Post-mapping topology rewriting for FPGA area minimization

by

Lei Chen

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2009

Abstract

Circuit designers require Computer-Aided Design (CAD) tools when compiling designs into Field Programmable Gate Arrays (FPGAs) in order to achieve high quality results due to the complexity of the compilation tasks involved. Technology mapping is one critical step in the FPGA CAD flow. The final mapping result has significant impact on the subsequent steps of clustering, placement and routing, for the objectives of delay, area and power dissipation. While depth-optimal FPGA technology mapping can be solved in polynomial time, area minimization has proven to be NP-hard.

Most modern state-of-the-art FPGA technology mappers are structural in nature; they are based on cut enumeration and use various heuristics to yield depth and area minimized solutions. However, the results produced by structural technology mappers rely strongly on the structure of the input netlists. Hence, it is common to apply additional heuristics after technology mapping to further optimize area and reduce the amount of structural bias while not harming depth. Recently, SAT-based Boolean matching has been used for post-mapping area minimization. However, SAT-based matching is computationally complex and too time consuming in practice.

This thesis proposes an alternative Boolean matching approach based on NPN equivalence. Using a library of pre-computed topologies, the matching problem becomes as simple as performing NPN encoding followed by a hash lookup which is very efficient. In conjunction with Ashenhurst decomposition, the NPN-based Boolean matching is allowed to handle up to 10-input Boolean functions. When applied to a large set of designs, the proposed algorithm yields, on average, more than 3% reduction in circuit area without harming circuit depth. The priori generation of a library of topologies can be difficult; the potential difficulty in generating a library of topologies represents one limitation of the proposed algorithm.

Acknowledgements

I would like to thank Prof. Andrew Kennings for supervising this work and for giving me the chance to step into the EDA industry. Without all the help he gave me, this thesis would not have come into existence. I would also like to thank my family and friends for their support over these years.

Contents

List of Tables

List of Figures

Chapter 1

Architecture and CAD for FPGAs

1.1 Introduction

Application-Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs) are two primary platforms for integrated circuit design. FPGAs are programmable semiconductor fabric that are based on an array of prebuilt Configurable Logic Blocks (CLBs) and programmable interconnects. As opposed to ASICs which are built and customized for one particular design, FPGAs can be programmed to the desired application. Although one-time programmable FPGAs are available, the dominate types are those that can be reprogrammed a large number of times as the design evolves.

FGPAs and ASICs have different value propositions, and the following are the pros and cons of each platform. First, the ASIC is fixed once manufactured, whereas the FPGA is completely reconfigurable. The reconfigurability of the FPGA allows easy modification of its application. In the case of ASICs, new chips must be manufactured again and old ones have to be discarded, with any change required in the circuit design. However, if the circuit design is based on the FPGA, the design can be updated by simply reconfigurating the FPGA. There are no non-recurring expenses (NREs) for FPGA designs. For this reason, FPGAs are very suitable for ASIC prototyping, and ASICs tend to be reserved for high volume products in the future. Second, FPGAs also have an advantage over traditional ASICs in terms of simpler design cycle and faster time-to-market. The ASIC chip manufacturing process can take months, whereas FPGAs are pre-fabricated and provide off-the-shelf applications. Even though FPGAs seem a compelling proposition for circuit designs in many situations, FPGAs are not without drawbacks. Because programmable logic blocks and routing resources are not fully utilized, PFGAs usually consumes larger silicon area, which leads to higher per-unit cost and larger power consumption. Besides, FPGAs have slower circuit speed compared with their ASIC counterpart, under the constraints of their logic blocks and routing resources [28, 29]. For applications that require high performance and have strict power consumption requirement, ASICs are still the preferred choice.

1.2 FPGA Architecture Overview

Traditionally, FPGAs include programmable logic blocks which implement logic functions, programmable routing blocks to interconnect these functions and input/output (I/O) blocks to make off-chip connections. They can be programmed to implement any logic functions. However, today's FPGAs have evolved far beyond the basic capabilities present in their predecessors, and incorporated blocks of commonly used functionalities such as embedded memories, embedded processors, Phase-Locked Loops (PLLs), Digital Signal Processing (DSP) units and other special feature blocks. These features make it possible to build a System on Chip (SoC) inside of a single FPGA. An example of modern FPGA architecture is the Altera Stratix IV device, as shown in Figure 1.1.

Every FPGA relies on the underlying programming technology that gives FPGAs their programmability. FPGA vendors offer three fundamental programming technologies for modern FPGAs. SRAM-based FPGAs are widely used and can be found in devices such as Xilinx Virtex-5 [8] and Altera Stratix IV [7]. Non-volatile flash-based technology is used in Actel ProASIC devices [4] and antifuse-based technology is used in Actel Axcelerator devices [3]. SRAM programming technology has become dominant for FPGAs for its re-programmability and use of standard CMOS manufacturing processes. Although, flash-based devices offer benefits in terms of power consumption due to their technology and antifuse-based devices are popular for aerospace and military applications due to their radiation tolerance.

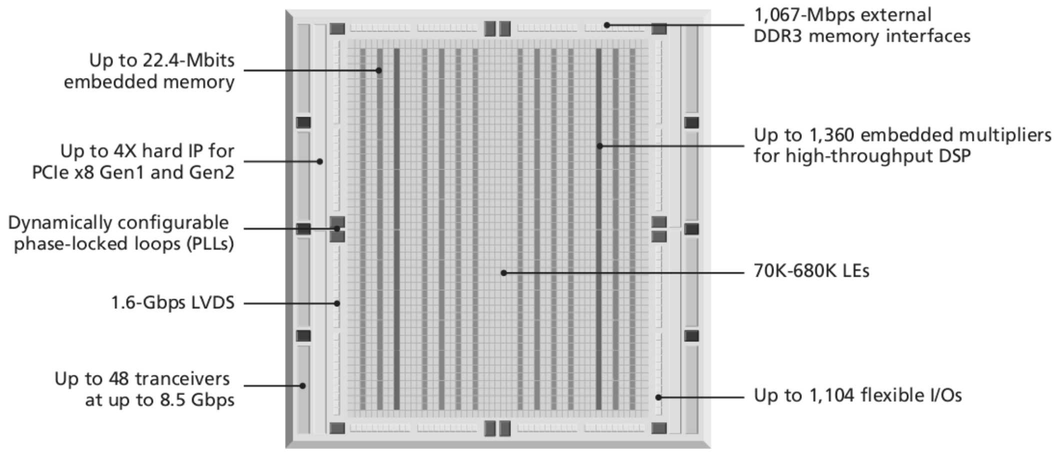

1,067-Mbps external DDR3 memory interfaces
Up to 22.4-Mbits embedded memory
Up to 4X hard IP for PCIe x8 Gen1 and Gen2
Up to 1,360 embedded multipliers for high-throughput DSP
Dynamically configurable phase-locked loops (PLLs)
70K-680K LEs
1.6-Gbps LVDS
Up to 48 tranceivers at up to 8.5 Gbps
Up to 1,104 flexible I/Os

Figure 1.1: Example of a modern FPGA architecture consisting of many different types of resources; e.g., the Altera Stratix IV architecture [6].

1.2.1 LUT-based FPGAs

A simplified model of the architecture of an FPGA is illustrated in Figure 1.2. In Figure 1.2, the FPGA is arranged in an island-style structure based on an array of identical programmable logic blocks surrounded by routing channels with the I/O pads evenly distributed around the perimeter of the FPGA [5].

The logic blocks consist of circuitry for implementing the functionality of the given circuit. Logic blocks are also called logic clusters, or configurable logic blocks (CLBs). The structure of a simplified cluster-based logic block is illustrated in Figure 1.3. Each cluster contains *N* Basic Logic Elements (BLEs). Typically, the number of input pins on the CLB is less than the sum of the number of inputs of the contained BLEs. Therefore, if the BLEs within the CLB are to be fully utilized, some of the BLEs will need share inputs. Typically, the BLE outputs are fed back to the BLE inputs within the cluster; i.e., BLEs within the same CLB can drive each other's inputs without having to go outside of the CLB. Because the delay of intra-CLB connections is typically much less than the delay of inter-CLB connections (it is unnecessary to use external routing resources when making intra-CLB connections), it is preferable to group BLEs together that share many

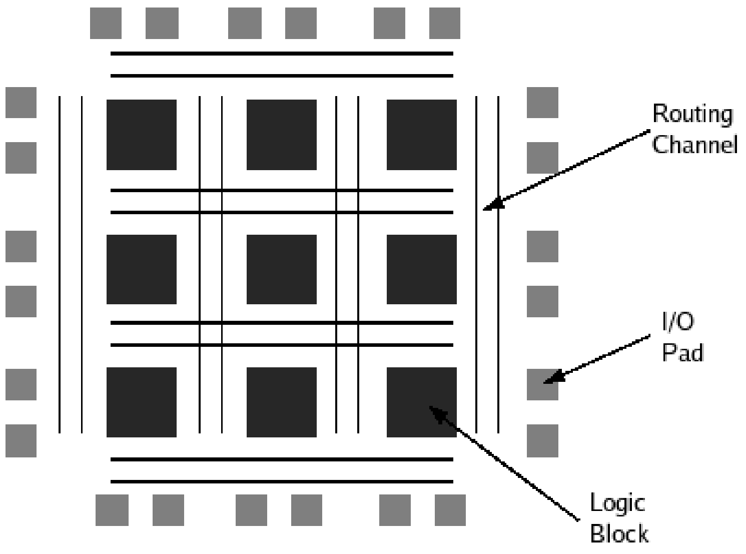


Figure 1.2: A simplified view of an island-styled FPGA consisting of CLBs. [1]

interconnections. Increasing the number of inputs to each logic block can increase the number of logic functions realized by each logic block as well as improve the performance of the logic block. However, this comes on the expense of wasted resources because not all the logic blocks will have all of their inputs fully utilized. The Altera Stratix and Xilinx Virtex FPGA devices are commercial examples of FPGA architectures that use the idea of BLEs clustered together inside of CLBs [7, 8].

The simplified architecture of a BLE is shown in Figure 1.4. Each BLE consists of a k-input Look-Up Table (LUT) for implementing combinational logic and a register for implementing sequential logic. A multiplexer permits either the LUT or the register to feed the output of the BLE. A k-input LUT consists of 2^k configuration bits and can realize any Boolean logic function of up to k variables by programming the truth table of the Boolean logic function directly into the memory bits of the LUT (the programming of the LUT is done during the configuration of the FPGA). Figure 1.5 shows a two-input LUT with 4 programmable memory bits and a multiplexer to select one of the memory bits based on the two select lines which serve as inputs to the 2-input logic function. In commercial LUT-based architectures, k typically varies from 3 to 6.

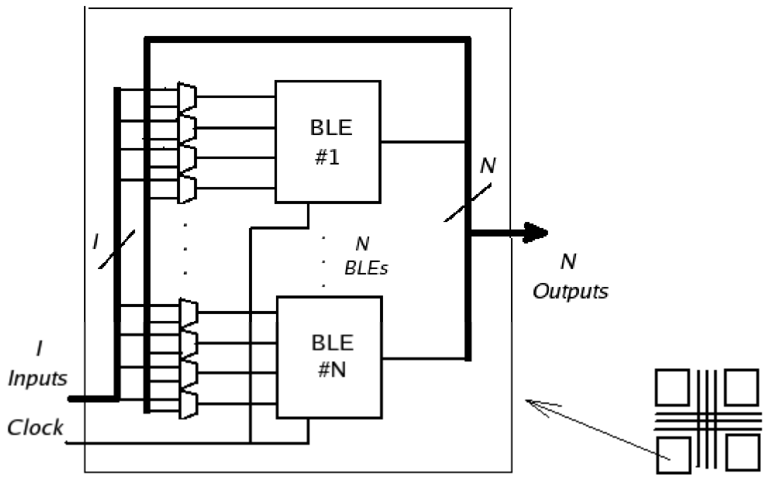


Figure 1.3: A simplified illustration of a Configuration Logic Block (CLB).

In the island-style FPGA architecture, each BLE can be connected to other BLEs via horizontal and vertical routing channels. A generic FPGA routing architecture is illustrated in Figure 1.6. Switch boxes connect horizontal and vertical channels through programmable switches. Connection boxes are used to connect routing channels to the CLBs [41]. The routing architecture of an FPGA is prefabricated and most modern FPGA architectures use various wires of different segment lengths to achieve the optimal performance in terms of circuit delay, routability or both. Finally, in addition to programmable routing for the purposes of connecting logic blocks, modern FPGAs also contain dedicated global routing networks for the purpose of distributing low skew clocks and control signals throughout the programmable logic fabric.

1.3 FPGA CAD Flow

FPGAs have become very popular in recent years, due to their programmability and fast time-to-market. With the advances in process technology, the number of gates and features on a single FPGA device has increased dramatically to compete with capabilities that have traditionally only been offered through ASIC devices. To effectively use FPGAs, circuit

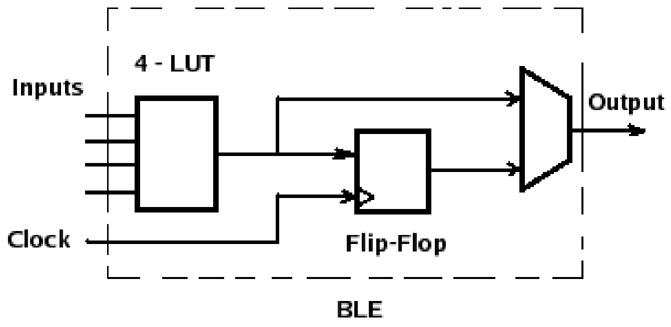


Figure 1.4: A simplified illustration of a Basic Logic Element (BLE).

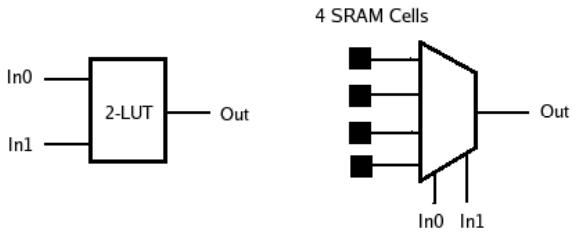


Figure 1.5: An illustration of a 2-input LUT.

designers must resort to and rely on Computer-Aided Design (CAD) tools. These tools play a critical role in delivering high-quality results when implementing a circuit via an FPGA. Since designs and devices continue to increase in size and capability, CAD tools must continually be improved in order to achieve high-quality results with reasonable computational effort.

Today, most FPGA vendors provide a fairly complete set of design tools that automatically transform design specification, entered either as a schematic or using a hardware description language (HDL), such as Verilog or VHDL, all the way down to a stream of "1"s or "0"s that program the FPGA chip during the configuration time. The main steps of the FPGA CAD flow are illustrated in Figure 1.7 and are described below.

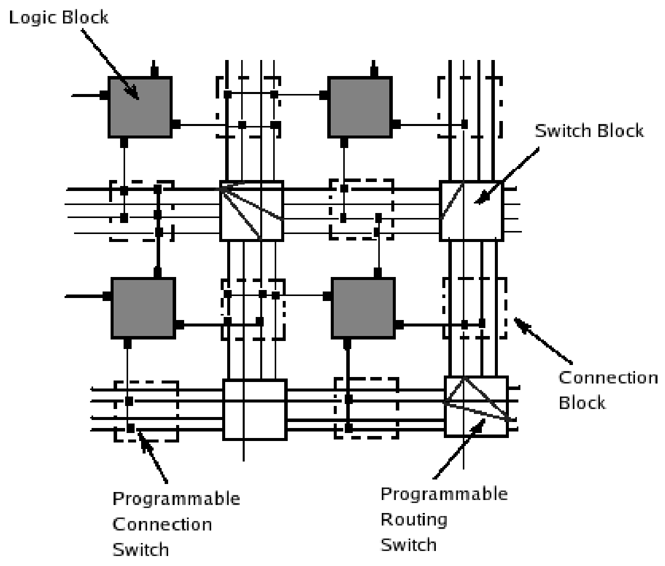


Figure 1.6: An illustration of the programmable routing architecture in an FPGA.

1.3.1 High-level Synthesis

High-level synthesis (HLS) is the process of intepreting an algorithmic description of a desired behaviour and creating hardware that implements the algorithm [32, 24, 22]. This process involves tasks such as identifying and instantiating datapaths and control logic (e.g., finite state machines) that implement the desired circuit behaviour. High-level synthesis also performs logic scheduling and resource binding subject to any provided architectural constraints. The output of HLS is typically a Register-Transfer Level (TRL) netlist consisting of combinational logic, registers and, finally, clock and control signals.

