLB and one of its fragments generated by a buffering fragmentation operation. The HLB shown in Figure 3-4(a) is composed of 3 two-input muxes. The connection of the inputs **a** and **b** to the same signal has the effect of making **o1** into a primary input. Another way of converting **o1** into a primary input is to ground **s1**. The resulting buffered fragment is shown in Figure 3-4(b).

The fragment pattern library generation algorithm for LUT-based HLBs is described in Figure 3-5. Initially, the pattern library list, **P**, consists of the complete HLB tree, **h**. The patterns to decompose list, **ToDecom**, initially contains only **h**. While there are patterns to decompose, the outer while loop of the algorithm continues to generate new pattern graphs using the edge deletion and edge buffering operations. For each graph **g** in **ToDecom**, all internal edges in **g** (the set e_g) are enumerated to determine the order of deletion. Each internal edge is deleted to yield two graphs, **g1** and **g2**, which are tested for isomorphism with respect to the other generated HLB patterns. Every isomorphically unique pattern is added to the **AddLib** list. After restoring the deleted internal edge to the graph **g**, the edge deletion operation is applied to the next internal edge in e_g. Also, for each graph **g** in **ToDecom**, all internal edges are enumerated to determine the order for the buffering operation to yield new pattern graphs. Again, only isomorphically unique patterns are added to the **AddLib** list. After completion of the deletion and buffering operations on the graphs in the **ToDecom** list, the new patterns in the **AddLib** list are added to the set **P**. The **ToDecom** list takes on the value of the **AddLib** list if this list is not empty, the outer loop continues to generate patterns from the new **ToDecom** list patterns. Because the simple matching algorithm outlined later in this chapter does not check for permutations of the internal node inputs when doing matching, the last step of the algorithm permutes the fan-in edges of each node of the patterns in **P**.



3-4(a): A mux-based HLB **3-4(b): Buffered fragment**

Figure 3-4: Mux-based HLB and a Buffered HLB fragment

Fragment Pattern Library Size

The algorithm described in Figure 3-5 exhaustively generates all possible fragment patterns from an HLB template and this can lead to a large pattern library. The fragment pattern subgraphs are created by all possible combinations of edge deletions, edge bufferings and permutations.

```
P := { h }
ToDecom := P
while ToDecom ≠ ϕ do
    AddLib := ϕ

    for every graph g ∈ ToDecom do
        for every internal edge e in edge set e_g do
            delete e from g to create graphs g1, g2
            if g1 is not isomorphic to any graph in P, AddLib or ToDecom then
                AddLib := AddLib U {g1}
            end if
            if g2 is not isomorphic to any graph in P, AddLib or ToDecom then
                AddLib := AddLib U {g2}
            end if
            restore the deleted edge in g
        end for

        for every internal edge e connected to a leaf node in edge set e_g do
            buffer e to create graph g1
            if g1 is not isomorphic to any graph in P, AddLib, ToDecom then
                AddLib := AddLib U {g1}
            end if
            restore the buffered edge in g
        end for
    end for

    P := P U AddLib
    ToDecom := AddLib
end while

P := Permute(P)
```

Figure 3-5: Pseudocode describing the generation of the HLB fragment pattern library

In a fragment pattern for the HLB template, each hard-wired link can either be (1) present, (2) deleted or (3) connected to a LUT implementing a buffer. Thus the number of fragments that can be generated by an exhaustive application of the fragmentation operations is $O(3^m)$, where m is the number of internal edges in the HLB tree. Note that some of these patterns may be isomorphic to each other. Only the isomorphically unique fragment patterns are then permuted to generate the final library. This permutation of input edges may lead to an expansion factor of up to $K!$, where K is the number of inputs to the LUT basic blocks. This gives an upper bound of $O(K! * 3^m)$ fragment patterns.

However, symmetries in the patterns keeps the number of isomorphically unique HLB fragment patterns to much fewer than $O(3^m)$. For example, of the HLBs considered in the architectural studies of Chapter 6, the one that resulted in the most patterns is composed of nine 4-LUTs (the L3-9.3.2.2 topology). This HLB has 8 internal edges but resulted in only 150 ($<< 3^8 = 6561$) isomorphically unique fragment pattern trees. After permutation of the L3-9.3.2.2 4-LUT HLB library patterns, the library expanded to 18372 ($<< 4! * 3^8 = 157464$) patterns.

The generation of permuted patterns before covering allows the use of a simple matching function. The expansion of the library due to permutations can be avoided by using a more complicated matching function that generates the permutations of the isomorphically unique fragment patterns during run-time. However, a more complicated matching function would increase the time required for matching. This trade-off increased memory usage to reduce the matching execution time.

3.4.4 Selection of the Set of Covering Fragments

Given the subject DAG and the fragment pattern library, the next step is to select a set of fragment patterns that together form a minimum cost cover of the subject DAG. The covering algorithm finds the feasible fragments at each node and uses dynamic programming to select the best set of feasible fragments. The covering algorithms used for area and delay optimization have

several differences. The essence of both algorithms is captured in the **`findOptimalCover`** procedure outlined in Figure 3-6. The features unique to each of the delay- and area-optimization algorithms are related to their respective cost measures and pattern matching options, and these differences will be discussed in Section 3.4.5.

To map the entire subject network, the **`findOptimalCover`** procedure is invoked at every output node of the subject DAG. This recursive procedure maps the transitive fan-in of each node before mapping the node itself. The **`findOptimalCover`** procedure finds the lowest cost matching HLB fragment pattern at each node in the subject DAG and at completion of the procedure records the lowest cost pattern and lowest cost for that node. After completion of **`findOptimalCover`** at a given node, the node is said to be *mapped*. Each node has a **`Mapped`**

```
procedure findOptimalCover(n)
    if (isInput(n) or n.Mapped) then
        return
    end if

    /* find optimal cover for fan-in nodes of n */
    foreach fan-in node f of n do
        findOptimalCover(f)
    end for

    /* using optimal covers of fan-in of n construct optimal cover at n */
    n.Cost := INFINITY
    n.Match := NULL_PATTERN
    foreach fragment pattern p ∈ P do
        if isMatch(p, n) then
            currentCost := p.Cost + faninCost(n, p)
            if (currentCost < n.Cost) then
                n.Cost := currentCost
                n.Match := p
            end if
        end if
    end for
    n.Mapped := TRUE
end findOptimalCover
```

Figure 3-6: Fragment Covering Algorithm

flag to record its current state as well as attributes to indicate the lowest **Cost** and best **Match** pattern seen so far. Note that the area and delay cost of a primary input node is 0.

The transitive fan-in of several outputs may have overlapping regions and thus a node may be visited more than once by the covering algorithm. The **Mapped** flag is used to ensure that a node in the overlapping regions is only mapped once. The first time a node is visited and mapped its **Mapped** flag is set so that on subsequent visits the node will not be mapped again.

A detailed description of applying **findOptimalCover** at a node **n** is as follows: First the node **n** is checked to see if it is a primary input or has already been mapped, and if so the procedure returns. If **n** has not been mapped, the fan-ins of **n** are mapped first and then the algorithm proceeds to find the best matching pattern at **n**. All matching patterns are found using the algorithm outlined in Figure 3-7, and for each matching pattern **p**, the area- or delay-cost of using that fragment pattern at **n** is determined by summing the cost of **p** and the value of the function called **faninCost(n, p)**. The **faninCost(n, p)** is the cost of the fan-ins of the subgraph of the subject DAG matched by **p**. Let **s** be the subgraph of the subject DAG matched by **p**. For area, the **faninCost(n, p)** is the *sum* of the costs of the nodes that fan-in to **s.** For delay, the **faninCost(n, p)** is the *maximum* of the costs of the nodes that fan-in to **s**. Since the cost of the current node, **currentCost**, is found using the previously computed costs of the fan-in nodes, this algorithm uses a dynamic programming approach. The final mapped cost of **n** is the lowest cost over all matched patterns. The matching pattern that leads to the lowest cost is also retained. Finally, the node **n** is marked as being mapped so that each node is only mapped once. An example showing the construction of an optimal cover using a similar library-based mapping algorithm was given in Section 2.2.2.

The matching of HLB fragment pattern trees at a given node in the subject DAG is similar to the graph theoretic problem of finding all subgraph isomorphisms of the pattern trees on the subject digraph. Each isomorphism is called a *match*. A pattern tree match is defined in [38] as follows:

Definition 3-2 A (full-node) match of a pattern graph $G_p = (V_p, E_p)$ on a subject subgraph $G_s = (V_s, E_s)$ is a *one-to-one* mapping of the pattern graph nodes into the subject graph nodes $(I: V_p \rightarrow V_s)$ such that the following properties hold:

(i) for every edge e defined by the pair of nodes $(p_1, p_2) \in E_p$, the corresponding edge defined by nodes $(I(p_1), I(p_2)) \in E_s$.

(ii) for every non-input vertex v in V_p, $| i(v) | = | i(I(v)) |$.

Property (ii) adds a fan-in constraint to the definition of a match. Note that without (ii) we would simply say that E_p is isomorphic to an induced subgraph of E_s.

The *full-node* matching problem at a given node in the subject DAG is to find if a pattern matches according to Definition 3-2. It is called full-node because every node in the pattern graph is mapped to a single node in the subject graph. The mapping is described as one-to-one because each pattern node is matched to a single subject node and vice versa. There is also a unique mapping between each edge in the pattern graph and an edge in the subject graph. Note that a one-to-one mapping between pattern nodes and subject nodes and the fact that the pattern is a tree ensures that only matches to tree subgraphs of the subject DAG will be found. The first property of Definition 3-2 states that the edge relationships are preserved between the pattern and subject graph nodes and the second property states that the fan-ins of matching subject and pattern nodes must be equal.

However, recall that each internal node of an HLB pattern corresponds to a lookup table, and some of the inputs to a LUT may be left unused. Thus, when a LUT-based HLB fragment pattern matches the subject graph, some of the primary input edges of the pattern tree need not have corresponding subject graph edges. Nevertheless, each internal node of the subject digraph must have a corresponding internal node of a pattern tree. This leads to the following modified definition of a match, which gives a related problem referred to as the *internal-node* matching problem.

Definition 3-3 An internal-node (or non-input node) match of a pattern graph $G_p = (V_p, E_p)$ on a subject subgraph $G_s = (V_s, E_s)$ is an onto mapping of the *internal* pattern graph nodes into the subject graph nodes $(I: V_p \rightarrow V_s)$ such that the following properties hold:

(i) for every edge *e* defined by the pair of internal nodes $(p_1, p_2) \in E_p$, there is a corresponding edge defined by $(I(p_1), I(p_2)) \in E_s$.

(ii) for every internal node *v* in V_p, $| i(v) | \geq | i(I(v)) |$.

Note that the mapping between subject nodes and pattern nodes in Definition 3-2 is not necessarily one-to-one. A subject node with fan-out greater than one may be mapped to more than one pattern node during covering. The resulting cover may not be exact.

Property (i) of the Definition 3-2 states that the edge relationships between LUT basic blocks in the HLB pattern are exactly duplicated in the matching subject subgraph. The second property uses the ignorable input property of LUTs and states that the fan-in of an internal pattern node can be either greater than or equal to the fan-in of the matching subject node.

The internal-node matching algorithm function is outlined in Figure 3-7. This algorithm checks to see if a pattern subtree rooted at a node **p** matches the subject subgraph rooted at node **s**. Note that the matching algorithm allows a subject node to be mapped to more than one pattern node, that is, matches to subgraphs that have nodes with fan-out greater than one are allowed. Also the fan-in edge ordering in the subject DAG is preserved in the matching pattern tree.

In the non-trivial case, the algorithm recursively tests if the fan-in nodes of **p** matches the fan-in nodes of **s**. The function **fan-inNode(n, i)** returns the fan-in node on the **i**th fan-in edge. If each of the fan-ins of **s** match each of the first **fan-in(s)** nodes of **p** and all leftover fan-ins of **p** are input nodes, then the pattern match is successful. The pattern matching algorithm is *O(m)* where *m* is the number of edges in the pattern tree.

Figure 3-8 illustrates the matching of a pattern tree on a subgraph of the subject DAG. The pattern graph in Figure 3-8(a) shows a pattern with seven internal nodes, with each internal node

having two fan-in nodes. Figure 3-8(b) shows the matching of the pattern graph on a tree subgraph of the subject DAG. The subject DAG in Figure 3-8(b) consists of ten nodes and the matching pattern covers the seven nodes D, E, F, G, H, I and J. Note that nodes F and G have only one fan-in node each, so the matching pattern nodes will each have one unused input. Figure 3-8(c) illustrates the matching of the pattern tree on a subgraph with a fan-out greater than one node[1]. Nodes A and B in the subject DAG each have fan-out of two. In this match, nodes 1 and 3 of the

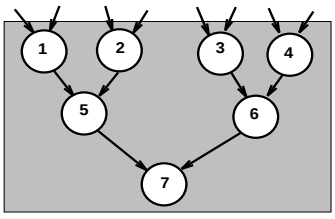
```
function isMatch(p, s)
    if isInput(p) then
        /* pattern input node matches any subject node */
        return MATCHED
    else if isInput(s) then
        /* no match since subject is an input node but pattern is not */
        /* for tree-matching, treat subject nodes with fan-out as inputs */
        return FAILED
    else if fan-in(p) < fan-in(s) then
        /* no match since fan-in of pattern is less than fan-in of subject */
        return FAILED
    else
        /* compare fan-in nodes of pattern and subject */
        /* assumes that fan-in edges of p, s have been enumerated */
        for i = 1 to fan-in(s) do
            if isMatch(fan-inNode(p, i), fan-inNode(s, i)) == FAILED then
                return FAILED
            end if
        end for

        /* if any leftover fan-ins of p are not inputs then matching FAILED */
        for i = fan-in(s)+1 to fan-in(p) do
            if isInput(fan-inNode(p, i)) == FALSE then
                return FAILED
            end if
        end for
        return MATCHED
    end if
end isMatch
```

Figure 3-7: Internal-node pattern matching algorithm

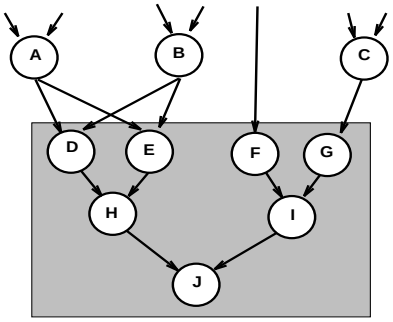
1. Note that hereafter a “fan-out node” will be assumed synonymous with “fan-out greater than one node”.

pattern graph are mapped to node A of the subject graph. Nodes 2 and 4 of the pattern graph are mapped to node B of the subject graph. If the covering algorithm chooses a pattern match that spans a fan-out node, then this node (and its fan-in edges) must be replicated in the final stage of the covering algorithm in order to realize the chosen fragment in the HLB circuit. In the example shown in Figure 3-8(c), nodes A and B and their fan-in input edges would each be duplicated as illustrated in Figure 3-8(d).

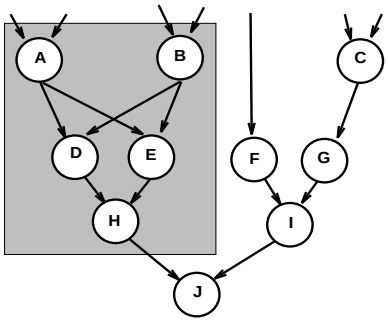
The final stage of the mapping algorithm constructs a netlist of the covering fragments. As was previously mentioned, some of the subject DAG nodes may have been matched to more than one fragment pattern node. During this stage, replication of subject DAG nodes is used to create a new subject network, which has the property that every one of its nodes is matched to a single



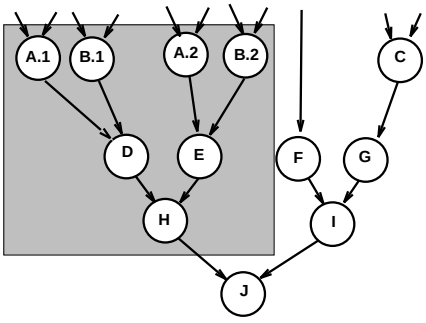
3-8(a): Fragment pattern



3-8(b): Match to tree subgraph



3-8(c): Match to subgraph with fan-out > 1 nodes



3-8(d): Replication of fan-out nodes in Figure 3-8(c)

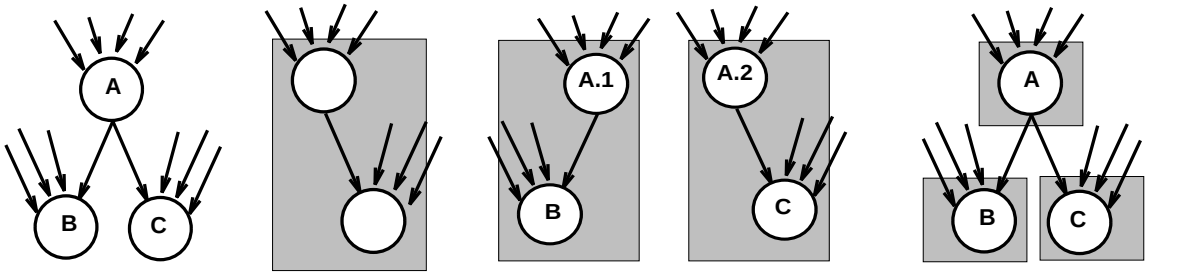
Figure 3-8: Pattern matches to a tree and non-tree subgraphs

fragment pattern node. This stage begins at the subject DAG output nodes and proceeds towards the primary inputs. If the best fragment pattern matches a *tree* subgraph of the subject DAG, then the correspondence between pattern and subject nodes is a simple one-to-one mapping and so there is no replication. However, if the best pattern matches a subgraph with fan-out nodes, then the fan-out subject nodes (and their fan-in edges) are replicated to make the pattern-to-subject node correspondence one-to-one. The algorithm then proceeds to visit the subject DAG nodes in the fan-in of the subgraph mapped to the current covering fragment.

3.4.5 Delay versus Area Optimization

The primary measure of delay cost used during fragment covering is the number of HLB covering fragments in the longest path. This measure is equivalent to counting the number of programmable connections in the longest path or assuming a unit delay for each HLB fragment in the fragment library. However, the actual delay of the HLB circuit is a combination of basic block delays and programmable connection delays. The basic block delays for every path are fixed before the mapping to HLBs and only change if extra basic blocks are introduced by buffered patterns. It is assumed that basic block delays are smaller than programmable connection delays and that few extra basic blocks are added to the critical path. Thus the only component of delay to be minimized is the number of programmable connection delays along the path.

In contrast, the primary measure of area used during fragment covering is the total number of HLB covering fragments. Note that the actual area cost of the HLB circuit is the number of HLBs after the fragments are packed together as tightly as possible (packing will be covered in Section 3.5). The rationale for minimizing the number of HLB fragments during covering is that the use of fewer fragments implies a greater total number of hard-wired connections are employed in the fragments. Since the packing of distinct fragments into the same HLB leads to a wastage of hard-wired connections, a greater number of hard-wired links in the fragments means that fewer are wasted when packing occurs.



3-9(a): Subject DAG **3-9(b): L2-2 HLB** **3-9(c): With replication** **3-9(d): No Replication**

Figure 3-9: Example where replication reduces area

The fragment pattern library and matching algorithm differ when optimizing for speed versus area. In speed optimization, the primary goal is to minimize the number of HLB fragment levels between primary inputs and outputs. In some cases all fan-ins of a basic block are needed to achieve the minimum number of programmable connections, even at the expense of added basic block delays on non-critical paths. Thus to minimize delay, the fragment pattern library includes all possible buffered patterns. Another feature of the delay-optimization algorithm is that the pattern matching step allows matches across fan-out nodes, and thus replication of fan-out nodes is used to reduce delay.

In contrast, the area-optimization library does not include buffered patterns, but has only the patterns generated by edge deletion. Recall that a buffered input represents a basic block used to implement a buffer and this is an inefficient use of a basic block. Contrary to the delay-optimization algorithm, during area optimization, matching does not occur across fan-out nodes. Thus the area algorithm does not allow replication of fan-out nodes.

It is true, however, that in some cases replication may reduce the number of packed HLBs. For example, suppose the subject DAG in Figure 3-9(a) is mapped to the L2-2 HLB in Figure 3-9(b). The solution with replication in Figure 3-9(c) has exactly two HLBs. Without replication, the solution in Figure 3-9(d) has three single-block fragments, each with fan-in equal to four. Only the uppermost basic block of the L2-2 HLB has fan-in of four and so, each single-block fragment must be packed in an HLB by itself. Thus, the no replication solution has three HLBs.

However, in many cases, replication may increase the area cost of the final packed HLB circuit by the following argument: Assume that each basic block has the maximum K inputs. For each basic block added by replication, not only is there an increase by one in the number of basic blocks, but there is also an attendant increase by K in the total number of edges in the subject DAG. The increase in the number of blocks and edges to be packed into HLBs tends to make circuits with replication require more packed HLBs and hence less area-efficient.

In this work, the benefits of replication during area-optimization were not investigated further.

