A Tutorial on Logic Synthesis for Lookup-Table Based FPGAs

Robert J. Francis

Department of Electrical Engineering, University of Toronto

Abstract

The ability to shorten development cycles has made Field-Programmable Gate Arrays (FPGAs) an attractive alternative to Standard Cells and Mask Programmed Gate Arrays for the realization of ASICs. One important class of FPGAs are those that use lookup tables (LUTs) to implement combinational logic. The ability of a K-input LUT to implement **any** *Boolean function of K variables differentiates the synthesis of LUT circuits from synthesis for conventional ASIC technologies. The major difference occurs during the technology mapping phase of logic synthesis. For values of K greater than 3, the large number of functions that can be implemented by a K-input LUT makes it impractical to use conventional library-based technology mapping. However, the* **completeness** *of the set of functions that can be implemented by a LUT eliminates the need for a library of separate functions. In addition, this completeness can be leveraged to optimize the final circuit.*

1 Introduction

Field-Programmable Gate Arrays (FPGAs) now provide an alternative to Standard Cells and Mask Programmed Gate Arrays for the realization of ASICs. An FPGA consists of an array of logic blocks that implement combinational and sequential functions, and a user-programmable routing network that provides connections between the logic blocks. User-programability allows for rapid and inexpensive prototype development [1]. This tutorial discusses combinational logic synthesis for FPGAs that use lookup tables (LUTs) to implement combinational logic, and focuses on issues that differentiate LUT synthesis from conventional logic synthesis.

A K-input lookup table is a digital memory that can implement any Boolean function of K variables. The K inputs are used to address a 2^K by 1-bit digital memory that stores the truth table of the Boolean function. For example, Figure 1a illustrates the truth table for the function $x = ab + \bar{b}c$, and Figure 1b illustrates the structure of a 3-input LUT implementing this function. The truth table is stored in an 8 by 1-bit memory, and an 8 to 1 multiplexer, controlled by the variables a, b, and c, selects the output value x.

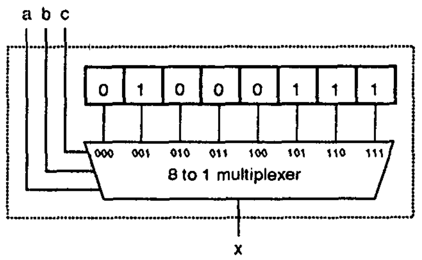
The goal of combinational logic synthesis is to produce an optimized circuit implementing a given combinational function. The original function can be represented by a Directed Acyclic Graph (DAG) where each node represents a local function of the global functions represented by its immediate fanin nodes. For example, in the DAG illustrated in Figure 2a the local function at the node z is $z = wxy$, and the global function is $z = ((abc) + (def))(g + h)(i + j)$.

The netlist describing a circuit of LUTs can be represented by a similar DAG. In this case, each node represents a single LUT and the node's local function specifies the function implemented by the LUT. Figure 2b illustrates a circuit of 5-input LUTs implementing the function illustrated in Figure 2a. The dotted boundaries in Figure 2b indicate the local function implemented by each LUT. The local function implemented by the LUT z is $z = (u + (def))t$. Unless stated otherwise, all examples in the remainder of this tutorial will use 5-input LUTs.

Combinational logic synthesis can be conceptually divided into two phases; technology-independent **logic optimization**, and **technology mapping** [2]. Logic optimization restructures the original network, without changing the function of its primary outputs, and technology mapping implements the optimized network using the circuit elements available in the given ASIC

a	b	c	$x = ab + \bar{b}c$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

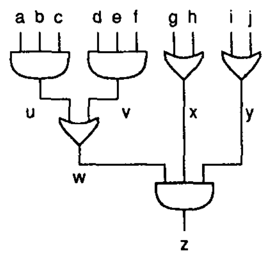
a) Truth Table



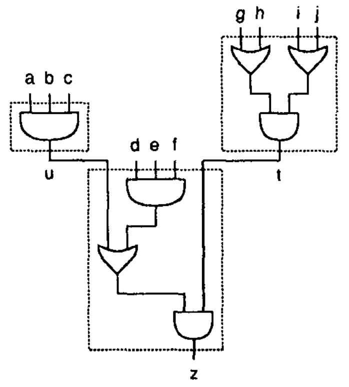
b) 3-Input LUT

Figure 1: A LUT Implementing $x = ab + \bar{b}c$

0-8186-3010-8/92 $03.00 © 1992 IEEE



a) Boolean Network



b) Circuit of 5-input LUTs

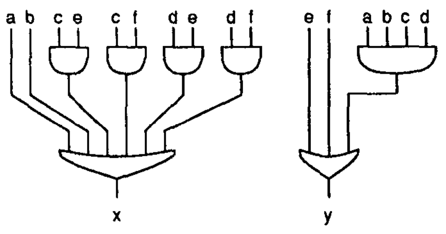
Figure 2: Network and Circuit

technology.