1.3.2 Logic Synthesis

Logic synthesis takes an RTL netlist and performs both technology independent and technology dependent netlist optimizations. Technology independent optimization is typically referred to as logic optimization and technology dependent optimization is typically referred to as technology mapping.

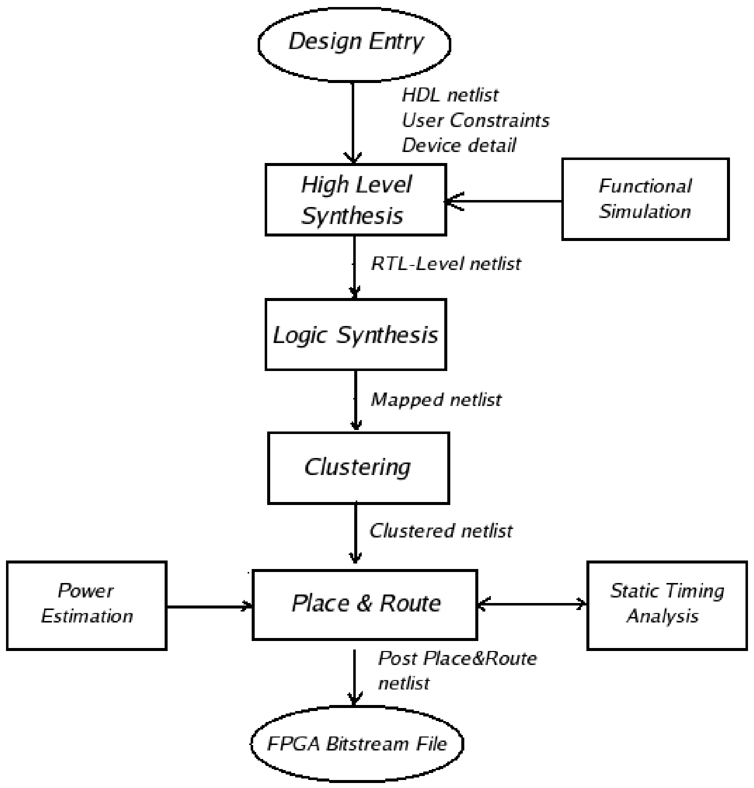


Figure 1.7: A typical FPGA CAD flow.

Logic optimization performs tasks such as the removal of redundant logic and simplification of logic functions in a technology independent fashion. It may also perform sequential optimizations including register retiming. Generally, the purpose of logic optimization is to generate an optimized netlist without the use of any technology dependent information which is useful for transforming the circuit netlist into something more suitable for the subsequent technology mapping phase.

Technology mapping is the phase in which a circuit netlist is converted, or mapped, into a netlist which is realizable within the target technology. For FPGA designs, this means conversion of combinational logic into k-input LUTs. Technology mapping is not only responsible for converting the netlist into something which is implementable within the given target technology, but it is also responsible for performing optimizations; i.e., minimizing circuit depth, area and any other important objectives. *The topic of this thesis involves FPGA technology mapping. Hence, technology mapping is described in greater detail later on in this thesis.*

1.3.3 Clustering

In the clustering phase, LUTs and registers are first packed into BLEs. Then, a set of CLBs are generated from the set of BLEs. The clustering phase has to consider specific parameters of the target FPGA architecture, such as the maximum number of BLEs inside of a CLB as well as other architectural limitations/constraints on the CLBs. The objective of the clustering algorithm is to minimize the number of logic blocks and/or minimize the delay. Certainly, commerical FPGA CAD tools perform clustering. Examples of academic clustering tools would include VPack or T-VPack [10].

1.3.4 Placement and Routing

In the placement phase, the packed logic blocks are assigned to specific physical logic blocks in the prefabricated two-dimensional array. In the case of island-style FPGAs, CLBs are moved around to determine the best location for each CLB. After the placement step, every CLB is assigned an X and Y location which represents its physical location. The procedure tries to minimize the delay along the critical path and enhance the routability of the resulting circuit [10].

The routing phase assigns the nets that connect CLBs in the placed netlist to specific segmented wires in the routing channels. Routing for FPGAs is complicated by the fact that the amount of routing resources in the FPGA device is fixed. Generally, routing can be divided into two steps: global routing and detailed routing [11, 12]. Global routing selects the channel for every net. Next, detailed routing assign each net to a specific wire in the channel. The objective of the routing algorithm is to minimize the delay along the critical path and avoid congestion in the FPGA routing resources.

Finally, the mapped, placed, and routed design generates the bit-stream file to program the logic and interconnect resources to implement the desired logic design on the target

FPGA device.

1.4 Contributions of the Thesis

The topic of this thesis involves the technology mapping stage of the FPGA CAD flow. In particular, this thesis focuses on post-technology mapping for area minimization subject to depth (i.e., delay) constraints. Hence, technology mapping is described in greater detail later in this thesis. Stated succinctly, the contributions of this thesis are as follows:

- An algorithm is proposed for performing area minimization on a technology mapped netlist. Essentially, the algorithm is based on the idea of circuit rewriting which replaces, or rewrites, cones of LUTs with alternative topologies of LUTs which serve to reduce circuit area (as measured in terms of the number of LUTs). Circuit functionality and delay are always preserved.

- The aforementioned algorithm requires the off-line creation of a library of topologies of LUTs. This turns out to be a potential time and space consuming task. The thesis proposes several strategies to improve the time and space efficiency of the off-line library generation.

- The thesis presents numerical results on a large set of design circuits to demonstrate the potential efficacy of the proposed algorithm in terms of its effectiveness at area minimization and in terms of runtime.

1.5 Summary

This introductory chapter has provided some background on FPGA architectures and the importance of the FPGA CAD flow. The remainder of this thesis is as follows. Chapter 2 provides additional background on the problem of FPGA technology mapping. Background on SAT-based Boolean matching as a means of determining if a particular topology of LUTs can implement a function is described in Chapter 3. The use of SAT-based Boolean matching as a post-technology mapping optimization algorithm is also described.

Thesis contributions are described in the following chapters. Chapter 4 describes the use of NPN equivalence classes for performing Boolean matching. Prior to the use of NPN equivalence classes for matching, it is shown that a topology of LUTs must be simulated and encoded. This procedure is shown to be complex in terms of both space and time. The idea of *essential bits* and *practical bits* are introduced as a means of reducing both the space and time complexity of the procedure. In Chapter 5, the overall algorithm for post-technology mapping area minimization based on NPN equivalence classes is proposed. Numerical results on a large suite of designs are also provided to demonstrate the efficacy of the proposed algorithm. Finally, the contributions of the thesis are described in Chapter 6. Future research possibilities are also described.

Chapter 2

Technology Mapping for LUT-based FPGAs

The purpose of FPGA technology mapping is to convert a netlist composed of simple logic gates into a netlist composed of k-input LUTs. The mapping procedure attempts to minimize some combination of area and delay. Other objectives, including power dissipation and routability may also be taken into account.

Modern FPGA technology mappers can be divided into functional mappers and structural mappers. Functional mappers perform Boolean decomposition of the logic functions of the nodes into sub-functions of limited support size realizable by individual LUTs [37]. Since functional mappers explore a larger solution space, they tend to be time-consuming, which limit their use to small designs. Structural mappers find a covering of the circuit graph with k-input subgraphs which correspond to k-input LUTs [17]. Due to both their efficiency and effectiveness, most modern state-of-the-art FPGA technology mappers are structural mappers.

Structural technology mappers typically proceed as follows: A circuit netlist is first converted into a simple network which consists of 2-input logic gates; e.g., a network such as an And-Inverter Graph (AIG) [17, 33]. This network is typically referred to as the subject graph. Subsequently, for mapping to an FPGA architecture consisting of k-input LUTs, all cuts of size k or less are computed for every node in the subject graph. This step is known as cut enumeration. Finally, the best cuts for a subset of nodes are selected to form k-input LUTs which cover all the gates in the subject graph. In performing the covering of the subject graph, the covering is done in order to minimize some combination of delay and area (or other objectives). Technology mapping for minimum circuit depth is known to be polynomial, but not for other objectives such as area. Hence, technology mapping is performed using a variety of heuristics during the covering of the subject graph.

The purpose of this chapter is provide background information on technology mapping. This includes cut enumeration and the different heuristics used by modern state-of-the-art technology mappers to select a set of cuts to cover the input subject graph.

2.1 Background

A combinational Boolean circuit can be represented as a directed acyclic graph (DAG) $G = (V, E)$ in which a node v represents a logic gate, and a directed edge e corresponds to a wire that connects two gates. A *primary input* (PI) is a node without incoming edges and a *primary output* (PO) is a node without outgoing edges. Inputs and outputs of flip-flops (FFs) can be treated as additional cases of POs and PIs. If there is a path from node v to node u, v is a *predecessor* of u and u is a *successor* of v. The topological order exists in the DAG, such that each node appears after all its predecessors and before all its successors in the ordering.

For any node v, *fanin(v)* denotes the set of nodes which are *fanins* of node v. Similarly, *fanout(v)* denotes the set of nodes which are *fanouts* of node v. A primary input node has no fanins and a primary output node has no fanouts. A node is *k-feasible* if $|fanin(v)| \leq k$. If every node in a graph is *k-feasible*, then the graph is *k-bounded.* A *subject graph* is typically the terminology reserved for the 2-bounded network which is used as input for the technology mapper. Subject graphs are typically represented as And-Inverter Graphs (AIGs) which are networks consisting of 2-input AND gates and inverters.

A cone of logic C_v, rooted at node v, is a sub-graph composed of v and some of its non-PI predecessors such that any node $u \in C_v$ has a path to v which is entirely included in C_v. We use *fanin(C_v)* to denote the set of nodes outside of C_v which drives nodes inside of C_v. The set of fanins to a cone is also known as a *cut* in the graph. There is a one-to-one correspondence between a cut and a cone. The notion of k-feasibility also applies to cones and cuts. A *fanin (fanout) cone* of node v is a cone that consists of all nodes reachable through the fanin (fanout) edges from the node v. A *maximum fanout free cone* (MFFC) of node v is a subset of the fanin cone, such that every path from a node in the subset to the POs passes through v. Hence, the MFFC of a node contains all the logic used by the node.

The *level* of a node v, denoted as $level(v)$, is the number of edges on the longest path from a PI to v. The level for a PI node is zero. Often, the terminology *level* and *depth* are used interchangeably. The network *depth* is the largest level of an internal node in the network. The depth and area of a technology mapped netlist is measured by the depth of the network and the number of LUts in the network, respectively.

2.2 Cut Enumeration

Cut enumeration is the process of determining all different cones (or cuts) of logic that can be used to implement the function at node v such that each cut is k-feasible. Cuts for a node v are created by using the cuts from its fanin nodes and combining them to form cuts for node v. An example of this is shown in Figure 2.1(a) where, to generate a cut for node c, two cuts from nodes a and b are duplicated and combined to form a larger cut that includes nodes a, b and c. Modern FPGAs consist of an array of k-input LUTs connected together through programmable interconnect. Since a k-input LUT can implement any function of fewer than k variables, in technology mapping it suffices to find all k-feasible cuts of each node v. The correspondence of k-input LUTs and k-feasible cuts is shown in Figure 2.1(b).

Let $u_1, u_2, \cdots, u_t$ denote the fanins of node v and let S_{u_i} denote the set of all k-feasible cuts for node u_i for $1 \leq i \leq t$. The set operation called *merge* is defined as follows [40]:

$$\begin{aligned} &merge(S_{u_1}, S_{u_2}, \ldots, S_{u_t}) = \\ &\quad \{s = s_1 \cup s_2 \cup \ldots s_t | s_i \in S_{u_i} \ and \parallel s \parallel \leq k\}. \end{aligned} \tag{2.1}$$

The set of k-feasible cuts for node v, denoted by S_v is equal to $merge(S_{u_1}, S_{u_2}, \ldots, S_{u_t}) \cup \{v^0\}$ where $\{v^0\}$ represents the trivial cut containing node v only. Hence, if all k-feasible cuts of the fanins of a node v are given, we can find all non-trivial k-feasible cuts of the

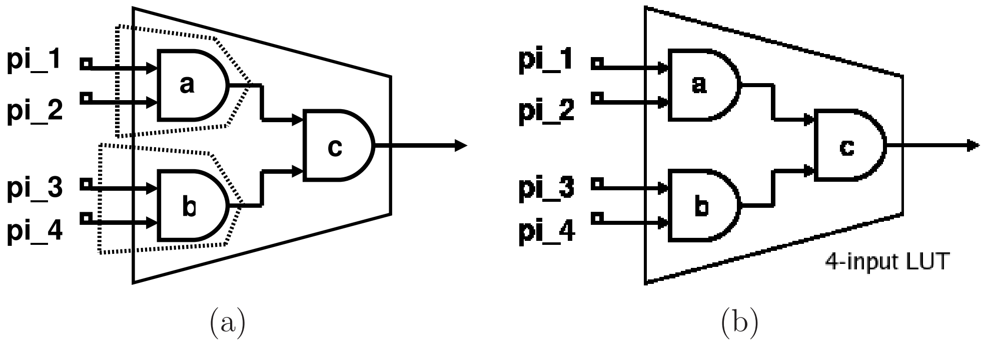


Figure 2.1: An illustration of cut enumeration: (a) A larger cut for node c derived from the cross product of the cutsets for nodes a and b; (b) a 4-LUT derived from the 4-input cut.

node v using the *merge* operation. This suggests that cut enumeration works by visiting each node in the network in topological order from PIs to POs and applying the *merge* operation to each node v. Any newly formed cuts that are not k-feasible are discarded during the process. Note that PIs only have a trivial cut. Cut enumeration as described is not scalable to larger cut sizes (e.g., $k > 7$) and for cuts containing a large degree of reconvergent paths.

In the process of merging the cut sets to form the resulting cut set, it is necessary to delete duplicated cuts and remove dominated cuts. Removing useless cuts before computing the cuts for the next node reduces the number of cut pairs considered without impacting the quality of mapping. In practice, the total number of cut pairs tried during the merging greatly exceeds the number of k-feasible cuts found. This makes checking k-feasibility of the unions of cut pairs, and testing duplication and dominance of individual cuts, the performance bottleneck of the cut enumeration.

The most serious problems in cut enumeration are the high time and space complexities. For a circuit with n nodes, the number of cuts of size k can be as large as $O(n^k)$ [21] and computing all cuts can take significant runtime. For small cuts (i.e. $k = 6$ or 7), the enumerative procedure is feasible to compute all cuts. But for large cuts, the enumerative procedure fails simply because there are too many cuts. Fortunately, modern FPGAs are built using LUTs with ≤ 6 inputs.

There are some heuristic approaches to make cut enumeration more manageable. In [21], this problem is addressed by selectively pruning cuts that seems to be useless. However, for large cut sizes, pruning tends to remove many valuable cuts in the following mapping solution. In [14], the notion of *cut factorization* is introduced where one enumerates both *global* and *local* cuts and uses these cuts to generate other cuts. It is possible to generate any cut from the factor cuts (i.e., *complete factorization*), but this is still an expensive operation. Partial factorization can also be used, but in this scheme there is no guarantee of generating all cuts from the factor cuts. In [16, 36], the notion of *prioity cuts* is introduced. In this scheme, only the most promising set of cuts for any given node are kept and propagated through the network. This reduces the runtime and memory requirements of cut enumeration, but can prevent some good cuts from being generated. Nevertheless, priority cuts are empirically demostrated to be useful for mapping to k-inputs LUTs. Given the cutset for a node v, each cut C_v is ranked according to several different criteria, such as (1) the logic depth of the cut, (2) area of the cut, (3) number of inputs of the cut $|fanin(v)|$. Since cuts are computed in topological order, the depth of a cut is computed easily by finding the maximum depth of the inputs of the cut. The area of the cut is a measure of the effectiveness of the cut in covering the underlying subject graph. Finally, the number of inputs of the cut is also a useful metric since those cuts which have good depth and large area, but only a small number of inputs, are likely to result in good mappings.

2.3 Timing Analysis

In order to optimize performance, technology mappers must have some concepts of circuit delay. Although arbitrary delay models can be used during technology mapping, it is most often the case that exact layout information is not available (i.e., placement and/or routing has not yet been performed). Hence, it is common for technology mappers to use a *unit delay* model to measure performance. In this model, each LUT contributes one unit of delay to any circuit path that passes through the LUT. Hence, under the unit delay assumption, the delay of a network becomes synonymous with circuit depth (or circuit level).