3.5 Fragment Packing

After the selection of the set of covering fragments, the fragments are packed together as tightly as possible to minimize the final number of packed HLBs. The optimization goal of the algorithm presented in this section is to minimize the number of HLBs, without regard to the connectivity of the HLB fragments. Other optimization goals that take connectivity or placement of the HLB fragments into account during packing may yield a more routable solution, and hence a smaller and faster circuit after routing. However, there are often many alternative and equivalent combinations of covering fragments that may be placed within the same packed HLB. Thus, a well-chosen permutation of equivalent fragments in the minimized packed HLBs solution may lead to a solution with good routability, without resorting to optimization goals that take placement into account.

The following section begins with definitions for the fragment packing problem. The subsequent subsection describes the generation of the packing sets used to verify whether fragments can be packed together. The next subsection contains the fast heuristic algorithm used to pack the fragments together. The final subsection describes the method used by the packing algorithm to order the fragment trees.

3.5.1 Fragment Packing Problem Definitions

This section presents several definitions needed to understand the fragment packing problem and the algorithms used to solve the problem.

Denote by $P = \{ P_1, P_2, P_3, .. , P_l \}$ the set of l isomorphically unique fragment patterns in the library generated from the HLB template. Denote by $C = \{ C_1, C_2, C_3, .. , C_n \}$ the set of n pattern trees in the cover of the subject DAG, S, where each C_i is isomorphic to a member of P.

A *packing set* of an HLB is a set of fragment patterns such that the fragment pattern trees may be legally packed together in the same HLB template. A precise definition is as follows:

Definition 3-4 A set S is a *packing set* iff S is a cover of a subgraph of the HLB template.

The maximal packing sets are the largest sets of fragments that can be legally packed together in the same HLB template. A maximal packing set is defined as follows:

Definition 3-5 Given a packing set S, define a corresponding packing set S', which is equivalent to S except that each buffered input in S is transformed into a basic block in S'. Then S is a *maximal* packing set iff S' is a cover of the entire HLB template.

The algorithm outlined in Section 3.5.3 generates the set $M = \{ M_1, M_2, M_3, .. , M_p \}$, of p maximal packing sets from the HLB template. The maximal packing sets are used to validate HLB packings as follows:

Definition 3-6 A set X_i is a valid HLB packing if $X_i \subseteq M_j$ for some $M_j \in M$.

Given the above definitions, the fragment packing problem is defined as follows:

Definition 3-7 The **fragment packing problem**: Given P, C and M as defined above, find a partition of C into k valid HLB packings such that k is minimum.

3.5.2 Unique Ordering for Fragment Trees

A unique (descending) ordering for fragment trees is needed for efficient subset checking during fragment packing. The ordering of fragment pattern trees is accomplished by comparing the root-node label strings for the trees after each tree has been converted into a canonical form. The technique for labelling the root-nodes of the trees is derived from an algorithm in [45] that generates a unique label string for isomorphically unique trees.

The canonical labelling algorithm is shown in Figure 3-10 and is called **`labelNode`**. Input nodes are labelled with the string "1" and the size of these nodes is also set to the integer 1. In the general case, each of the fan-in nodes of the current node **n** are first labelled and then the fan-in labels are used to generate the label for **n**. The labels for the fan-ins are used to sort the fan-in

```
procedure labelNode(n)
    if isLeaf(n) then
        n.Label := "1"
        n.Size := 1
    else
        /* label fan-in nodes then put fan-ins in descending order of string labels*/
        for i := 1..fan-in(n) do
            labelNode(fan-inNode(n, i))
        end for
        sortFaninsByLabel(n)

        /* the size of current node equals sum of fan-in sizes plus 1*/
        n.Size := 1
        for i := 1 to fan-in(n) do
            n.Size := n.Size + fan-inNode(n,1).Size
        end for

        /* the current node'slabel is the node size concatenated with fan-ins' labels */
        n.Label := integerToString(n.Size)
        for i := 1 to fan-in(n) do
            n.Label := concatenate(n.Label, fan-inNode(n,1).Label)
        end for
    end if
end labelNode
```

Figure 3-10: Canonical Labelling Algorithm

nodes so that the left fan-in node has the largest label string and the rightmost fan-in node has the smallest label string. The size of **n** is then set to be the sum of the sizes of the fan-in nodes plus 1. The current node's label string is composed of the size of **n** concatenated with the fan-in label strings added onto the end of **n**'s label string from left to right. For example, the string label of node **A** in Figure 3-11 is {"5", "1", "1", "1", "1"} and the string label of node **B** is {"8", string label of **A**, "1", "1"}.

The ordering of the fragment trees into descending order of labels is done by comparing the string labels of the fragment trees character by character left to right. Since the label for any isomorphic tree is unique, the canonical labels can be used to make a unique ordering for the fragment pattern trees. The string label of node **B** starts with an "8" and the string label of node **A** starts with a "5", and so the tree rooted at **B** is greater than the tree rooted at **A**. One property of this ordering scheme is that the label of a tree, T_i which is a subgraph of another tree, T_j, is smaller and thus the ordering will have T_j before T_i. The scheme also orders single-block fragments in descending order of fan-in.

3.5.3 Generation of Maximal Packing Sets

The maximal packing sets used for validating each packed HLB are generated before proceeding to the actual packing. The input to this step is the HLB template and the output is a set M of maximal packing sets.

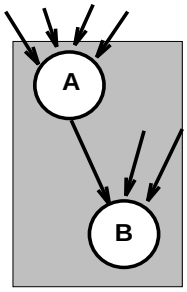


Figure 3-11: String Label Example

Assuming that the basic block has the properties of input ignorability and buffering capability, the maximal packing sets are generated using the same internal edge deletion and buffering operations as in the pattern generation step described in Section 3.4.3. As in the fragment pattern generation algorithm, the maximal packing set generation algorithm exhaustively deletes or buffers all internal edges. Thus, the set M contains all possible maximal packing sets. Similar to the case for fragments, the number of possible ways to select edges to buffer or delete implies that the number of maximal packing sets is $O(3^m)$. Note that during the generation of each maximal packing set, the list of fragment patterns in each M_i is kept in a canonical order to make checking for subsets of maximal packing sets more efficient. The method of determining the order for fragment trees is described in Section 3.5.2. If a generated packing set is a subset of another packing set (or vice versa), then only the larger packing set will appear on the list of maximal packing sets.

Figure 3-12 shows the L2-3 HLB template on the left and two maximal packing sets, M_1 and M_2, generated by the edge deletion operation on the right. Each maximal packing set is enclosed within a dotted rectangle. The first internal edge deletion operation generates a maximal packing set M_1 with two HLB fragment patterns, a two-basic block fragment pattern plus a single 4-input basic block fragment pattern. The dashed line between the two fragment patterns represents the deleted hard-wired input edge. Deletion of the internal edge from the two-basic block fragment pattern in M_1 generates a second maximal packing set, M_2, which consists of two 4-input basic blocks and one 2-input basic block. If buffered patterns are allowed then a third maximal packing set, consisting of a single two-LUT fragment, is generated by converting one of the non-root LUTs into a buffer.

3.5.4 The Fragment Packing Algorithm

The algorithm used to solve the fragment packing problem is shown in Figure 3-13. The input to the algorithm is a list of n unpacked HLB covering fragments, $C = \{ C_1, C_2, C_3, .. , C_n \}$. The output is a set Y of k packed HLBs, $Y = \{ Y_1, Y_2, Y_3, .. , Y_k \}$ for which k is minimized. Initially Y

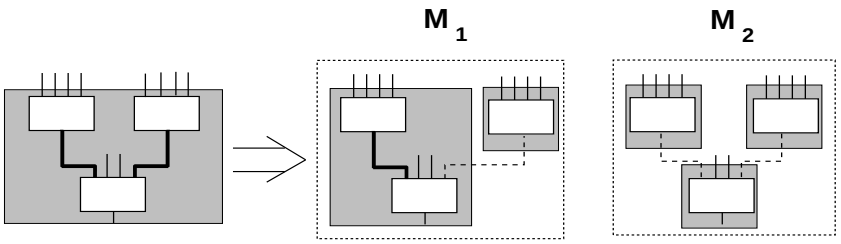


Figure 3-12: L2-3 HLB and maximal packing sets due to edge-deletion

is empty and as packed HLBs are constructed they are added to the set. The algorithm to construct each packed HLB is a "first-fit" packing algorithm on the set *C* of the fragments sorted into descending order of size. Because it is easier to fit smaller fragments into unused portions of an HLB, it is more effective to pack them last. This type of packing algorithm is generally known as a first-fit decreasing (FFD) packing algorithm because it processes a descending list by placing the largest object into the first container into which it can fit.

Initially the collection of packed HLBs, **Y**, is empty. First a sorted copy of the fragments in the set **C** are placed in the set **C'**. The outer loop constructs packed HLBs by adding the first remaining fragment from **C'** (called **addedFrag**) to the first packed HLB in the collection **Y** that can

```
Y := ϕ
C' := descendingSort(C)

while C' not= ϕ do
    addedFrag := deleteNext(C')

    if ∃ y ∈ Y such that isValidPackingSet(y U {addedFrag}) == TRUE then
        y := y U { addedFrag }
    else
        newPackedHLB := { addedFrag }
        Y := Y U newPackedHLB
    end if

end while
```

Figure 3-13: HLB Fragment Packing Algorithm

accommodate it. If there exists a **y** ∈ **Y** that can accommodate **addedFrag**, then **addedFrag** is added to **y**. If none of the packed HLBs in **Y** can accommodate **addedFrag**, then a new packed HLB, composed of only **addedFrag**, is added to **Y**. The outer while loop of the algorithm terminates when there are no remaining elements in **C'**.

The **isValidPackingSet** function checks if the candidate packed HLB is a subset of one of the maximal packing sets by calling the **subsetChecking** function in Figure 3-14. Before invoking the **subsetChecking** function, the fragment patterns in the candidate packed HLB are ordered using the scheme in Section 3.5.2. Recall that the fragment patterns in the maximal packing set are also ordered by the same scheme. The **nextFragment** function returns the next fragment in the maximal packing set or **NIL** if at the end of the list. The first time **nextFragment** is called it returns the first fragment.

First, the **subsetChecking** function compares the largest of the fragment patterns in the packed HLB against each fragment pattern in the maximal packing set until it finds a match. If a

```
function subsetChecking(packedHLB, packingSet)
    for each fragment h in packedHLB do
        matched := FALSE
        s := nextFragment(packingSet)
        repeat
            if isMatch(h, s) then
                matched := TRUE
                exit repeat loop
            else
                s := nextFragment(packingSet)
            end if
        until (s == NIL)
        if (matched == FALSE) then
            return FALSE
        end if
    end for
    return TRUE
end subsetChecking
```

Figure 3-14: The subset checking function

match is found, then the next largest fragment pattern in the packed HLB is compared to the remaining fragments in the maximal packing set. For any fragment pattern in the packed HLB, if a match is not found then the packed HLB is not a subset of this maximal packing set and the function returns **FALSE**. If all packed HLB fragments are matched the function returns **TRUE**.

3.6 Conclusion

This chapter presented the algorithms used to map LUT networks to HLBs. The following chapter discusses the complexity and optimality of the HLB technology mapping algorithms. The subsequent chapter (Chapter 5) contains an evaluation of the effectiveness of the HLB mapping algorithms with respect to theoretical bounds and to a technology mapper for a commercial LUT-based FPGA architecture with hard-wired connections.

Chapter 4 Complexity and Optimality of the HLB Mapping Algorithms

This chapter derives the complexity of the HLB mapping algorithms and gives statements and proofs concerning the optimality of the algorithms described in Chapter 3. One key theoretical result is that the fragment covering algorithm is delay optimal with respect to the number of programmable connections in a critical path of the HLB cover. The other important result is that the fragment packing algorithm results in a minimal number of packed HLBs when packing the covering fragments of any two-level HLB topology.

The first section of this chapter contains the complexity derivations and optimality statements and proofs for the fragment covering algorithm. The second section derives the complexity of the fragment packing problem and the heuristic fragment packing algorithm and shows sufficient conditions for the heuristic algorithm to be optimal. The second section also demonstrates the optimality of the fragment packing algorithm for all two-level HLB topologies. The final section concludes this chapter.

4.1 Complexity and Optimality of Fragment Covering Problem and Algorithm

This section reviews the fragment covering problem in Section 4.1.1 and then examines the run-time complexity of the fragment covering algorithm in Section 4.1.2. The next subsection (Section 4.1.3) considers the optimality of the fragment covering algorithm with respect to the

minimization of the delay and area of an HLB cover. The fragment covering algorithm is shown to be optimal for delay but sub-optimal for area.

4.1.1 Covering Problem Definition and Algorithm Review

The inputs to the fragment covering algorithm are a subject DAG, *S*, and the HLB template, *H*. In the HLB covering algorithm described in Section 3.4, the template *H* is used to generate the fragment pattern library, $P = \{ P_1, P_2, P_3, .. , P_l \}$. Every pattern in *P* is matched against the nodes in *S* to find feasible fragments. The goal of the fragment covering algorithm is to select the least cost feasible fragment matches for the covering set of fragments, $C = \{ C_1, C_2, C_3, .., C_n \}$. The area cost of the cover is the number of fragments in the cover and the delay cost of the cover is the number of fragments along the critical path. The last stage of the covering algorithm generates the final HLB fragment netlist by using replication whenever the subgraphs of *S* covered by the C_i overlap. The algorithm replicates the overlapping regions to ensure that the final mapping between fragment pattern nodes and subject nodes is one-to-one.

4.1.2 Complexity of Fragment Covering Algorithm

The general problem of constructing an optimal area or delay cover of a subject DAG using pattern trees has been proven to be NP-hard [19] [38]. Note that in the general problem, the delay model includes the effect of loading on the outputs of gates, whereas the problem in this dissertation assumes unit delay for each gate, and so the delay calculations are not dependent on fan-out. The algorithm for computing the best pattern match at each node during fragment covering entails finding all pattern matches at each subject node, and for each pattern match, evaluating the cost of using that match. The evaluation of the cost of each match is a simple computation and so the computational complexity of the finding the best pattern matches is dominated by the matching algorithm. After finding the best pattern matches at each node in the subject DAG, the fragment covering algorithm assembles the set of covering fragments using a pre-order traversal from the outputs.

The complexity of selecting the best pattern matches has running time proportional to the number of matches at each node in the subject DAG. In the worst case the running time will be $O(|S||P|)$, where $|P|$ is the total number of fragment patterns in the library and $|S|$ is the total number of nodes in the subject DAG. An upper bound on $|P|$ was shown in Section 3.4.3 to be $O(K! * 3^m)$, where K is the number of inputs to the LUT basic block and m is the number of internal edges in the HLB. Thus, the complexity of finding the best matches is $O(|S| * K! * 3^m)$. The complexity of assembling the covering set of fragments is the same as the complexity of a pre-order traversal, which is $O(|S|)$. Thus the overall complexity of the fragment covering algorithm is $O(|S| * K! * 3^m)$.

Cost of Replication

The covering algorithm uses replication during delay optimization (but not during area optimization). We now show that the run-time cost of replication is bounded by a finite factor.

The replications may be done for every fragment pattern in the cover, and the upper bound on the number of fragment patterns is $|S|$. The HLB template has m internal edges and $m+1$ nodes. Thus, for each of the subject nodes, $O(m)$ new nodes may be created for each fragment pattern in the cover during replication. The cost of node creation is constant, and so the complexity of the replication stage is $O(|S| * m)$. This calculation also shows that in the worst case replication may increase the size of the subject network by a factor of m.

4.1.3 Optimality of Fragment Covering Algorithm

This section will discuss the optimality of the fragment covering algorithm, described in Figure 3-6, with respect to delay and area.

Delay Optimality of Fragment Covering Algorithm

This sub-section proves that the fragment covering algorithm generates a delay-optimal solution with respect to the number of programmable connections on the critical path, for an arbitrary subject DAG. The delay-optimizing version of the fragment covering algorithm has *all*

possible buffered and delete-edge fragment patterns in the pattern library *P*, uses matching across fan-out nodes during covering, and uses replication of subject nodes to construct the final HLB netlist.

This sub-section first defines delay and delay optimality of the covering set of fragments. Then the fragment covering algorithm is shown to be delay optimal for subject DAGs that are trees. Finally, the optimality of the algorithm is extended to arbitrary DAGs. Note that in the proof of delay optimality it is assumed that the subject DAG, *S*, corresponds to the transitive fan-in of a single primary output *o*. After proving delay optimality for mapping *S* with a single primary output, it is shown that delay optimality holds for mapping *S* with multiple outputs.

Note that in our definition of delay, we only consider the delay due to programmable connections. Each fragment pattern has an associated delay of one programmable connection.

Definition 4-1 Given an HLB template, *H*, a subject DAG, *S*, and the set of covering fragments $C = \{ C_1, C_2, C_3, .., C_n \}$, the delay of a node $v \in V(S)$ is defined as follows:

i) If *v* is a primary input then $delay(v) = 0$.

(ii) Otherwise, $delay(v) = 1 + \max_{u \in I(F)} (delay(u))$, where F is the subgraph of *S* rooted at *v* and covered by a $C_i \in C$. Recall that $I(F)$ is the fan-in of *F*.

A less formal way of defining the $delay(v)$ is that it is equal to the maximum number of fragments between *v* and any primary input.

The *cone* rooted at a subject DAG node *v* denotes the entire transitive fan-in of *v*, including primary inputs.

Definition 4-2 Given an HLB template, *H*, and a subject DAG, *S*, which is the cone for a primary output, *o*, a *delay optimal* set of covering fragments, *C*, is one such that *delay(o)* is minimal.

Note that all possible HLB fragment pattern trees implementable by *H* will be considered by the delay-optimizing fragment covering algorithm because the fragment generation algorithm is exhaustive.

The following lemma states that for any subject node *v*, the matching algorithm in Figure 3-7 will correctly find any feasible match between a pattern tree and a subject subgraph (with or without fan-out nodes) rooted at *v*. A match is defined in Definition 3-2. The matching algorithm compares the number of fan-in nodes of the subject node *v* and the root node of the pattern tree P_i to see if there is a match. If the match between the two nodes is successful, the algorithm recursively compares the fan-in nodes of *v* and the root of P_i in the same fan-in edge order.

Lemma 1 The matching algorithm in Figure 3-7 will correctly find a match (according to Definition 3-2) between a pattern tree *p* and a subject DAG subgraph rooted at subject node *s*, if such a match exists. If there is no match then the algorithm fails, correctly.

Proof by induction:

Base cases: If *p* and *s* are both primary inputs OR *p* is a primary input and *s* is not a primary input then a match is found. In this match *p* is mapped to *s* and the match is correct because both properties of Definition 3-2 are satisfied. If *p* is not a primary input and *s* is a primary input node then the matching algorithm terminates and fails correctly because it does not satisfy the fan-in constraint (property (ii) of Definition 3-2).

General Case: This is the case when both *p* and *s* are not primary inputs. Assume that the fan-ins of *p* and *s* have been matched properly. The matching algorithm compares the in-degrees of *p* and the subject node *s* to ensure that $|i(p)| \geq |i(s)|$, that is, property (ii) of Definition 3-2 is satisfied. If this condition is false then the algorithm fails correctly for nodes *p* and *s*. The fan-in node

checking sets up an edge correspondence between E_p and E_s by mapping the fan-in edges of p to those of s in the order they are visited. If $|i(p)| \geq |i(s)|$, then this mapping satisfies property (i) of Definition 3-2. Thus if the fan-in nodes of p and s match, then the algorithm will also match p and s correctly. If the match of the fan-in nodes fails then the algorithm will fail correctly.

The matching algorithm has been shown to work properly when p or s are primary inputs. For the case when both p and s are not primary inputs, if the algorithm works properly for the fan-ins of p and s, it has been shown to work properly for p and s. Thus, by induction it follows that the matching algorithm will work properly for any given p and s.

QED.

The fragment covering algorithm in Figure 3-6 uses dynamic programming and tree-matching. A well known result [19] [38] states that if the subject DAG is a tree and the library of patterns also consists of trees, then a dynamic programming and tree-matching covering algorithm produces a delay optimal cover of library gates for the unit-delay gate model. The proof of the optimality of a dynamic programming and tree-matching covering algorithm, when applied to a subject tree, was given in [38]. A similar proof will be repeated here for the sake of completeness.

Theorem 4 Given a subject tree S, corresponding to a primary output o, and a library of fragment pattern trees $P = \{ P_1, P_2, P_3, .. , P_l \}$, corresponding to the HLB template H, the fragment covering algorithm (**`findOptimalCover`**) outlined in Figure 3-6 produces a delay optimal cover of fragments $C = \{ C_1, C_2, C_3, .. , C_n \}$ for o.

Proof by induction:

Base case: The subject tree S is a primary input. The algorithm terminates with the optimal delay of 0 for S. The optimal fragment choice when S is a primary input is the empty set.

General case: The subject tree S is not a primary input and is rooted at a node v.

Since all possible HLB fragment pattern trees for *H* are in the library *P*, Lemma 1 ensures that the matching algorithm will correctly find *all* possible feasible HLB fragments for the subject node *v*.

Suppose that the fragment choices at each of the nodes in the transitive fan-in of *v* result in the optimal delay. The definition of *delay(v)* is $delay(v) = 1 + \max_{u \in I(F)} (delay(u))$, where a feasible $C_i \in P$ matches the subject subgraph *F* rooted at *v*. The fragment covering algorithm selects from among all possible feasible fragment patterns, a $C_{opt} \in P$ that gives the minimum delay. Since the optimal fragment choice for all nodes in the transitive fan-in of *v* have been determined, this C_{opt} will be the delay optimal fragment choice for the cover of the tree rooted at *v*. Thus the fragment covering algorithm generates the delay optimal fragment choice for a given non-input node *v* given the optimal fragment choices for the fan-in of *v*.