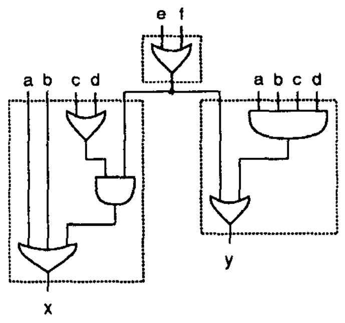
The modifications performed by logic optimization typically include redundancy removal and common sub-expression elimination. The intention is to improve the final circuit by simplifying the underlying network. For example, consider the network shown in Figure 3a. The common sub-expression $e + f$ can be factored out of the functions x and y leading to the simplified network covered by the circuit shown in Figure 3b. Conventional techniques for logic optimization can be effective for LUT circuits particularly at a level of granularity where factors have more than K inputs. These techniques have been summarized in [2] and will not be discussed here.

Technology mapping selects sub-networks of the optimized network to be implemented by the available circuit elements. In the case of LUT-based FPGAs, any sub-network with at most K inputs can be implemented by a K-input LUT. The final circuit must include a LUT implementing each of the primary outputs and all of the LUT inputs that are not primary inputs.

The optimization goal for the synthesis of LUT circuits is typically the minimization of the total number of LUTs, the number of *levels* of LUTs, or both. Minimizing the number of LUTs in the circuit increases the size of designs that can fit into the fixed number of LUTs available in a given FPGA. The minimization of the number of levels of LUTs can improve the performance of the circuit by reducing the number of logic block delays and programmable routing delays



a) Boolean Network



b) Circuit of 5-input LUTs

Figure 3: Common Sub-Expression Elimination

on the longest path. To illustrate the issues that differentiate LUT synthesis from conventional logic synthesis, this tutorial will focus on the minimization of the total number of LUTs in the final circuit. The following section discusses the limitations of conventional library-based synthesis when applied to LUT circuits, and Section 3 discusses approaches to logic synthesis that deal specifically with LUT circuits.

2 Library-Based Synthesis

Standard Cells and Mask Programmed Gate Arrays both implement combinational functions using a limited set of simple gates. The most successful approach to synthesis for these ASIC technologies has used library-based technology mapping [3]. This approach traverses the network from the primary inputs to the primary outputs, and at each node the local structure of the network is *matched* against a library of patterns representing the set of available gates. For each successful match, the cost of the circuit using that gate is calculated from the cost of the gate, and the cost of the previously constructed circuits implementing the inputs to the gate. The minimum cost circuit among all the matches is then retained.

The major obstacle to applying library-based technology mapping to LUT circuits is the large number of different functions that a K-input LUT can implement. The function implemented by a K-input LUT is determined by the values stored in its 2^K memory bits. Since each bit can independently be either 0 or 1, there are 2^{2^K} different Boolean functions of K variables. For values of K greater than 3 the library required to rep-

K	without permutations and inversions	with permutations	with permutations and inversions
2	16	12	4
3	256	80	14
4	65536	3984	232

Table 1: Number of Patterns for a K-Input LUT

resent a K-input LUT becomes impractically large.

The size of the library can be reduced by noting that some patterns are equivalent after a permutation of inputs [4]. The inversion of outputs or inputs, which is trivially accomplished with a LUT, can also produce equivalent patterns. Table 1 lists the number of different patterns, with and without permutations and inversions, for $K = 2$, 3, and 4. To match a sub-network against a pattern in the reduced library it may be necessary to permute or invert the sub-network. Hashing functions have been proposed to simplify the matching of permuted patterns [5], but the increased complexity of pattern matching limits the benefits of the reduced library.

Another alternative is to use a partial library tuned to take advantage of the network structure likely to be produced by technology independent logic optimization [6]. The limitation of this approach is that it precludes some opportunities for optimization of the final circuit. The following section discusses approaches to LUT synthesis that exploit the full functionality of a K-input LUT to obtain improved results.

