Complexity Bounds for Lookup Table Implementation of Factored Forms in FPGA Technology Mapping

Wenyi Feng[1], Fred J. Meyer[2], and Fabrizio Lombardi[2]

[1] FPGA Software Core Group, Lucent Technologies, 1247 S Cedar Crest Blvd, Allentown PA 18103
[2] Electrical & Computer Engineering, Northeastern University, 360 Huntington Avenue, Boston MA 02115

Abstract. We consider technology mapping from factored form (binary leaf-DAG) to lookup tables (LUTs), such as those found in field programmable gate arrays. Polynomial time algorithms exist for (in the worst case) optimal mapping of a single-output function. The worst case occurs when the leaf-DAG is a tree. Previous results gave a tight upper bound on the number of LUTs required for LUTs with up to 5 inputs (and a bound with 6 inputs). The bounds are a function of the number of literals and the LUT size. We extend these results to tight bounds for LUTs with an arbitrary number of inputs.

1 Introduction

We view computer-aided synthesis of a logic circuit in two major steps: (1) the optimization of a technology-independent logic representation, using Boolean and/or algebraic techniques, and (2) technology mapping. Logic optimization is used to transform a logic description such that the resultant structure has a lower cost than the original [1].

Technology mapping is the task of transforming an arbitrary multiple-level logic representation into an interconnection of logic elements from a given library of elements. Technology mapping is very crucial in the synthesis of semicustom circuits for different technologies, such as sea-of-gates, gate arrays, or standard cells. The quality of the synthesized circuit, both in terms of area and performance, depends heavily on this step.

We focus on the problem of technology mapping onto Field-Programmable Gate Arrays (FPGAs). FPGAs are prewired circuits that are programmed by the users to perform the desired functions [13]. In particular, we consider FPGAs where the logic functions are implemented with lookup tables (LUTs). In a LUT-based FPGA, the basic block is a K–input, single-output LUT (K–LUT) that can implement any Boolean function of up to K variables. The technology mapping problem for LUT-based FPGAs is to generate a mapping of a set of Boolean functions onto K–LUTs. Traditional library binding algorithms for standard cells and Mask-Programmable Gate Arrays (MPGAs) are not applicable to FPGAs because the virtual library of a LUT is too large to enumerate (a K–LUT can realize 2^{2^K} logic functions). Many papers have proposed algorithms for LUT-based technology mapping. They can be divided into 3 categories: (1) minimization of number of levels of LUTs in the mapped network [5]; (2) minimization of the number of LUTs used in the mapped network [3, 10, 7, 6], (3) routability of the mapping solution [2, 11], or combinations of these topics [4, 3].

Minimizing the number of levels is solvable in polynomial time in Flow-Map [5]. The key feature in Flow-Map is to compute a minimal height K–feasible cut in the input network. Minimization of the number of LUTs is a much harder problem. It was shown to be $\mathcal{NP}$–hard even for restricted cases [6]. So, heuristics are used in all mapping systems.

In this paper, we restrict our attention to mapping a single-output function onto LUT technology. We specify the input function with a graph, where each node represents a function of 2 inputs. We constrain the problem so that the synthesis must be conducted without being aware of (taking advantage of) the underlying function at each 2–input node.

2 Preliminaries

Definition 1. A *leaf-DAG* is a general case of a tree—the leaves of the tree (primary inputs) are allowed to fan out. If node i is one of the inputs to node j in a leaf-DAG, we say that i is a *child* of j and that j is a parent of i.

In mapping, we will not take any special advantage of leaf-DAGs; instead, we will regard the inputs to the various nodes in the DAG as coming from distinct primary inputs—i.e., we will not take advantage of any knowledge of fan-out at the primary inputs. This yields bounds that are applicable in any case and, in particular, in the worst case of a tree.

• $p(v)$. Apart from the leaves, each node, v, in a leaf-DAG has a unique parent, $p(v)$.

• $l(S)$. The number of literals of the input function, S. This is the sum of the number of inputs to all nodes of the input graph. We simply use l, instead of $l(S)$, whenever S is understood.

Definition 2. The size or complexity, $C(S)$, of a circuit, S, is the number of gates (number of nodes in its DAG). The circuit complexity of a function, f, with respect to a basis, Ω, is $C_\Omega(f)$, which is the minimal number of gates from the set Ω in order to compute f.