Since the fragment covering algorithm finds the delay optimal fragment choice for primary input nodes and also generates the delay optimal fragment choice for a given non-input node *v* given the optimal fragment choices for the fan-in of *v*, then by induction, the fragment covering algorithm generates a delay optimal fragment choice for any given subtree, including the one rooted at a primary output node *o*.

Given the delay optimal fragment choices for an output node *o* and its transitive fan-in nodes, a delay optimal cover of fragments $C = \{ C_1, C_2, C_3, .. , C_n \}$ for the tree rooted at *o* can be trivially generated by a pre-order traversal. The traversal starts at the $C_i \in C$ rooted at *o*, adds this C_i to *C*, then recursively adds the $C_j \in C$ at each of the fan-in nodes of the subject subgraph covered by C_i. Note that there is no replication during the construction of the delay optimal cover because the mapping between nodes in the C_i and *S* are one-to-one.

QED.

What remains to be shown is that the fragment covering algorithm will also produce a delay optimal solution if the output cone rooted at primary output *o* is not a tree, that is, if the cone that produces *o* contains fan-out nodes. The key to the proof is to show that the inclusion of matches between fragment pattern trees and subject DAG subgraphs with fan-out nodes[1] does not affect the delay optimality of the set of covering fragments.

The selection of a fragment that matches to a subject DAG subgraph with fan-out nodes implies replication of subject nodes when constructing the final HLB circuit. During construction of the final netlist, if the best pattern P_i matches a subgraph F of the subject DAG with fan-out nodes, the fan-out nodes of F are replicated so that the final mapping has a one-to-one correspondence between the nodes and edges of P_i and F. The next lemma states that these replications do not affect the delay optimality of the cover.

Lemma 2 For a subject DAG node *v*, if any subject DAG subgraph F rooted at *v* matches a fragment pattern P_i, the replication of internal nodes of F does not affect the delay of *v*.

Proof Given a node *v*, there is the subject subgraph F covered by the fragment P_i that matches at *v*. The calculation of the *delay(v)* uses the maximum of the delays of the fan-ins of F. Because the replication of internal nodes (and their fan-in edges) of subject subgraph F does not introduce another fragment level between the inputs of F and the node *v*, it does not affect the delay of the fan-ins of F and thus does not affect the *delay(v)*.

QED.

Given that replications of the subject nodes do not affect the delay of the cover, the delay optimality of the fragment covering algorithms can be extended to subject DAGs in the following theorem.

1. Recall that “fan-out” nodes is short for “fan-out greater than one nodes”.

Theorem 5 Given a subject DAG *S*, corresponding to a primary output *o*, and a library of fragment pattern trees $P = \{ P_1, P_2, P_3, .. , P_l \}$, corresponding to the HLB template *H*, the fragment covering algorithm produces a delay optimal cover of fragments $C = \{ C_1, C_2, C_3, .. , C_n \}$ for *o*.

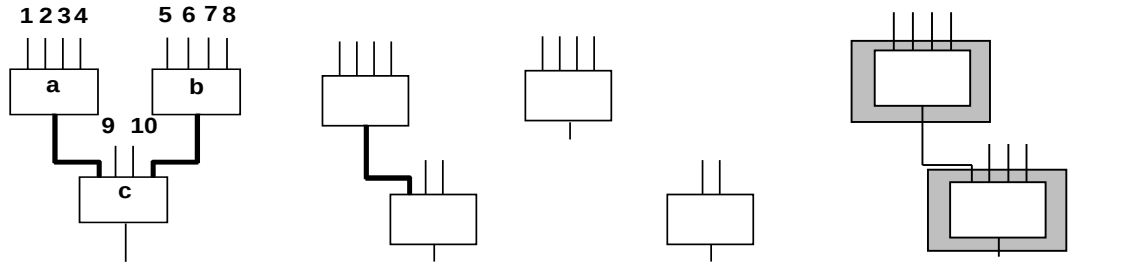
Proof The induction proof proceeds exactly as in the proof of Theorem 4 except that the construction of the final HLB netlist may require the replication of fan-out nodes in matching subject subgraphs. Lemma 2 states that the delay of the node *v* implemented by a fragment pattern match to a subject subgraph *F* with fan-out is unchanged by the replication of fan-out nodes that are internal to *F*. Thus the delay optimality of the cover is unchanged by these replications. If the subject DAG node to be mapped is rooted at the primary output *o*, then the covering set of fragments, $C = \{C_1, C_2, C_3, .. , C_n \}$, is delay optimal for the output *o*.

QED.

The delay optimality of the fragment covering algorithm can be easily extended to multiple output DAGs. The covering fragments for each output cone can be determined separately. The construction of the final HLB netlist may require replication for the overlapping regions between output cone covers. However, by Lemma 2 this does not affect the delay optimality of the individual output nodes' covers and so the combination of the separate output covers will also be delay optimal.

Sub-optimality of Area Optimization Algorithm

This section shows that the fragment covering algorithm is not area optimal for an arbitrary subject DAG. The area-optimizing version of the fragment covering algorithm uses only delete-edge fragment patterns in the pattern library *P* and does not allow matching across fan-out nodes during the covering stage. Since matching across fan-out is not allowed, replication is not needed to construct the final HLB netlist.



4-1(a): L2-3 HLB **4-1(b): L2-3 HLB fragments** **4-1(c): Mapped Subject DAG**

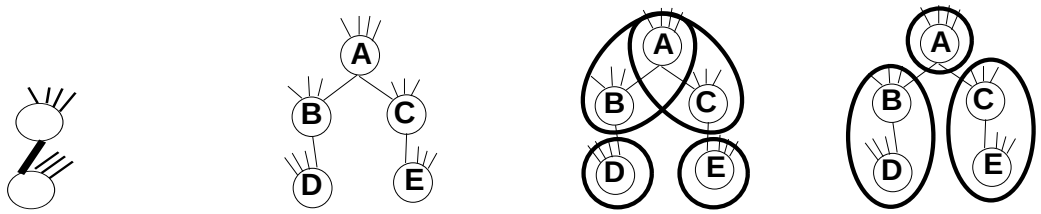
Figure 4-1: Example to show sub-optimality of area algorithm

This section first defines area and area optimality of the covering set of fragments. Then the fragment covering area-optimizing algorithm is shown to be sub-optimal for subject trees using a counter-example.

Definition 4-3 Given an HLB template, *H*, a subject DAG, *S*, and the covering set of fragments for *S*, $C = \{C_1, C_2, C_3, .. , C_n\}$, the area of the cover of *S* is the number of fragments in the covering set of fragments, *n*. An area optimal set of covering fragments, is one such that *n* is minimal.

One reason that the area algorithm is sub-optimal is because the fragment pattern library used in the area algorithm does not contain the HLB fragment patterns with buffers. The use of buffered fragments may reduce the total number of HLB fragments (this is the area cost function in covering) after mapping. However, buffered patterns waste basic blocks and the extra basic blocks will often lead to a greater number of HLBs after fragment packing.

Figure 4-1 contains an example that demonstrates that the area-optimizing version of the fragment covering algorithm is not optimal with respect to minimizing the number of fragments. Figure 4-1(a) shows the L2-3 HLB and Figure 4-1(b) shows its isomorphically unique delete-edge fragments. If the subject tree in Figure 4-1(c) is mapped using the library in Figure 4-1(b), then the area-optimizing algorithm gives the set of two covering fragments enclosed in the shaded boxes. However, a smaller area cost in terms of fragments may be obtained if the library were



4-2(a): L2-2 HLB **4-2(b): Subject network** **4-2(c): Sub-optimal mapping** **4-2(d): Optimal mapping**

Figure 4-2: Sub-optimal replication example

expanded to include HLB fragments with buffers. With the complete isomorphically unique library shown in Figure 3-1, the subject tree could be mapped to only one HLB fragment. Since the area-optimizing algorithm resulted in a solution with more fragments, it is sub-optimal for trees and will also be sub-optimal for DAGs.

Extensions to the Area Optimization Algorithm

There are extensions to the area optimization algorithm that may result in mappings with fewer HLB fragments. As shown above, a complete fragment pattern library that includes the HLB fragment patterns with buffers results in solutions with fewer fragments. Similar to the delay optimizing algorithm, the area-optimizing algorithm will be optimal for trees if the subject graph is a tree and a complete HLB pattern tree library is used [19] [38].

The use of replication during area optimization may also reduce the number of HLB fragments in subject networks with fan-out. However a covering algorithm with replication may not lead to an area-optimal cover if the area cost function is simply the sum of the area costs (in terms of number of HLB fragments) of the transitive fan-in nodes. Note that we also assume that if two solutions have the same cost in HLB fragments, then the solution with the smallest root HLB fragment is selected. For example, Figure 4-2 shows an example where replication results in a sub-optimal solution. Figure 4-2(a) shows the L2-2 4-LUT HLB template and Figure 4-2(b) shows the subject network to be covered by the L2-2 HLB.

Assuming replication were allowed, the mapping of the subject network would proceed as follows. Node **A** of the subject network is trivially mapped to a single-block fragment and has area cost of one. Nodes **B** and **C** are each mapped to an entire L2-2 HLB (that includes node **A**) and both have area cost of one. Note that this mapping implies a replication of node **A**. Node **D** is mapped to a single-block fragment and its area cost is one plus the cost of node **B**, which is equal to two. Node **E** is also mapped to a single-block fragment and its area cost is also two. The entire area cost of the mapped network is the sum of the costs of **D** and **E**, which is four HLB fragments. The final mapping of the subject network is shown in Figure 4-2(c). The nodes covered by each of the four HLB fragments are encircled by a thick line.

However, an optimal covering algorithm would not replicate node **A**. The optimal covering algorithm would implement nodes **D** and **E** with complete L2-2 HLBs and node **A** with a single-block fragment. The cost of the optimal cover is three HLB fragments. The optimal cover is illustrated in Figure 4-2(d). Thus the covering algorithm with replication and using the simple area cost function is sub-optimal.

4.2 Complexity and Optimality of Fragment Packing Problem and Algorithm

This section reviews the fragment packing problem in Section 4.2.1 and then shows that the fragment packing problem can be solved optimally in polynomial time in Section 4.2.2. The complexity of the heuristic fragment packing algorithm is discussed in Section 4.2.3. The optimality of fragment packing is defined in Section 4.2.4 and then the conditions under which the heuristic algorithm is optimal are described in Section 4.2.5. The final subsection (Section 4.2.6) gives a counter-example that shows that the heuristic fragment packing algorithm is, for the general case, sub-optimal.

4.2.1 Packing Problem Definition Review

The fragment packing problem is defined as follows: Given a set $P = \{ P_1, P_2, P_3, .. , P_l \}$, where each P_i corresponds to a fragment pattern, the covering set of fragments $C = \{ C_1, C_2, C_3, .. , C_n \}$, where each C_i is isomorphic to a member of P, and a collection of maximal packing sets $M = \{ M_1, M_2, M_3, .. , M_m \}$ where each M_i is a subset of P, the fragment packing problem is to find a partition of C into a collection of disjoint subsets, $X = \{ X_1, X_2, X_3, .. , X_k \}$, such that each $X_i \subseteq M_j$ for some j and k is as small as possible.

4.2.2 Complexity of Fragment Packing

The HLB fragment packing problem shares some features with the integer bin-packing problem. The integer bin-packing problem can be stated as follows: Given a set of n items, each item having an integer size s_i, $1 \leq s_i \leq K$, pack the n items into a minimum number of integer bins of integer size K. Assuming that the integer size K is a constant, the integer bin-packing problem can been shown to be polynomial in n [42]. It will now be shown that the fragment packing problem can also be solved optimally in polynomial time.

Theorem 6 The fragment packing problem described in Section 3.5.1 can be solved optimally in polynomial time.

Proof The proof shows that an exhaustive algorithm to solve the fragment packing problem optimally can be executed in polynomial time. An outline of the exhaustive algorithm is as follows: Given n fragments to pack, all possible sets of n or fewer packed HLBs are generated. Each generated set of packed HLBs, X, is checked to determine if it contains all of the n covering fragments in C, that is, if X is a *valid* set of packed HLBs. The valid set of packed HLBs with smallest cardinality is the optimal packing.

The following proof assumes that the HLB has b LUTs and is represented by a directed tree consisting of b internal nodes and $m = b\text{-}1$ internal edges. The covering set of n fragments to be

packed is denoted by C. It is also assumed that the cardinality of the collection M of maximal packing sets is p. The upper bound on the number of maximal packing sets p was shown in Section 3.5.3 to be 3^{b-1}. Since b is a constant, p will also be bounded by another constant.

We now determine an upper bound on the number of sets of packed HLBs that the algorithm will generate given n fragments to pack. Suppose the integer n is partitioned into integers n_0, n_1, n_2, .. , n_p such that $\sum_{i=0}^{p} n_i = n$. Let each n_i, where $i = 1 .. p$, correspond to the number of occurrences of packing set M_i in the generated packed HLB set X and n_0 correspond to the unused part of n that has not been assigned to any of the p packing sets. The fragments in X correspond to the union of the fragments in n_1 packing sets of type M_1, n_2 packing sets of type n_2, .. , and n_p packing sets of type M_p. Note that if $n_0 > 0$ then the number of packed HLBs in X is less than n and thus if X is valid then some of the packed HLBs in X contain more than one fragment from C. There is no need to consider partitioning an integer greater than n because there are at most n packed HLBs for n fragments. The inclusion of n_0 in the partitioning of n ensures that all solutions with fewer than n packed HLBs will also be considered. The number of possible sets of packed HLBs is bounded by the number of ways of partitioning the integer n into $p+1$ partitions, which is less than or equal to $\binom{n+p}{p}$.

The validity checking can be formulated as a bipartite matching problem as follows. Let G be a bipartite graph with a vertex set equal to the union of the two sets of vertices, V_c and V_x and a set of edges E. Each vertex in V_c corresponds to a fragment in the covering set C and each vertex in V_x corresponds to a fragment in the packing sets of X. If a fragment f in X can cover a fragment g in C, then there is an edge between the corresponding vertices in V_x and V_c. If the cardinality of the maximum matching of G is equal to $| V_c |$ then X is a valid set of packed HLBs. For all valid solutions, the one with the lowest number of packed HLBs (that is, the lowest value of $n-n_0$) is the optimal solution.

The cost of validity checking is the sum of the costs of creating the bipartite graph *G* plus the costs of solving the bipartite matching problem. The cost of constructing *G* is the cost of applying the fragment matching algorithm times the number of possible fragment matches. The cost of using the fragment matching algorithm is *O(m)*, where *m* is a constant equal to the number of internal edges in the HLB. There are at most *n* packed HLBs in *X* and each packed HLB has at most *b* fragments, so there are *O(b n) = O(n)* fragments in *X*. There are *n* fragments in *C*, and so there are $O(n^2)$ possible applications of the fragment matching algorithm and thus $O(n^2)$ edges in *G*. The complexity of generating the bipartite graph is therefore $O(m\ n^2) = O(n^2)$. The cost of solving the bipartite matching problem is $O(n^{1/2}E) = O(n^{1/2}n^2)$ [44]. Therefore the overall complexity of validity checking is $O(n^{5/2})$.

Thus the expression for the overall run-time complexity of the exhaustive algorithm, *T(n)*, is bounded as follows:

$$T(n) \leq \binom{n+p}{p} \times n^{5/2}$$

$$T(n) \leq (n+p)^p \times n^{5/2}$$

Therefore:

$$T(n) = O\left(n^{p+\frac{5}{2}}\right)$$

Since the number of maximal packing sets *p* is bounded by a constant, *T(n)* is polynomial in *n*, and thus the run-time complexity of the fragment packing problem is also polynomial.

QED.

However, the exponent of the polynomial for the above algorithm is large and thus the algorithm may not be practical.

4.2.3 Complexity of the Heuristic Fragment Packing Algorithm

The heuristic fragment packing algorithm described in Section 3.5.4 first sorts the elements of C into descending order, and then proceeds to create the list of packed HLBs, X. The largest remaining C_i is placed in the first current packed HLB that can accommodate it. If no current packed HLB can accommodate the largest remaining C_i then a new packed HLB is created for that C_i and added to X. The fragment packing is *greedy* because it packs the largest remaining fragment, C_i, into the unused part of the first packed HLB in X into which C_i can fit. This type of heuristic algorithm is referred to as a first fit decreasing (FFD) fragment packing algorithm.

The two steps of the heuristic fragment packing algorithm that determine its complexity are the sorting of the covering fragments list, C, and the construction of the packed HLBs from the sorted list. Assume that there are n fragments to pack and that there are p maximal packing sets. The fragments can be sorted using a sorting algorithm, such as heapsort, in $O(n \log n)$ time [45].

The construction of the packed HLBs requires scanning each of the n covering fragments exactly once. Each scanned fragment is added to each of the current packed HLBs, one after the other, to give candidate packed HLBs. Each candidate packed HLB is then validated by comparing it to each of the p maximal packing sets for containment. If the candidate packed HLB is valid then the fragment is packed there.

If there are b basic blocks in the HLB template, then there are at most b fragments in each maximal packing set and at most b fragments in each packed HLB. Checking if a candidate packed HLB is a subset of a given maximal packing set requires a linear comparison of both lists and is thus $O(b)$. Because there are p maximal packing sets, the complexity of validating a packed HLB is $O(p\ b)$. There may be up to n candidate packed HLBs for each fragment and so the cost of determining where to pack a fragment is $O(n\ p\ b)$. Since there are n fragments to pack, the total complexity of constructing the packed HLBs is $O(n^2\ p\ b)$. Since p and b are constants, the complexity of constructing the packed HLBs is $O(n^2)$.

Since the complexity of sorting the fragments is $O(n \log n)$ ($< O(n^2)$) and the cost of constructing the packed HLBs is $O(n^2)$, the overall complexity of the FFD fragment packing algorithm is $O(n^2)$.

4.2.4 Definition of Optimality for Fragment Packing

Given the packing problem definition in Section 4.2.1, an *optimal* algorithm for an instance of the fragment packing problem is one that finds the minimal number of packed HLBs, k. More formally stated, the optimality of a valid packing of fragments is as follows:

Definition 4-4 The valid packing of the n covering fragments of $C = \{ C_1, C_2, C_3, .. , C_n \}$ into a collection of disjoint subsets $X = \{ X_1, X_2, X_3, .. , X_k \}$ is *optimal* if and only if for any other valid packing $Y = \{ Y_1, Y_2, Y_3, .. , Y_y \}$ of C, $y \geq k$.

The following section proves that the FFD fragment packing algorithm is optimal under certain constraints on the maximal packing sets and that all two-level HLBs fit these constraints.

4.2.5 HLBs for which FFD Fragment Packing is Optimal

The following theorem defines a set of sufficient conditions for the maximal packing sets of an HLB so that the FFD fragment packing algorithm will be optimal. Note that in the following theorem, the HLB is assumed to consist of several basic blocks with an identical fan-in of K.

Theorem 7 The FFD fragment packing algorithm described in Section 3.5.4 is optimal if the maximal packing sets of the HLB satisfy the following two constraints:

(i) Every maximal packing set with multi-block fragments can have only one multi-block fragment and zero or more single-block fragments.

(ii) There is only one maximal packing set consisting of only single-block fragments.

Proof Because the FFD fragment packing algorithm sorts the covering fragments in descending order of labels it packs the multi-block fragments first. Constraint (i) implies that a packed HLB

with multi-block fragments can accommodate exactly one multi-block fragment. Thus, the packing of each multi-block fragment is trivial. No other rearrangement of the multi-block fragments can lead to a better packing and so the FFD fragment packing algorithm packs the multi-block fragments optimally. The unused part of an HLB will be referred to as a *hole*. The FFD fragment packing algorithm must now pack the single-block fragments into either the holes left after packing the multi-block fragments or some newly created HLBs.

The holes left after packing multi-block fragments are single-block holes. A single-block hole can accommodate a single-block fragment if the fan-in of the fragment is less than or equal to the fan-in of the hole. FFD fragment packing places the largest remaining single-block fragment in any single-block hole that can accommodate the fragment. The packing of each single-block hole is independent of each other because only one fragment can fit in each hole. The next paragraph shows that the FFD fragment packing algorithm fills the single-block holes optimally with single-block fragments.

Assume that the strategy of greedily placing the largest remaining single-block fragment, *f*, in any hole, *h*, that can accommodate *f* leads to a sub-optimal packing. This means that there exists a subsequent single-block fragment *g* that can fill *h* better than *f*, that is, *g* has greater fan-in than *f*. However, this is a contradiction because no fragment after *f* can have greater fan-in because the single-block fragments have been sorted in descending order of fan-in.