3 LUT-Specific Synthesis

There has been a great deal of recent work on logic synthesis that deals specifically with LUT circuits. [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18]. The key to all of these approaches is the ability of a K-input LUT to implement *all* functions of K variables. This *completeness* simplifies the matching of a sub-network to a LUT. To determine if a sub-network matches a K-input LUT it is not necessary to match the sub-network against a library of separate patterns, as described in the preceding section. It is sufficient to count the number of inputs to the sub-network, and verify that the number of inputs does not exceed the constraint K.

Technology mapping optimizes the final circuit by selecting which sub-networks are covered by LUTs. If the original network includes nodes with more than K inputs, referred to as *infeasible* nodes, it may not be possible to find a circuit of LUTs covering the network. In many mapping algorithms, to ensure that a circuit covering the network exists, each infeasible node is decomposed into a set of *feasible* nodes, each with at most K inputs. In addition, the decomposition of both feasible and infeasible nodes presents an opportunity to optimize the final circuit.

The next section discusses the decomposition of infeasible nodes, Section 3.2 discusses how decomposition and covering can be combined to improve the final circuit, and Section 3.3 describes how covering can exploit fanout nodes in the original network.

3.1 Decomposition of Infeasible Nodes

The general strategy for the decomposition of infeasible nodes is to decompose each infeasible node into sub-functions that use fewer inputs than the original infeasible node. Any sub-function that uses no more than K inputs is feasible and is decomposed no further. Any sub-function that has more that K inputs is recursively decomposed. Eventually the original infeasible node is decomposed into a set of feasible nodes. Four methods that have been proposed for the decomposition of infeasible nodes are; disjoint decomposition, algebraic factorization, AND-OR decomposition, and Shannon cofactoring.

3.1.1 Disjoint Decomposition

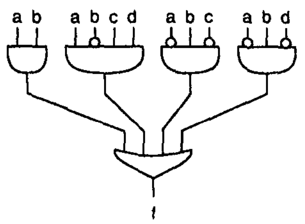
A disjoint decomposition is based on a partition of the inputs to the infeasible node into two disjoint sets referred to as the *bound set* and the *free set*. One or more functions of the bound set are extracted from the infeasible node, and the infeasible node is replaced by a function of the outputs of the extracted functions and the inputs in the free set. The attraction of a disjoint decomposition is that the number of inputs in the each of the two sets must be less than the number of inputs to the infeasible node.

Disjoint decompositions can be found by searching through all possible partitions of the inputs to the infeasible node, and using well known methods such as *residues* [19], to determine if each each partition leads to a disjoint decomposition. A residue function is obtained by replacing the inputs in the free set with constant values. If the set of all possible residue functions for a given partition consists of the constants 0 or 1, or a single function h of the bound variables, or its inverse $\overline{h}$, then the partition is a disjoint decomposition, with one extracted function. For example, consider the 4-input function $f = ab + a\overline{b}cd + \overline{a}b\overline{c} + \overline{a}b\overline{d}$ shown in Figure 4a, and the partition of its inputs into the free set $\{a, b\}$ and the bound set $\{c, d\}$. The set of residue functions for this partition, shown in Figure 4b, consists of the constants 0 and 1, and the function cd and its inverse $\overline{(cd)}$. Therefore, this partition leads to the disjoint decomposition of the function f, shown in Figure 4c.

The number of partitions grows exponentially with number of inputs to the infeasible node, and the search for disjoint decompositions can become prohibitively expensive if the infeasible node has a large number of inputs.

3.1.2 Algebraic Decomposition

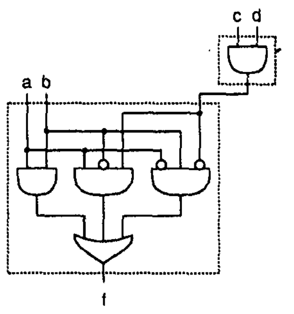
Algebraic factorization techniques developed for technology independent logic optimization can also be used for the decomposition of infeasible nodes [7]. For example, the function $x = ac + bc + bd + ce$ can be algebraically factored into the factor $y = a + b + e$, and the remainder $x = cy + bd$. Since the variable b is used by