Under the unit delay model, and assuming that cut enumeration has been performed, it is possible to compute the optimal delay for any given node in the subject graph. Subsequently, the optimal delay for the mapped network can be computed; i.e., the best achievable circuit delay is known. The minimum delay at any node v, denoted by $arrival(v)$, can be calculated by

$$arrival(v) = min[max[arrival(u_i)] + 1], \forall\ C_v \in cutset(v), u_i \in fanin(C_v) \tag{2.2}$$

where C_v represents a cut in the cutset for node v and $arrival(u_i)$ is the arrival time on the fanin u_i of C_v. The calculation of arrival times is computed in topological order from the PIs towards to POs of the network. The worst arrival time at any of the POs represents the worst case delay of the entire circuit; i.e., the optimal (minimum) worst case delay of the circuit.

Given a selected set of cuts which cover the subject graph, the required time at any node v, denoted as $required(v)$, can be computed. The required time denotes the largest delay allowed at the node v in order to achieve the best network delay. Required times are computed in reverse topological order from POs backward toward the PIs of the network. The required time at any node v is computed by

$$required(v) = min[required(v), required(n) - 1], n \in fanout(v) \tag{2.3}$$

where n is a fanout of node v. Finally, the *slack* can be computed for each node in the technology mapped network. The slack for any node v in the mapped network, denoted by $slack(v)$ is computed in terms of its arrival and required times by

$$slack(v) = required(v) - arrival(v). \tag{2.4}$$

Slack gives an indication of how far away a node is from being considered as timing critical. Nodes with slack > 0 are not on the critical path; i.e., they are not timing critical. Nodes with slack ≤ 0 are timing critical nodes and are important for performance.

2.4 Practical Technology Mapping

As mentioned, modern FPGA technology mapping algorithms [17, 18, 15, 31, 35] are based on cut enumeration. This is because every k-input cut maps directly into a k-input LUT.

```
    Procedure: Technology Mapping
    Inputs: An And-Inverter netlist $N$ and LUT architecture $K$
    Outputs: A mapped LUT-based netlist $N'$
1   Perform cut enumeration and save all K-feasible cuts at each node;
2   Find a depth-optimal cut at each node and mark it as the representative;
3   Modify the representative cut at each node on the non-critial paths to save area;
4   Replace each representative cut with a corresponding LUT;
5   Return the netlist $N'$ of LUTs as the result of the final mapping;
```

Figure 2.2: The typical FPGA technology mapping flow.

Based on the previous descriptions of cut enumeration and timing analysis, it is possible to develop an intuitive approach for the objectives of any technology mapping solution. The overall flow of technology mapping is described in Figure 2.2. Specifically, the goal of any technology mapper is to select the "correct" set of cuts such that (1) the delay of the circuit is minimized (delay optimal) while (2) requiring a minimum number of cuts (area optimal) to cover the original network. Intuitively, selecting a set of cuts which yield a delay optimized solution is straightforward (select the delay minimizing cut for each node v from the cutset for node v). In fact, the depth-optimal technology mapping problem can be solved in polynomial time using a dynamic programming procedure known as FlowMap [17]. Conversely, the area minimization problem has been shown to be NP-hard for k-input LUTs of size $k \geq 3$ [23]. The selection of a set of cuts which minimizes area is not straightforward.

Thus, heuristics are necessary to solve the area-minimization problem. The area minimizing mapping procedure based on depth relaxation is shown in Figure 2.3. Before the area minimizing procedure, the new required time is computed on each node in reverse topological order. The node with slack larger than 0 is claimed to be on the non-critical path, and the representative cut on the node can be changed to minimize local mapping area without violating the timing constraints. As discussed in IMap [31] and ABC [36, 35], two complementary heuristic measures of area cost are applied as guidance to good practical results. The first heuristic *area flow* has a global view and selects logic cones with more shared logic. It can be computed in one pass over the network from the PIs to the POs. Area flow for the PIs is set to 0, and area flow at node v is:

$$AreaFlow(v) = [Area(v) + \sum_i AreaFlow(fanin_i(v))]/numFanouts(v) \qquad (2.5)$$

```
Procedure: AREA RECOVERY HEURISTICS
Inputs: A optimum-depth netlist, $N$
Returns: A mapped netlist with area recovery, $N'$
Update the required times from POs to PIs for the last mapping;
for each node $n$ in topological order from PIs to POs do
    for each cut $C$ at node $n$ do
        $C_r$ is the current representative cut;
        if $Arrival\_Time(C) <= Required\_Time(n)$ then
            if $AreaFlow(C) < AreaFlow(C_r)$ then
                $C_r = C$; continue;
            end if
            if $LocalArea(C) < LocalArea(C_r)$ and $AreaFlow(C) == AreaFlow(C_r)$ then
                $C_r = C$; continue;
            end if
        end if
    end do
end do
return $N'$;
```

Figure 2.3: Pseudocode for area recovery mapping procedure.

where $AreaFlow(v)$ is the number of LUTs needed to cover the representative cut at the node v. $fanin_i(v)$ is the i-th fanin of the representative cut at v, and $numFanouts(v)$ is the number of fanouts of node v in the last mapping solution. The second heuristic *local area* looks at the area added to the mapping by locally modifying the representative cut at a node. The local area is equal to the sum of the LUTs in the MFFC of the cut, i.e. the LUTs to be added to the mapping if the cut is selected as the representative cut. It has a local view by minimizing the area exactly at each node. In one pass of area recovery, area flow is used as the primary metric and local area as a secondary metric to break the tie. The heuristic area minimizing procedure will run several iterations until no area improvement happens. At last, one pass of LUT covering is done in reverse topological order. For each PO, the representative cut is chosen and a LUT is constructed in the mapped netlist to implement it. Then, recursively for each fanin node of the LUT, this procedure is repeated until the whole netlist is covered by LUTs.

2.5 Summary

This chapter has introduced the concept of cut enumeration and has described the typical flow of state-of-the-art structural FPGA technology mappers. These technology mappers are capable of producing delay optimized solutions given the structure of the provided subject graph. However, they make local decisions based on depth relaxation and various heuristic measures of area cost (e.g., area flow and exact local area evaluation) in order to minimize area. Further, because these technology mappers are structural in nature and cut enumeration is performed on the provided subject graph, the ability of the technology mappers is somwhat restricted by the structure of the provided subject graph. Hence, because of the limitations imposed based on the structure of the subject graph and due to the heuristic nature of the area minimization, it is quite likely that the technology mapped netlists can be further improved if additional post-processing is applied directly to the technology mapped netlist.

Chapter 3

SAT-Based Boolean Matching

As mentioned in Chapter 2, structural technology mappers suffer from two main issues with respect to area minimization. First, the results produced by structural mappers depend strongly on the structure of the initial subject graph. Second, these mappers make local choices when selecting a cut for any given node; choices are made based on either local or heuristic measures of area. Hence, the LUT netlists produced by structural mappers can be further optimized by post-technology mapping improvement techniques which target the minimization of area under delay constraints [33, 30, 43, 20, 26].

This chapter describes how LUT netlists can be further optimized through the use of Boolean matching. The Boolean matching problem is described. Subsequently, it is demonstrated how the Boolean matching problem can be solved via the solution of a Boolean Satisfiability (SAT) problem. These approaches are computationally demanding and too slow for practical purposes.

3.1 Boolean Matching

For the purposes of this thesis, the problem of Boolean matching can be stated as follows. *Can a logic function f of n variables, e.g., expressed in terms of its truth table, be implemented by some topologies of LUTs?* For example, Figure 3.1(a) shows a 3-input logic function $f = b(a + c)$. The question to be answered is whether or not this function can be implemented by the topology of LUTs illustrated in Figure 3.1(b). In this example, the

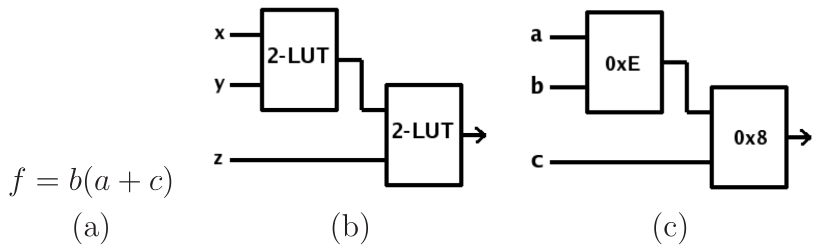


Figure 3.1: An example of Boolean matching: (a) A 3-input logic function f; (b) A topology of LUTs consisting of two 2-input LUTs; (c) An implementation of f.

function can be implemented as showing in Figure 3.1(c) by connecting the function inputs as shown and setting the configuration bits of the LUTs appropriately; other solutions are also possible and are obtained by permutation of the inputs and changing the values of the configuration bits for the LUTs.

Within the FPGA community, the problem of determining whether or not a logic function f can be implemented via a topology of LUTs has been formulated as a SAT problem. This is accomplished as follows: First, an expression representing the topology of LUTs is created in Conjunctive Normal Form (CNF). Second, this CNF expression is duplicated once for each row of the truth table of f. Finally, the conjoined CNF expression is passed to a SAT solver. If the SAT solver returns *false*, then the logic function can not be implemented by the topology of LUTs. On the other hand, if the SAT solver returns *true*, then the logic function f can be implemented by the topology of LUTs. Further, the solution to the SAT problem provides the proper permutation for connecting the inputs of the logic function and the proper configuration bits for the LUTs within the topology.

To illustrate this process, consider the problem of determine whether or not a logic function f with ≤ 3 inputs can be implemented by the LUT topology consisting of two 2-input LUTs shown in Figure 3.1(b). To create the CNF for the topology, it is necessary to break down the topology into individual circuit elements and compute the CNFs for each individual element. The individual CNFs can then combine to form the CNF for the entire topology. In order to account for input permutations, it is necessary to add *virtual multiplexers* at the inputs of the topology. These multiplexers do not exist in the topology and are "artifically" added into the SAT problem in order to support the permutation

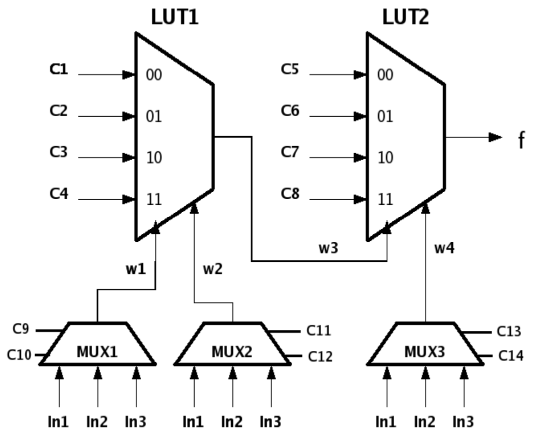


Figure 3.2: Illustrating of the CNF creation for a particular topology of LUTs. Virtual multiplexers are included to allow for input permutation.

of the logic function inputs. The topology consisting of two, 2-input LUTs in Figure 3.1 is illustrated in terms of its basic logic elements in Figure 3.2. The addition of virtual multiplexers to allow for input permutation is also illustrated in Figure 3.2.

In Figure 3.2, the CNF for LUT1, denoted by $G_{LUT1}(\vec{x}, \vec{C}, f)$, involves the two inputs $w_1w_2 = \vec{x}$, the configuration bits $C_1C_2C_3C_4 = \vec{C}$, and the output $w_3 = f$ and is given by

$$\begin{aligned} G_{LUT1} =& (w_1 + w_2 + \overline{C_1} + w_3) \cdot (w_1 + w_2 + C_1 + \overline{w_3}) \cdot \\ & (w_1 + \overline{w_2} + \overline{C_2} + w_3) \cdot (w_1 + \overline{w_2} + C_2 + \overline{w_3}) \cdot \\ & (\overline{w_1} + w_2 + \overline{C_3} + w_3) \cdot (\overline{w_1} + w_2 + C_3 + \overline{w_3}) \cdot \\ & (\overline{w_1} + \overline{w_2} + \overline{C_4} + w_3) \cdot (\overline{w_1} + \overline{w_2} + C_4 + \overline{w_3}) \end{aligned} \tag{3.1}$$

Similarily, The CNF for MUX1, denoted by $G_{MUX1}(\vec{x}, \vec{C}, f)$, involves inputs $i_1i_2i_3 = \vec{x}$, the configuration bits $C_9C_{10} = \vec{C}$, and output the $w_1 = f$.

$$\begin{aligned} G_{MUX1} =& (C_9 + C_{10} + \overline{i_1} + w_1) \cdot (C_9 + C_{10} + i_1 + \overline{w_1}) \cdot \\ & (C_9 + \overline{C_{10}} + \overline{i_2} + w_1) \cdot (C_9 + \overline{C_{10}} + i_2 + \overline{w_1}) \cdot \\ & (\overline{C_9} + C_{10} + \overline{i_3} + w_1) \cdot (\overline{C_9} + C_{10} + i_3 + \overline{w_1}) \cdot \\ & (\overline{C_9} + \overline{C_{10}}) \end{aligned} \tag{3.2}$$

As mentioned, since MUX1 is a virtual multiplexer, its configurations bits are used to determine which of the logic function inputs should be connected to w_1; the configuration bits are not required for the programming of any LUT.

Now combining these two equations according to Figure 3.2, along with similar equations for the other LUTs and virtual multiplexers, yields the CNF equation given by

$$\begin{aligned}G_{Total} = &G_{MUX1}(i_1 i_2 i_3, C_9 C_{10}, w_1)\cdot\\ &G_{MUX2}(i_1 i_2 i_3, C_{11} C_{12}, w_2)\cdot\\ &G_{MUX3}(i_1 i_2 i_3, C_{13} C_{14}, w_4)\cdot\\ &G_{LUT1}(w_1 w_2, C_1 C_2 C_3 C_4, w_3)\cdot\\ &G_{LUT2}(w_3 w_4, C_5 C_6 C_7 C_8, f)\end{aligned} \tag{3.3}$$

which represents the input for the entire LUT topology with inputs $i_1 i_2 i_3$ and output f.

In order to test whether function f could be implemented by the LUT configuration, G_{Total} would have to be duplicated for each of the possible 2^3 inputs (for an n-input function, the G_{Total} would need to be duplicated 2^n times). Each copy of G_{Total} corresponds to one specific row of the function's truth table and has appropriate values substituted for the input variables and the output variable in the CNF. The conjunction of the 2^3 duplicates of G_{Total} represent the CNF equation specific to the function f which is being tested against the generic topology of LUTs. This CNF equation can be fed into a SAT solver to determine whether or not the function f can be implemented by the topology.

The SAT-based approach for Boolean matching is complete and guarantees to find a match if it exists. It can be easily customized to different topologies of LUTs and therefore offers great flexibility. However, it suffers from excessive runtime due to high computational complexity even with improvements [43, 20, 26]. The complexity of SAT-based Boolean matching increases dramatically with the number of inputs in the topology; e.g., it can take seconds or minutes to solve a problem involving the matching of a 10-input logic function. With an imposed timing limit, the SAT-based Boolean matching procedure becomes incomplete with only a limited success rate; e.g., as shown in [20], only 50% Boolean matching problems are solved for 10-input Boolean functions under the timeout limit of 60 seconds.

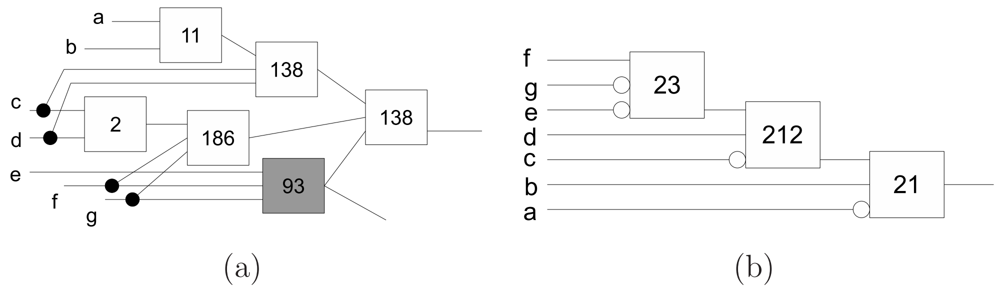


Figure 3.3: Using matching to reduce area: (a) A 7-input logic cone implementing a logic function f which can be implemented by (b) an alternative circuit.