• K. The LUTs in the technology to be mapped onto have K inputs. We call them K–LUTs. A K–LUT implements the basis B_K.

• $L_K(f)$. The number of LUTs needed to map function f to K–LUTs. We use $L(f)$, L_K, and L whenever f and/or K are understood.

• $C_K(l)$. This is the circuit complexity for leaf-DAGs mapped onto K–LUTs. It is the worst case, over all functions represented by leaf-DAGs with l literals, of the minimal number of K–LUTs required to implement the function.

Definition 3. A *factored form* of a one-output function is a generalized sum-of-products form allowing nested parentheses and arbitrary binary operations.

A factored form is represented by a binary leaf-DAG (all gates are in B_2).

For example, the function $ab'c' + a'bc' + d$ can be represented in a factored form with 7 literals as $(((ab')c') + ((a'b)c')) + d$, and it can be written more compactly in factored form as $((a \oplus b)c') + d$ with 4 literals.

When all l literals of a factored form are different, its corresponding binary leaf-DAG is a binary tree. The binary tree has l inputs and $l-1$ internal nodes. Figure 1 shows a binary tree with $l = 7$ inputs and $l - 1 = 6$ internal nodes.

If all inputs of a binary leaf-DAG, D, are different, we have a binary tree, B. So, a realization of B would also serve as a realization of D. Perhaps some other realization of D requires fewer LUTs, using some structural information of D.

Lemma 4. *Suppose a binary tree, B, is obtained from a binary leaf-DAG, D, by viewing all D's inputs as different.* $L_K(D) \leq L_K(B)$

This lemma tells us that, in order to analyze the worst case complexity of binary leaf-DAG mapping, it is enough to analyze binary trees.

In [9], some results are provided on the complexity bound of a function, f, given in a factored form. The results are summarized in the following theorem.

Theorem 5. *For the class of functions with l literals in factored form,*

$$\begin{aligned} C_2(l) &= \quad l-1 \quad (l \geq 2) \\ C_3(l) &= \lfloor (2l-1)/3 \rfloor \ (l \geq 2) \\ C_4(l) &= \lfloor (l-1)/2 \rfloor \ (l \geq 3) \\ C_5(l) &= \lfloor (2l-1)/5 \rfloor \ (l \geq 4) \\ C_6(l) &\leq \lfloor (l-1)/3 \rfloor \ (l \geq 6) \end{aligned}$$

Reference [6] presented an optimal algorithm, Tree-Map, for technology mapping where the input is a tree. Tree-Map uses a greedy dynamic programming approach, which happens to guarantee an optimal mapping.

Our approach to determining a tight bound for $C_K(l)$ for all l is to analyze a technology mapping algorithm that is optimal on trees. We use the Tree-Map algorithm [6], because it is easiest to analyze.

Definition 6. For a tree, $T(V, E)$, its *height* is the number of nodes on the longest path from an input to the root. The *level* of the root is the height of the tree. The level of a node (excluding the root) is the level of its parent minus 1.

Definition 7. Consider a tree, $T(V, E)$, with vertex (node) set, V, and edge set, E. Let $V1$ be a subset of V such that a LUT is assigned to precisely those vertices in $V1$. Two quantities are defined for each vertex $v \in V$. These quantities are its *dependency*, $d(v)$, and its *contribution*, $Z(v)$, defined according to:

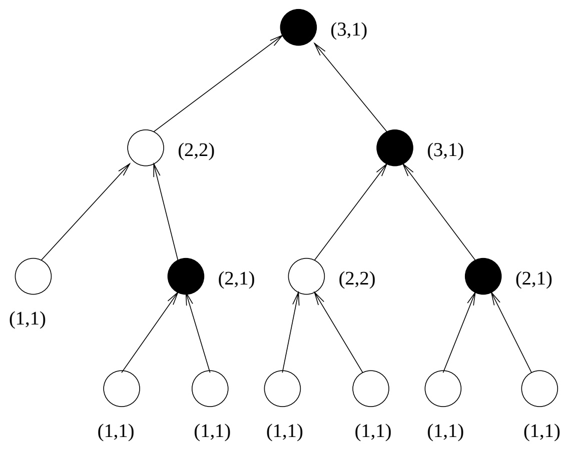
– Contribution, $Z(v)$:

  - For each primary input (or literal), v, $Z(v) = 1$.
  - For each $v \in V1$, $Z(v) = 1$.
  - For all other vertices $v \in V$, $Z(v) = Z(u_1) + \ldots + Z(u_{c(v)})$—where v has $c(v)$ children: $u_1, \ldots, u_{c(v)}$.
- Dependency, $d(v)$:
  - For each primary input (or literal), v, $d(v) = 1$.
  - For all other vertices $v \in V$, $d(v) = Z(u_1) + \ldots + Z(u_{c(v)})$—where v has $c(v)$ children: $u_1, \ldots, u_{c(v)}$.