After exhausting the holes due to multi-block fragments, any remaining single-block fragments must be packed in HLBs that consist of only single-block fragments. Constraint (ii) says that there is a unique maximal packing set that consists of only single-block fragments. Thus, there is no alternative way of packing together only single-block fragments in an HLB. The single-block fragments in the maximal packing set can be considered as single-block holes. As

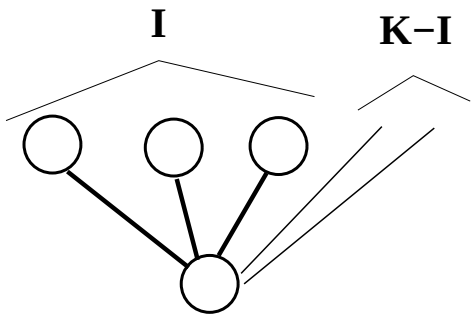


Figure 4-3: A generic two-level HLB

shown earlier in this proof, the FFD strategy for packing single-block fragments in single-block holes is an optimal strategy.

QED.

All two-level HLBs are shown to fit the constraints of Theorem 7 in the next subsection and thus FFD fragment packing will be optimal for these HLB topologies.

Two-level HLBs

A generic two-level HLB consisting of *I+1* K-input basic blocks and *I* hard-wired links is shown in Figure 4-3. The root basic block is hard-wired to the *I* leaf blocks and the root block has *K–I* primary inputs. Note that the primary inputs of the *I* leaf blocks are not shown.

First we show that a two-level HLB has at most one multi-block fragment in any maximal packing set and thus satisfies constraint (i) of Theorem 7. Every multi-block fragment must include the root block. Since there is only one root block there cannot be more than one multi-block fragment.

We now show that for all two-level HLBs there is only one maximal packing set consisting of only single-block fragments (constraint (ii) of Theorem 7). Assume that the HLB template has only two blocks, a leaf block and a root block. Using the leaf block to implement a buffer will yield one buffered single-block fragment with *K* inputs. A delete-edge operation on the same input edge yields two single-block fragments, a *K*-input block and another block with *K-1* inputs.

Thus the delete-edge operation leads to a packing set that is a superset of the packing set generated by using a buffer operation. A similar argument can be applied to an HLB template with three blocks. To create a buffered single-block fragment, one or both of the leaf blocks must be used to implement a buffer. In either case, the packing set with the buffered fragment is a subset of the one created by only using delete-edge operations. The argument can be extended to HLB templates with more than three blocks. Thus we conclude that the maximal packing set of single-block fragments is generated by using only delete-edge operations on all edges and since there is only one way to delete all edges, this maximal packing set is unique.

For example, Figure 4-4(a) shows the L2-4 5-LUT HLB and Figure 4-4(b)-(d) are the maximal packing sets created by one, two and three edge deletions. Other maximal packing sets (with buffered patterns) are generated by using blocks **A**, **B** and/or **C** to implement buffers.

4.2.6 An HLB for which FFD Fragment Packing is Sub-optimal

This section presents an HLB topology and an instance of the fragment packing problem for which the FFD fragment packing algorithm is sub-optimal. The counter-example assumes that the HLB has 4-LUT basic blocks. Thus, the FFD fragment packing algorithm is not optimal in general for all HLBs.

The counter-example uses the L3-4.2 4-LUT HLB. This HLB is labelled as **ps1** in Figure 4-5 and consists of four 4-LUTs. Figure 4-5 shows the L3-4.2 HLB and the 7 maximal packing sets generated by delete-edge operations. The maximal packing sets are labelled **ps1**, **ps2**, .. , **ps8**.

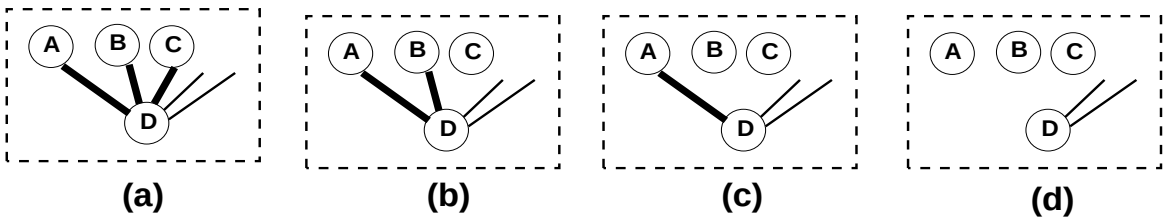


Figure 4-4: A two-level HLB (L2-4) and its maximal packing sets

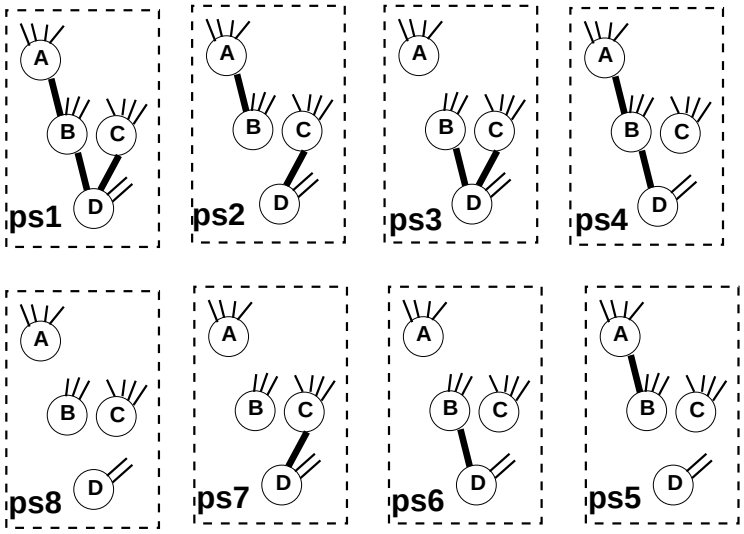


Figure 4-5: L3-4.2 HLB and its maximal packing sets

Figure 4-6 shows a set of fragments that are not packed optimally by the FFD fragment packing algorithm. This covering set of fragments consists of two multi-block fragments, two single-block fragments with fan-in of four and two single-block fragments with fan-in of three. The six fragments in Figure 4-6 are in descending order and are labelled **f1**, **f2**, .. , **f6**.

The result of using the FFD fragment packing algorithm is illustrated in Figure 4-7(a). FFD packs **f1** and **f2** into the same packed HLB since these two fragments can fit into maximal packing set **ps2**. Fragment **f1** is isomorphic to the fragment with blocks **C** and **D** in **ps2**. Fragment **f2** has one fewer primary input than the fragment with blocks **A** and **D** in **ps2**, but still fits the matching criteria. Fragments **f3**, **f4** and **f5** fit in the second packed HLB, but since fragment **f6** does not fit, the FFD algorithm requires another HLB, for a total of three packed HLBs. In contrast, the optimal packing in Figure 4-7(b) requires only two packed HLBs, in which

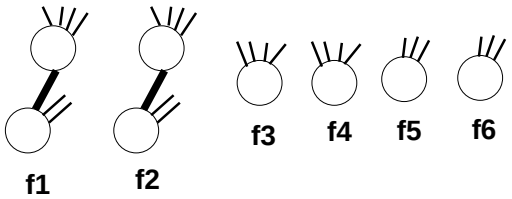


Figure 4-6: Covering Fragments that give sub-optimal packing for L3-4.2 HLB

4-7(a): Sub-optimal packing by FFD **4-7(b): Optimal Packing**

Figure 4-7: Sub-optimal packing and Optimal packing for L3-4.2 HLB example

each of the two packed HLBs contain three fragments that are isomorphic to the fragments in maximal packing set **ps7**. Since the FFD algorithm generates a solution with more packed HLBs than an optimal algorithm, it is sub-optimal for the L3-4.2 HLB.

4.3 Conclusion

This chapter has presented proof of the delay optimality of the fragment covering algorithm. The FFD fragment packing algorithm was shown to be optimal for all two-level HLBs. The following chapter evaluates the effectiveness of the HLB mapping algorithms with respect to theoretical bounds and a commercial mapping tool.

Chapter 5 Effectiveness of the HLB Mapping Algorithms

This chapter evaluates the effectiveness of the HLB technology mapping algorithms. The overall HLB synthesis methodology consists of technology-independent logic optimization, followed by mapping to LUTs and then the HLB fragment covering and packing algorithms detailed in Chapter 3. The first section of this chapter evaluates the effectiveness of the covering and packing algorithms using theoretical bounds. Section 5.2 contains an empirical study to compare the efficacy of our overall HLB synthesis methodology to a commercial technology mapper [11] for an FPGA with hard-wired connections, the Xilinx 4000 FPGA. The last section summarizes this chapter.

5.1 Comparison to Theoretical Bounds

This section compares the effectiveness of the HLB mapping algorithms with respect to the minimization of area. The measure of area is the total number of packed HLBs. We do not discuss the optimality of the algorithms with respect to delay-optimization because in Chapter 4 it was shown that the delay-optimization algorithm produces an HLB netlist with the minimal delay.

5.1.1 Performance of the Area-optimization Algorithm

In Chapter 4, the area-optimization HLB mapping algorithm was proven sub-optimal with respect to producing the minimal number of HLBs for a given input LUT network. The fragment covering stage of the algorithm does not necessarily generate the minimal number of HLB fragments. However, the packing stage of the algorithm produces a minimal number of packed

HLBs for certain HLB topologies and so the area-optimization algorithm may be reasonably effective.

The overall effectiveness of the area-optimization HLB mapping algorithm can be measured with respect to easily calculated lower bounds: Given a network of N basic blocks and an HLB composed of B basic blocks, a simple lower bound, LB, on the number of HLBs is $LB = \lceil N/B \rceil$. This lower bound may not be achievable whenever the basic block network precludes the selection of a set of fragments that can be packed with no wasted internal blocks in the packed HLBs. When there are no wasted basic blocks in the packed HLBs the packing is said to be a *perfect* packing.

An application of the lower bound is shown in Table 5-1. In this table, 15 MCNC benchmark circuits [49] are mapped to the L2-3 HLB with three 4-LUTs shown in Figure 3-1. Note that the L2-3 HLB is a two-level HLB and thus its fragments will be packed optimally. The first column

Circuit	# LUTs	Lower Bound	Actual HLBs
9symml	71	24	26
alu2	142	48	49
alu4	236	79	86
apex7	74	25	29
b9	45	15	15
c1355	91	31	33
c8	38	13	15
cc	25	9	9
cm162a	12	4	5
comp	36	12	13
count	39	13	16
decod	20	7	10
mux	17	6	6
vda	208	70	71
z4ml	6	2	3
Totals	1060	358	386

Table 5-1: Comparison with Lower Bound on Area of L2-3 HLB circuits

gives the circuit name, the second column is the number of 4-LUTs in the circuit, the third column has the predicted lower bound for the number of L2-3 HLBs and the last column gives the actual number of HLBs after using our technology mapper. These circuits are also used in Section 5.2 for the comparison with a commercial HLB mapper. The summary at the bottom of the table shows that the lower bound predicts a total of 358 HLBs, while the actual mappings totalled 386 packed HLBs. This 8% average difference is caused by the following: (1) the covering algorithm makes sub-optimal fragment choices which cannot pack well together and (2) the properties of the basic block network do not allow choosing a set of fragments that will pack perfectly together.

In general, as the number of LUTs (and hard-wired links) in the HLB increases, the size of the area-optimized circuits with respect to the lower bound increases. This effect is caused by the increase in the number of hard-wired links in the HLBs. More hard-wired links in the HLB reduce the connection flexibility, and this makes it more difficult to efficiently utilize the larger HLB.

Table 5-2 lists the most area-efficient HLBs consisting of a given number of 4-LUTs, and compares the number of actual HLBs and the number of HLBs predicted by the lower bound. Column 1 contains the name of the HLB, column 2 contains the predicted lower bound on the total number of HLBs in all circuits, column 3 lists the actual totals after mapping and the last column shows the percent difference between the actual number and the lower bound number.

Topology	Lower Bound totals	Actual HLBs totals	% difference
L2-2	533	547	3
L2-3	358	386	8
L3-4.2	270	285	6
L3-5.2.2	219	244	11
L3-6.3.2	183	207	13
L3-7.2.2	158	183	16
L3-8.2.2.2	139	157	13
L3-9.4.3	125	139	11

Table 5-2: Comparison with Lower Bound on Area of 4-LUT HLB circuits

The percent differences from the lower bound in Table 5-2 range from small (3%) to significant (16%). This shows that the area-optimizing HLB synthesis procedure is reasonably effective when mapping 4-LUT HLBs and may also be effective for other HLBs.

5.2 Effectiveness of Overall HLB Mapping Procedure

This section compares our overall HLB mapping procedure with a technology mapper for the Xilinx 4000 FPGA architecture called PPR [11]. We used version 1.21 of PPR for our experiments. The CAD systems are compared for delay and area in an empirical study. The benchmark circuits used in these evaluations are chosen randomly from the multi-level logic synthesis benchmark suite [49], with the constraint that every circuit could fit in the Xilinx 4005 FPGA when implemented by PPR. These same circuits were used to evaluate the Xilinx 4000 logic block architecture in a previous study [9] and some of the circuits are used in the empirical FPGA architecture study in Chapter 6.

The Xilinx 4000 Configurable Logic Block (X4000 CLB) is a commercial LUT-based FPGA architecture with hard-wired connections. The combinational logic portion of the X4000 CLB, which is illustrated in Figure 5-1, consists of two 4-LUTs whose outputs are hard-wired to the inputs of a 3-LUT. Note that the X4000 CLB only allows any two of the three LUT outputs to be accessed simultaneously. This is in contrast to the architectural assumption in Section 1.1 that states that *all* LUT outputs are accessible through tapping buffers.

Figure 5-1: The Xilinx 4000 CLB

The overall HLB synthesis procedure includes technology-independent logic optimization, technology mapping to LUTs and then the HLB covering and mapping steps. The common starting point for PPR and for our HLB synthesis methodology is a Boolean network that has undergone technology-independent logic optimization. PPR maps the delay- or area-optimized Boolean network directly to a netlist of X4000 CLBs. For the following comparison, PPR was set to use its default optimization parameters. The HLB synthesis methodology in this dissertation first maps the Boolean network to an area- or delay-optimized network of LUTs using Chortle [20] [21] and then uses the HLB covering and packing algorithms, implemented in a CAD tool called TEMPT, to produce the final HLB netlist. Note that TEMPT had to be modified to map to the X4000 CLB because it violated the assumption that there is a tapping buffer on every LUT basic block output and because its H LUT has 3 inputs while the other two LUTs each have 4 inputs.

Table 5-4 compares PPR and the Chortle+TEMPT combination for delay optimization. The first column lists the MCNC benchmark circuit name, the second column lists the number of programmable connections (PCs) in the critical path when that benchmark is mapped using PPR, the third column lists the PCs required when using Chortle+TEMPT, the fourth column lists the number of X4000 CLBs required for that benchmark when using PPR, and the last column lists the number of X4000 CLBs when using Chortle+TEMPT. Chortle+TEMPT used an average of 22% fewer programmable connections than PPR. This is a significant difference in the number of programmable connections. However, this may not translate into a significant reduction in routing delay after placement and routing because, when optimizing for delay, Chortle+TEMPT uses 89% more CLBs than PPR. A large part of the area overhead is due to the Chortle mapper, which produces an excessive number of LUTs when optimizing delay. A more area-efficient delay-optimizing LUT mapper, such as Flowmap [33], would greatly reduce the area overhead and thus the reduction in programmable connection delay would more likely translate into a similar reduction in routing delay.

Benchmark Circuit	Number of Prog. Conn.		Number of CLBs.	
	PPR s	Ch+TEMPT	PPR	Ch+TEMPT
9symml	8	5	36	41
alu2	15	12	71	145
alu4	20	14	123	283
apex7	5	5	36	51
b9	3	3	21	25
c1355	7	8	47	111
c8	5	3	18	17
cc	2	2	8	20
cm162a	3	3	5	8
comp	7	5	17	29
count	9	7	16	21
decod	2	1	10	16
mux	2	3	5	6
vda	9	5	98	193
z4ml	3	2	3	5
Totals	100	78	514	971

Table 5-3: Comparison of PPR and TEMPT for Delay-optimization

Table 5-4 compares PPR and the Chortle+TEMPT synthesis methodology for area-optimization. The first column lists the MCNC benchmark circuit name, the second column lists the number of X4000 CLBs required for that benchmark when using PPR, the third column lists the number of CLBs required when using Chortle+TEMPT, the fourth column lists the number of programmable connections (or PCs) along the critical path when using PPR, and the last column lists the number of critical path PCs when using Chortle+TEMPT. Overall, Chortle+TEMPT uses about 4% fewer X4000 CLBs to implement the same benchmark suite than PPR, so the two CAD systems are similar in effectiveness for area-optimization. When optimizing area, Chortle+TEMPT has 10% more PCs in the critical paths because the first priority is keeping the number of CLBs as low as possible.

The above tables have shown that Chortle+TEMPT are similar in effectiveness to PPR when optimizing area, but is significantly more effective when optimizing delay. However, it should be

Benchmark Circuit	Number of CLBs		Number of Prog. Conn.	
	PPR s	Ch+TEMPT	PPR	Ch+TEMPT
9symml	36	36	8	8
alu2	71	69	15	20
alu4	123	113	20	19
apex7	36	36	5	7
b9	21	20	3	5
c1355	47	38	7	7
c8	18	18	5	4
cc	8	11	2	3
cm162a	5	6	3	4
comp	17	15	7	6
count	16	16	9	8
decod	10	10	2	2
mux	5	8	2	4
vda	98	97	9	10
z4ml	3	3	3	3
Totals	514	496	100	110

Table 5-4: Comparison of PPR and TEMPT for Area-optimization

noted that TEMPT can be used to map to any HLB topology, while PPR is restricted to mapping to the X4000 CLB. The next chapter will demonstrate how TEMPT can be used to explore many different HLB topologies to find the HLB-based FPGA architectures with good speed and density.

5.3 Conclusion

This chapter showed that the HLB mapping algorithm performs reasonably well compared to theoretical lower bounds for area. Compared to a commercial CAD tool for a commercial HLB-based FPGA architecture, the synthesis procedure performed similarly in terms of area (4% fewer CLBs) and significantly better in terms of delay (22% fewer programmable connections in critical paths).

The next chapter will demonstrate the use of the HLB mapping algorithms in an empirical study of HLB-based FPGA architectures.

Chapter 6 An Empirical Study of HLB Architectures

This chapter describes the experiments used to evaluate a set of alternative hard-wired logic block (HLB) architectures. As discussed previously, hard-wired connections in logic blocks may lead to FPGA circuits that are faster and have reduced routing area. However, area-efficiency may be reduced since the fixed interconnections in HLBs make it difficult to utilize the logic efficiently. This chapter describes the empirical approach used to explore the relationships between the basic block functionality, the hard-wired connection topology of the HLB and the speed and density of the resulting HLB circuits.

The methodology is to implement several benchmark circuits in an FPGA using each HLB architecture and then measure the area and delay of the resulting circuits. These results are then compared to determine the best HLBs for speed and/or density.

The goals of this empirical study are:

i) Find the basic block and HLB topology that will lead to fast FPGA architectures with good area-efficiency.

ii) Determine the best basic block and topology for the most area-efficient HLB circuits.

This chapter is organized as follows. Section 6.1 describes the space of hard-wired logic blocks explored in the experiments. The experimental method, which includes the synthesis steps and the area and delay models, is given in Section 6.2. Section 6.3 contains the results of the HLB

architectural investigations, as well as some discussion regarding the effect of non-ideal HLB synthesis tools and the area and delay model parameters on the results. Section 6.4 summarizes the key empirical results. The final section discusses some limitations of the empirical study.

6.1 The Hard-wired Logic Block Design Space

An HLB is defined by its basic blocks and the topology used to connect the basic blocks with hard-wired links. The HLBs investigated are tree topologies and all the basic blocks in the HLB are identical. Each basic block has a tapping buffer to make its output accessible to the routing.

The results of previous studies [6] [8] have shown that for FPGA architectures without hard-wired connections, lookup table (LUT) basic blocks are a good choice from both a density and speed perspective. Thus, in this dissertation, we restrict our attention to LUTs as the basic block of the HLB. The HLBs are also constrained to have LUT basic blocks with the same number of inputs. This simplifies the mapping problem and restricts the size of the design space, although other research indicates that there may be reasons to use a heterogeneous mixture of LUT basic blocks [13] [46].

The HLBs investigated in this study are as follows:

i) All possible HLB topologies consisting of 2-LUTs (that is, 2-LUT HLB topologies) with 4 or fewer levels of LUTs.

ii) All possible 3-LUT HLB topologies with 3 or fewer levels.

iii) All possible 4-LUT HLB topologies with 3 or fewer levels and 9 or fewer LUTs.

iv) All possible 5-, 6- and 7-LUT HLB topologies with two levels and those with 3 levels and 6 or fewer LUTs.

The investigated HLBs were restricted to the above types because the run-times and memory requirements of the HLB technology mapper increase greatly with the number of hard-wired links in the HLB. In addition, the results of the HLB architecture studies presented later in this chapter show that there are diminishing improvements in speed as the size of the HLBs approach the limits of the above types. As the number of hard-wired links increase, the incremental speed improvements become more costly in terms of area because the reduced connection flexibility makes it more difficult to utilize the HLB efficiently.

In total, there were over 200 different HLBs investigated. Some of the 4-LUT HLB tree topologies investigated in this study are illustrated in Figure 6-1. The circles in Figure 6-1 represent LUT nodes in the HLB tree. The thick lines in Figure 6-1 represent a hard-wired connection edge between two LUTs, while the thin lines represent a primary input edge. The naming convention adopted for each HLB topology is described in Section 3.4.2.