3.2 Post-Technology Mapping Optimization

The procedure for SAT-based Boolean matching can be used to post-process technology mapped netlists to perform further area minimization without harming the delay of the mapped netlist. This is accomplished as follows. In turn, each node in the technology mapped netlist (i.e., a netlist of LUTs) can be examined. This examination can be done randomly or in some specific order such as in a topological order. For any selected node v, an n-input cone of logic C rooted at node v can be computed for some reasonable n (typically $n \leq 10$). Once the cone of logic is selected, it can be simulated to determine its logic function f. Subsequently, assuming a topology of LUTs is provided with $\geq n$ inputs, a SAT problem can be formulated to determine if the function f of the cone C can be implemented by the topology of LUTs. If f cannot be implemented, either another cone of logic is attempted for node v, or the algorithm proceeds to another node in mapped netlist. If, however, f can be implemented by the topology of LUTs, this topology provides an *alternative* circuit structure that can be used to replace the original cone of logic C without changing the function f. If the new circuit topology for implementation of f reduces the number of required LUTs in the network and does not cause the slack at node v to become negative, then the topology of LUTs is a suitable replacement for the cone of logic which reduces the total number of LUTs in the mapped network without impacting the worst-case network delay. An example of replacing a cone of logic with an alternative topology of LUTs is illustrated in Figure 3.3. In Figure 3.3(a), a 7-input cone of logic is identified in the technology mapped netlist. Here, the unsigned integer values inside of each LUT indicate the configuration bits of the LUT. The selected logic cone consists of 6 LUTs. Note that one of the LUTs in the logic cone has multiple fanouts; it cannot be removed by replacing the logic cone. In Figure 3.3(b), an alternative implementation for the logic cone is shown and was obtained by matching the logic function to a topology of three, 3-input LUTs connected serially. Here, the unsigned integers again represent the LUT configuration bits. The bubbles represent inversions which can be absorbed into the LUTs. This alternative topology requires only 3 LUTs. Hence, assuming that the delay at the output of the new topology does not worsen the worst-case delay of the network, the alternative circuit structure in Figure 3.3(b) can be used to replace the original implementation of the logic function in Figure 3.3(a) while removing two LUTs from the mapped netlist.

3.3 Summary

This chapter has described the Boolean matching problem and shown that it can be solved by formulating a SAT problem. Further, this chapter has described how Boolean matching can be used in an algorithm applied to post-technology mapping optimization. Although very general, the SAT-based approach to Boolean matching is rather time consuming which makes it less attractive and an actual optimization algorithm applied after technology mapping.

Chapter 4

NPN-based Boolean Matching

As discussed in Chapter 3, SAT-based Boolean matching is not feasible for practical applications due to the time complexities involved in formulating and solving the necessary SAT problem each time a function is matched. In this chapter, an alternative approach based on the use of precomputed libraries of implementable functions and NPN equivalence classes is proposed. The NPN-based Boolean matching approach finds matches through simple hash table lookups in a pre-computed library of implementable functions created a priori from the analysis of different topologies of LUTs. Since hash table lookups are fast, the resulting Boolean matching is fast enough for practical application. The proposed approach requires a library of topologies of LUTs and the explicit need to determine all of the implementable functions. The creation and storage of the library of implementable functions is time consuming and memory intensive; it is the potential inability to create a library that limits the practicality of the approach. However, if the library can be created, then the proposed method is fast and efficient for performing Boolean matching.

This chapter describes how a pre-computed library of implementable functions, created by the analysis of a set of topologies of LUTs, can be used along with the idea of NPN equivalence classes as an alternative to SAT-based Boolean matching. Some critical improvements on computation time and memory usage are proposed to make the a priori library creation more efficient and practical.

4.1 Matching Via Hash Table Lookups

Intuitively, SAT-based Boolean matching is an implicit way to answer the question whether or not a logic function f with $\leq k$ inputs can be implemented by a k-input topology of LUTs; i.e., given a topology of LUTs, all implementable functions with $\leq k$ inputs are *implicitly* represented by a set of Boolean clauses. Determining whether or not a logic function f is implementable involves tailoring the SAT problem to the particular function and performing a search (accomplished via the SAT solver) through the clauses for a solution.

However, there is an alternative way to answer the question of whether or not a k-input logic function f is implementable by a k-input topology of LUTs. Specifically, given a k-input topology of LUTs, all implementable k-input functions can be enumerated *explicitly* via a priori simulation and stored (e.g., via a hash table) for future reference. Consequently, the Boolean matching problem is simplified to the problem of looking to see if f is found in the hash table.

Unlike the SAT-based Boolean matching, the proposed alternative approach is only capable of matching a k-input logic function to a k-input topology of LUTs. This restriction stems from the use of hash table lookups and that everything in the proposed alternative approach is explicit rather than implicit (In SAT-based Boolean matching, it is the use of virtual multiplexers that allows a single variable to be connected to multiple inputs on the topology, c.f. Chapter 3. The use of virtual multiplexers is yet another example of the representation of implicit information). This restriction, however, is not a problem and is easily addressed by creating a *library of topologies of LUTs* in which each topology has a different number of inputs. The connection of a single logic function input to multiple inputs on a topology is accomplished by matching the function to the proper topology of LUTs. Without further detail, illustrations of the topologies of LUTs used in this thesis are provided in Appendix B. Topologies of LUTs obey the naming convention "LUTx_y_v", where "x" means a x-LUT architecture (i.e., the architecture consists of x-input LUTs), "y" represents the number of inputs of the topology, and "v" represents the version of the structure in the topology. For example, the topologies "LUT3_6_1" and "LUT3_6_4" are the names of two different 6-input topologies for an architecture which consists of 3-input LUTs.

```
Procedure: Enumeration of Implementable Logic Functions
Inputs: A LUT Topology $T$
Returns: A Hash Table, $H$.
1  for each set of configuration bits $(c_1, c_2, \ldots, c_n)$ in the topology $T$ do
2      Simulate the current logic function $f$ that is implemented by the topology $T$;
3      if $f$ does not exist in the hash table $H$ then
4          Insert $f$ and $(c_1, c_2, \ldots, c_n)$ into the hash table $H$;
5      end if
6  end do
7  return $H$;
```

Figure 4.1: Enumeration of implementable logic functions for a particular topology of LUTs using simulation.

Hence, the proposed alterative approach to Boolean matching can be described as follows for matching a k-input logic function f to a k-input topology of LUTs: (1) create a library of k-input topologies of LUTs suitable for the target architecture; (2) determine through simulation all logic functions implementable by each topology of LUTs and insert these functions into a hash table; (3) determine which, if any, k-input topologies are capable of implementing the logic function f through hash table lookups.

When simulating a set of topologies of LUTs, a hash table is built for each separate topology of LUTs. Each hash key corresponds to a logic function, and the associated hash value corresponds to a set of configuration bits of LUTs necessary to implement the simulated logic function. The pseudo-code of the exhaustive enumeration procedure for a topology composed of n LUTs is described in Figure 4.1. For each set of configuration bits $(c_1, c_2, \ldots, c_n)$, the given topology can be simulated to determine its current logic function f. If the logic function f does not exist in the hash table, it is inserted into the table, with the current set of configuration bits.

Obviously, in the proposed approach, the use of hash tables makes the matching of logic functions extremely fast when compared to SAT-based matching. Whereas SAT-based matching amortizes the matching time over all matching problems, the proposed approach pushes all computational effort to the a priori determination of all implementable functions. Hence, while the exhaustive simulation of a topology of LUTs can be very expensive both in terms of time and memory usage, it only needs to be performed once and can therefore be treated as an off-line process. After the exhaustive enumeration, the functions (along with configuration bit information) which are implementable are saved to data files. Prior to Boolean matching, these data files are loaded into memory.

However, the Boolean matching approach based on hash lookups proposed above is still facing several severe problems:

- The number of implementable logic functions in a topology is extremely large, due to the flexibility of LUTs. It results in consuming a large portion of memory to keep hash tables online.
- The enumeration of all implementable logic functions for topologies with large number of inputs is time consuming. Runtime improvement is indispensable for pre-computation of some topologies.
- The pre-computed library is requested to contain a large number of topologies with different structure, which has a large impact on both computation time and memory usage.

In the following sections, these problems are discussed in detail and relevant solutions are proposed.

4.2 NPN Equivalence

The issue of memory usage may be addressed by relying on the concept of NPN equivalence classes to reduce the size of the hash tables required to store those functions implementable by a particular topology of LUTs. Any k-input logic function can be represented through its NPN equivalence class in a canonical way. Specifically, two Boolean functions f and g are said to belong to the same NPN equivalence class if f can be derived from g (or visa-versa) by negating (N) and permuting (P) inputs and negating (N) the output [9]. For example, functions $f = ab + cd$ and $g = ac + bd$ are NPN equivalent because swapping b and c makes them functionally equivalent. However, functions $f = ab + cd$ and $g = abc + d$ are not NPN equivalent because any permuting and complementing of variables can not make them functionally equivalent. A canonical form, denoted as the NPN equivalence class f_{npn} can be used to check NPN equivalence between functions f and g by testing under all input permutations/complementations and output complementation.

```
Procedure: NPN EQUIVALENCE CLASS ENUMERATION
Inputs: A LUT Topology $T$
Returns: A Hash Table, $H$.
1  for each set of configuration bits $(c_1, c_2, \ldots, c_n)$ of LUTs in the topology $T$ do
2      Simulate the current logic function $f$ that is implemented by the topology $T$;
3      Obtain its NPN equivalence class $f_{npn}$ using a NPN encoder
4      if $f_{npn}$ does not exist in the hash table then
5          Insert $f_{npn}$, configuration bits, and complementation/permutation information into the hash table
6      end if
7  end do
8  return $H$;
```

Figure 4.2: Enumeration of implementable logic functions for a particular topology of LUTs using simulation. Only one representative logic function is recorded for each NPN equivalence class.

When simulating a topology of LUTs, rather than inserting all implementable logic functions into a hash table, it is only necessary to store one representative logic function for each NPN equivalence class found during simulation. This requires an extra NPN encoding step in the procedure for the simulation of a topology of LUTs. Specifically, for each set of configuration bits $(c_1, c_2, \ldots, c_n)$ of LUTs, the topology is simulated to determine the current logic function f. This is followed by an efficient NPN encoding step [2, 13] to obtain the NPN equivalence class, f_{npn}, of f. If the NPN equivalence class f_{npn} does not exist in the hash table, it is inserted into the table, with the current set of configuration bits and the specific permutations/complementations on inputs and output provided by the NPN encoder as its associate hash value. Thus, all NPN equivalence classes are enumerated after going through all sets of configuration bits of LUTs. The pseudo-code of the enumeration of NPN equivalence classes for a given topology is described in Figure 4.2. The compressed hash table with all NPN equivalence classes are saved into a data file that corresponds to the name of the given topology. In this way, the number of hash keys maintained in the hash table significantly decreases. For example, for a k-input topology of LUTs, there are as many as 2^{2^k} logic functions possible. However, the number of NPN equivalence classes can be significantly less. As one specific example, there are 65536 possible logic functions for a simple 4-input topology of LUTs. However, there are at most 222 NPN equivalence classes for 4-input logic functions [39].

While the use of NPN equivalence classes effectively reduces the memory usage required to store the hash tables, it does affect the matching process itself. Specifically, given a function f, it is no longer "just sufficient" to look for the function f in the hash table of implementable functions; this is because the hash table only stores equivalence class representatives. Rather, to determine if some arbitrary logic function f is implementable, it is now (additionally) necessary to encode the function and compute the necessary inversions and/or permutations to arrive at a matching for the function. Finally, it is important to note that the permutations and/or inversions do not increase the complexity of the match in terms of the number of LUTs in the topology. Permutations are handled simply by connecting signals as is appropriate. Inversions, rather than creating explicit inverters in the circuit, are simply absorbed into the LUTs within the topology (which corresponds to simple and straight-forward changes to the LUT configuration bits).

Numerical results are presented later in this chapter to demonstrate that the use of NPN equivalence classes results in fairly compact encoding of the logic functions implementable by different topologies of LUTs.

4.3 Essential Bits

Although the use of NPN equivalence classes addresses the potential storage issues, it does not address the runtime implications of determining all of the logic functions implementable by a topology of LUTs. The enumeration of all NPN equivalence classes is time consuming, due to the heavy computation tasks of simulating a topology for a particular set of configuration bits and NPN encoding the resulting simulated function for each set of configuration bits. The complexity of enumerating the implementable logic functions for a topology of LUTs increases dramatically with both the number of topology inputs, the number of LUTs in the topology and the number of LUT inputs. For example, as shown in Table 4.3, it takes about 20 hours to enumerate and encode all implementable logic functions for a 7-input topology composed of three 3-input LUTs (c.f. topology "LUT3_7_1" in Table 4.3 and in Appendix B).

However, during the enumeration of the topology, it is possible that certain values of the configuration bits for certain LUTs can be *skipped* and those bits which can be skipped

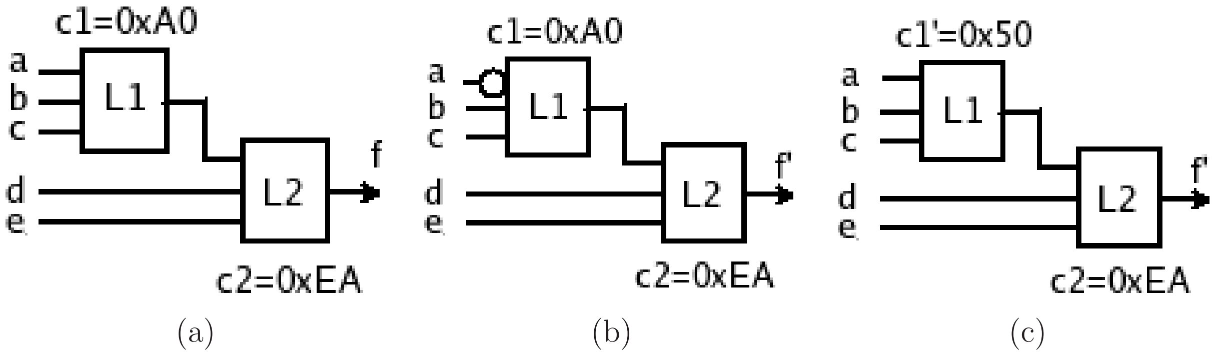


Figure 4.3: Three NPN equivalent topologies with complementation: (a) Topology $\{f, c_1, c_2\}$; (b) Topology $\{f', c_1, c_2\}$; (c) Topology $\{f', c_1', c_2\}$

can be determined a priori. Of course, skipping many configuration bits is beneficial, because it will reduce the total amount of simulation and NPN encoding required during the enumeration of the topology to find the implementable functions. Those configuration bits which *cannot* be skipped during enumeration are referred to as the *essential bits* while those bits which can be skipped are referred to as *redundant bits*.

The question arises as to which bits are essential bits and which bits are redundant for a particular LUT in any given topology. Redundant and essential bits follow from the definition of NPN equivalence; based on the definition of NPN equivalence, it is possible to know a priori that two *different* logic functions implemented by two different sets of configuration bits will lead to the same NPN equivalence class, *without knowing which particular equivalence class*. Thus, since we only need to store one representative for any particular equivalence class, only one of the two functions needs to be simulated and NPN encoded, thus reducing the overall computational effort required to enumerate the topology. Further, skipping configuration bits in the fashion described will *yield the same results* as performing full enumeration of the topology (i.e., not skipping any bits).

A simple example is used to illustrate the methodology of determining which configuration bits can be skipped for each LUT in a topology. Consider the 5-input topology consisting of two 3-input LUTs illustrated in Figure 4.3(a). The configuration bits for these two 3-input LUTs named L_1 and L_2 are denoted as c_1 and c_2, respectively. The

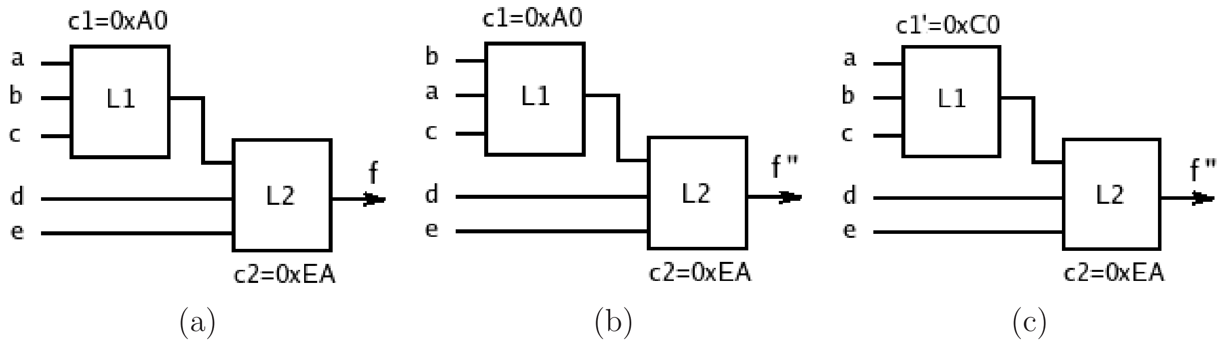


Figure 4.4: Three NPN equivalent topologies with permutation: (a) Topology $\{f, c_1, c_2\}$; (b) Topology $\{f'', c_1, c_2\}$; (c) Topology $\{f'', {c_1}', c_2\}$

logic function implemented by the above 5-input topology is denoted as f. Assume that configuration bit $c_1 = $ 0xA0 and $c_2 = $ 0xEA. This yields a logic function $f = abc + a\bar{b}c + de$. In Figure 4.3(b), an inverter represented as a bubble is manually added on the first input pin a of L_1. In this case, the implementable logic function, denoted as f', is given by $f' = \bar{a}bc + \overline{ab}c + de$. According to the concept of NPN equivalence, f and f' are NPN equivalent since one is obtained from the other by inversion of an input. Hence, the NPN encoding of both functions would lead to the same NPN equivalence class. Finally, in Figure 4.3(c), the inverter is absorbed into into the LUT L_1, which modify the configuration bit of L_1 from c_1 to ${c_1}'$ with ${c_1}' = $ 0x50. Since two logic functions f and f' belong to the same NPN equivalence class, it is *not necessary* to simulate and NPN encode both the circuit shown in Figure 4.3(a) and Figure 4.3(c); it is guaranteed that none of the NPN equivalence classes will be lost if one of $\{c_1, c_2\}$ or $\{{c_1}', c_2\}$ is skipped during the enumeration of the topology.