Definition 8. In a mapping, if a node is assigned a LUT, we say it is a *LUT node.* Otherwise, we say it is a *free node.*

From Def. 7, we know that, for a free node, its contribution is equal to its dependency, but for a LUT node its contribution is set to 1.

Note that $d(v)$ is the summation of the number of inputs or LUTs that directly or indirectly supply input to vertex v, and it represents the number of signals that would need to be placed at v if the signal at v were implemented with a LUT. The quantity $Z(v)$, on the other hand, represents the contribution of vertex v to the dependency of its parent vertex. Figure 1 shows an example of a tree and the assignment of LUTs to its vertices. The shaded vertices in the figure represent the LUT nodes. The dependency and contribution values for each node, v, in the tree are shown with an ordered pair, $(d(v), Z(v))$.



Notation (d,Z) represents the dependency and contribution values for a vertex in the tree.

Fig. 1. The dependency and contribution values for a tree

The Tree-Map algorithm scans from leaves to the root, assigning LUTs as necessary. Whenever it encounters a node with dependency exceeding K, it must assign LUTs. It assigns LUTs to that node's children, starting with the child with largest contribution, until the node's dependency has been sufficiently reduced. This greedy mapping is optimal with respect to the number of LUTs [6].

Our objective is to derive a tight bound for general K–LUT technology mapping. We use Tree-Map [6] as a unified optimal mapping algorithm. Although the dynamic programming algorithm in [7] is also optimal for tree mapping, it is hard to work from it to derive bounds on the circuit complexity. Tree-Map takes a DAG input. In this paper, we constrain it to be a leaf-DAG—i.e., each internal node has fanout 1, the primary inputs may fan out arbitrarily. This is a generalization of trees [6] through allowing the primary inputs to fan out. Generally, the leaf-DAGs will be in factored form, because that is the worst case in terms of LUT complexity.

We do not assume that we know the individual functions used in the formula. For example, if an output of an AND gate goes to another AND gate, we do not allow any inputs to be rearranged between the two gates. In short, the output of the technology mapping must be valid, even if arbitrary functions are substituted for each of the input leaf-DAG's gates.

3 Worst Case Mapping to K–LUTs

Tree-Map proceeds from level to level in the tree. When we deal with vertices at level j, all vertices below level j have dependencies less than or equal to K. Tree-Map had processed all nodes at lower levels. Whenever any of them had dependency more than K, it assigned sufficient LUTs to reduce the dependency to at most K.

Lemma 9. *If K is even,*

$$d(v_i) \geq (K/2 + 1), \qquad 1 \leq i \leq L - 1 \tag{1}$$

Proof: i ranges up to $L-1$, so it includes all the LUTs, except the one assigned to the root. According to the Map-Tree algorithm, a node i (except the root) is assigned a LUT only because its parent has dependency larger than K before the assignment. Furthermore, it is selected to be assigned a LUT because its dependency is at least as large as the (only) other child of its parent. So, its dependency must be $\geq K/2 + 1$. □

Lemma 10. *If K is odd,*

$$d(v_i) \geq (K + 1)/2, \qquad 1 \leq i \leq L - 1 \tag{2}$$

Proof: Similar to the proof for Lemma 9. □

Lemma 11. *Suppose K is odd, and v_i ($i \neq L$) is a node with $d(v_i) = (K+1)/2$. Suppose v_j is the first LUT node on the path from v_i to the root. Then:*

$$d(v_j) \geq (K + 3)/2 \tag{3}$$

and we say v_j is the pair node *of v_i.*

Proof. Omitted for brevity. □

Lemma 12. *Suppose K is odd, and v_i and v_j $(i, j \neq L)$ both have dependency $(K+1)/2$. Then their pair nodes are two different nodes.*

Proof. Contrariwise, suppose v_1 were the pair node of both v_i and v_j. According to the proof of Lemma 11, v_1 must resolve $\geq 2 \cdot (K+3)/2$ dependency, which cannot be true. □

Lemma 13. *Suppose $L > 1$ and $v_L = r$. Suppose v_i is a LUT node nearest to r (if there are multiple such nodes, select any one). So, on the path from v_i to r, no other LUT node exists. Then:*

$$d(r) + d(v_i) \geq K + 2 \tag{4}$$

and we say that v_i is the pair node of r.