a) An infeasible function for $K = 3$

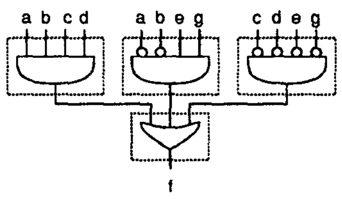
a	b	$h(c,d)$
0	0	0
0	1	$\overline{(cd)}$
1	0	cd
1	1	1

b) Residues for the partition $\{a,b\},\{c,d\}$

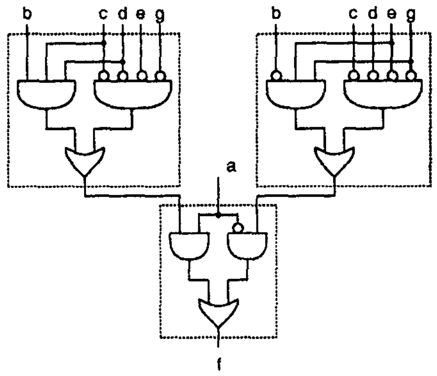


c) The disjoint decomposition

Figure 4: Identifying Disjoint Decompositions



a) Original circuit, 4 LUTs



b) With Shannon cofactoring, 3 LUTs

Figure 5: Shannon Cofactoring

both the factor y and the remainder x this is a not a disjoint decomposition.

3.1.3 AND-OR Decomposition

Disjoint decompositions and algebraic factorization are not sufficient to decompose all infeasible nodes. For example, the majority function $z = ab + ac + bc$, has no disjoint decomposition. AND-OR decomposition can be used to ensure that any infeasible node is decomposed into a set of feasible nodes. The AND (OR) operator is associative and commutative, which allows an AND (OR) node to be divided into smaller AND (OR) nodes using any partition of its inputs. An infeasible node is represented as a sub-graph of AND and OR nodes, each of which is then decomposed using AND-OR decomposition. For example, the above 3-input majority function can be decomposed into the 2-input functions $v = ab$, $w = ac$, $x = bc$, $y = v + w$, and $z = y + x$.

AND-OR decomposition can also be used to decompose large infeasible nodes into infeasible nodes that are small enough to make an exhaustive search for disjoint decompositions practical [9].

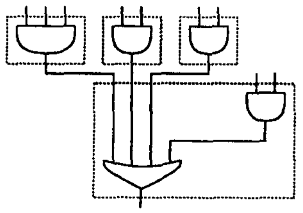
3.1.4 Shannon Cofactoring

Another form of decomposition that will always successfully decompose an infeasible node is Shannon cofactoring [20]. An infeasible function of n variables, $f(x_1 \ldots x_j \ldots x_n)$, is decomposed into the three functions $f_{x_j} = f(x_1 \ldots 1 \ldots x_n)$, $f_{\overline{x_j}} = f(x_1 \ldots 0 \ldots x_n)$ and $f = x_j f_{x_j} + \overline{x_j} f_{\overline{x_j}}$. The function f now depends on the three inputs x_j, f_{x_j}, and $f_{\overline{x_j}}$, and can therefore be implemented by a single K-input LUT for $K \geq 3$. The functions f_{x_j}, and $f_{\overline{x_j}}$, each depend on at most $n-1$ variables. If $n-1$ equals K then the completeness of a K-input LUT ensures that these functions can be implemented by a single LUT. Otherwise, the functions f_{x_j}, and $f_{\overline{x_j}}$, can be recursively decomposed. For example, the 6 input function $f = abcd + \overline{a}\overline{b}eg + \overline{c}\overline{d}eg$ can be covered by the 4 LUT circuit shown in Figure 5a. This function can be cofactored about the variable a to produce the cofactors $f_a = bcd + \overline{c}\overline{d}\overline{e}\overline{g}$, $f_{\overline{a}} = \overline{b}eg + \overline{c}\overline{d}\overline{e}\overline{g}$, and the 3 LUT circuit shown in Figure 5b.

3.2 Decomposition and Covering

An important observation is that the decomposition of feasible nodes, as well as infeasible nodes, can lead to a superior circuit. For example, in the circuit shown in Figure 6a the AND and OR nodes in the underlying network are all feasible, and four 5-input LUTs are needed to cover the network. In Figure 6b the original 4-input OR node has been decomposed into a 2-input and a 3-input OR node and only 2 LUTs are needed to cover the network.