Similarly, input permutation and output complementation can also be used to find out the sets of configuration bits which implement NPN equivalent logic functions. Consider Figure 4.4(a) and Figure 4.4(b) where the first two inputs pins a and b are swapped. The logic function in Figure 4.4(b), denoted by f'', has logic function $f'' = abc + \bar{a}bc + de$ and is NPN equivalent to f in Figure 4.4(a). In Figure 4.4(c), to absorb the permutation of a and b into LUT L_1, the configuration bit c_1 of L_1 changes from c_1 to ${c_1}''$ (${c_1}'' = $ 0xC0), so as to maintain the same logic function f''. Thus, the configuration bits $\{{c_1}'', c_2\}$ can

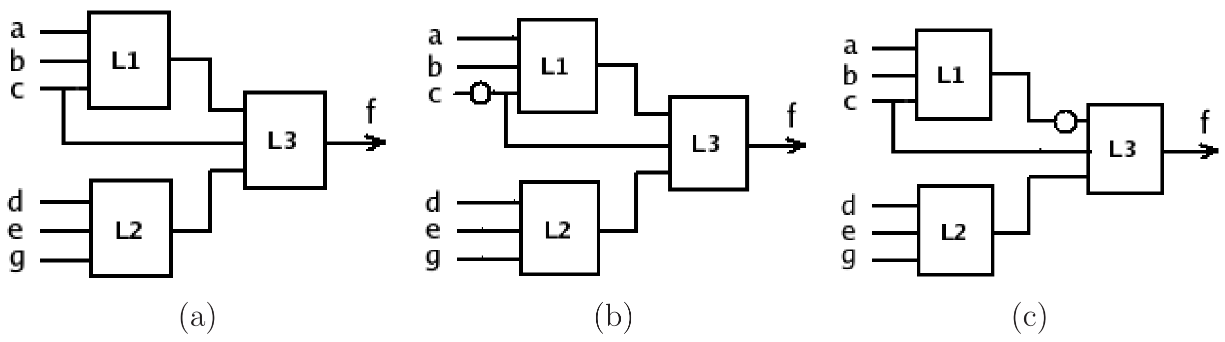


Figure 4.5: A 6-input topology of LUTs illutrating situations where complementation is not permitted to find essential and redundant bits; (a) Original topology; (b) Complementation not allowed on bridged inputs; (c) Complementation not allowed on internal inputs.

also be skipped during the enumeration of the topology as long as the configuration bits $\{c_1, c_2\}$ are not skipped. In summary, for the preceding examples, $c_1{}'$ and $c_1{}''$ are defined as *redundant bits* for LUT L_1, while c_1 (c_1 = 0xA0) is defined as the *essential bit* of $c_1{}'$ ($c_1{}'$ = 0x50) and $c_1{}''$ ($c_1{}''$ = 0xC0) for LUT L_1.

As discussed above, the simulation and NPN encoding steps must only applied on its essential bits and redundant bits can be skipped. Before the simulation and NPN encoding steps, another procedure is required to enumerate all essential bits for each LUT in the given topology. It needs to be mentioned that the procedure has to be modified for each LUT in a specific topology structure, because not every input or output pin is allowed for complementation and permutation. An example of a 6-input topology of 3-LUTs is shown in Figure 4.5(a). First, bridged input pins are not allowed to have complementation or permutation, as shown in Figure 4.5(b). Second, the internal pins connecting two LUTs are not allowed to have complementation or permutation, as shown in Figure 4.5(c). In this case, complementation and permutation are only allowed to be on the input pins a and b of LUT L_1, the input pins d, e and f of LUT L_2, and the output pin of LUT $L3$.

The pseudo-code of the enumeration of NPN equivalence classes using essential bits for a given topology is described in Figure 4.6. Numerical results demonstrating the benefits of essential bits in reducing the amount of computational effort required to enumerate topologies of LUTs are presented later in the chapter.

```
Procedure: NPN Equivalence Class Enumeration Using Essential Bits
Inputs: A LUT Topology $T$
Returns: A Hash Table, $H$.
1  Enumerate all essential bits for each LUT in the topology $T$;
2  for each set of essential configuration bits $(c_1, c_2, \ldots, c_n)$ of LUTs in the topology $T$ do
3      Simulate the current logic function $f$ that is implemented by the topology $T$;
4      Obtain its NPN equivalence class $f_{npn}$ using a NPN encoder;
5      if $f_{npn}$ does not exist in the hash table then
6          Insert $f_{npn}$, configuration bits, and complementation/permutation information into
7          the hash table;
8      end if
9  end do
10 return $H$;
```

Figure 4.6: Enumeration of implementable logic functions for a particular topology of LUTs using essential bits

4.4 Practical Bits

The NPN-based approach for Boolean matching is *complete* and guarantees to find a match if it exists, as long as the enumeration of NPN equivalence classes is complete. However, the enumeration of NPN equivalence classes still suffers excessive runtime using essential bits. Based on the practical observation, some values of the configuration bits for certain LUTs rarely show up in practical circuit designs, due to the large flexibility of LUTs. Hence, certain values of the configuration bits can also be skipped, and it further reduces the total amount of simulation and NPN encoding required during the enumeration of the topology to find the implementable functions, at the expense of some implementable functions missed. As a result, the NPN-based Boolean matching approach becomes *incomplete*.

The popularity of a configuration bit refers to the frequency of its appearance in practical circuits. Certain configuration bits are denoted as *practical bits* when their popularities are above a particular threshold value. To find out appropriate practical bits, the following work is done on a large set of practical circuits. First, all k-feasible ($k = 3$ or 4) cuts are computed on the AIG of each practical circuit. Next, each k-feasible cut is simulated to determine its implementable logic functions. Obviously, the truth table of the logic function corresponds to the configuration bit of a k-input LUT. Last, all configuration bits of LUTs are collected from the set of practical circuits. The popularity of a particular configuration bit is calculated using the number of its appearance divided by the total number of appearance for all configuration bits. The threshold value for picking up practical bits has to be chosen carefully. If the threshold moves up, more configuration bits can be skipped and more possible implementable functions will be missed during enumeration. For the 3-input LUT architecture, the threshold value is set as 0.00%, 0.01%, 0.02%, 0.03%, 0.04% and 0.05% individually. For the 4-input LUT architecture, the threshold value is set as 0.0%, 0.1% and 0.01% individually. The numbers of practical bits whose popularities above different threshold values are shown in the table 4.1 and table 4.2.

Table 4.1: The number of practical bits of 3-input LUTs

3-input LUT architecture						
Popularity	0.00%	0.01%	0.02%	0.03%	0.04%	0.05%
# of Practical Bits	243	126	111	102	93	84

Table 4.2: The number of practical bits of 4-input LUTs

4-input LUT architecture			
Popularity	0.00%	0.01%	0.1%
# of Practical Bits	8376	774	140

It is clear that the use of both essential bits and practical bits can significantly reduce computation time and memory usage further. The practical bits are collected and saved in a data file. During the enumeration for topologies of LUTs, practical bits are loaded into memory. The essential bits for each LUT are enumerated on the basis of its practical bits, instead of all its configuration bits. Hence, less essential bits are necessary to be fed into the following simulation and NPN encoding steps. However, some valuable NPN equivalence classes are possibly missed and it leads to the incomplete Boolean matching. The pseudo-code of the enumeration of NPN equivalence classes using essential bits for a given topology is described in Figure 4.7. The efficacy of practical bits is presented later in the chapter, which shows that the use of practical bits significantly speeds up the simulation and NPN encoding steps, and also results in loss of some NPN equivalence classes.

```
Procedure: NPN EQUIVALENCE CLASS ENUMERATION USING ESSENTIAL AND PRACTICAL BITS
Inputs: A LUT Topology $T$
Returns: A Hash Table, $H$.
Load the practical bits for the LUTs in the topology;
Enumerate all essential bits on the basis of practical bits for each LUT in the topology $T$;
for each set of essential configuration bits $(c_1, c_2, \ldots, c_n)$ of LUTs in the topology $T$ do
    Simulate the current logic function $f$ that is implemented by the topology $T$;
    Obtain its NPN equivalence class $f_{npn}$ using a NPN encoder;
    if $f_{npn}$ does not exist in the hash table then
        Insert $f_{npn}$, configuration bits, and complementation/permutation information into
        the hash table;
    end if
end do
return $H$;
```

Figure 4.7: Enumeration of implementable logic functions for a particular topology of LUTs using essential and practical bits

4.5 Numerical Results

The pre-computation of a library of topologies is executed on an Intel D920 machine running the Linux operating system. The numerical result is shown in the table 4.3 and 4.4. In these two tables, "TO" means that the enumeration of NPN equivalence classes exceeds the 48 hours timeout limit. In this case, the enumeration is terminated before completion, so the correspondent number of partial NPN equivalence classes is labelled as "N/A".

The columns "Baseline", "Essential Bits" and "Essential and Practical Bits" show the computation time for enumeration of NPN equivalence classes and the number of resulting NPN equivalence classes using all sets of configuration bits exhaustively, sets of essential bits on the basis of all configuration bits, and sets of essential bits on the basis of all practical bits. Under the condition of "Essential and Practical Bits", the practical bits belong to the group of configuration bits with popularities larger than 0.03%. From the experimental result, the same number of NPN equivalence classes are obtained between using the sets of configuration bits directly and the sets of essential bits on the basis of all configuration bits. It is obvious that a significant improvement in computation time are achieved using essential and practical bits. For example, almost $1700X$ speedup is obtained for the topology "LUT3_7_2". The enumeration for three 9-input topologies of

Table 4.3: Numerical Results for Topologies of 3-input LUTs

	Baseline		**Essential Bits**		**Essential and Practical Bits**	
Topology	**NPN classes**	**Time**	**NPN classes**	**Time**	**NPN classes**	**Time**
LUT3_5_1	255	16.5 sec	255	0.3 sec	88	0.1 sec
LUT3_6_1	1010	24.7 min	1010	18.1 sec	435	4.6 sec
LUT3_6_2	769	25.0 min	769	12.4 sec	332	3.1 sec
LUT3_6_3	845	23.9 min	845	12.2 sec	319	2.4 sec
LUT3_6_4	999	23.8 min	999	16.0 sec	407	3.8 sec
LUT3_6_5	299	23.9 min	299	6.8 sec	91	1.4 sec
LUT3_6_6	3096	6.9 hrs	3096	6.4 min	1025	32.4 sec
LUT3_6_7	9200	6.2 hrs	9200	5.2 min	1937	27.4 sec
LUT3_6_8	3421	6.2 hrs	3421	5.3 min	1025	25.8 sec
LUT3_6_9	7644	6.0 hrs	7644	5.2 min	1652	22.3 sec
LUT3_6_10	3067	6.0 hrs	3067	3.8 min	937	17.4 sec
LUT3_7_1	5610	22.1 hrs	5610	4.1 min	1197	59.4 sec
LUT3_7_2	3081	17.9 hrs	3081	3.8 min	605	36.6 sec
LUT3_9_1	N/A	TO	N/A	TO	2015	5.00 hrs
LUT3_9_2	N/A	TO	N/A	TO	5367	24.0 hrs
LUT3_9_3	N/A	TO	N/A	TO	10197	30.0 hrs
LUT3_9_4	N/A	TO	N/A	TO	14682	6.70 hrs

Table 4.4: Numerical Results for Topologies of 4-input LUTs

	Baseline		**Essential Bits**		**Essential and Practical Bits**	
Topology	**NPN classes**	**Time**	**NPN classes**	**Time**	**NPN classes**	**Time**
LUT4_5_1	1705	10.9 min	1705	4.35 sec	231	0.52 sec
LUT4_5_2	810	11.5 min	810	1.87 sec	155	0.26 sec
LUT4_5_3	255	43.1 sec	255	0.60 sec	88	0.11 sec
LUT4_5_4	N/A	TO	12661	3.03 min	908	4.43 sec
LUT4_5_5	N/A	TO	6599	33.0 sec	772	4.00 sec
LUT4_5_6	N/A	TO	N/A	TO	6474	5.11 min
LUT4_6_1	N/A	TO	9744	5.18 hrs	692	2.31 sec
LUT4_6_2	N/A	TO	5418	6.00 hrs	642	2.28 sec
LUT4_6_3	N/A	TO	N/A	TO	10655	90.9 sec
LUT4_7_1	N/A	TO	N/A	TO	5050	63.0 sec
LUT4_10_1	N/A	TO	N/A	TO	N/A	TO
LUT4_10_2	N/A	TO	N/A	TO	N/A	TO

3-input LUTs and two topologies of 4-input LUTs is allowed to be complete before the timeout limit, although the enumeration is still terminated for the two 10-input topologies of 4-input LUTs.

In conclusion, essential bits offer an impressive solution to runtime speedup, and practical bits suggest tradeoff between CPU time and the number of NPN equivalence classes. The aforementioned improvements make the pre-computation of a library of topologies practical for the 3-input LUT architecture, however it is still not practical for the 4-input LUT architecture. To address this issue and the third problem mentioned in section 4.1, a modified NPN-based Boolean matching is proposed in the next section.

4.6 Functional Decomposition

The SAT-based Boolean matching approach is flexible, and it can be customized to different topologies of LUTs. Given a k-input function with a given n-input topology of LUTs ($k \leq n$), it is guaranteed to find a match if it exists. However, the NPN-based approach is limited to match a k-input function with a k-input topology of LUTs. As a result, the pre-computed library is required to include a large number of topologies with different structure, due to the existence of bridged inputs and flexible choice of k-input LUTs. The number of k-input topologies with different structure is shown in the table 4.5. The large number of topologies with different structure has a large impact on computation time and

Table 4.5: Number of k-input topologies with different structure

k-variable LUT topology	**# different topologies**
5-input topology of 3-LUTs	**1**
6-input topology of 3-LUTs	**10**
7-input topology of 3-LUTs	**2**
8-input topology of 3-LUTs	**22**
9-input topology of 3-LUTs	**4**
5-input topology of 4-LUTs	**5**
6-input topology of 4-LUTs	**3**
7-input topology of 4-LUTs	**1**
8-input topology of 4-LUTs	**27**
9-input topology of 4-LUTs	**10**
10-input topology of 4-LUTs	**2**

memory consumption. For example, 8-input topologies of 3-input LUTs have 22 different structures. Not only the runtime of enumerating NPN equivalence classes is excessive for all these 22 topologies, but also a large portion of memory allocation is required when hash tables in these topologies are loaded priori to the online matching. Meanwhile, as described above, the number of NPN equivalence classes is still too large for 10-input topologies of 4-input LUTs, even though the enumeration is incomplete using practical bits. To alleviate these problems, a NPN-based Boolean matching in conjunction with functional decomposition is proposed.

In order to find a match for a 10-input logic function, the aforementioned NPN-based Boolean matching is looking for some 10-input topologies of 4-input LUTs that can implement the NPN equivalence class of the given logic function. Instead of straightforward hash lookups, functional decomposition is applied to pre-process the logic function. First, the 10-input function can be decomposed into a 4-input function and a 7-input function. It is obvious that any 4-input function is implementable in a single 4-input LUT. Thus, the Boolean matching problem is simplified from a 10-input function to a smaller 7-input function.