Figure 6-1: Some 4-LUT HLB topologies

An important assumption is that each HLB features a *tapping buffer* on the output of each LUT basic block, which makes the output accessible to the routing. Figure 6-2 illustrates the tapping buffers on each LUT output. Tapping buffers offer two major advantages:

i) Tapping buffers lead to faster HLB circuits since the output of one LUT can be accessed directly instead of propagating it through another LUT.

ii) Tapping buffers improve logic density since unrelated pieces of logic can be packed together in the same HLB, with each piece using a separate tapped basic block output.

The disadvantage of tapping buffers is that they require significant area because they have large signal driving capability and require extra routing resources to access them.

The next section describes how the speed and density of FPGAs with these kind of hard-wired logic blocks can be explored empirically.

6.2 Empirical Method for Exploring HLBs

To evaluate the various HLB-based FPGA architectures, a set of 15 MCNC combinational benchmark circuits are implemented in each FPGA architecture (each with a different HLB) using available synthesis tools and a novel CAD tool based on the algorithms described in Chapter 3. Over 200 different HLB-based FPGAs were investigated, and thus, over 3000 different HLB circuits were constructed. The area and delay of each implemented circuit is then calculated, and the

Figure 6-2: HLB tapping buffers

results over all circuits are summarized for each HLB architecture. The summaries are used to produce measures of the goodness of each HLB-based FPGA architecture for the comparisons discussed in Section 6.3.

The following subsections describe the benchmark circuits used in this study, the synthesis steps used to implement each circuit, and the models used to measure the area and delay of the circuit.

6.2.1 Benchmark Circuits

The 15 benchmark circuits were chosen from the MCNC multi-level logic synthesis benchmark suite [49]. The selected benchmarks include seven of the circuits that were used to evaluate the Xilinx 4000 logic block architecture in a previous study [9] and to determine the effectiveness of the HLB mapping algorithms in Chapter 3. Table 6-1 contains the information about the benchmark circuits. The circuit's name is listed in column 1 and a brief description of the function of the benchmark circuit is in column 2 [49]. These limited functional descriptions are the only ones available in [49]. The third and fourth columns respectively contain the size of the circuit in terms of the number of 4-input LUTs and the maximum primary output depth, when the circuit is mapped to 4-LUTs using Chortle [21] in speed-optimizing mode. The benchmark circuits contain a mixture of random logic and arithmetic circuits and vary in size from 13 to 608 4-LUTs. The maximum output depth varies from 3 to 12 LUT levels. The final row summarizes the total number of 4-LUTs and the sum of the critical path depths over the entire benchmark suite.

6.2.2 Synthesis Steps

The input to the experimental procedure is a Boolean description of the benchmark circuit and the output is a place-and-routed netlist of HLBs that implement the circuit. The FPGA routing architecture illustrated in Figure 6-3 is assumed. In Figure 6-3, the FPGA on the right consists of the HLB tile on the left repeated in a 2-dimensional square array. Each HLB tile contains a hardwired logic block (labelled "L"), two connection boxes (labelled "C"), a switch box (labelled "S")

Benchmark	Description	Size (# of 4-LUTs)	Max. Output Depth
9symml	count ones	87	6
c1355	error correcting	426	7
c432	priority decoder	265	12
c499	error correcting	383	8
alu2	ALU	365	11
apex7	logic	128	5
cm150a	logic	13	4
cm151a	logic	13	3
cm162a	logic	25	3
cm163a	logic	22	3
count	counter	117	4
frg1	logic	58	5
k2	logic	608	7
mux	MUX	17	3
parity	parity	21	3
Totals		2548	84

Table 6-1: Benchmark Circuit Information

and channel segments between the connection and switch boxes. The connection boxes are used to connect HLB I/O pins to the channel segments. The switch boxes are used to connect vertical and horizontal channel segments. Each channel segment contains *W* routing tracks, where *W* is determined by the placement and routing step described below.

Figure 6-3: FPGA layout tile and the routing architecture

The mapping from MCNC benchmark circuit to the HLB-based FPGA consists of the following steps:

1. Perform technology-independent logic optimization on the MCNC benchmark circuit using mis2.2 [17] to produce an optimized Boolean network. The goal of this step is to minimize the technology-independent cost function that measures the area or delay of the Boolean description of the circuit.

2. Map the optimized Boolean description to an optimized network of LUTs using the Chortle [20] [21] LUT technology mapper. Chortle has two modes: (a) area-optimizing mode, in which it minimizes the number of LUTs and (2) speed-optimizing mode, in which it minimizes the depth of the LUT network.

3. Map the optimized LUT network to a netlist of hard-wired logic blocks using the algorithms outlined in Chapter 3. The algorithms are implemented in a program called the TEMPT HLB technology mapper [9]. When optimizing area, TEMPT minimizes the number of HLBs, N_{HLB}, in the circuit. When optimizing delay, TEMPT minimizes the number of programmable connections, N_R, in the critical path. Note that it is assumed that the FPGA will have *exactly* the number of HLBs specified by this mapping step and this N_{HLB} is used to calculate the area of the FPGA resources needed to implement the circuit. This assumption of having FPGA resources that "float" according to the circuit being implemented will be further discussed in Section 6.2.3.

4. Perform global placement and routing on the HLB netlist, using Altor [22] and PGAroute [8] respectively, to find *W*, the maximum channel width over all the channel segments of the FPGA. The placement and routing step minimizes the number of tracks in the routing channels between the logic blocks and attempts to make the resulting implementation as square as possible. This leads to a circuit that is both small and fast. Note that the FPGA has exactly *W* tracks when implementing this particular benchmark circuit, and *W* is used to find the size of the

FPGA routing resources needed to implement the circuit. This assumption of having the channel width "float" according to the circuit being implemented will be further discussed in Section 6.2.3.

Note that in the study concerned with HLB speed, the first three steps are set to minimize delay with area being a secondary concern. In the area-efficiency study, the first three steps are set to minimize area with delay being a lesser consideration.

6.2.3 Fixed vs. Free Variable Number of HLBs and Channel Width

In the experiments presented below, the area costs for implementing a set of benchmark circuits using a given HLB is calculated using two free variables, N_{HLB} and W. These free variables are set to whatever value is needed by the CAD tools to implement the circuit in the HLB-based FPGA. However, the use of free variables may not be relevant because in a "real" HLB-based FPGA, the amount of each resource (N_{HLB} and W) would be fixed at the point of fabrication. Thus an alternative to using free variables in the experiments would be to fix the value of N_{HLB} and W variables and then implement each circuit in the FPGA within these constraints. The only consideration with respect to area, when dealing with an FPGA with fixed resources, is whether or not the circuit will fit on the FPGA.

However, suppose one has several FPGA architectures with a fixed set of resources sufficient to accommodate any of the benchmark circuits. To compare the different fixed resource FPGAs on a level playing field, one would have to use some measure of the *utilization* of each FPGA's total resources when implementing each circuit. One can argue that this is what the free variable values of N_{HLB} and W do indeed measure. The values of the free variables indicate the minimum amount of resources needed to implement a circuit in an FPGA with a particular HLB architecture. Since the fixed resource FPGA can accommodate the circuit, these free variable values would be less than the amount of fixed resources in the FPGA and can be used to measure the uti-

lization of the total resources of the FPGA. Thus, the free variable numbers are suitable for comparing the area costs of the various HLB architectures.

6.2.4 Delay Model

The delay of a circuit of hard-wired logic blocks is taken to be the longest combinational path delay. The longest path delay, D_{tot}, is given by:

$$D_{tot} = N_{LB} * D_{LB} + N_R * D_R \quad (6.1)$$

where N_{LB} is the number of basic logic blocks along any of the longest paths, D_{LB} is the delay of each basic block, N_R is the number of programmable routing connections along a longest paths and D_R is a constant that represents the average delay per programmable routing connection. Note that each programmable connection counted by N_R may consist of one or more switching stages. The delay due to the hard-wired links is assumed to be zero. The first product in this expression is the combinational delay along the longest path. The D_{LB} delays for the various LUT sizes in a 1.2μm CMOS process are summarized in Table 6-2. These delays were determined from SPICE simulations [7]. The second product is the routing delay. The average delay of a programmable connection is difficult to determine since it depends upon several circuit and routing architecture properties, such as connection fanout, number of switching stages in the connection, switch delays and number of switches connected to each routing track. The simplest possible model for average delay, a constant parameter, was chosen. The D_R constant parameter will be varied to investigate its effect on the conclusions of the FPGA architecture experiments.

# LUT inputs	**D_{LB} Delay (*ns*)**
2	1.39
3	1.44
4	1.71
5	2.03
6	2.38
7	2.85

Table 6-2: Delays of Lookup Tables in 1.2μm CMOS process

Discussion about the Delay of a Hard-wired Connection

In the delay model it was assumed that a hard-wired link has zero delay. This sub-section discusses whether this assumption is reasonable.

A hard-wired connection consists of a simple metal wire, which by itself would have an insignificant delay compared to a basic block or programmable connection. However, the load of the tapping buffer on the LUT output driving the hard-wired connection may increase the delay through a hard-wired connection. Figure 6-4 shows the detailed view of an implementation of a hard-wired connection between two 4-input LUTs labelled **LUT1** and **LUT2** and the tapping buffer on the output of **LUT1** [25]. The details of one quarter of the SRAM cells and decoding tree of **LUT1** are shown in Figure 6-4. The hard-wired link between the LUTs is shown as a bold line. The output of **LUT1** has a small buffer since it must drive both the tapping buffer and the hard-wired input of **LUT2**. Input **D** of **LUT1** (and input **L** of **LUT2**) is the least loaded of the four inputs because it has the fewest pass transistors connected to it. Note that the hard-wired link

Figure 6-4: Detailed view of Hard-wired Connection and Tapping Buffer [25]

between the LUTs is connected to the least loaded input (input **L**) of **LUT2** to minimize the delay of the hard-wired connection [25]. The total load on the **LUT1 output buffer** is small and so the hard-wired link delay is insignificant compared to the delay of a LUT or a programmable connection. In a layout of an FPGA in a 1.2μm CMOS technology [25] [26] the delay due to a hard-wired connection was 0.01ns and the delay of a 4-LUT was 1.71ns. Thus the assumption of zero delay for hard-wired connections is reasonable.

Discussion about the Assumption of a Constant Value for D_R

The delay model assumes that the average delay of a programmable connection D_R is constant over all FPGAs, regardless of the HLB architecture or if the FPGA has hard-wired connections. This sub-section examines the validity of this assumption.

The routing architecture shown in Figure 6-3 was assumed because of the availability of CAD tools suitable for this architecture. In this architecture, the LUTs in each HLB are grouped closely together and the HLB I/O pins are spread evenly on the periphery of the group of LUTs. For example, Figure 6-5 shows an HLB that has four K-input LUTs connected with three hard-wired

Figure 6-5: Assumed Routing Architecture

links (bold lines). With this architecture, as the total number of LUTs in the HLB increases, the average number of pins that have to be connected to the channel segments on each side of the HLB also increases. For example, if each LUT has K inputs and one output, then the HLB in Figure 6-5 will have on average $\frac{(4(K+1)-3)}{4} \cong K$ pins on each side, and thus there is an average of about $2K$ pins per channel segment. A K-LUT FPGA without hardwired connections would have on average about $K/2$ pins per channel segment. Since there are more pins per channel segment for HLBs with several LUTS, there will tend to be a greater number of routing tracks per channel segment, W, relative to an FPGA without hard-wired connections. A higher W means that the parasitic capacitance connected to each programmable switch is greater and hence the delay per programmable connection is also greater.

A better routing architecture is one in which the routing tracks are between the LUTs of the HLBs and the hard-wired links between the LUTs span the routing channels in a manner similar to the direct connect in the Xilinx 3000 architecture [10]. For example, Figure 6-6 shows an HLB with four LUTs using the alternative routing architecture. This routing scheme leads to almost the

Figure 6-6: Better routing architecture

same number of pins to connect to each channel segment as in the FPGA has no hard-wired connections. Thus, assuming the number of basic blocks in the HLB circuit and input basic block circuit is the same, then it is likely that the circuits will have a similar number of routing tracks per channel segment, *W*. Thus if the assumption of similar numbers of basic blocks holds, the delay per programmable connection in the circuits of the HLB-based FPGA and the FPGA without hard-wired connections would be about the same.

For area-optimized circuits, it was shown in Section 5.1.1 that the mapping algorithm, when mapping to a variety of HLB architectures, produced a number of HLBs close (from 3 to 16% difference) to a lower bound based on the number of basic blocks in the input basic block circuit. Hence the number of basic blocks does not vary significantly among the HLB architectures for area-optimized circuits. Thus, we expect that with the better routing architecture, *W* will not vary significantly with the HLB architecture and thus the average programmable connection delay will be about the same independent of the HLB architecture.

However, it should be noted that when optimizing for delay, the HLB mapper increases the number of LUT basic blocks by significant amounts because it uses replication to improve the delay of *every* fan-out node. A large increase in the number of LUTs would lead to greater programmable connection delays. However, the use of an improved HLB mapper that only uses replication along the critical paths would alleviate this problem.

6.2.5 Area Model

The total area of a circuit is calculated as the product of the number of HLBs, N_{HLB}, times the area per HLB tile. Each HLB tile consists of two portions, a logic portion and a routing portion. The logic area per tile is the sum of three components. The first component is the area of the lookup tables themselves. The second part is the fixed area per LUT and accounts for the circuitry used to access the LUT outputs. D flip flops are needed to implement sequential circuits and so

they are assumed to exist in the FPGA. The third component is the area due to D flip flops. The expression for the logic area per tile, *LA*, is as follows:

$$LA = M * 2^K * LB + M * FA + D * DFFA \quad (6.2)$$

where *M* is the number of lookup tables in the HLB, *K* is the number of inputs per LUT, *LB* is the area per logic bit in the LUT, *FA* is the fixed area per LUT, *D* is the number of D flip flops per HLB and *DFFA* is the area of each D flip flop.

The area per logic bit, *LB*, consists of the area of a programming bit, the pass transistor from the decoding tree used to select the bit[1], and the buffer for each bit used to drive the decoding tree. Since the entire truth table of the function is in the LUT, there are 2^K bits. The fixed area per LUT, *FA*, consists of the area for the LUT output buffer needed to drive the next LUT input and the tapping buffer. Because of the relatively small size of a DFF compared to a LUT, we have found that varying *D* does not significantly affect the ranking of the sizes of the various HLBs. For the following studies, we assume $D = 1$ (only 1 D flip flop per HLB) in the area model.

In the studies, the area model parameter values were derived from the layout of an FPGA in a 1.2μm CMOS technology [25] [26]. The parameter values are $LB = 1400\ \mu m^2$, $FA = 1800\ \mu m^2$ and $DFFA = 3000\ \mu m^2$.

For the routing area model, it is assumed that the area of the programmable switches will dominate the routing area per tile (*RA*). Thus the expression for *RA* counts the number of routing bits and is as follows:

$$RA = (N_{IO} * F_c + 2 * F_s * W) * RB. \quad (6.3)$$

where N_{IO} is the number of input and output pins per HLB, F_c is the flexibility of the connection box, F_s is the flexibility of the switch box, *W* is the number of routing tracks in each channel seg-

1. Note that the decoding tree for a LUT with 2^K bits has 2^K-1 pass transistors. Thus, there is approximately one pass transistor per LUT bit.

ment after global routing and *RB* is the area of each programmable routing bit. The terms F_c and F_s are defined in Section 2.3.3. The $N_{IO} * F_c$ product is the number of programmable switches used to connect the I/O pins to the channel segments. The $2 * F_s * W$ term is the number of switches in the switch boxes used to connect channel segments to each other. The area of each bit in the programmable routing resources, *RB*, consists of the area of a programmable bit plus the area of a pass transistor controlled by the bit.

The results in [27] showed that an F_c which is close to W and $F_s = 3$ gives good routability without excessive routing resources, and this leads us to the following simplified equation for *RA*:

$$RA = (N_{IO} * W + 6 * W) * RB \tag{6.4}$$

The number of I/O pins, $N_{IO,}$ is equal to $M * K + 1$ and so the final expression for *RA* is:

$$RA = (M * K + 7) * W * RB \tag{6.5}$$

The 1.2μm CMOS layout in [25] [26] used static-RAM programmable routing bits and the derived value of *RB* is equal to 1000μm^2.

6.3 Experimental Results

There were two types of studies conducted, one that optimized the HLB-based FPGA circuits purely for speed and another that optimized purely for area. The goal of the speed study is to find the LUT basic block and connection topologies that lead to a high-performance HLB architecture, with reasonably good logic density. The goals of the area study are to find the LUT basic block and topologies that give the most area-efficient HLB-based FPGAs and to determine if HLBs can provide better density than logic blocks without hard-wired connections. Section 6.3.1 and Section 6.3.2 describe the speed and area studies with D_R and *RB* values that correspond roughly to an FPGA in a 1.2μm CMOS layout technology with static-RAM programming bits in the routing.

In the speed and area studies, the HLBs were chosen based on one optimization criterion (either speed or area), with the other criterion being the tie-breaker. However in both studies, our

CAD tools focus on optimizing either speed or area, with little regard to optimizing the other. The CAD tools' singular optimization goal may lead to inaccurate conclusions. Section 6.3.3 discusses how this limitation of the HLB synthesis procedure may affect the accuracy of the speed study of Section 6.3.1 and the area study of Section 6.3.2.

The conclusions of the speed and area study, with respect to the best LUT basic block, can also be affected by the parameters of the delay and area models. The parameters with the greatest impact are the average interconnection delay, D_R, and the routing programming bit size, *RB*. The effect of these parameters on the architecture conclusions will be discussed in Section 6.3.4 and Section 6.3.5.

6.3.1 Speed of HLB Architectures

As mentioned earlier, in the speed study, all the CAD tools were set to optimize speed with area being a secondary goal. In the context of this study, the *speed* of a circuit refers to the maximum operating frequency of the circuit. We assume that the combinational benchmark circuits will be placed between latches or flip flops, and thus the speed or maximum operating frequency is the fastest rate at which this sequential circuit can be clocked.

We define the "speed" for a particular HLB to be the arithmetic mean of the normalized speed for each benchmark circuit implemented in an FPGA with that HLB. The speed is normalized with respect to the speed of the same benchmark circuit implemented in an FPGA composed of 4-LUT basic blocks without hard-wired connections (an L1 4-LUT HLB-based FPGA). The normalized speed is given by $S = \frac{1/D_{HLB}}{1/D_{K4}}$, where D_{K4} is the delay along the longest path between primary inputs and outputs of the circuit in the L1 4-LUT FPGA and D_{HLB} is the delay of the circuit in the HLB-based FPGA. D_{K4} and D_{HLB} are derived from the delay model given in Section 6.2.4. In this section, we assume that $D_R = 4\ ns$ and $RB = 1000\ \mu m^2$. After speed-optimized imple-

mentation, the 4-LUT FPGA circuits have an average system speed of 39 *MHz*, corresponding to an average longest path delay of 2*6ns*.

The area metric for a particular HLB is the arithmetic mean of the normalized areas of the circuits found by using the area model presented in Section 6.2.5. The average area of the L1 4-LUT FPGA circuits is 19 x 10^6 μm^2.

Every HLB has a corresponding (area, speed) point that is the average normalized area and speed when the benchmark circuits are implemented in an FPGA with that hard-wired logic block. The speed versus area curve for a given set of K-LUT HLBs is constructed by connecting the points in the *envelope* set, which is the set of (area, speed) co-ordinates for the "best" K-LUT HLB architectures. An HLB's (area, speed) point belongs to the envelope set if and only if no other point in the entire set has both greater speed and lower area cost.

Figure 6-7 shows the speed versus area curve corresponding to the envelope point set for the delay-optimized 4-LUT HLBs and Table 6-3 lists the normalized values of speed and area for

Topology	Normalized Speed	σ(speed)	Normalized Area	σ(area)
L1	1.00	0	1.00	0
L2-2	1.14	0.10	1.19	0.16
L2-3	1.27	0.14	1.22	0.22
L3-4.2	1.28	0.14	1.33	0.34
L2-4	1.34	0.13	1.34	0.29
L3-5.2	1.37	0.13	1.42	0.47
L2-5	1.42	0.10	1.44	0.39
L3-8.4.3	1.44	0.19	1.69	0.67
L3-8.3.2	1.48	0.15	1.73	0.75
L3-9.4.3	1.49	0.21	1.74	0.70
L3-9.4.2	1.51	0.16	1.86	0.88

Table 6-3: Envelope Point Set for Speed-optimized 4-LUT HLB circuits

each point in the graph. The first column in Table 6-3 lists the name of HLB for that point, the sec-

ond column lists the average normalized speed of the benchmark circuits, the third column lists the standard deviation of the average normalized speed, the fourth column lists the average normalized area and the last column lists the standard deviation of the average normalized area.

The speed versus area curve for 4-LUT HLBs in Figure 6-7 illustrates the trade-off of speed for area, when optimizing purely for speed. In general, with more added hard-wired links in the HLB, the HLB speed increases while the area-efficiency of the HLB decreases because of the reduced connection flexibility. This effect is partly due to the mapping process. When the HLB mapper is optimizing purely for delay, it seeks to place as many hard-wired links as possible along critical paths by replicating LUT basic blocks as needed. This replication of LUTs leads to an increase in area because of the additional logic bits in the replicated LUTs and the additional programmable connections to connect the inputs of the replicated LUTs. For example, the L2-2 HLB is 14% faster than the L1 4-LUT HLB but at the cost of 19% more area.