The AND-OR decomposition of feasible, and infea-


a) Original 4-input OR node, 4 LUTs

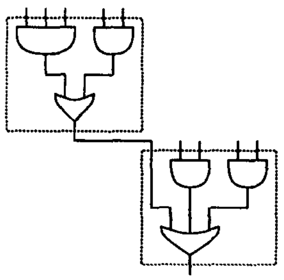

b) OR node decomposed, 3 LUTs

Figure 6: Decomposition of a Feasible Node

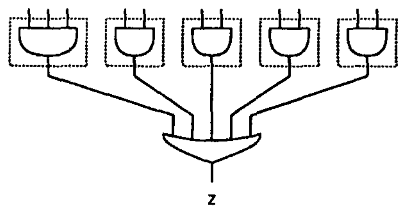
sible nodes, can be combined with a covering algorithm similar to the library-based approach described in Section 2 to optimize the final circuit [10]. The original network consists of AND and OR nodes, and is traversed from the primary inputs to the primary outputs. A circuit implementing each node is constructed from the circuits implementing its immediate fanin nodes. This circuit is optimized to minimize the total number of LUTs, and to minimize the number of inputs used by its root LUT. This increases the number of unused inputs available at the root LUT, and these may reduce the number of LUTs required to implement a subsequent node.

At each node, the root LUTs of the fanin circuits are referred to as the *fanin LUTs*. For example, Figure 7a illustrates the fanin LUTs for the node z. In this example, the LUTs preceding the fanin LUTs are not shown, and the functions implemented by the fanin LUTs are simple AND gates. In general, the fanin LUTs can implement more complex functions.

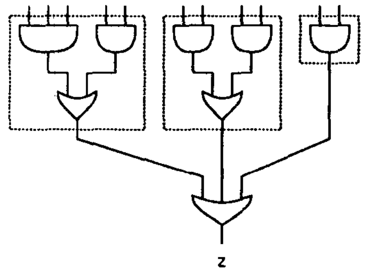
At each node, a tree of LUTs replacing the fanin LUTs and implementing a decomposition of the node being mapped is constructed in two steps. The first step selects decompositions, as shown in Figure 7b, that allow several fanin LUTs to be packed together into a single LUT. The second step connects these LUTs to form the circuit implementing the node being mapped, as shown in Figure 7c. The following sections describe these two steps.

3.2.1 Decomposition Using Bin Packing

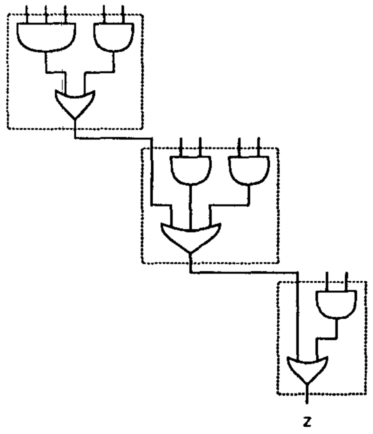
The objective of the first step is the minimization of the number of LUTs into which the fanin LUTs are packed. To determine if a group of fanin LUTs can be packed into a single LUT it is sufficient to count the total number of inputs used by this group. The completeness of K-input LUTs ensures that any group of fanin LUTs that together have no more than K inputs can be implemented by a single LUT. This allows the minimization of the number of packed LUTs to be restated and solved as a bin packing problem.



a) Fanin LUTs



b) After bin packing



c) The final circuit

Figure 7: Decomposition and Covering

The goal of the bin packing problem is to find the minimum number of fixed capacity bins into which a given set of arbitrary sized boxes can be packed. In this case, the boxes are the fanin LUTs and the bins are the LUTs into which they are packed. The size of each box is the number of inputs used by the fanin LUT and the capacity of each bin is K. In Figure 7a the boxes have sizes 3, 2, 2, 2, and 2.

Bin packing is a well known combinational optimization problem, and there exists several effective algorithms for its solution [21]. In particular, the First Fit Decreasing (FFD) algorithm is optimal for boxes and bins with integer sizes less than equal to 6 [22]. The

FFD algorithm begins with an empty list of bins. The boxes are sorted by size and then each box, beginning with the largest, is packed into the first bin in the list into which it fits. If the box does not fit into any bin then it is packed into a new bin added to the end of the list. In Figure 7b the FFD algorithm has packed the fanin LUTs from Figure 7a into LUTs having filled capacities of 5, 4, and 2. Note that packing boxes into bins implies decomposition of the node being mapped.