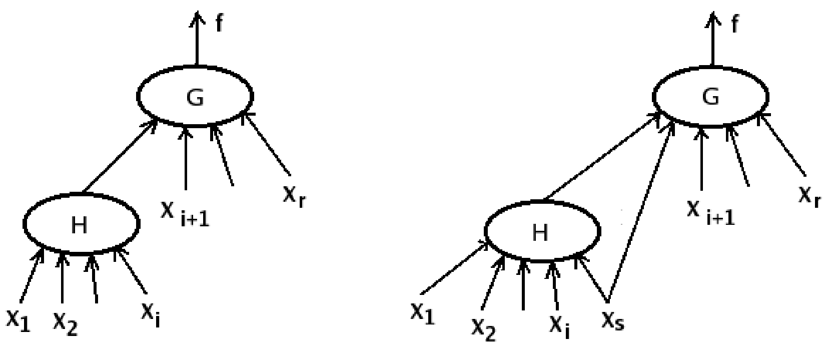
In general, a functional decomposition is of the form:

$$f(x_1, \ldots, x_r) = G(H_1(x_1, \ldots, x_i), \ldots, H_m(x_1, \ldots, x_i), x_j, \ldots, x_r) \tag{4.1}$$

where $i, j, m \leq r$ and $0 < i < j - 1$. Intuitively, this is a procedure of encoding the first $j - 1$ variables using m new variables. Therefore the functions $H_1, \ldots, H_m$ are referred as the *encoding functions*, and g as the *base function*. If $i = j - 1$, it is called a *disjunctive decomposition*, in which variables $x_1, \ldots, x_i$ form the *bound set*, and $x_j, \ldots, x_r$ form the *free set*; otherwise it is a *non-disjunctive decomposition*. If $m = 1$, it is called a *simple decomposition*; otherwise it is a *complex decomposition*. If $m < j - 1$, g has fewer variables than f, the decomposition is *nontrivial*; otherwise it is *trivial* [17].

For the Boolean matching problem, the functional decomposition is based on the classical *Ashenhurst decomposition*. Ashenhurst decomposition solves the simple disjunctive decomposition problem, as shown in Figure 4.8(a). Thus, the equation 4.1 is simplified as:

$$f(x_1, \ldots, x_r) = G(H(x_1, \ldots, x_i), x_{i+1}, \ldots, x_r) \tag{4.2}$$



(a) disjunctive (b) nondisjunctive

Figure 4.8: Ashenhurst decomposition

It uses a *partition matrix*, in the form of a 2-D truth table, for a given variable partitioning of bound set $B = x_1, \ldots, x_i$ and free set $F = x_{i+1}, \ldots, x_r$. Each column corresponds to one possible assignment (i.e. a minterm) of the bound set variables, and each row corresponds to one possible assignment of the free set variables. The partition matrix implies a simple disjunctive decomposition if and only if there are at most two different distinct column (thus the bound set variables can be encoded into one bit, using one function). To have a nontrivial decomposition, the size of the bound set must be at least two. The derivation of the base and encoding functions is easy: the encoding function $H(B)$ can be defined by the bound set minterms corresponding to the columns of one pattern (which is equivalent to assigning $H = 1$ for these columns); the base function $g(H, F)$ can be defined by the compressed truth table obtained after merging the identical columns and assigning the value of H for each column. Assuming to decompose a 10-input logic function into a 4-input function and a 7-input function, the truth table of the logic function H corresponds to the configuration bit of a 4-input LUT, and the other logic function G is used to find a match in the pre-computed library. The functional decomposition is also computationally complex. For the case above, there are $\binom{10}{4}$ different way for decomposition. Fortunately, functional decomposition is only applied on the function once, thus the complexity of decomposition is not an issue.

It is important that Ashenhurst decomposition can also be modified to handle non-disjoint decomposition. To make the decomposition simple and efficient, only one common

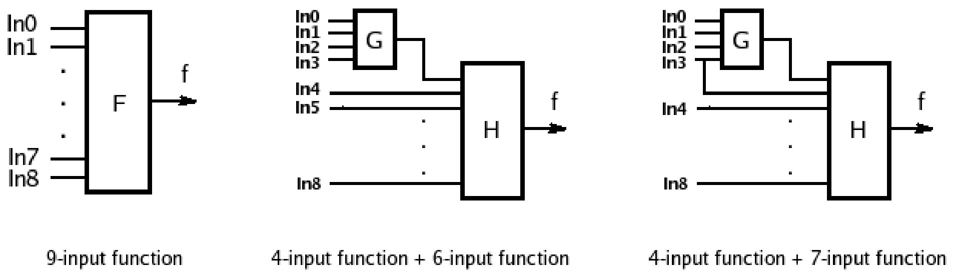


Figure 4.9: Functional decomposition of a 9-input Boolean function

variable is allowed to be shared between the bound set and the free set, as shown in Figure 4.8(b). Thus, equation 4.2 can be modified as:

$$f(x_1, \ldots, x_r) = G(H(x_1, \ldots, x_s, \ldots, x_i), x_s, x_{i+1}, \ldots, x_r) \tag{4.3}$$

where, x_s exists in both the bound set and the free set. The existence of common variable allows to explore larger solution space, making it possible to handle matching problems for those topologies with bridged inputs.. For example, a 9-input logic function can be decomposed into a 4-input logic function and a 6-input logic function, where no common variable exists between these two functions, or it can be decomposed into a 4-input logic function and a 7-input logic function, where one common variable is shared between these two logic functions, as shown in Figure 4.9. The Table 4.6 shows a full list of possible functional decomposition for all k-input functions ($8 \leq k \leq 10$).

In conclusion, for the 3-input LUT architecture, the straightforward NPN-based Boolean matching is feasible but limited to an incomplete appraoch. Also, many different topologies are required for simulation and NPN encoding; for the 4-input LUT architecture, the NPN-based Boolean matching is not feasible for k-input ($k > 7$) logic functions. The Ashenhurst decomposition in conjunction with NPN equivalence addresses the problem. Its benefits are (1) only k-input topologies ($5 \leq k \leq 7$) are necessary to exist in the library, which saves a lot of computation time and memory usage; (2) for the 3-input LUT architecture, the NPN equivalence classes for k-input topologies ($5 \leq k \leq 9$) can be enumerated only using essential bits in the timeout limit, thus the Boolean matching will be complete. (3) for the 4-input LUT architecture, the enumeration of NPN equivalence classes for k-

Table 4.6: A list of decompositions for k-input logic functions

k-input topology	**Architecture**	**Encoding Function (H)**	**Base Function (G)**
8-input topology	3-LUT	2-input function	7-input function
8-input topology	3-LUT	3-input function	6-input function
8-input topology	3-LUT	3-input function	7-input function
8-input topology	4-LUT	2-input function	7-input function
8-input topology	4-LUT	3-input function	6-input function
8-input topology	4-LUT	4-input function	5-input function
8-input topology	4-LUT	3-input function	7-input function
8-input topology	4-LUT	4-input function	6-input function
9-input topology	3-LUT	3-input function	7-input function
9-input topology	4-LUT	3-input function	7-input function
9-input topology	4-LUT	4-input function	6-input function
9-input topology	4-LUT	4-input function	7-input function
10-input topology	4-LUT	4-input function	7-input function

input topologies ($5 \leq k \leq 10$) has to involve practical bits, thus the Boolean matching will be incomplete. Besides, the proposed approach has to miss *seven* topologies with two bridged input pins, when to find a match for some 8-input functions. These missed 8-input topologies are illustrated in Figure 4.10.

4.7 Summary

This chapter has proposed the NPN-based Boolean matching problem and shown it can be solved through simple hash lookups in a pre-computed library of topologies. The pre-computation of a library of topologies suffers excessive computation time and memory consumption. Two critical concepts of essential bits and practical bits are discussed to solve runtime and memory problems. Further, Ashenhurst decomposition is involved as a pre-process for logic functions, to make the NPN-based Boolean matching efficient to handle functions with larger number of inputs. These technologies are combined together to allow the NPN-based Boolean matching approach applicable on large industrial circuits under the 3- or 4-input LUT architecture.

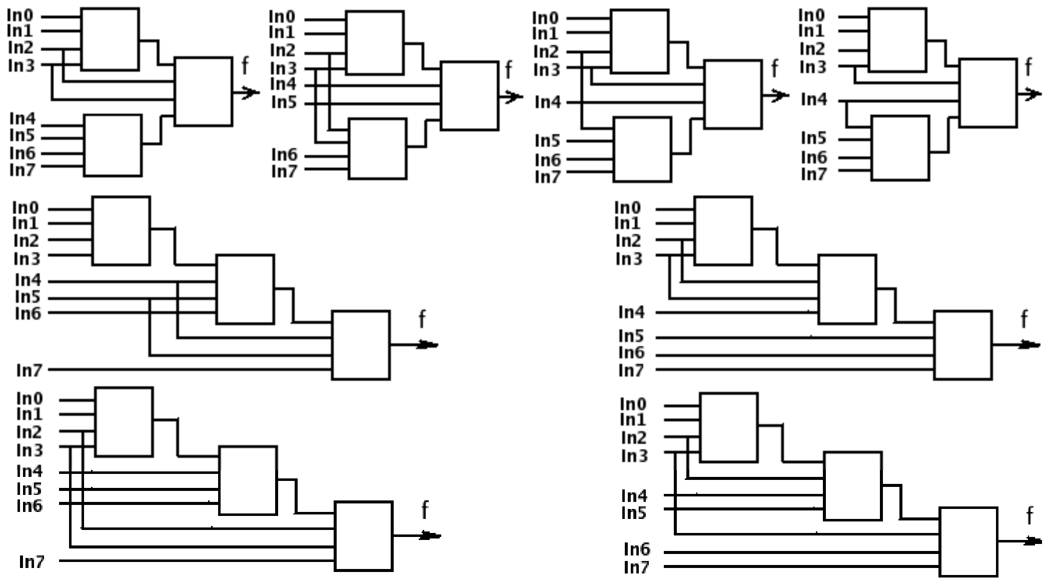


Figure 4.10: Seven missed 8-input topologies of 4-input LUTs

Chapter 5

NPN-based Post-mapping Topology Rewriting

Given the availability of the fast Boolean matching algorithm described in Chapter 4, this chapter describes an algorithm that can be applied post-technology mapping to reduce the number of LUTs in the mapped design without damaging the critical path of the circuit. The presented algorithm is based on topology rewriting and focuses on 3- and 4-input LUT architectures. Numerical results are presented to demonstrate the efficacy of the proposed algorithm.

5.1 Top-level Rewriting Algorithm

The proposed algorithm for area minimization on the post-technology mapped network is now described. The goal of the algorithm is to reduce the number of LUTs in the technology mapped network without worsening the critical path in the design. The proposed algorithm is based on the concept of circuit rewriting. Basically, it is a fast greedy algorithm which minimizes the area of a mapped network by iteratively selecting subgraphs of LUTs rooted at nodes and replacing, or *rewriting*, the subgraphs with alternative and smaller (i.e., less LUTs) precomputed subgraphs. Of course, the repcomputed subgraphs must implement the same functionality as the subgraphs they are replacing. Circuit rewriting is a very local algorithm in so far as it rewrites small cones of logic. However, rewriting is very fast and can be applied to the network many times. By an iterative application of rewriting, the scope of the changes made to the network will no longer be local; the cumulative effect of several rewriting passes is likely to achieve good results in terms of area minimization over the entire network.

A single pass of the proposed algorithm is described as follows with pseudo-code provided in Figure 5.1. The proposed algorithm accepts as input a technology mapped network. The output is also a mapped network with reduced area; i.e., a functionally equivalent network of LUTs which contains fewer LUTs than the original network. Each pass of rewriting begins with static timing analysis in which arrival times are propagated from PIs to POs; this is followed by the propagation of required times from POs to PIs. Subsequently, each node in the network is considered as a candidate for rewriting in topological order; i.e., nodes are processed from PIs to POs.

For node n, the following operations are considered to determine if the function implemented by node n should be rewritten (and, hence, implemented by an alternative circuit topology). At the node n, a set of cuts *cutset* is computed for node n, filtered and sorted by a ranking weight (largest weights first); each cut C in the cutset of node n is considered as a candidate for rewriting. The selected cut C is simulated to determine its corresponding logic function f.

If the number of inputs of the logic function f is ≤ 7, then NPN-based matching is used to find an alternative set of topologies of LUTs, denoted by *topset*, which implements the logic function f. Each alternative topology implementing f is considered to determine the impact on the area and delay of the circuit. An alternative topology is considered to improve the circuit if either the area of the circuit is reduced without violating the timing of the circuit or if the area of the circuit remains the same, but the delay of the circuit is reduced (although the goal of the algorithm is to perform area minimization, it is possible that a reduction in circuit delay can be found by performing rewriting). The topology which yields the maximum improvements is selected for replacing the cut C. Pseudo-code for selecting the best topology from the *topset* is shown in Figure 5.2.

If the number of inputs of the logic function f is ≥ 8, then NPN-based matching is combined with Ashenhurst decomposition to find an alternative topology of LUTs which implements the logic function f. First, disjoint Ashenhurst decomposition is applied to

```
**Procedure:** NPN-BASED TOPOLOGY REWRITING
**Inputs:** Mapped network, $N$
**Returns:** Area minimized network, $N'$
perform static timing analysis on network $N$
for each node $n$ in topological order from PIs to POs do
    compute sorted $cutset$ for node $n$;
    for each cut $C$ in $cutset$
        $Area = Area(C)$; $Delay = Delay(C)$; $Required\_Time = Required\_Time(C)$;
        simulate $C$ to obtain logic function $f$;
        if $fanin(C) \leq 7$ do
            find alterative topologies $topset$ for $f$ using NPN-based matching;
            select best topology $T_{best}$ for $f$ from $topset$;
            if $T_{best}! = 0$ do
                replace cut $C$ with topology $T_{best}$ and update timing;
            end if
        else
            for each disjoint decomposition $(g, h)$ of $f$ do
                find alterative topologies $topset$ for $h$ using NPN-based matching;
                select best topology $T_{best}$ for $h$ from $topset$;
                replace $(g, h)$ with $(LUT, T_{best})$;
            end do
            if $T_{best}! = 0$ do
                for each non-disjoint decomposition $(g, h)$ of $f$ do
                    find alterative topologies $topset$ for $h$ using NPN-based matching;
                    select best topology $T_{best}$ for $h$ from $topset$;
                    replace $(g, h)$ with $(LUT, T_{best})$;
                end do
            end if
            if $T_{best}! = 0$ do
                replace cut $C$ with topology $(LUT, T_{best})$ and update timing;
            end if
        end if
    end do
end do
```

Figure 5.1: The top-level algorithm for post-technology mapping area minimization using topology rewriting.

```
    Procedure: Topology selection
    Inputs: Set of topologies $topset$, current $Area$, $Delay$ and $Required\_Time$
    Returns: Best topology $T_best$ in $topset$
    $T_{best} = 0$;
    for each topology $T$ in $topset$
        if $Area(T) < Area$ and $Delay(T) <= Required\_Time$ do
            $Area = Area(T)$; $Delay = Delay(T)$; $T_{best} = T$;
        else
            if $Area(T) == Area$ and $Delay(T) < Delay$ do
                $Delay = Delay(T)$; $T_{best} = T$;
            end if
        end if
    end do
end do
```

Figure 5.2: The means by which a topology is selected to replace a cut.

decompose f into the base function g and encoded function h. Of course, there are multiple decompositions possible. The encoded function h is matched using NPN-based matching. If disjoint Ashenhurst decompostion cannot find a candidate topology for f, then non-disjoint Ashenhurst decomposition is used. The ground rule is that the disjoint decomposition will always be tried first since non-disjoint decompositions will generate reconvergent paths which leads to more complicated topologies.

5.2 Revised Cut Computation

The presented rewriting algorithm requires the computation of a cutset for the nodes in the mapped network. The cut computation for topology rewriting is somewhat different when compared with the cut enumeration used prior to technology mapping. Recall that cut enumeration applied prior to technology mapping is perfomed on a subject graph and the size or volume of the cut (measured in terms of its area) is not particularly important; it is only important that each cut fits into a single LUT. Further, for cut enumeration prior to technology mapping, the number of inputs to the cut is typically small since cuts are mapped into single LUTs. However, during topology rewriting, larger cuts with additional inputs need to be computed as well. Finally, since the topology rewriting algorithm is continually changing the network, cut computation needs to be dynamic to remain in sync

```
    Procedure: Cut Computation for Topology Rewriting
    Inputs: $root, N, S$
    Returns: A set of cuts sorted by their weights, cutset
1   Compute the MFFC $mffc(root)$ for the root node;
2   insert the trivial cut of the root node in the cutset;
3   for each cut $C$ in cutset do
4       for each fanin $L$ of $C$ do
5           if fanin $L$ is not primary input then
6               $expandCut(C, L, N, S, mffc, cutset)$;
7           end if
8       end do
9   end do
10  sort cuts in the cutset by their weights;
11  return cutset;
```

Figure 5.3: Cut computation for topology rewriting.

with the changing network. Hence, the strategy for enumerative cut generation which is used prior to technology mapping is not well suited for topology rewriting and a different strategy is required.