Figure 6-7: Speed versus Area Curve for Speed-optimized 4-LUT HLB Circuits

Figure 6-8: Speed versus Area curves for Speed-optimized HLB Circuits

The standard deviations for normalized speed and area reflect the variation in the characteristics of the benchmark circuits. There tends to be a higher variation in area among the speed-optimized circuits in Table 6-3 because some circuits require more replication of LUT basic blocks to achieve the maximum speed.

Figure 6-8 illustrates the speed versus area curves for each size of basic block LUT, when optimizing purely for speed. Figure 6-8 is used in the next subsection to determine the best basic block for constructing fast HLB-based FPGAs.

Best LUT Basic Block for Speed

Figure 6-8 demonstrates that the fastest HLBs are those made of the coarsest-grained basic block, the 7-LUT. With 7-LUT basic blocks, the HLB speeds range from 132% to 177% of the

speed of the L1 4-LUT HLB. However, Figure 6-8 shows there is a relatively high area cost (from 23% to 204% higher than the L1 4-LUT HLB) associated with using this large-grained basic block. The high area costs arise because of two reasons: (1) it is difficult to utilize the logic functionality of large-grained 7-LUT HLBs efficiently and (2) because of the large number of pins associated with 7-LUT HLBs, there is a high routing area cost for connecting the many logic block pins. Although the 7-LUT basic block leads to the fastest HLBs, the high area costs make the 7-LUT undesirable for HLBs with reasonable density.

In contrast, Figure 6-8 shows that the slowest HLBs are made with the finest grained basic block, the 2-LUT. The speeds of 2-LUT HLBs range from only 75% to 101% of the speed of the L1 4-LUT HLB. However, the 2-LUT HLBs also have a lower area-efficiency than the coarse-grained LUT HLBs (from 40% to 144% more area than the L1 4-LUT HLB). The logic area-efficiency of fine-grained LUT HLBs tends to be higher than that of coarse-grained LUT HLBs because it is easier to efficiently utilize the smaller LUTs. However, the fine-grained LUT HLB circuits have many LUTs and thus many pins. The routing area cost of connecting the many pins is high and this leads to low overall area-efficiency [8]. The slow speeds and high area costs make the fine-grained 2-LUT basic block undesirable for fast HLBs with good density.

Figure 6-8 shows that the intermediate granularity of a 6-LUT leads to the best basic block for fast HLB circuits with reasonable density. Most of the 6-LUT speed versus area curve lies to the left and above all other LUT speed versus area curves. This shows that for almost any given speed, there exists a 6-LUT HLB that can implement the benchmark circuits using a smaller area than an HLB composed of any other LUT basic block. Conversely, for almost any given area, a 6-LUT HLB has the fastest speed. The location of the 6-LUT HLB curve demonstrates 6-LUT is the best LUT basic block for fast FPGA circuits with good density (although we note that some of the points on the 5-LUT and 7-LUT curves are within σ of the 6-LUT curve).

Table 6-4 contains the speed and area of the 6-LUT HLBs in Figure 6-8. The first column is the HLB topology, the second and third columns respectively contain the average normalized speed for speed-optimized circuits and standard deviation of the speed of the HLB. The fourth and fifth column lists the average normalized area of the HLB circuits relative to the L1 4-LUT HLB and the standard deviation of the area. This table shows that for speed optimized circuits, the single 6-LUT HLB is 25% faster than the L1 4-LUT HLB and has about the same area cost. The logic area costs of 6-LUT circuits are higher than for 4-LUT circuits. However, the 4-LUT circuits have many more LUTs in the circuits (2548 versus 1743) and this leads to higher routing area costs for 4-LUT circuits. Thus, this results in the 6-LUT and 4-LUT circuits having similar total areas for speed-optimized circuits.

The L2-2 HLB has an additional 6-LUT, which leads to a 6% increase in speed at the cost of an 11% increase in area over the L1 6-LUT HLB. The fastest 6-LUT HLB investigated is the L3-6.5 HLB, which is 72% faster than the L1 4-LUT HLB but requires 77% more area.

Topology	Normalized Speed	σ(speed)	Normalized Area	σ(area)
L1	1.25	0.18	1.02	0.21
L2-2	1.31	0.18	1.13	0.19
L2-3	1.45	0.20	1.30	0.38
L2-4	1.54	0.26	1.46	0.52
L3-5.2	1.56	0.25	1.50	0.47
L3-6.3	1.58	0.27	1.61	0.53
L2-5	1.67	0.28	1.62	0.62
L3-6.2	1.70	0.28	1.70	0.56
L3-6.5	1.72	0.30	1.77	0.78

Table 6-4: Envelope Point Set for Speed-optimized 6-LUT HLB circuits

Best HLB Topologies for Speed

The topologies of the 6-LUT HLBs on the speed versus area curve are listed in Table 6-4 and the fastest topologies for a given number of 6-LUTs (from one to six) are illustrated in Figure 6-9.

Figure 6-9 shows that the fastest HLB topologies for one, two, three, four, five and six 6-LUTs have balanced subtrees.

Since the circuits are optimized for speed, the LUT technology mapper tries to make all paths from inputs to outputs as even as possible. This tends to make portions of the LUT networks appear as balanced trees with high fan-in nodes, and this explains why the fastest HLB topologies contain balanced subtrees and nodes with high fan-in of hardwired connections. The topologies illustrated in Figure 6-9 are also reasonably good for area-optimized circuits. The area-efficiency of these topologies will be further discussed in Section 6.3.3.

An inspection of the envelope set for speed-optimized 4-LUT HLBs in Table 6-3 reveals that the fastest HLB topologies for one, two, three, four, and five 4-LUTs are identical to the first five topologies in Figure 6-9. This further substantiates the view that balanced topologies lead to the fastest HLBs. Note that the fastest HLB with six 4-LUTs is the L3-6.5 topology, but this HLB is not in the envelope set listed in Table 6-3.

6.3.2 Area-efficiency of HLB Architectures

The goal of the area study is to determine the LUT basic block and topologies that give the best HLB for maximizing area-efficiency. All CAD tools were set to optimize area with little

Figure 6-9: Fastest 6-LUT HLB topologies

Figure 6-10: Speed versus Area Curve for Area-optimized 4-LUT HLB Circuits

regard for speed. Even though the circuits are not optimized for speed, there will likely be hard-wired links present in the HLBs along the critical paths. These hard-wired connections will make the area-optimized HLB circuits faster than circuits built from the same LUT basic blocks without hard-wired connections.

Figure 6-10 shows the speed versus area curve for the best 4-LUT HLBs when optimizing purely for area. In this section, we assume that $D_R = 4\ ns$ and $RB = 1000\ \mu m^2$. The area-optimized L1 4-LUT HLB circuits have an average maximum operating frequency or speed of 26 *MHz* (delay of 39*ns*) and an average area of 8 x $10^6\ \mu m^2$. The point at co-ordinate (1.00, 1.00) is labelled “L1” in Figure 6-10 and corresponds to the L1 4-LUT HLB.

Table 6-5 lists the average normalized speed and areas of the 4-LUT HLBs in Figure 6-10. The first column contains the topology name, the second column contains the average normalized area for area-optimized circuits and the third column lists the standard deviation of the area. The fourth and fifth columns list the average speed and standard deviation of the speed respectively.

Topology	Normalized Area	σ(area)	Normalized Speed	σ(speed)
L2-2	0.94	0.19	1.18	0.13
L3-4.2	0.95	0.17	1.28	0.15
L3-6.3.2	1.04	0.11	1.32	0.17
L3-7.4.2	1.20	0.21	1.33	0.17

Table 6-5: Envelope Point Set for Area-optimized 4-LUT HLB circuits

Table 6-5 demonstrates that FPGAs with HLBs can be more area-efficient than those without hard-wired connections. For example, the L2-2 HLB, which has two 4-LUTs hard-wired together uses 6% less area than the L1 4-LUT HLB. In the case of the L2-2 HLB, the increase in area due to loss of flexibility caused by hardwired connections is more than offset by the decrease in area due to the reduction in the number programmable connections. Note that the presence of hard-wired connections in the critical path of the L2-2 HLB circuits resulted in an 18% speedup with respect to the L1 4-LUT HLB circuits.

The standard deviations for the average normalized area in Table 6-5 for area-optimized circuits are much smaller than those for speed-optimized circuits in Table 6-3. In area-optimized circuits the standard deviations for normalized area are smaller because there is no replication during area-optimization. In speed-optimization, the amount of replication depends upon the circuit properties, and so there is more variation than in the standard deviation of the area-optimized circuits.

Figure 6-11 shows the speed versus area curves for each of the LUT basic blocks when optimizing area. Figure 6-11 is used to determine the best LUT basic block for making area-efficient HLB-based FPGAs.

Best LUT Basic Block for Density

Figure 6-11 shows that the 2-LUT and 7-LUT HLBs require significantly more area than the L1 4-LUT HLB to implement circuits. The fine-grained 2-LUT basic block has good logic area-efficiency, but because 2-LUT circuits have many logic blocks and associated pins to connect, the

Figure 6-11: Speed versus Area curves for Area-optimized HLB Circuits

cost of routing is high [8]. The coarse-grained 7-LUT basic block, because of their large functionality, have the worst logic area-efficiency among the LUTs investigated. There are also many logic block pins in 7-LUT circuits and this leads to a high routing area cost to connect the pins. Thus neither the finest-grained or coarsest-grained LUTs are suitable for high density HLB circuits.

Figure 6-11 demonstrates that the 4-LUT basic block leads to the HLBs with greatest area-efficiency. The HLB with the best density has the L2-2 topology. The L2-2 4-LUT HLB uses 6% less area and is 18% faster than the L1 4-LUT HLB. The HLB with the second lowest area-costs is the L3-4.2 4-LUT HLB, which uses 5% less area than the L1 4-LUT HLB. Note that the L3-4.2 4-LUT HLB has similar area-efficiency to the L2-2 HLB but is significantly faster (28% faster than the L1 4-LUT HLB).

However, Figure 6-11 shows that the 5-LUT may provide a better basic block for making area-efficient FPGAs because, while using only slightly more area than the densest 4-LUT HLBs,

Figure 6-12: Densest 5-LUT HLB topologies

the 5-LUT HLBs result in much faster circuits. For example, the L2-3 5-LUT HLB uses 3% less area than the L1 4-LUT HLB but is 47% faster. By using slightly more area (3% more) than the most area-efficient 4-LUT HLB, the most area-efficient 5-LUT HLB realizes an additional 29% speedup. The 5-LUT speed versus area curve lies almost wholly to the left and above all other LUT speed versus area curves. This shows that for area-efficient FPGA circuits, a 5-LUT HLB will be the best for almost any given speed or area.

Best HLB Topologies for Density

The topologies of the 5-LUT HLBs on the speed versus area curve are summarized in Table 6-6 and the 5-LUT HLB topologies with the best area-efficiency are illustrated in Figure 6-12. The first column in Table 6-6 contains the HLB name, the second column lists the average normalized area for area-optimized circuits in that HLB and the standard deviation for the area. The fourth and fifth columns contain the average speed of the HLB circuits and the standard deviation of the speed. In general, the most area-efficient HLBs have topologies with long chains and the HLB has all LUTs with two or more non-hard-wired inputs.

In the area-optimizing mode of the fragment covering algorithm, the algorithm only uses matches to subject *trees*. Since the LUT networks tend to have small subject trees, the covering set of fragments contains a high ratio of single-block fragments. Thus, for good density every HLB should be able to accommodate many single-block fragments.

In order to pack a single-block fragment, the LUT must have at least two non-hard-wired inputs because LUTs with zero or one non-hard-wired inputs cannot be used for packing any single-LUT fragment. Any LUTs with zero or one non-hard-wired inputs are wasted when packing single-block fragments. For the densest 5-LUT HLB topologies shown in Figure 6-12, all LUTs have two or more non-hard-wired inputs.

Topology	Normalized Area	σ(area)	Normalized Speed	σ(speed)
L2-3	0.97	0.17	1.47	0.45
L3-5.2.2	1.03	0.18	1.52	0.44
L3-6.2.2	1.09	0.27	1.54	0.42
L3-6.3.2	1.15	0.28	1.56	0.53

Table 6-6: Envelope Point Set for Area-optimized 5-LUT HLB circuits

The HLBs whose LUTs have two or more non-hard-wired inputs have topologies in which all internal LUTs have sparsely populated fan-ins of hard-wired connections. This is in contrast to the fastest HLB topologies, which have a few internal LUTs with fully populated fan-ins of hard-wired connections. These HLBs with fully populated fan-ins have lower densities compared to HLBs with sparsely populated fan-ins with the same number of LUTs because some of the LUT basic blocks cannot be used to pack single-block fragments.

HLBs with the Highest Density

Table 6-7 lists all HLBs with better logic density (that is, normalized area less than 1) than the L1 4-LUT HLB. The first column lists the number of inputs for the LUT basic block, the second column gives the topology, the third column lists the average normalized area of area-optimized circuits, the fourth column contains the standard deviation of the area, the fifth column contains the average normalized speed and the sixth column contains the standard deviation of the area.

The 4-LUT basic block is the most area-efficient logic block for FPGAs without hard-wired connections [8]. In general, the most area-efficient HLBs have a small number of basic blocks (less than five) because as the number of basic blocks increase in an HLB, the more difficult it is

to utilize the functionality of the larger HLB efficiently. Also, all of the HLBs in Table 6-7 have no LUTs with one or zero non-hard-wired inputs. Note that all the HLBs made of LUTs with 4 or more inputs are faster than the 4-LUT basic block. The fastest HLB with better logic density than the 4-LUT, the 5-LUT L2-3 HLB, is also 47% faster.

# LUT inputs	Topology	Normalized Area	σ(area)	Normalized Speed	σ(speed)
3	L2-2	0.98	0.12	0.90	0.20
4	L2-2	0.94	0.19	1.18	0.13
4	L2-3	0.96	0.08	1.18	0.13
4	L3-4.2	0.95	0.17	1.28	0.15
5	L2-2	0.97	0.15	1.45	0.42
5	L2-3	0.97	0.17	1.47	0.45

Table 6-7: HLBs with logic density better than the 4-LUT

6.3.3 Limitations of the HLB Synthesis Procedure

In the FPGA architecture studies of Section 6.3.1 and Section 6.3.2, the set of envelope points was used to determine the best basic block and HLB topologies in terms of speed and density. Recall that the set of envelope points corresponds to the HLBs with the best (area, speed) points. The method for gathering the envelope point data is important for establishing confidence in the architecture study results.

In the speed study, the goal was to find the HLBs that result in the fastest circuits with reasonable density, and thus when optimizing the speed of a circuit, the area was the tie-breaker. In the area-efficiency study, the goal was to find the HLBs that result in the densest circuits with reasonable speed. Thus when minimizing the area of a circuit, the speed was the tie-breaker. An ideal HLB mapping procedure would maximize the speed of a circuit and from among the circuits with maximum speed choose the one with minimal area, or it would minimize the area of a circuit and choose the fastest of the smallest circuits. The use of this ideal HLB mapping procedure would yield the set of envelope points that perfectly describes the trade-offs between speed and area when the optimization goal is either maximum speed or minimum area.

However, the technology mapping tools used in our FPGA architecture studies are not ideal and this may lead to an erroneous set of envelope points. There are two sources of inaccuracy in the HLB mapping procedure: the LUT basic block technology mapper (Chortle [20]) and the HLB technology mapper described in this dissertation (TEMPT).

Accuracy of the Speed Study

When minimizing delay for the speed study, the Chortle mapper, when compared to the optimal depth Flowmap LUT mapper [33], yields close to the optimal delay for 5-LUT circuits. However, Chortle uses an average of about 50% more LUTs than Flowmap because Chortle does not attempt to conserve area when optimizing speed. The TEMPT HLB mapper yields the optimal delay HLB circuits but uses more than minimal area because it does not conserve area when mapping the non-critical parts of the circuit. Thus, during speed optimization, this combination of technology mappers would yield circuits with close to optimal speeds but with areas that are significantly greater than optimal. This results in the envelope points being shifted to the right relative to their locations when using an ideal mapping procedure. For example, Figure 6-13 shows the speed versus area curve that would be produced by an ideal mapping procedure as a solid line, and the non-ideal speed versus area curve that would be produced by our mapping procedure as a dashed line. The non-ideal speed versus area curve is to the right of the ideal curve because the areas of the points on the non-ideal curve are larger but the speeds are the same.

If the areas of the envelope points in the speed study are greater by similar proportions then the general shape of the speed versus area curve would remain the same and the conclusions of the speed study would not be changed. However, if the mapping procedure increases area costs by widely varying amounts for different HLBs, then the (area, speed) points of the HLBs may be shifted so that some points that should not be in the envelope set now falsely appear there. In order to support the conclusions of the speed study, we shall demonstrate that our mapping procedure does not significantly alter the HLBs on the speed versus area curve due to widely varying effects on different HLBs.

Figure 6-13: Speed versus Area curve shift due to non-ideal speed-optimization

One way to show that the mapping procedure is reasonably consistent across the various HLBs is to check if the area results of the speed-optimization study shows some consistency with the area results of the area-optimization study. The set of envelope points for best 6-LUT HLBs from the speed study are shown in Table 6-8. The first column lists the HLB topology. The second

Topology	Speed-optimized Circuits		Area-optimized Circuits	
	Normalized Speed	Normalized Area	Normalized Speed	Normalized Area
L1	1.25	1.02	1.11	1.17
L2-2	1.31	1.13	1.22	1.17
L2-3	1.45	1.30	1.26	1.20
L2-4	1.54	1.46	1.24	1.29
L3-5.2	1.56	1.50	1.31	1.34
L3-6.3	1.58	1.61	1.32	1.40
L2-5	1.67	1.62	1.27	1.40
L3-6.2	1.70	1.70	1.34	1.42
L3-6.5	1.72	1.77	1.31	1.46

Table 6-8: Best 6-LUT HLBs for Speed-optimized circuits

and third columns list the average normalized speed and area of the HLB circuits relative to the speed-optimized circuits, when using the L1 4-LUT speed-optimized circuit speed and area values as a basis for normalization. The fourth and fifth columns list the average normalized speed and area relative to the area-optimized circuits, when using the L1 4-LUT area-optimized circuit speed and area values as a basis for normalization. The rank of the HLB circuit areas for speed-optimized circuits (column 3) and area-optimized circuits (column 5) is consistent. This shows that the relative areas of the various HLBs are the same between the speed-optimized and area-optimized results. This supports the notion that our non-ideal speed-optimization mapping procedure is relatively unbiased by the HLB topology and the area increase due to our non-ideal procedure is reasonably consistent across the HLBs on the speed versus area curve. Thus, the shapes of the speed versus area curves and the results of the speed study derived from the curves are likely to be valid.

Accuracy of the Area-efficiency Study

When optimizing area for the area study, the Chortle mapper is among the best LUT technology mapper for minimizing the number of LUTs in 5-LUT circuits [20]. However, Chortle does not attempt to maximize speed when performing area optimization. When chaining together the bin-packed LUTs, Chortle first sorts the bin-packed LUTs in descending order of used inputs, and then greedily utilizes the unused inputs in the chaining. When the unused inputs are exhausted new LUTs are created for the chaining. If all the bin-packed LUTs have only one or two unused input before chaining, this can lead to a LUT network with a long critical path.

For example, Figure 6-14 shows the chaining of three bin-packed LUTs with 5 used inputs. When mapping to a 5-LUT circuit, because there are no unused inputs available, a new LUT has to be created to chain the three bin-packed LUTs together as in Figure 6-14(a). When mapping to a 6-LUT circuit, the LUTs can be chained together using the unused input of two of the bin-packed LUTs as in Figure 6-14(b). The cascaded arrangement in Figure 6-14(b) has a longer critical path.

The above effect may explain why the speeds of the 6-LUT and 7-LUT HLBs are significantly slower than the 5-LUT HLBs in Figure 6-11. As the number of LUT inputs increase, the probability of unused inputs and a cascaded arrangement along the critical path also increases. Thus the 6-LUT and 7-LUT networks have more LUTs along the critical paths than the 5-LUT networks and the resulting HLB networks have more programmable connections on the critical paths.

In Section 5.1.1 the TEMPT HLB mapper in area-optimization mode was shown to give area-efficient implementations. However, during area-optimization, TEMPT does not optimize across fan-out and since the fan-out free trees tend to be small, TEMPT does not effectively use larger HLBs to speed up the circuits. An examination of the normalized speeds of area-optimized circuits in Table 6-8 (column 4) shows that the speed of all the HLBs with 5 or 6 LUTs are close to each other (ranges from 1.27 to 1.34).

When using the combination of Chortle and TEMPT to implement area-optimized HLB circuits, the resulting circuits have close to minimal area but are significantly slower than the maximum speed circuits with minimal area. The speed versus area curve generated from this non-ideal mapping procedure would appear below the curve generated by an ideal mapping procedure because the speeds of the points on the non-ideal curve are lower, but the areas are about the same.