Proof. Omitted for brevity. □

Now we are able to prove our key theorem.

Theorem 14.

$$C_K(l) = \begin{cases} \lfloor (2l-2)/K \rfloor, & \text{if } K \text{ is even} \\ \lfloor (2l-1)/K \rfloor, & \text{if } K \text{ is odd} \end{cases}$$

Proof. For brevity, we omit the half of the proof that the bound is always achievable. We only give the half of the proof that the bound is tight.

To show tightness, we need to show some trees that meet the upper bound. We consider two cases.

(1) K is even. Figure 2 shows an example. Node v_a is the root of a binary tree with $K/2+1$ inputs; each of the nodes v_b, v_d, v_f, ... is the root of a binary tree with $K/2$ inputs. The shaded nodes show the nodes to which a K–LUT should be assigned according to the Tree-Map algorithm. For example, when node v_c is visited, the dependency $d(v_c)$ is $K+1$, and we put a K–LUT at node v_a, and so on. The value $K/2+1$ beside node v_a represents the amount of dependency resolved at node v_a—i.e., $d(v_a)$. The value $K/2$ beside node v_b represents $d(v_b)$.

Suppose the number of LUTs in the figure is L. The total tree inputs is

$$l = L(K/2) + 1 \tag{5}$$

So, the tree needs the upper bound number of K–LUTs. Therefore, the bound is tight in this case.

(2) K is odd. We show two subcases in Fig. 3. In the first (second) case, according to the Tree-Map algorithm, an odd (even) number of LUTs is needed.

Suppose the number of K–LUTs needed is L.

In the first subcase (L is odd), the number of inputs is:

$$l = (LK + 1)/2 \tag{6}$$

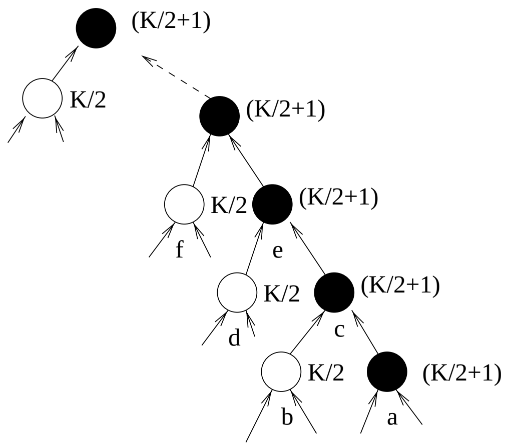


Fig. 2. Proof of tightness when K is even

It meets the upperbound. (In this case, $\lfloor(2l-1)/K\rfloor$ is 1 more than $\lfloor(2l-2)/K\rfloor$, and we need $\lfloor(2l-1)/K\rfloor$ K–LUTs.) Therefore, this also shows that, for each l that makes $(2l-1)/K$ an (odd) integer, there exists a binary tree that needs $(2l-1)/K$ K–LUTs.

In the second subcase (L is even), the number of inputs is:

$$l = L(K/2) + 1 \tag{7}$$

It also meets the upperbound. (In this case, $\lfloor(2l-1)/K\rfloor$ is equal to $\lfloor(2l-2)/K\rfloor$). □

4 Conclusion

Arbitrary functions can be mapped onto FPGAs that use lookup tables (LUTs). If the input function is in the form of a tree or leaf-DAG [9], a greedy algorithm can process the input in polynomial time. In the case of a tree, the greedy algorithm minimizes the number of LUTs, subject to the constraint that the algorithm is not allowed to exploit any knowledge of the particular functions represented by the nodes in the input graph. In the case of a leaf-DAG, the number of LUTs needed is bounded by that required for an equivalent tree representation using unique literals.