During the previously proposed algorithm for topology rewriting, cuts were computed separately for each node starting with its trivial cut. New cuts are obtained by *expanding existing cuts* towards the PIs. The pseudo-code of the cut computation is shown in Figure 5.3. The procedure starts on the node *root* for which the cuts are being computed along with the limit on the cut size N and the limit S on the number of duplicated nodes. A duplicated node is defined as a node which can be covered by a cut for node *root*, but is not in the MFFC of the node *root*. By allowing nodes to be duplicated, more cuts can be generated for node *root*, thereby allowing the rewriting algorithm to explore a potentially large solution space which can lead to a better solution. Procedure *expandCut()* tries to expand the cut by moving a fanin node L to the set of covered nodes and adding the node's fanins to the set of fanins. If the fanin node doesn't belong to the MFFC of the root $mffc(root)$ and the number of duplicated nodes of the cut $numDups(C)$ has already saturated, i.e., $numDups(C) \geq S$, the new cut is not constructed. If the new cut has more fanins than the limit N or dominated by any of the previously computed cut, the new cut is not appended to the cutset. Finally, if none of the above conditions holds, the cut is appended to the cutset. Later in the cut computation, this cut is used to derive other cuts

```
Procedure: CUT EXPANSION
Inputs: $C, L, N, S, mffc, cutset$
Returns: $cutset$
if $L \notin mffc$ and $numDups(C) == S$
    return $cutset$;
end if
A new cut $C' = (C - L) \cup fanins(L)$ is constructed;
if $numFanins(C') > N$ // check if the cut $C'$ is $N$-feasible
    return $cutset$;
end if
if $cutsFilter(cutset, C')$ // check if $C'$ is dominated by any cut in $cutset$
    return $cutset$;
end if
$cutset = cutset \cup C'$;
return $cutset$;
```

Figure 5.4: Cut expansion

by calling *expandCut()*, whose pseudo-code is shown in Figure 5.4.

At the end of cut computation, some cuts may not be useful for topology rewriting because the network structure covered by these cuts cannot be improved. That is, some cuts should be pruned before rewriting consideration because they obviously offer little to no potential improvement for reducing the area of the network; any further consideration of these cuts will simply waste runtime without yielding any benefit in terms of circuit area. In order to filter out the useless cuts and prioritize other cuts, the notion of *cut weight* is involved. Weights are computed for all cuts in the cutset and only those cuts with large enought weights are retained for topology rewriting.

The weight of a cut is defined and given by

$$cutWeight(c) = [numCovered(c) - numDup(c)]/numLuts(c) \tag{5.1}$$

where $numLuts(c) = ceiling[(numFanins(c) - 1)/(K - 1)]$. The procedures *numCovered* and *numDup* return the total number of covered nodes and the number of covered nodes whose fanout nodes not included in the cut, respectively. *numLuts* is the minimum number of k-input LUTs needed to implement the cut of the given size. An example of a cut of LUTs is illustrated in Figure 5.5. This cut contains 6 nodes, and the node F is not covered by the cut and needs duplication. Meanwhile, this cut has 7 fanins and the minimum number of 3-input LUTs to implement the cut will be 3. Thus, the weight of cut will be

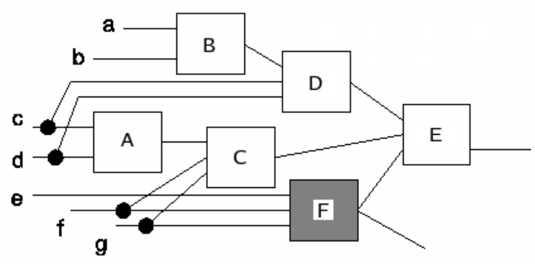


Figure 5.5: An example of cut weight calculation.

1.67. Intuitively, the weight of a cut shows how likely the cut will be useful in topology rewriting. Larger weights imply cuts that might yield larger reductions in area. Those cuts with weights ≤ 1 can be skipped because they offer no potential for improvement.

5.3 Numerical Results

The proposed rewriting algorithm has been tested on a large set of designs which have been technology mapped to both architectures composed of 3-input LUTs and 4-input LUTs. In terms of circuit size, the designs range in size from 51 to 16K LUTs when mapped to 3-input LUTs and from 39 to 12K LUTs when mapped to 4-input LUTs. All numerical results were obtained on a dual processor Xeon machine with two separate hyper-threaded processors, which runs a Linux operating system.

Table 5.1 presents a summary of the results for 3- and 4-input LUT architectures. Table 5.1 compares the area and delay of the original technology mapped network to the network obtained after the application of our proposed algorithm; area and delays are reported as the geometric mean over the entire set of designs. As shown in Table 5.1, the proposed algorithm yields and area improvement of roughly 3.5% on the 3-input LUT network and 3.1% on the 4-input LUT network. Table 5.1 also shows the average runtime required by the proposed algorithm demonstrating that, on average, the algorithm is reasonably fast. The geometric mean of runtime for the entire rewriting is around 20 seconds. The longest runtime for the rewriting process is 403.16 seconds on the 3-input LUT network, and 894.06 seconds on the 4-input LUT network.

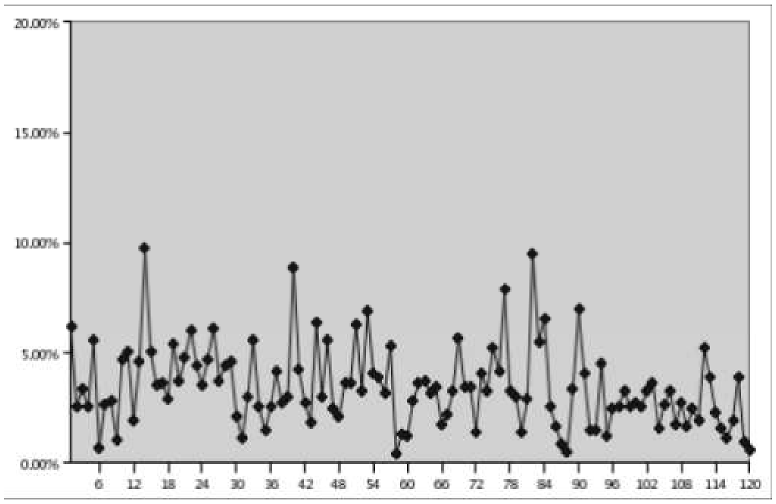
Table 5.1: Experiment result of post-mapping topology rewriting

	Before Rewriting		**After Rewriting**			
Designs	**Area**	**Delay**	**Area**	**Delay**	**Ratio of Area**	**Runtime (sec)**
3-input LUT architecture	2097.80	11.67	2024.60	11.67	0.965	19.87
4-input LUT architecture	1648.97	8.21	1597.17	8.20	0.969	23.45

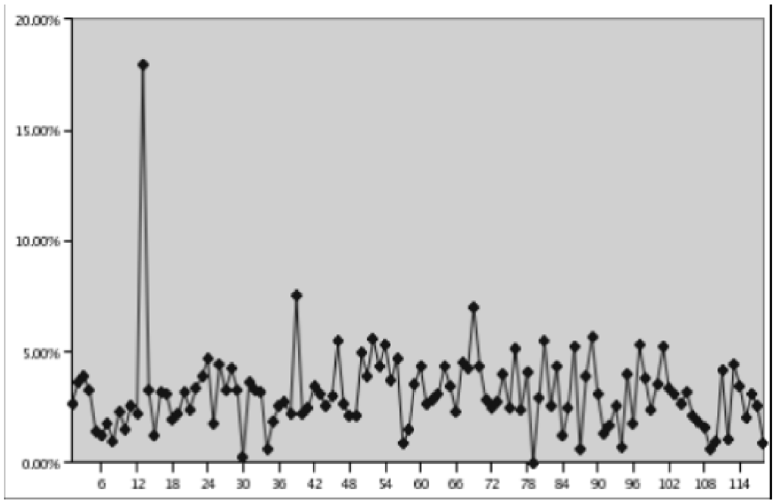
Numerical results on a per design basis are shown in Figures 5.6(a) and 5.6(b) for the the 3-input and 4-input LUT architectures respectively. From these figures, it can be observed that the maximum area improvement is 10% for the 3-input LUT architecture and 18% for the 4-input LUT architecture.

5.4 Summary

This chapter has discussed the entire proposed rewriting algorithm for area minimization after technology mapping. It has shown how larger cuts are computed and how NPN-based Boolean matching (along with Ashenhurst decomposition) is used to rewrite a circuit to reduce the required number of LUTs. Numerical results have been presented to demonstrate the efficacy of the proposed rewriting algorithm. In general, the application of the proposed rewriting algorithm is reasonably fast and can effectively reduce the number of LUTs in a given technology mapped netlist.



(a) 3-input LUT architecture



(b) 4-input LUT architecture

Figure 5.6: Experiment result of post-mapping topology rewriting

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Technology mapping is an important step in the FPGA CAD flow and has the responsibility of optimizing simultaneous objectives, including area and delay. Unfortunately, this is a difficult problem and technology mappers are forced to resort to efficient heuristic methods to accomplish their task.

Additional heuristics can be applied after technology mapping. In particular, it is possible to apply circuit rewriting in order to further reduce the area of a technology mapped network without harming the delay of the network. Unfortunately, area minimization through rewriting requires the use of some sort of Boolean matching algorithm in order to determine alternative circuit topologies which require less area, while implementing the same function. It is possible to use SAT-based Boolean matching. However, SAT-based Boolean matching is slow due to the need to continually formulate and solve the individual SAT problems. Hence, despite the flexibility and elegance of the SAT-based approach, it is not necessarily practical in terms of runtime.

This thesis has proposed the use of NPN-based matching as an alternative to SAT-based matching. In this approach, only hash table lookups are required to determine if a function is implementable within a particular topology of LUTs. Consequently, the use of the proposed approach during circuit rewriting for area minimization offers significant speedups in terms of runtime if compared to a SAT-based matching algorithm.

Unfortunately, the proposed NPN-based matching algorithm requires off-line enumeration of the logic functions which can be implemented by different topologies of LUTs. Although this procedure only needs to be performed once for a given topology of LUTs, it has both large runtime and memory requirements. To offset these negatives, several contributions have been proposed. The use of NPN equivalence classes has been proposed to reduce the amount of memory required to store the functions which are implementable by a particular topology of LUTs. To offset the amount of runtime required to process a topology of LUTs, the ideas of *essential bits* and *practical bits* was introduced to avoid unnecessary and unneeded simulation and NPN encoding. Numerical results were presented to demonstrate the efficacy of the proposed ideas. The use of NPN encoding was shown to keep the memory requirements low while the use of essential and practical bits was shown to significantly speed up the processing of different topologies of LUTs (many topologies of LUTs which could not be processed within 24 hours could, in fact, be processed by using essential and practical bits). Finally, to further extend the range and ability of the proposed matching algorithm, it was combined together with simple Ashenhurst decomposition.

Finally, the thesis proposed a post-technology mapping algorithm aimed at area minimization. The proposed algorithm was based on the concept of circuit rewriting using the proposed NPN-based matching algorithm. Experimental results obtained from testing the proposed algorithm on a set of designs demonstrated that the proposed algorithm was quite capable of achieving area minimization. For FPGA architectures consisting of 3-input LUTs, an average area reduction of 3.5% was obtained. For FPGA architectures consisting of 4-input LUTs, an average area reduction of 3.1% was obtained. Further, the proposed algorithm was demonstrated to be computationally efficient due to the use of NPN-based matching. On average, the proposed algorithm took, on average, about 20 seconds to perform rewriting. It is therefore concluded that the proposed algorithm can be applied to large designs.

6.2 Future Work

From the obtained results, substantial area improvement can be achieved after post-mapping topology rewriting in the reasonable computation time. However, the proposed rewriting algorithm is only applicable on the 3- or 4-input LUT architectures. Besides, the algorithm can only rewrite on the 10-feasible cuts. Future work in the post-mapping topology rewriting will include:

- Discovering new canonical forms to represent logic functions instead of NPN equivalence classes, which provides a more compressed way to contain all implementable logic functions in a topology of LUTs. This will reduce the memory consumption problem and make the Boolean matching feasible on topologies of large LUTs.
- Exploring other functional decomposition algorithms in corporation with NPN-based Boolean matching, to allow the rewriting algorithm to work for functions with more than 10 inputs.

APPENDICES

Appendix A

Glossary of Terms

ASIC Applicaton-Specific Integrated Circuit.

BLE Basic Logic Element.

Boolean Matching The question whether a logic function f can be implemented by a topology of logic gates.

CLB Configurable Logic Block.

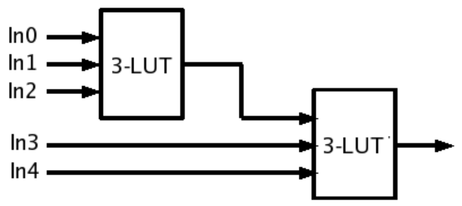
FPGA Field Programmable Gate Array.

NPN equivalent Two Boolean functions are NPN equivalent if one function can be derived from the other function (or visa-versa) by negating (N) and permuting (P) inputs and negating (N) the output.

IO Input/Output.

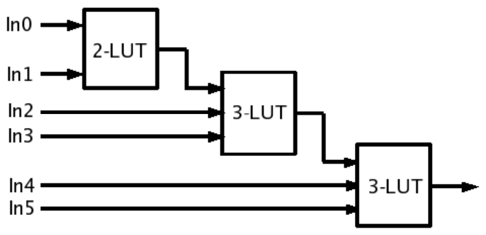
Appendix B

List of LUT Topologies

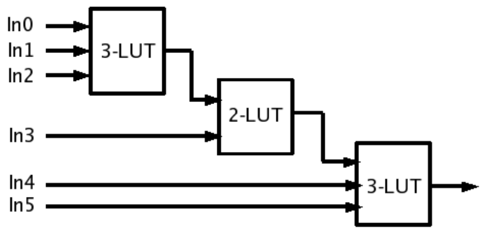


(a) LUT3_5_1

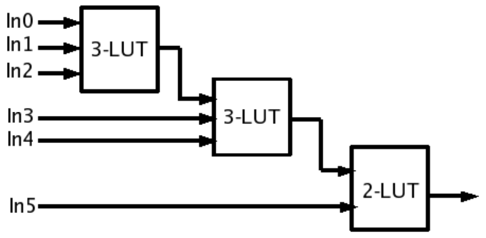
Figure B.1: 5-input topologies of 3-input LUTs