3.2.2 Completing the Circuit

After the fanin LUTs have been packed into the bins, the final circuit, shown in Figure 7c, is formed by sorting the bins by filled capacity and then connecting the output of each bin to an unused input of one of the following bins. If no unused inputs are available then a new LUT is added to the root of circuit. Connecting the bins alters the decomposition of the node being mapped, however, the completeness of a K-input LUT ensures that each sub-function can be implemented by a single LUT. In addition to minimizing the number of LUTs in the circuit, this approach minimizes the number of inputs used at the root LUT of the circuit. This is an important consideration, since the root LUT becomes a box when the following node is mapped. Smaller boxes can reduce the number of bins required by the bin packing step and lead to a superior circuit.

3.2.3 Optimality

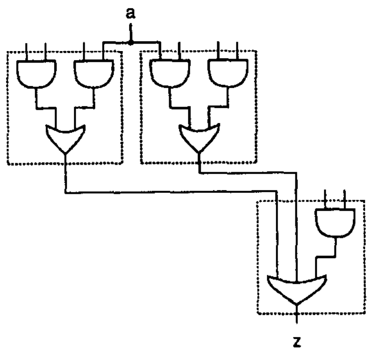
If the original network is a fanout-free tree then the above approach constructs the circuit containing the minimum number of K-input LUTs for values of $K \leq 5$. A similar approach can map fanout-free trees into the the minimum depth circuit for values of $K \leq 6$ [14]. The mapping of trees can be used as part of a divide and conquer strategy to map arbitrary networks by partitioning the network at fanout nodes into a forest of trees that are then mapped separately.

3.3 Covering of Fanout Nodes

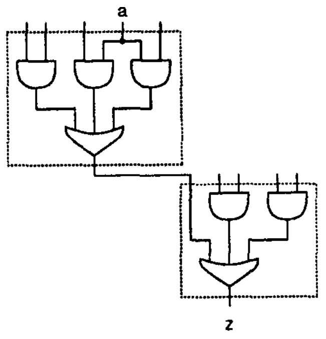
While separate trees can be mapped effectively using the approach described in the previous section, general networks containing fanout nodes present additional challenges and opportunities. The following two sections describe the opportunities presented by reconvergent paths and the replication of logic at fanout nodes, and Section 3.3.3 describes an approach to LUT technology mapping that takes advantage of these opportunities.

3.3.1 Covering Reconvergent Paths

In some networks separate paths that originate at a fanout node reconverge at a subsequent node. For example, in Figure 8a, there are a pair of reconvergent paths originating at the node a and terminating at the node z. If the reconvergent paths are realized by separate LUTs, as in Figure 8a, then each path requires one LUT input connected to the fanout node. If the reconvergent paths are contained within a sub-network that



a) Reconvergent paths covered separately, 3 LUTs



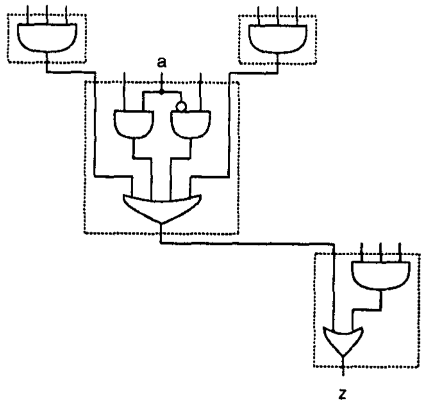
b) Reconvergent paths covered together, 2 LUTs

Figure 8: Covering Reconvergent Paths

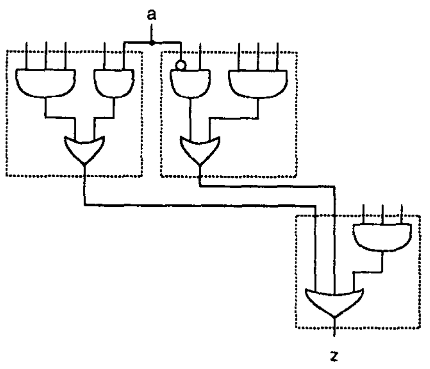
has at most K distinct inputs, as shown in Figure 8b, then the completeness of a K-input LUT ensures that the paths can be covered by a single LUT with only one input connected to the fanout node. The reduction in the number of inputs connected to the fanout node can lead to a superior circuit, as shown in Figure 8b. However, it is not always advantageous to cover reconvergent paths with a single LUT. For example, in the circuit shown in Figure 9a the reconvergent paths originating from the node a are realized within a single LUT, and 4 LUTs cover the network. This network can be covered with a circuit of 3 LUTs, as shown in Figure 9b if the reconvergent paths are covered separately. The challenge for LUT synthesis is to determine when reconvergent paths should be covered by a single LUT.