6-14(a): Chaining for 5-LUT circuit **6-14(b): Chaining for 6-LUT circuit**

Figure 6-14: Chaining together three LUTs with 5 used inputs

Table 6-9 lists the set of envelope points corresponding to area-optimized 5-LUT HLB circuits. The first column contains the HLB topology. The second and third columns contain the average normalized speed and area of the HLB circuits relative to the area-optimized 4-LUT circuits. The fourth and fifth columns contain the average normalized speed and area relative to the speed-optimized 4-LUT circuits.

Topology	Area-optimized Circuits		Speed-optimized Circuits	
	Normalized Speed	Normalized Area	Normalized Speed	Normalized Area
L2-3	1.47	0.97	1.38	1.33
L3-5.2.2	1.52	1.03	1.39	1.63
L3-6.2.2	1.54	1.09	1.44	1.59
L3-6.3.2	1.56	1.15	1.42	1.59

Table 6-9: Best 5-LUT HLBs for Area-optimized circuits

The low variation in the speed of the HLBs in Table 6-9 shows that the large-grained HLBs with five or six 5-LUTs are not used effectively to speed up circuits during area-optimization. Since speed is the tie-breaker in the area-efficiency study the conclusions of the area-efficiency study with respect to large-grained HLBs are questionable. There may be other large-grained HLBs with greater speeds and higher areas that belong to the envelope set. Thus the rightmost portions of the area-optimization study envelope points may be incorrect.

6.3.4 The Effect of Changing the Average Routing Delay, D_R

This subsection examines the effect on the conclusions of the speed and area studies when the average programmable connection delay, D_R, is varied. The speed and area studies in Section 6.3.1 and Section 6.3.2 assumed static-RAM controlled routing bits (RB = 1000 μm^2) and an average programmable connection delay, $D_R = 4ns$. The speed study led to the conclusion that the 6-LUT is the best basic block for fast circuits with reasonable density. The area study led to the conclusion that the 5-LUT is the best basic block for high area-efficiency. However, the best LUT basic block may change with variations in D_R.

Effect of D_R on Speed Study

At lower values of D_R, the delay of the programmable connections in critical paths have less effect on the critical path delay. This also means that the delay of the LUTs in critical paths have a greater impact on the critical path delay. Table 6-10 shows the total combinational delay along the critical paths of the speed-optimized LUT circuits. The first column contains the LUT basic block size, the second column contains the sum of the number of LUTs along the critical path over all circuits, TN_{LB}, the third column contains the delay of each LUT, D_{LB}, and the last column has the total LUT delay over all circuits, $TN_{LB}*D_{LB}$. If $D_R = 0$, that is, total routing delay is zero, then this table predicts that 5-LUTs lead to the fastest circuits because they have the lowest combinational

# LUT inputs	TN_{LB}	D_{LB} (*ns*)	$TN_{LB}*D_{LB}$ (*ns*)
2	186	1.39	258.5
3	115	1.44	165.6
4	84	1.71	143.6
5	68	2.03	138.0
6	60	2.38	142.8
7	53	2.85	151.0

Table 6-10: Total LUT delays for Speed-optimized LUT circuits

delay. The 6-LUT, 4-LUT and 7-LUT circuits are almost as fast as the 5-LUT circuits. The fine-grained 2-LUT and 3-LUT basic blocks lead to significantly slower circuits.

Figure 6-15 shows the speed versus area curves when D_R has a value of 1*ns*. With this small value of D_R, the HLB speedups relative to the 4-LUT are now smaller because of the lower impact of routing delay. The maximum speedups due to hard-wired links is only 29% versus the 77% speedups when D_R was 4*ns*. The increase in area costs are similar to when D_R was 4*ns* (up to 150%), and so hard-wired links are not as attractive at the lower D_R of 1*ns*.

At D_R = 1*ns*, the 6-LUT is still the best basic block choice for fast circuits because its speed versus area curve is almost entirely above and to the left of the other curves. However, at this low value of D_R, since the combinational delay of the 5-LUT circuits is almost the same as the 6-LUT,

the 5-LUT curve is very close to the 6-LUT curve. Thus, for small D_R, the 5-LUT is an equivalently good basic block for fast circuits.

When D_R is increased, the delay of the programmable connections in the critical path have a greater impact on the critical path delay and so the HLB speedups have become larger. Figure 6-16 shows the speed versus area curves when D_R has a value of 10*ns*. The maximum speedups due to hard-wired links has increased to 119% versus the 77% maximum speedup when D_R = 4*ns*. The increase in area costs are similar to when D_R was 4*ns*, and so hard-wired links are more attractive when D_R is 10*ns*.

The increase in D_R to 10*ns* makes the coarse-grained 7-LUT HLBs slightly more attractive. For this value of D_R, the 7-LUT curve crosses the 6-LUT curve at one point (corresponding to the L2-3 7-LUT HLB), whereas for D_R = 4*ns*, the 6-LUT curve never intersected the 7-LUT curve. At this single point in the speed versus area design space, the 7-LUT provides a better HLB than the

Figure 6-15: Speed versus Area curves for Speed-optimized HLB circuits, D_R = 1*ns*

Figure 6-16: Speed versus Area curves for Speed-optimized HLB circuits, D_R = 10*ns*

6-LUT HLB point just below it. However, for most speed and area combinations, the 6-LUT is a superior basic block to the 7-LUT.

Effect of D_R on Area study

The value of D_R does not affect the area model and thus the area study results are the same. All HLBs have the same ranking in terms of area-efficiency. The 5-LUT is still the best basic block for area-efficiency. Increasing D_R gives the 5-LUT a larger speed advantage over the smaller grained LUTs, while decreasing D_R reduces the speed advantage.

The overall conclusion is that the value of D_R has little effect on the choice of best LUT for speed or area-optimized circuits.

6.3.5 The Effect of Changing the Routing Bit Area, *RB*

This subsection examines the effect on the conclusions of the speed and area studies when the programmable routing bit size, *RB*, is varied. The speed and area studies in Section 6.3.1 and Section 6.3.2 assumed static-RAM controlled routing bits (RB = 1000μm^2) and D_R = 4*ns*, and concluded that the 6-LUT is the best basic block for fast circuits with reasonable density and that the 5-LUT is the best basic block for high area-efficiency. The static-RAM routing bit is one of the largest programming technologies for FPGAs. This subsection will examine the effect of using a much smaller routing bit technology, which has a routing bit size, *RB*, equal to 250μm^2.

Effect of *RB* on Area Costs in the Speed Study

For smaller routing bit size, the routing area has less impact on the total area, and the logic area has an increased impact. Table 6-11 shows the average normalized area for the logic-only

# LUT inputs	$\overline{LOA}$
2	0.97
3	0.87
4	1.00
5	1.44
6	2.40
7	3.94

Table 6-11*:* Average Logic Area for Area-optimized HLB circuits

part of each area-optimized HLB circuit, $\overline{LOA}$. The values of $\overline{LOA}$ are derived by assuming a routing area cost of 0, that is RB = 0, in the area model. $\overline{LOA}$ is the average of the logic-only areas of each circuit implemented in K-LUTs normalized with respect to the logic-only areas of the same circuit implemented in 4-LUTs. Column 1 contains the number of inputs for the LUT basic block and column 2 contains the average logic area if the circuits were implemented in the given LUT. The 3-LUT circuits have the lowest logic area cost and in general, the finer-grained LUTs, with 2 to 4 inputs, have significantly lower logic area costs than the LUTs with 5 or more inputs.

Figure 6-17: Speed versus Area curves for Speed-optimized HLB circuits, *RB* = 250 μm^2

Figure 6-17 shows the speed versus area curves for *RB* = 250μm^2. Since all areas are normalized to the 4-LUT, the 4-LUT curve is in about the same position as when *RB* = 1000μm^2. The smaller-grained 2- and 3-LUTs have higher logic area-efficiency than the 4-LUT and so their speed versus area curves for *RB* = 250μm^2 are shifted to the left relative to their locations for *RB* = 1000μm^2. For example, when *RB* = 1000μm^2, the L1 3-LUT HLB at the bottom of the 3-LUT curve required 13% more area than the L1 4-LUT HLB, but when *RB* = 250μm^2, the L1 3-LUT HLB required only 3% more area. The larger-grained 5-, 6- and 7-LUTs have lower logic area-efficiency than 4-LUTs and so their curves are shifted to the right. For example, when *RB* = 1000μm^2, the L1 7-LUT HLB at the bottom of the 7-LUT curve required only 23% more area than the L1 4-LUT HLB, but when *RB* = 250μm^2, the L1 7-LUT HLB required 98% more area.

The relative area cost of the 6-LUT is much higher than when *RB* = 1000μm^2 (a maximum area increase of 125% versus 60%) and so the 6-LUT is no longer the best LUT basic block for

Figure 6-18: Speed versus Area curves for Area-optimized HLB circuits, *RB* = 250 μm^2

high speed circuits. The 4-LUT and 5-LUT HLBs have become relatively more area-efficient compared to the 6-LUT HLBs, so that for most speeds the HLBs on the 4-LUT and 5-LUT curves are the best for high speed circuits with good density. Thus for small programming technologies, these finer-grained LUTs are the best choices for constructing fast HLB circuits.

Effect of *RB* on Area Costs in the Area study

For small routing bit sizes, the routing area is less of a factor, and thus the LUTs that lead to the lowest logic area costs have a greater influence on the overall area-efficiency. According to Table 6-11, the 3-LUT has the best logic area-efficiency of any LUT. When $RB = 1000\mu m^2$, the most area-efficient HLB had 4-LUTs. However, when the cost of routing is lowered significantly, $RB = 250\mu m^2$, Figure 6-18 shows that a 3-LUT HLB has the best area-efficiency amongst all HLBs. The L2-2 3-LUT HLB has 8% less area than the 4-LUT but is 10% slower. Another 3-LUT

HLB with better area-efficiency than the 4-LUT is the L3-4.2 HLB (uses 2% less area than the 4-LUT and is 5% faster).

However, the fastest HLBs that have better density than a 4-LUT, belong to the set of 4-LUT HLBs. The L2-2 4-LUT HLB uses 3% less area than the 4-LUT and is 18% faster. The L3-4.2 4-LUT HLB uses about the same area as the 4-LUT but is 28% faster.

The 5-LUT curve is shifted to the right because of the relatively high logic area cost of the 5-LUT with respect to the 4-LUT. There are now no 5-LUT HLBs with better logic density than the 4-LUT L1 HLB. However, there are 5-LUT HLBs with high speeds for moderate area costs. For example, with a 16% increase in area with respect to the 4-LUT L1 HLB, the L2-3 5-LUT HLB yields a 47% increase in speed.

The overall conclusion is that *RB* affects the choice of best LUT for fast and dense circuits. A lower value of *RB* makes smaller LUTs relatively more area-efficient with respect to large LUTs and thus changes the choice of best LUT for speed or density. This section showed that a significant decrease in the value of *RB* from 1000 to 250μm^2 makes 4-LUTs and 5-LUTs a better basic block choice than 6-LUTs for use in HLB-based FPGAs aimed at making fast circuits with good density. This lower value of *RB* also makes 3-LUTs better than 4-LUTs for making HLB-based FPGAs that lead to the densest circuits.

6.4 Summary of Results

The empirical studies of this chapter have demonstrated that hard-wired links can be used effectively in FPGA logic blocks, to not only improve FPGA speed, but also to increase density.

For area and delay models corresponding to a static RAM programming technology in 1.2μm CMOS layout technology (*RB* = 1000μm^2) and assuming D_R = 4*ns*, the best LUT basic block for high speed HLB architectures with reasonable logic density is the 6-input lookup table. Under the same assumptions, the 5-input lookup table was the best basic block in terms of logic density.

For high speed HLB-based FPGA circuits, the best HLB topologies contained balanced and fully populated trees. For dense circuits, the most area-efficient HLB topologies employ LUTs with two or more non-hard-wired inputs.

The average programmable connection delay, D_R, did not affect the choice of best LUT basic block when D_R was varied from 1 to 10*ns*, with $RB = 1000\mu m^2$. The 6-LUT is the best basic block over this range of D_R.

The average programmable routing bit size, *RB*, affects the choice of best LUT for fast and dense HLB-based FPGA circuits. Small values of *RB* makes finer-grained LUTs more attractive because of their higher logic area-efficiencies. For speed-optimized HLB circuits and a small *RB*, the 4- and 5-LUTs become better choices for basic block than the 6-LUT because of the improved relative area-efficiency of 4- and 5-LUT HLBs. The 3-LUT has the highest logic area-efficiency among all LUTs. Thus, for area-optimized circuits and a small *RB*, the 3-LUT basic block is the best choice for making area-efficient HLB-based FPGAs.

6.5 Limitations of the Empirical Study

This section contains more discussion on how the CAD tools and the assumed routing architecture can affect the results and conclusions of the experimental study.

6.5.1 Effect of HLB synthesis tools

The current procedure for mapping an input Boolean network to a netlist of HLBs proceeds in two phases. First the Boolean network is mapped to basic blocks and then the basic block network is mapped to a netlist of HLBs. The basic block technology mapper optimizes the speed or area of the basic block network and does not optimize the network for particular HLB topologies. The resulting basic block networks may be more favourable for some HLB topologies. A technology mapper that maps the input Boolean network directly to a netlist of HLBs may be less biased and this may alter the results of the HLB architecture studies.

6.5.2 Effect of Routing Architecture Assumption

To derive our area model, we assumed a routing architecture in which all the routing tracks were between entire HLBs, that is, all the pins of the HLB are evenly distributed on its four sides. For HLBs with several LUTs and many pins this may lead to a large maximum channel width, *W,* because there are many pins to be connected in each channel segment. Thus the HLBs with several LUTs will appear unattractive because of high area costs.

An alternative routing architecture is one in which the routing tracks are between the LUTs of the HLBs and the hard-wired links between the LUTs span the routing channels in a manner similar to the direct connect in the Xilinx 3000 architecture [10]. With this scheme, *W* should be less than the *W* of an FPGA without hard-wired connections because the number of pins to be connected to each channel segment is reduced. A reduced *W* implies that the alternative routing architecture may lead to HLBs with more LUT basic blocks becoming more attractive because they would have lower area costs.

Chapter 7 Conclusions and Future Work

7.1 Thesis Summary and Contributions

This dissertation presents CAD algorithms for mapping a combinational digital circuit into a delay- or area-optimized netlist of hard-wired logic blocks (HLBs). The algorithm begins with a circuit of basic blocks and transforms it into a netlist of HLBs using two stages. The first step produces a delay- or area-optimized set of covering fragments. The second step packs the set of covering fragments together into a minimum number of packed HLBs.

The delay-optimization covering algorithm is shown to produce a netlist of HLB fragments with a minimal number of programmable connections along the critical path. Also, the fragment packing algorithm is proven to be optimal when packing the set of covering fragments for all two-level HLBs.

The effectiveness of the CAD algorithms is evaluated with respect to a lower bound and also compared with a commercial mapping tool [11]. Since the delay-optimization covering algorithm is optimal the theoretical minimum bounds for delay was achieved. Compared to a simple lower bound on area, the mapping algorithms uses only 3% to 16% more HLBs when mapping HLBs with two to nine 4-LUTs. When compared to a commercial mapping tool for the Xilinx 4000 Configurable Logic Block [11], which is a commercial HLB-based FPGA architecture, our overall synthesis procedure produces area-optimized circuits of about the same size and delay-optimized circuits with significantly (22%) fewer programmable connections along the critical paths.

The HLB mapping tools were used to investigate a wide range of HLB-based FPGA architectures for speed and density. In particular we sought to determine the relationships between the basic block functionality and hard-wired connection topology of the HLB versus the speed and density of the resulting HLB circuits. These are the results of the HLB architecture studies:

i) HLB architectures consisting of 6-LUTs yield the fastest HLB-based FPGA circuits with reasonable density.

ii) HLB architectures consisting of 4-LUTs yield the most area-efficient FPGA circuits. However, for only a slightly higher cost in area relative to the densest 4-LUT HLBs, some 5-LUT HLBs give much higher speeds.

iii) The HLB topologies that resulted in the fastest FPGA circuits had a high fan-in of hard-wired connections to some of its basic blocks.

iv) The HLB topologies that resulted in the most area-efficient FPGA circuits consisted of basic blocks which all had two or more non-hard-wired inputs.

7.2 Future Work

This section describes several improvements that could be made to the HLB mapping algorithm and also several avenues for future HLB-based FPGA architecture research.

7.2.1 Enhancements to the HLB Mapping Algorithms

The mapping algorithms in this thesis are the first attempts at synthesis for general HLB architectures. This section contains several suggestions for improving the quality of HLB synthesis.

Combining Basic Block Mapping and HLB Technology Mapping

The current procedure for mapping a Boolean network to a netlist of HLBs has two steps. First the Boolean network is mapped to a network of basic blocks, and then the basic block network is

mapped to a netlist of HLBs. The CAD tools that map to basic block networks do not utilize HLB topology information. A technology mapper that synthesized directly from Boolean network to the HLB structure could yield better mapping results because it would use HLB topology information during the basic block mapping process.

Mapping to Multi-output Fragments

HLBs have multiple outputs. However, the mapping algorithm in this dissertation first finds a covering set of single-output fragments and then packs these single-output fragments together. When several single-output fragments are packed together hard-wired links are wasted. A mapping algorithm that maps to multi-output fragments directly should give better area results since fewer hard-wired links would be wasted.

Placement-Based Cost Function

During the fragment packing stage, the connectivity of the fragments is not taken into account. Other optimization goals that take connectivity or placement of the HLB fragments into account during packing may yield a more routable solution, and hence a smaller and faster circuit after routing.

Delay-Area Trade-off

The delay- and area-optimization algorithms do not easily allow trade-offs between delay and area. The mapping algorithms either minimize area or delay, with no provisions for area or delay constraints. A better mapping algorithm would minimize area under a delay constraint.

7.2.2 HLB-based FPGA Architecture Investigation Avenues

This dissertation examined a subset of the possible avenues of research into HLB-based FPGA architectures. This section suggests some new directions for HLB architecture research.

Using More Effective Basic Block Mappers

The LUT mapper may also have an effect on the results. The Chortle LUT mapper either optimizes area or delay and does not devote much effort towards optimizing the secondary optimization goal. In the future, other studies could be done with technology mappers that are more effective for minimizing the secondary costs. For example, the Flowmap LUT mapper [33] minimizes delay with much better area results than Chortle.

Application Specific or Class Specific Architectures

The selection of benchmark circuits used to evaluate the architectures is a mixture of random logic and arithmetic circuits. One avenue to investigate is to determine if benchmark circuit topologies vary widely and if so, determine whether there are HLB topologies which are more suitable for a particular class of circuits, such as random logic or arithmetic circuits.

Other hard-wired connection topologies

This thesis assumed that basic blocks were hard-wired in tree topologies. There may be benefits to having fan-out of hard-wired connections within an HLB or to having sharing of LUT inputs as in the Xilinx 3000 architecture.

Heterogeneous LUT HLB architectures

This thesis assumed that the LUTs in the HLB were homogeneous. Other research indicates that the use of several sizes of LUT basic blocks in the FPGA may result in improved density with respect to homogeneous LUT basic blocks [13] [46]. Heterogeneous LUT HLB architectures may also result in similar improvements.

Focus on Certain LUT sizes

The results in Chapter 6 indicate that mid-grained LUTs (from 3 to 6 inputs) offer the best HLBs for speed and or density. Future HLB-based FPGA studies should target HLBs composed of these particular LUTs, and so more topologies could be examined. The run-time of the CAD

tools would have to be reduced to investigate HLBs with more LUTs than those studied in this dissertation.

Reduction of Tapping Buffers

The tapping buffers used to access each basic block output have a significant on-chip area cost. While this work assumed that there is a tapping buffer on *every* LUT output, this may not be necessary. A study of the utilization of the HLB tapping buffers may lead to methods to reduce the number of tapping buffers per HLB.

Improved Delay Modelling

Hard-wired logic blocks reduce the average programmable connection length in circuits because they reduce the number of logic blocks in the circuit. This reduction of the average programmable connection length also means the number of switching stages is lessened and this leads to faster circuits. A study to measure the speedup attained by reducing the programmable connection lengths in circuits would require a more accurate delay model.

Changed Architecture Assumptions

The routing architecture in this thesis assumed that each HLB (which consists of several basic blocks) is surrounded by the routing channels. A possibly better routing architecture is one in which the routing tracks are between the LUTs of the HLBs and the hard-wired links between the LUTs span the routing channels in a manner similar to the direct connect in the Xilinx 3000 architecture [10]. This leads to narrower routing channels because the number of pins to connect to each channel is lower. In fact, the routing channel widths should be closer to the widths of the circuits without hard-wired logic blocks. The drawback of this new scheme is that the programmable connections would have to go through more switching stages than the scheme that assumed the HLB was surrounded by the routing channels. However, the circuits using the new scheme will still experience speedups because of the presence of hard-wired connections in critical paths.

Appendix A Data from the HLB Architecture Studies

This appendix presents a summary of the data from the speed and area-efficiency studies in Chapter 6.