We differentiate between LUTs by the number of inputs they handle, K. We considered leaf-DAGs where all nodes are 2–input functions. This is the worst case in terms of how many K–LUTs are required. Previous work [9] had obtained bounds on the worst case number of K–LUTs for K up to 6 (tight bounds up to 5). We extended this to tight bounds for all K.

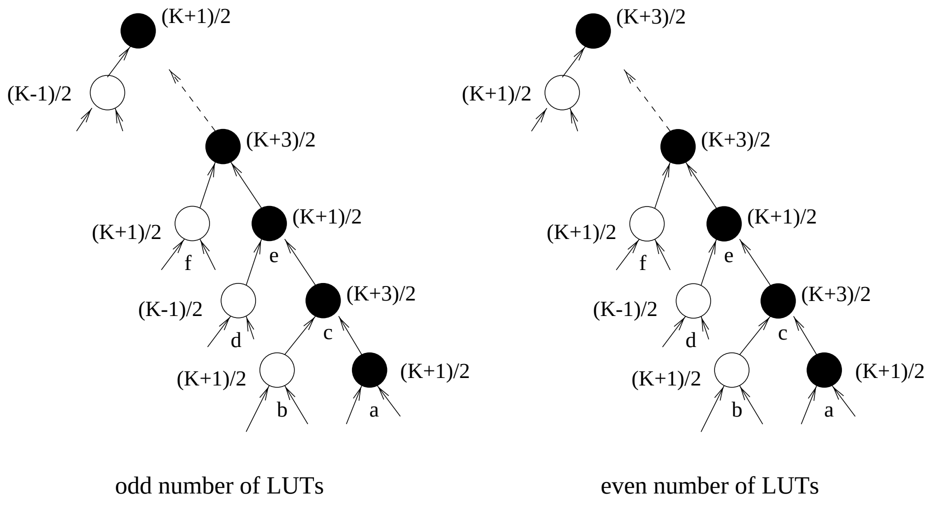


Fig. 3. Proof of tightness when K is odd

References

1. Brayton, R. K., Rudell, R., Sangiovanni-Vincentelli, A.: Mis: A multiple-level logic optimization system. IEEE Trans. CAD of Int. Circ. and Sys. **6** (1987) 1062–1081
2. Chan, P. K., Zien, J. Y., Schlag, M.: On routability prediction of FPGAs. IEEE/ACM Des. Auto. Conf. (1993) 326–330
3. Chaudhary, K., Pedram, M.: A near optimal technology mapping minimizing area under delay constraints. IEEE/ACM Des. Auto. Conf. (1992) 492–498
4. Cong, J., Ding, Y.: On area/depth trade-off in LUT-based FPGA technology mapping. IEEE/ACM Des. Auto. Conf. (1993) 213–218
5. Cong, J., Ding, Y.: Flowmap: An optimal technology mapping algorithm for delay optimization in look-up table based FPGA designs. IEEE Trans. CAD of Int. Circ. and Sys. **13** (1994) 1–12
6. Farrahi, A. H., Sarrafzadeh, M.: Complexity of the look-up table minimization problem for FPGA technology mapping. IEEE Trans. CAD of Int. Circ. and Sys. **13** (1994) 1319–1332
7. Francis, R. J., Rose, J., Chung, K.: Chortle: A technology mapping algorithm for lookup table based FPGAs. IEEE/ACM Des. Auto. Conf. (1990) 613–619
8. Francis, R. J., Rose, J., Vranesic, Z.: Chortle-crf: Fast technology mapping for lookup table based FPGAs. IEEE/ACM Des. Auto. Conf. (1991) 227–233
9. Murgai, R., Brayton, R. K., Sangiovanni-Vincentelli, A.: Logic Synthesis for FPGAs. Kluwer Academic Publishers (1995)
10. Murgai, R., Nishizaki, Y., Shenoy, N., Brayton, R. K., Sangiovanni-Vincentelli, A.: Logic synthesis algorithms for programmable gate arrays. IEEE/ACM Des. Auto. Conf. (1990) 620–625
11. Schlag, M., Kong, J., Chan, P. K.: Routability driven technology mapping for look-up table FPGAs. IEEE Int. Conf. Comp. Des. (1992) 89–90
12. Wegener, I.: The Complexity of Boolean Functions. Wiley-Teubner (1987)
13. Xilinx Corporation: Xilinx FPGA Data Book. (1996)