3.3.2 Replication of Logic at Fanout Nodes

The replication of logic at fanout nodes also presents an opportunity to improve the final circuit. For example, in the 3 LUT circuit shown in Figure 10a the fanout node x is explicitly implemented as the output of a LUT. Replicating the function of this LUT leads to the 2 LUT circuit shown in Figure 10b. The sub-circuits implementing the nodes y and z in Figure 10a have sufficient unused capacity to include a copy of the 3-input AND node, and the completeness of a K-input LUT ensures that the functions incorporating the copies can be realized. Replication of logic is not



a) Reconvergent paths covered together, 4 LUTs



b) Reconvergent paths covered separately, 3 LUTs

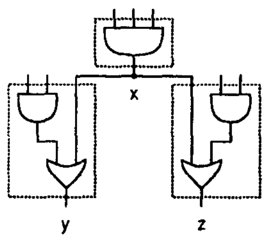
Figure 9: Covering Reconvergent Paths

always beneficial and LUT synthesis must determine at which fanout nodes replication should occur.

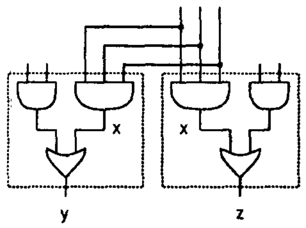
3.3.3 Covering using Edge Visibility

In the final circuit produced by technology mapping every edge of the original network is either driven as the output of a LUT or implemented within a LUT. The covering of reconvergent paths and the replication of logic at fanout nodes can be described by the assignment of edge visibility labels [12]. A *visible* edge is driven by the output of a LUT, whereas an *invisible* edge is implemented within a LUT. For example, the circuit shown in Figure 11b covers by the network shown in Figure 11a. In this circuit the invisible edges (x, y) and (w, y) are represented by curved lines. This assignment of edge labels implies the replication of the 3-input OR node, and leads to a circuit containing 3 LUTs. In Figure 11c the edges (x, z) and (y, z) are invisible, and the reconvergent paths originating at the node w are realized within a single LUT, leading to a circuit containing 2 LUTs.

After the decomposition of infeasible nodes, the assignment of edge labels can be optimized using a divide and conquer strategy. The network is partitioned into sub-networks, and within each sub-network, the



a) Without replication, 3 LUTs



b) With replication, 2 LUTs

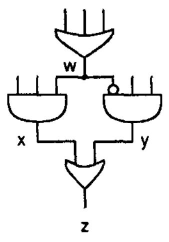
Figure 10: Replication of Logic at a Fanout Node

optimal assignment is found by exhaustively searching all possible combinations of edge labels. If the sub-network containing m edges, there are 2^m different combinations. For each combination a circuit is formed by combining the source and destination nodes of invisible edges into one LUT. If any of the resulting LUTs have more than K inputs, then the combination of edge labels is rejected. Otherwise, the combination resulting in the circuit containing the fewest LUTs is retained. The computational cost of the search is controlled by the limit on the number of edges in the sub-network. In addition, the search can be pruned whenever a combination leading to a LUT with more than K inputs is rejected.

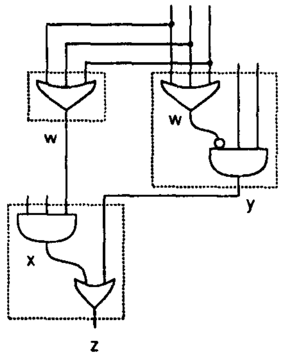
4 Conclusion

The key feature that differentiates the synthesis of LUT circuits from synthesis for conventional ASIC technologies is the completeness of the set of functions that can be realized by a K-input LUT. This completeness simplifies technology mapping by eliminating the need for a library of separate functions. In addition, completeness presents opportunities to select decompositions that improve the final circuit.