A.1 Envelope Set data from Speed Study

This section of the appendix contains the (area, speed) co-ordinates of the envelope set of points used for the speed study (Figure 6-8). Each of the following six tables, Table A.1 to Table A.6, corresponds to the speed versus area curves for the 2-LUT, 3-LUT, 4-LUT, 5-LUT, 6-LUT and 7-LUT HLBs respectively. The data in the following tables was generated using a delay model with $D_R = 4ns$ and an area model with $RB = 1000\mu m^2$. The first column in each table lists the HLB topology, the second column lists the average normalized area over all circuits, the third column lists the standard deviation of the average area, the fourth column lists the average normalized speed and the fifth column lists the standard deviation of the average speed.

HLB topology	Area	S.D. Area	Speed	S.D. Speed
L2-3	1.40	0.16	0.75	0.08
L3-5.2.2	1.55	0.23	0.83	0.11
L3-6.3.2	1.60	0.23	0.90	0.11
L3-7.3.3	1.79	0.29	0.91	0.09
L4-8.4.2.3	1.82	0.33	0.93	0.10
L4-11.6.3.2.4.3	1.90	0.47	0.95	0.12
L4-9.5.2.2.3	1.93	0.47	0.97	0.09
L4-12.6.3.2.5.3	1.95	0.56	1.00	0.10
L4-15.7.3.3.7.3.3	2.44	0.70	1.01	0.08

Table A-1: Speed-optimized 2-LUT HLB envelope set

HLB topology	Area	S.D. Area	Speed	S.D. Speed
L1	1.13	0.19	0.77	0.12
L2-3	1.24	0.28	1.10	0.16
L2-4	1.35	0.25	1.17	0.19
L3-6.3.2	1.55	0.45	1.21	0.18
L3-6.4	1.65	0.47	1.23	0.19
L3-7.3.2	1.68	0.58	1.28	0.22
L3-7.4.2	1.72	0.56	1.30	0.19
L3-9.4.3	1.82	0.64	1.32	0.19
L3-10.4.3.2	1.88	0.68	1.36	0.22
L3-11.4.4.2	2.04	0.80	1.38	0.24
L3-12.4.4.3	2.13	0.75	1.39	0.25

Table A-2: Speed-optimized 3-LUT HLB envelope set

HLB topology	Area	S.D. Area	Speed	S.D. Speed
L1	1.00	0.00	1.00	0.00
L2-2	1.19	0.16	1.14	0.10
L2-3	1.22	0.22	1.27	0.14
L3-4.2	1.33	0.34	1.28	0.14
L2-4	1.34	0.29	1.34	0.13
L3-5.2	1.42	0.47	1.37	0.13
L2-5	1.44	0.39	1.42	0.10
L3-8.4.3	1.69	0.67	1.44	0.19
L3-8.3.2	1.73	0.75	1.48	0.15
L3-9.4.3	1.74	0.70	1.49	0.21
L3-9.4.2	1.86	0.88	1.51	0.16

Table A-3: Speed-optimized 4-LUT HLB envelope set

HLB topology	Area	S.D. Area	Speed	S.D. Speed
L1	1.01	0.20	1.16	0.16
L2-2	1.13	0.18	1.21	0.15
L2-3	1.33	0.45	1.38	0.20
L3-4.2	1.39	0.47	1.39	0.20
L2-4	1.46	0.49	1.42	0.20
L3-5.2	1.49	0.51	1.43	0.22
L3-6.3	1.55	0.61	1.47	0.22
L3-6.2	1.56	0.64	1.52	0.26
L2-6	1.78	0.64	1.65	0.31

Table A-4: Speed-optimized 5-LUT HLB envelope set

HLB topology	Area	S.D. Area	Speed	S.D. Speed
L1	1.02	0.21	1.25	0.18
L2-2	1.13	0.19	1.31	0.18
L2-3	1.30	0.38	1.45	0.20
L2-4	1.46	0.52	1.54	0.26
L3-5.2	1.50	0.47	1.56	0.25
L3-6.3	1.61	0.53	1.58	0.27
L2-5	1.62	0.62	1.67	0.28
L3-6.2	1.70	0.56	1.70	0.28
L3-6.5	1.77	0.78	1.72	0.30

Table A-5: Speed-optimized 6-LUT HLB envelope set

HLB topology	Area	S.D. Area	Speed	S.D. Speed
L1	1.23	0.38	1.32	0.21
L2-2	1.42	0.35	1.35	0.21
L2-3	1.52	0.51	1.55	0.26
L2-4	1.69	0.58	1.56	0.27
L3-6.4	1.87	0.87	1.57	0.27
L2-5	2.02	0.98	1.70	0.30
L3-6.2	2.11	1.04	1.71	0.30
L2-7	2.89	1.73	1.74	0.31
L2-8	3.04	1.98	1.77	0.32

Table A-6: Speed-optimized 7-LUT HLB envelope set

A.2 Envelope Set data from Area-efficiency Study

This section of the appendix contains the (area, speed) co-ordinates of the envelope set used for the area-efficiency study (Figure 6-11). Each of the following six tables, Table A.7 to Table A.12, corresponds to the speed versus area curves for the 2-LUT, 3-LUT, 4-LUT, 5-LUT, 6-LUT and 7-LUT HLBs respectively. The data in the following tables was generated using a delay model with $D_R = 4ns$ and an area model with $RB = 1000\mu m^2$. The first column in each table lists the HLB topology, the second column lists the average normalized area over all circuits, the third column lists the standard deviation of the average area, the fourth column lists the average normalized speed and the fifth column lists the standard deviation of the average speed.

HLB topology	Area	S.D. Area	Speed	S.D. Speed
L4-5.3.2	1.21	0.24	0.89	0.27
L4-7.3.2.3.2	1.27	0.27	0.90	0.27
L4-7.4.2.2	1.29	0.36	0.92	0.26
L4-8.4.2.3.2	1.35	0.31	0.95	0.27

Table A-7: Area-optimized 2-LUT HLB envelope set

HLB topology	Area	S.D. Area	Speed	S.D. Speed
L2-2	0.98	0.12	0.90	0.20
L3-4.2	1.03	0.13	1.05	0.28
L3-5.3	1.10	0.27	1.10	0.37
L3-8.3.3	1.43	0.41	1.11	0.37
L3-9.3.3.2	1.50	0.38	1.12	0.36

Table A-8: Area-optimized 3-LUT HLB envelope set

HLB topology	Area	S.D. Area	Speed	S.D. Speed
L2-2	0.94	0.19	1.18	0.13
L3-4.2	0.95	0.17	1.28	0.15
L3-6.3.2	1.04	0.11	1.32	0.17
L3-7.4.2	1.20	0.21	1.33	0.17

Table A-9: Area-optimized 4-LUT HLB envelope set

HLB topology	Area	S.D. Area	Speed	S.D. Speed
L2-3	0.97	0.17	1.47	0.45
L3-5.2.2	1.03	0.18	1.52	0.44
L3-6.2.2	1.09	0.27	1.54	0.42
L3-6.3.2	1.15	0.28	1.56	0.53

Table A-10: Area-optimized 5-LUT HLB envelope set

HLB topology	Area	S.D. Area	Speed	S.D. Speed
L2-2	1.17	0.20	1.22	0.39
L2-3	1.20	0.24	1.26	0.42
L3-4.2	1.24	0.27	1.31	0.44
L3-5.2.2	1.26	0.30	1.36	0.53
L3-6.3.2	1.32	0.26	1.39	0.54

Table A-11: Area-optimized 6-LUT HLB envelope set

HLB topology	Area	S.D. Area	Speed	S.D. Speed
L1	1.43	0.37	1.27	0.26
L2-2	1.47	0.36	1.38	0.32
L3-4.2	1.56	0.42	1.44	0.39
L3-5.2	1.73	0.45	1.47	0.39

Table A-12: Area-optimized 7-LUT HLB envelope set

A.3 Envelope Set Data for Individual Circuits

Table A.13 lists the 4-LUT HLB topologies that lie on the speed versus area curve for each individual circuit when the HLB circuits are optimized for speed. The first column lists the benchmark circuit name, the second column lists the HLB topology, the third column lists the average normalized area and the fourth column lists the average normalized speed.

Benchmark Cct.	HLB topology	Area	Speed
9symml	L3-7.3.3	0.89	1.54
c1355	L1	1.00	1.00
¨	L2-2	1.27	1.11
¨	L2-3	1.41	1.25
¨	L3-6.3.2	1.68	1.43
¨	L3-9.4.2	3.05	1.67
c432	L1	1.00	1.00
¨	L2-2	1.22	1.21
¨	L2-3	1.59	1.30
¨	L2-5	2.05	1.54
¨	L3-9.5	4.02	1.69
c499	L1	1.00	1.00
¨	L2-2	1.34	1.10
¨	L2-3	1.52	1.36
¨	L3-6.4	1.67	1.54
alu2	L1	1.00	1.00
¨	L2-2	1.27	1.15
¨	L3-4.2	1.28	1.34
¨	L3-5.4	1.73	1.47
¨	L3-7.3	2.14	1.62
apex7	L1	1.00	1.00
¨	L2-2	1.13	1.16
¨	L3-4.2	1.23	1.39
¨	L3-7.3.2	1.64	1.73
cm150a	L3-7.4	0.67	1.54
cm151a	L3-7.4	0.77	1.30

Table A-13: Speed-optimized 4-LUT HLB envelope set for individual circuits

Benchmark Cct.	HLB topology	Area	Speed
"	L3-8.4.3	0.87	1.88
cm162a	L2-2	0.91	1.30
cm163a	L3-7.2.2	0.99	1.30
"	L3-7.5	0.99	1.30
"	L3-9.5.2	1.16	1.88
count	L1	1.00	1.00
"	L3-4.2	1.03	1.21
"	L3-6.2.2	1.16	1.54
"	L3-6.3.2	1.16	1.54
frg1	L3-4.2	0.96	1.39
"	L3-9.3.2.2	1.26	1.73
k2	L1	1.00	1.00
"	L2-2	1.14	1.11
"	L2-3	1.29	1.25
"	L2-4	1.50	1.43
"	L3-9.4.2	2.46	1.67
mux	L2-5	0.80	1.30
"	L3-5.2	0.80	1.30
"	L3-9.4.2	0.80	1.30
"	L3-9.5	0.80	1.30
parity	L2-5	0.90	1.30

Table A-13: Speed-optimized 4-LUT HLB envelope set for individual circuits

Table A.14 lists the 4-LUT HLB topologies that lie on the speed versus area curve for each individual circuit when the HLB circuits are optimized for area. The first column lists the benchmark circuit name, the second column lists the HLB topology, the third column lists the average normalized area and the fourth column lists the average normalized speed.

Benchmark Cct.	HLB topology	Area	Speed
9symml	L3-4.2	0.84	1.36
c1355	L2-2	0.92	1.37
"	L3-4.3	1.02	1.48
c432	L2-2	0.98	1.12
"	L2-3	1.04	1.16
"	L3-4.2	1.08	1.21
"	L3-6.3.2	1.16	1.27
c499	L3-3.2	0.85	1.37
"	L3-4.3	0.90	1.48
alu2	L2-2	0.77	1.21
"	L3-4.2	0.81	1.32
"	L3-8.4.2	1.52	1.39
apex7	L2-2	1.02	1.21
"	L2-3	0.89	1.10
cm150a	L3-7.3.3	0.92	1.54
"	L3-7.4	0.92	1.54
cm151a	L3-8.4.2	0.58	1.39
cm162a	L3-6.2.2	0.65	1.21
cm163a	L3-3.2	0.75	1.21
count	L2-2	0.92	1.08
frg1	L2-3	0.81	1.18
"	L3-3.2	0.89	1.30
k2	L2-2	0.90	1.18
mux	L3-7.3.3	0.65	1.54
parity	L3-9.2.2.2.2	0.78	1.73

Table A-14: Area-optimized 4-LUT HLB envelope set for individual circuits

References

[1] Panel discussion, "Will the Field-Programmable Gate Array replace the Mask-Programmed Gate Array," *Proceedings of the 28th Design Automation Conference (DAC-28)*, June 1991.

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

[3] W. Carter, K. Duong, R. H. Freeman, H. Hsieh, J. Y. Ja, J. E. Mahoney, L. T. Ngo and S. L. Sze, "A User Programmable Reconfigurable Gate Array," *Proceedings of the 1986 Custom Integrated Circuits Conference (CICC-86)*, May 1986, pp. 233-235.

[4] A. El Gamal, J. Greene, J. Reyneri, E. Rogoyski, K. El-Ayat and A. Mohsen, "An Architecture for Electrically Configurable Gate Arrays," *IEEE Journal of Solid State Circuits (JSSC)*, Vol. 24, No. 2, April 1989, pp. 394-398.

[5] A. Gupta, V. Aggarwal, R. Patel, P. Chalasani, D. Chu, P. Seeni, P. Liu, J. Wu and G. Kaat, "A User Configurable Gate Array Using CMOS-EPROM Technology," *CICC-90*, May 1990, pp. 31.7.1-31.7.4.

[6] S. Singh, J. Rose, D. Lewis, K. Chung and P. Chow, "Optimization of Field-Programmable Gate Array Logic Block Architecture for Speed," *CICC-91*, May 1991, pp. 6.1.1-6.1.6.

[7] S. Singh, J. Rose, P. Chow and D. Lewis, "The Effect of Logic Block Architecture on FPGA Performance," *JSSC*, Vol. 27, No. 3, March 1992, pp. 281-287.

[8] J. Rose, R. J. Francis, D. Lewis, P. Chow, "Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency," *JSSC*, Vol. 25, No. 5, Oct. 1990, pp. 1217-1225.

[9] K. Chung and J. Rose, "TEMPT: Technology Mapping for the Exploration of FPGA Architectures with Hard-Wired Connections," *Proceedings of the 29th Design Automation Conference (DAC-29)*, June 1992, pp. 361-367.

[10] "The XC3000 Logic Cell Array Family," in **The Programmable Gate Array Data Book**, Xilinx Inc, 1991.

[11] **The XACT 4000 User Guide**, Xilinx Inc., March 1991,

[12] "The XC4000 Logic Cell Array Family" Product Description, Xilinx Inc., August 1992.

[13] H. Hsieh, W. Carter, J. Y. Ja, E. Cheung, S. Schreifels, C. Erickson, P. Freidin and L. Tinkey, "Third-Generation Architecture Boosts Speed and Density of Field-Programmable Gate Arrays," *CICC-90*, pp. 31.2.1-31.2.7.

[14] S. Singh, "The Effect of Logic Block Architecture on FPGA Performance," *M.A.Sc. Thesis*, Department of Electrical Engineering, University of Toronto, 1991.

[15] K. Chung, S. Singh, J. Rose and P. Chow, "Using Hierarchical Logic Blocks to Improve the Speed of Field-Programmable Gate Arrays," in **FPGAs** - *Proceedings of the First International Workshop on Field Programmable Logic and Applications*, Oxford, Sept. 1991, pp. 103-113.

[16] D. Hill and N. Woo, "The Benefits of Flexibility in Lookup-Table FPGAs" in **FPGAs** - *Proceedings of the First International Workshop on Field Programmable Logic and Applications*, Oxford, Sept. 1991, pp. 127-136.

[17] R. Brayton, R. Rudell, A. Sangiovanni--Vincentelli and A. Wang, "MIS: a Multiple-Level Logic Optimization System," *IEEE Transactions on CAD (TCAD)*, Vol. CAD-6, No. 6, Nov. 1987, pp. 1062-1081.

[18] R. J. Francis, "A Tutorial on Logic Synthesis for Lookup-Table Based FPGAs," *ICCAD-92*, Nov. 1992, pp. 40-47.

[19] K. Keutzer, "DAGON: Technology Binding and Local Optimization by DAG Matching," *DAC-24*, June 1987, pp. 341-347.

[20] R. J. Francis, J. Rose and Z. Vranesic, "Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs," *DAC-28*, June 1991, pp. 227-233.

[21] R. J. Francis, J. Rose and Z. Vranesic, "Technology Mapping of Lookup Table-Based FPGAs for Performance," *Proceedings of IEEE International Conference on CAD (ICCAD-91)*, Nov. 1991, pp. 568-571.

[22] J. Rose, Z. Vranesic and W. M. Snelgrove, "ALTOR: An automatic standard cell layout program," in *Proceeding of the Canadian Conference on VLSI*, Nov. 1985, pp. 168-173.

[23] J. Vuillamy, "Performance Enhancement in Field-Programmable Gate Arrays," *M.A.Sc. Thesis*, Department of Electrical Engineering, University of Toronto, 1991.

[24] J. Vuillamy, Z. Vranesic and J. Rose, "Performance Evaluation and Enhancement of FPGAs," in **FPGAs** - *Proceedings of the First International Workshop on Field Programmable Logic and Applications*, Oxford, Sept. 1991, pp. 137-146.

[25] P. Chow, S. O. Seo, K. Chung, G. Paez and J. Rose, "A High-Speed FPGA Using Programmable Mini-tiles," **Research on Integrated Systems** - *Proceedings of the 1993 Symposium on Integrated Systems*, University of Washington, March 1993, pp. 103-122.

[26] S. O. Seo, “A High-Speed FPGA Using Programmable Mini-tiles,” M.A.Sc. Thesis in preparation, Department of Electrical Engineering, University of Toronto.

[27] J. Rose and S. Brown, “The Effect of Switch Box Flexibility on Routability of Field Programmable Gate Arrays,” *CICC-90*, May 1990, pp. 27.5.1-27.5.4.

[28] J. Rose and S. Brown, “Flexibility of Interconnection Structures in Field Programmable Gate Arrays,” *JSSC*, Vol. 26, No. 3, March 1991, pp. 277-282.

[29] J. Kouloheris and A. El Gamal, “FPGA Performance vs. Cell Granularity,” *CICC-91*, May 1991, pp. 6.2.1-6.2.4.

[30] J. Kouloheris and A. El Gamal, “FPGA Area vs. Cell Granularity - Lookup tables and PLA Cells,” *ACM/SIGDA Workshop on Field-Programmable Gate Arrays, FPGA-92*, Berkeley, CA, February 1992, pp. 9-14.

[31] J. Kouloheris and A. El Gamal, “PLA-based FPGA Area vs. Cell Granularity,” *CICC-92*, May 1992, pp. 4.3.1-4.3.4

[32] K. Chen, J. Cong, Y. Ding, A. Kahng and P. Trajmar, “DAG-MAP: Graph-based FPGA Technology Mapping for Delay Optimization,” *IEEE Design and Test of Computers*, Sept. 1992, pp. 7-20.

[33] J. Cong and Y. Ding, “An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs,” *ICCAD-92*, Nov. 1992, pp. 48-53.

[34] R. Murgai, N. Shenoy, R. K. Brayton and A. Sangiovanni-Vincentelli, “Improved Logic Synthesis Algorithms for Table Look Up Architectures,” *ICCAD-91*, Nov. 1991, pp. 564-567.

[35] R. Murgai, N. Shenoy, R. K. Brayton and A. Sangiovanni-Vincentelli, “Performance Directed Synthesis for Table Look Up Programmable Gate Arrays,” *ICCAD-91*, Nov. 1991, pp. 572-575.

[36] P. Sawkar and D. Thomas, “Area and Delay Mapping for Table-Lookup Based Field Programmable Gate Arrays,” *DAC-29*, June 1992, pp. 368-373.

[37] N. Woo, “A Heuristic Method for FPGA Technology Mapping Based on Edge Visibility,” *DAC-28*, June 1991, pp. 248-251.

[38] R. Rudell, “Logic Synthesis for VLSI Design,” PhD. Thesis, University of California, Berkeley, April 1989. Memorandum No. UCB/ERL M89/49.

[39] R. J. Francis, “Technology Mapping for Lookup-Table Based Field-Programmable Gate Arrays,” PhD. Thesis, University of Toronto, Toronto, Canada. December 1992.

[40] M. Garey and D. Johnson, **COMPUTERS AND INTRACTABILITY - A Guide to the Theory of NP-Completeness**, W. H. Freeman and Company, New York, 1979.

[41] D. Lewis, Personal communication regarding unique naming of HLB tree topologies.

[42] M. Hutton, Personal communication regarding complexity of the integer bin-packing problem, February 1993.

[43] M. Hutton, “Notes on Integer Bin-Packing for Technology Mapping on Trees,” Personal communication, February 1993.

[44] C. Papadimitriou and K. Steiglitz, **COMBINATORIAL OPTIMIZATION - Algorithms and Complexity**, Prentice-Hall Inc.,Englewood Cliffs, New Jersey, 1982.

[45] A. Aho, J. Hopcroft and J. Ullman, **The Design and Analysis of Computer Algorithms,** Addison-Wesley Publishing Company, Reading, Massachusetts, 1974.

[46] J. He and J. Rose, “Advantages of Heterogeneous Logic Block Architectures for FPGAs ,” *CICC-9*3, May 1993, pp. 7.4.1-7.4.5.

[47] R. Bellman. **Dynamic Programming**, Princeton University Press, 1957.

[48] L. Cooper and M. Cooper, **Introduction to Dynamic Programming**, Pergamon Press Ltd., 1981.

[49] S. Yang, “Logic Synthesis and Optimization Benchmarks User Guide,” Version 3.0, Microelectronics Centre of North Carolina (MCNC), January 1991.

[50] T. Cormen, C. Leiserson and R. Rivest, **Introduction to Algorithms**, The MIT Press, Cambridge, Massachusetts, McGraw-Hill Book Company, Toronto, Eighth printing, 1992.

[51] J. L. Hennessy and D. A. Patterson, **Computer Architecture: A Quantitative Approach**, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.