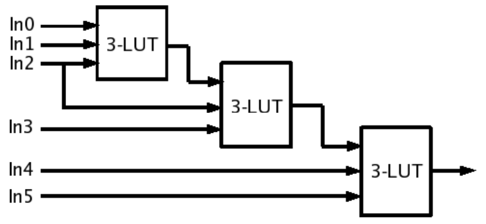
(a) LUT3_6_1



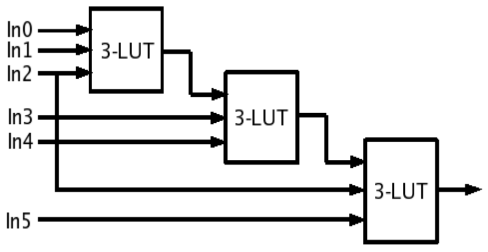
(b) LUT3_6_2



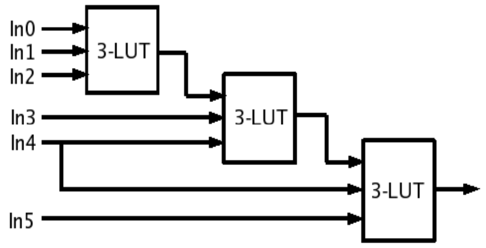
(c) LUT3_6_3



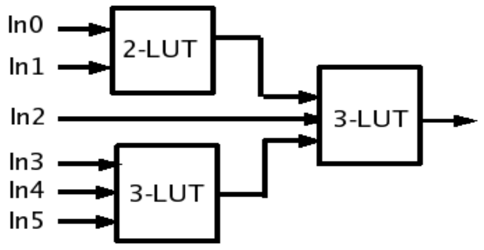
(d) LUT3_6_4



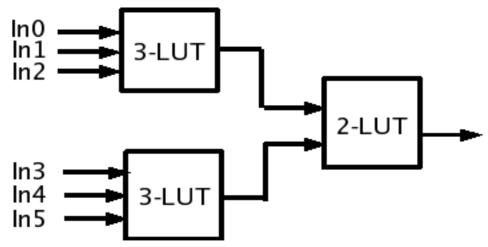
(e) LUT3_6_5



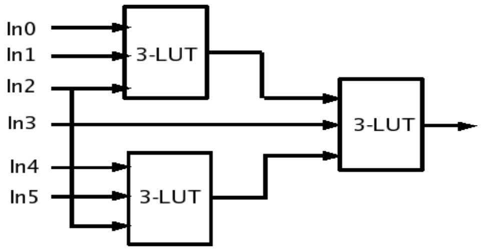
(f) LUT3_6_6



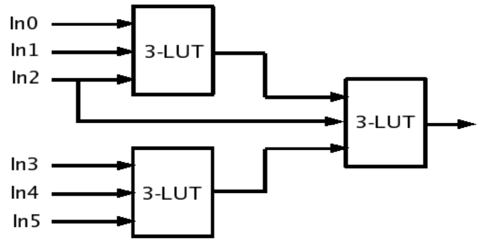
(g) LUT3_6_7



(h) LUT3_6_8

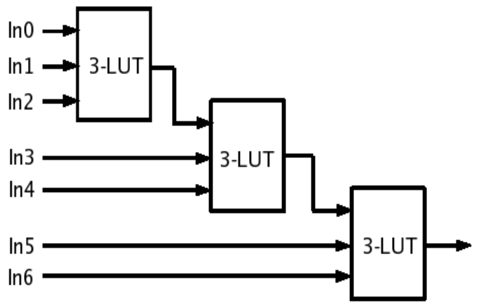


(i) LUT3_6_9

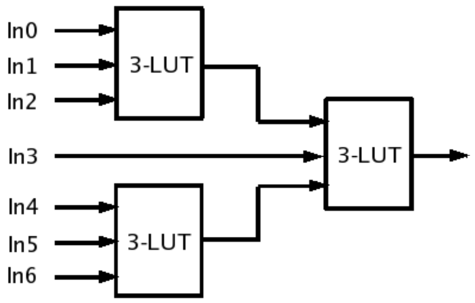


(j) LUT3_6_10

Figure B.2: 6-input topologies of 3-input LUTs

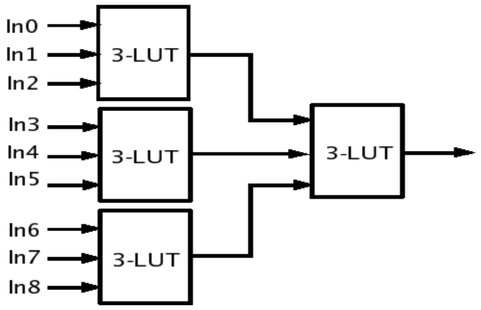


(a) LUT3_7_1

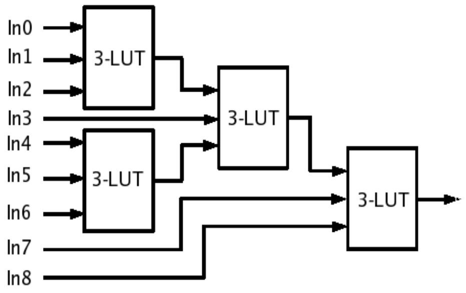


(b) LUT3_7_2

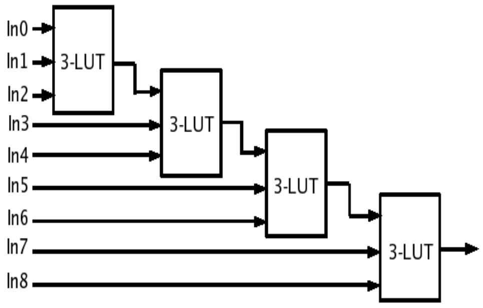
Figure B.3: 7-input topologies of 3-input LUTs



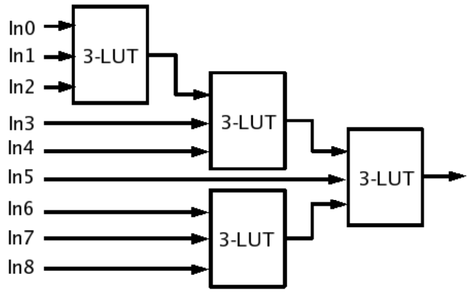
(a) LUT3_9_1



(b) LUT3_9_2

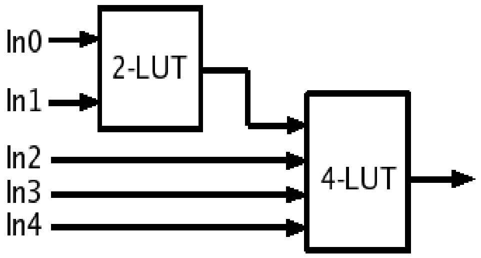


(c) LUT3_9_3

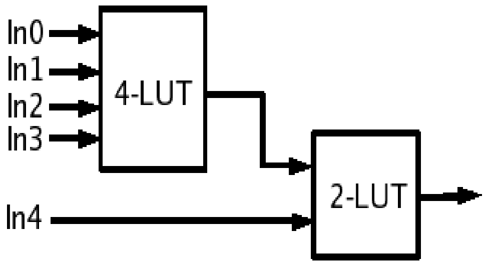


(d) LUT3_9_4

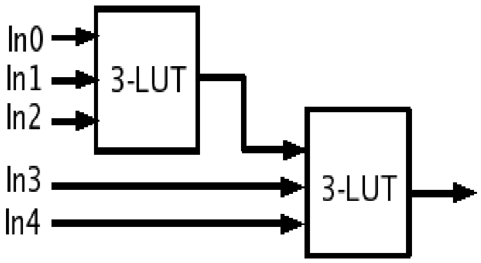
Figure B.4: 9-input topologies of 3-input LUTs



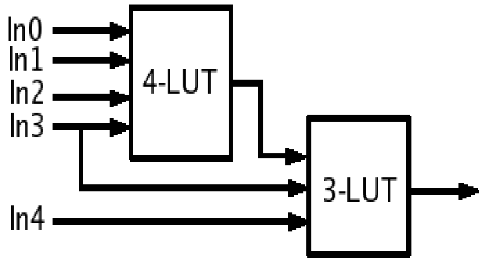
(a) LUT4_5_1



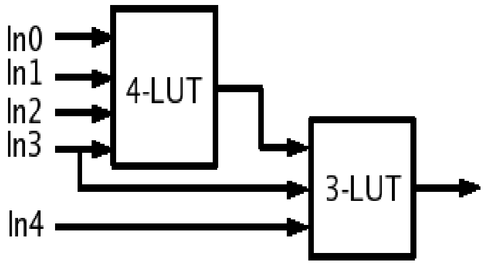
(b) LUT4_5_2



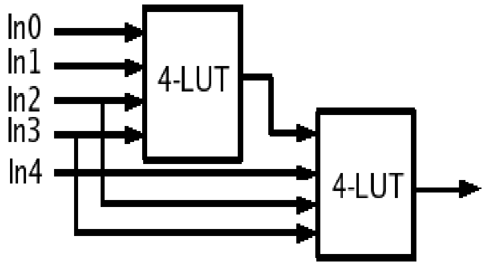
(c) LUT4_5_3



(d) LUT4_5_4

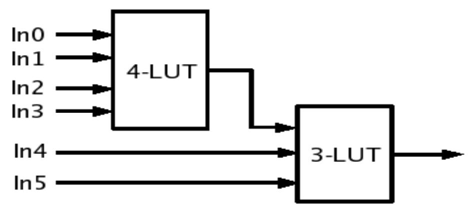


(e) LUT4_5_5

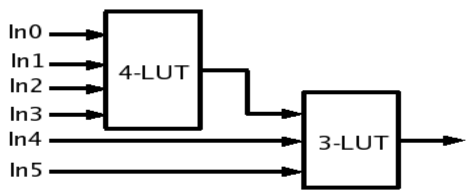


(f) LUT4_5_6

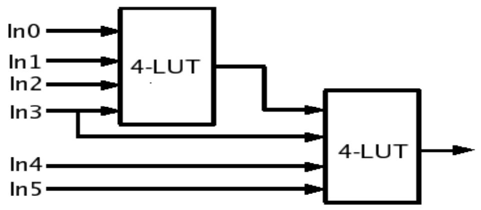
Figure B.5: 5-input topologies of 4-input LUTs



(a) LUT4_6_1

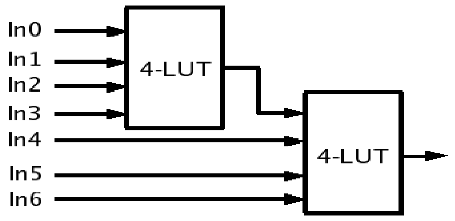


(b) LUT4_6_2



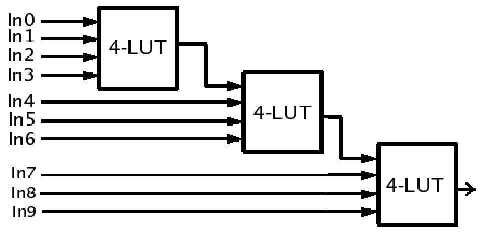
(c) LUT4_6_3

Figure B.6: 6-input topologies of 4-input LUTs

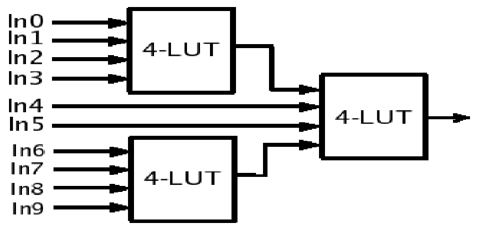


(a) LUT4_7_1

Figure B.7: 7-input topologies of 4-input LUTs



(a) LUT4_10_1



(b) LUT4_10_2

Figure B.8: 10-input topologies of 4-input LUTs

Bibliography

[1] FPGA architecture. `http://www.eecg.toronto.edu/~vaughn/challenge/fpga_arch.html`.

[2] A. Abdollanhi and M. Pedram. A new canonical form for fast Boolean matching in logic synthesis and verification. In *Proc. DAC*, pages 379–384, 2005.

[3] Actel Corporation, Mountain View, CA. *Actel Axcelerator family FPGAs*, November 2008.

[4] Actel Corporation, Mountain View, CA. *Actel ProASIC3 flash family FPGAs*, February 2009.

[5] E. Ahmed and J. Rose. The effect of LUT and cluster size on deep-submicron fpga performance and density. *IEEE Trans. VLSI*, 12(3):288–298, March 2004.

[6] Altera Corporation, San Jose, CA. *Altera 40nm Stratix IV FPGAs and HardCopy IV ASICs*, September 2008.

[7] Altera Corporation, San Jose, CA. *Altera Stratix IV device handbook*, June 2009.

[8] Altera Incorporated, San Jose, CA. *Xilinx Virtex-5 family overview*, February 2009.

[9] L. Benini and G. Micheli. A survey of Boolean matching techniques for library binding. *Trans. on DAES*, 2(3):193–226, July 1997.

[10] V. Betz and J. Rose. VPR: A new packing, placement and routing tool for FPGA research. In *Proc. FPL*, pages 213–222, 1997.

[11] Vaughn Betz, Jonathan Rose, and Alexander Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs.* Kluwer Academic Publishers, 1999.

[12] S. Brown, J. Rose, and Z. Vranesic. A detailed router for field-programmable gate arrays. volume 11, pages 620–628, May 1992.

[13] D. Chai and A. Kuelmann. Building a better Boolean matcher and symmetry detector. In *Proc. DATE*, pages 1079–1084, 2006.

[14] S. Chatterjee, A. Mishchenko, and R. Brayton. Factor cuts. In *Proc. ICCAD*, pages 143–150, 2006.

[15] D. Chen and J. Cong. DAmap: A depth-optimal area optimization mapping algorithm for FPGA designs. In *Proc. ICCAD*, pages 752–759, 2004.

[16] S. Cho, S. Chatterjee, A. Mishchenko, and R. Brayton. Efficient FPGA mapping using priority cuts. In *Proc. FPGA*, 2007.

[17] J. Cong and Y. Ding. FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Trans. CAD*, 13(1):1–12, January 1994.

[18] J. Cong and Y. Ding. Combinational logic synthesis for LUT based field programmable gate arrays. *Trans. on DAES*, 1(2):145–204, April 1996.

[19] J. Cong and Y. Hwang. Boolean matching for LUT-based logic blocks with applications to architecture evaluation and technology mapping. *IEEE Trans. CAD*, 20(9):1077–1090, September 2001.

[20] J. Cong and K. Minkovich. Improved SAT-based Boolean matching using implicants for LUT-based FPGAs. *Proc. FPGA*, pages 139–147, 2007.

[21] J. Cong, C. Wu, and Y. Ding. Cut ranking and pruning: enabling a general and efficient FPGA mapping solution. In *Proc. FPGA*, pages 29–35, 1999.

[22] P. Coussy and A. Morawiec, editors. *High-level synthesis: from algorithm to digital circuit.* Springer Publishers, 2008.

[23] A. Farrahi and M. Sarrafzadeh. Complexity of the lookup-table minimization problem for FPGA technology mapping. *IEEE Trans. CAD*, 13(11):1319–1332, Nov 1996.

[24] D. Gajski, N. Dutt, A. Wu, and S. Lin, editors. *High-level synthesis: introduction to chip and system design.* Springer Publishers, 1999.

[25] S. Hassoun and T. Sasao, editors. *Logic Synthesis and Verification.* Springer Publishers, 2001.

[26] Y. Hu, V. Shih, R. Majumdar, and L. He. Exploiting symmetry in SAT-based Boolean matching for heterogeneous FPGA technology mapping. In *Proc. DAC*, pages 379–384, 2007.

[27] Y. Kukimoto, R. Brayton, and P. Sawkar. Delay-optimal technology mapping by DAG covering. In *Proc. DAC*, pages 348–351, 1998.

[28] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. In *Proc. FPGA*, pages 21–30, 2006.

[29] I. Kuon, R. Tessier, and J. Rose. Fpga architecture: Survey and challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253, 2008.

[30] A. Ling, D. Singh, and S. Brown. FPGA technology mapping: a study of optimality. In *Proc. DAC*, pages 427–432, 2005.

[31] V. Manohararajah, S. D. Brown, and Z. G Vranesic. Heuristics for area minimization in LUT-based FPGA technology mapping. *IEEE Trans. CAD*, 25(11):2331–2340, November 2006.

[32] M. McFarland, A. Parker, and R. Camposano. Tutorial on high-level synthesis. In *Proc. DAC*, pages 330–336, 1988.

[33] A. Mishchenko. DAG-Aware AIG rewriting: a fresh look at combinational logic synthesis. pages 24–28, 2006.

[34] A. Mishchenko, S. Chatterjee, and R. Brayton. Fast Boolean matching for LUT structures. 2007.

[35] A. Mishchenko, S. Chatterjee, and R. Brayton. Improvements to technology mapping for LUT-based FPGAs. *IEEE Trans. CAD*, 26(2):250–253, February 2007.

[36] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton. Combinational and sequential mapping with priority cuts. In *Proc. ICCAD*, pages 354–361, 2007.

[37] R. Murgai, R. Brayton, and A. Sangiovanni-Vincentelli. Logic synthesis algorithms for programmable gate arrays. pages 620–625, 1990.

[38] R. Murgai, R. Brayton, and A. Sangiovanni-Vincentelli, editors. *Logic Synthesis for Field-Programmable Gate Arrays*. Springer Publishers, 1995.

[39] Muroga.S, editor. *Logic design and switching theory*. John Wiley & Sons, 1979.

[40] P. Pan and C. Lin. A new retiming-based technology mapping algorithm for LUT-based FPGAs. In *Proc. FPGA*, pages 35–41, 1998.

[41] J. Rose and S. Brown. Flexibility of interconnection structures for field-programmable gate arrays. *IEEE J. SSC*, 26(3):277–282, March 1991.

[42] J. Rose, R.J. Francis, D. Lewis, and P. Chow. Architecture of field-programmable gate arryas: The effect of logic block functionality on area efficiency. pages 1217–1225, 1990.

[43] S. Safarpour, A. Veneris, G. Baeckler, and R. Yuan. Efficient SAT-based Boolean matching for FPGA technology mapping. pages 466–471, July 2006.