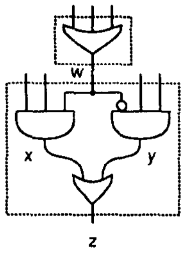
To date research in LUT synthesis has focused primarily on technology mapping. The challenge for the future is to exploit the completeness of LUTs to improve other phases of FPGA synthesis. In particular, delay penalties incurred by programmable routing in FPGAs, provide motivation for the investigation of the combined effect of logic synthesis, placement, and routing on circuit performance. Another important issue is the tradeoff between optimization to fit a design into the fixed logic and routing resources available on a given FPGA, and optimization to improve the performance of the final circuit.



a) Original network



b) Edges $(xz), (wy)$ invisible, 3 LUTs



c) Edges $(xz), (yz)$ invisible, 2 LUTs

Figure 11: Covering Using Edge Visibility

References

[1] S. D. Brown, R. J. Francis, J. Rose, Z. G. Vranesic, **Field-Programmable gate Arrays**, Kluwer Acedemic Publishers, 1992.

[2] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, "Multilevel Logic Synthesis," Proc. of IEEE, Vol. 78, No. 2, Feb. 1990, pp. 264-300.

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

[4] S. Trimberger, "A Small Complete Mapping LIbrary for Lookup-Table-Based FPGAs," 2nd Intl. Workshop on Field-Programmable Logic and Applications, Aug. 1992.

[5] U. Schlichtmann, F. Brglez, M. Hermann, "Characterization of Boolean functions for Rapid Matching in EPGA Technolgy Mapping," Proc, 29th DAC, June 1992, pp. 374-379.

[6] R. J. Francis, J. Rose, K. Chung, "Chortle: A Technology Mapping Program for Lookup Table-Based Field Programmable Gate Arrays," Proc. 27th DAC, June 1990, pp. 613-619.

[7] R. Murgai, Y, Nishizaki, N. Shenay, R. K. Brayton, A. Sangiovanni-Vincentelli, "Logic Synthesis for Programmable Gate Arrays," Proc. 27th DAC, June 1990, pp. 620-625.

[8] P. Abouzeid, L. Bouchet, K. Sakouti, G. Saucier, P. Sicard, "Lexicographical Expression of Boolean Function for Multilevel Synthesis of high Speed Circuits," Proc. SASHIMI 90, Oct. 1990, pp. 31-39.

[9] D. Filo, J. C. Yang, F. Mailhot, G. De Micheli, "Technology Mapping for a Two-Output RAM-based field Programmable Gate Array," Proc. EDAC, Feb. 1991, pp. 534-538.

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

[11] K. Karplus, "Xmap: a Technology Mapper for Table-lookup Field-Programmable Gate Arrays," Proc, 28th DAC, June 1991, pp. 240-243.

[12] N. Woo, "A Heuristic Method for FPGA Technology Mapping Based on Edge Visibility." Proc. 28th DAC, June 1991. pp. 248-251.

[13] R. Murgai, N. Shenoy, R.K. Brayton, A. Sangiovanni-Vincentelli, "Improved Logic Synthesis Algorithms for Table Look Up Architectures," Proc. ICCAD, Nov. 1991, pp. 564-567.

[14] R. J. Francis, J, Rose, Z. Vranesic, "Technology Mapping of Lookup Table-Based FPGAs for Performance," Proc. ICCAD, Nov. 1991. pp. 568-571.

[15] R. Murgai, N. Shenoy, R.K. Brayton, "Performance Directed Synthesis for Table Look Up Programmable," Gate Arrays, Proc. ICCAD, Nov. 1991 pp. 572-575

[16] K. C. Chen, "Logic Minimization of Lookup-Table Based FPGAs," 1st Intl Workshop on FPGAs, Feb. 1992, pp. 71-76.

[17] P. Sawkar, D. Thomas "Area and Delay Mapping for Table-Look-Up Based Field Programmable Gate Arrays," Proc, 29th DAC, June 1992, pp. 368-373.

[18] J. Cong, T. Ding, A. Kahng, P. Trajmar "Graph Based FPGA Technology Mapping for Delay Optimization," Proc. ICCD, Oct. 1992.

[19] E. J. McClusky, **Logic Design Principles**, Prentice Hall, 1986.

[20] C. E. Shannon, "The Synthesis of Two-Terminal Switching Circuits," Bell Syst. Tech. Journal, Vol. 28, 1949, pp. 59-98.

[21] M. R. Garey, D. S. Johnson, **Computers and Intractability, A Guide to the Theory of NP-Completeness**, W. H. Freeman and Co., 1979.

[22] R. J. Francis, **Technology Mapping for Lookup Table-Based FPGAs**, Ph.D. Thesis in preparation, University of Toronto, Department of Electrical Engineering.