# Depth Optimal Incremental Mapping for Field Programmable Gate Arrays

Jason Cong
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
cong@cs.ucla.edu

Hui Huang[1]
Computer Science Department
Stanford University
Stanford, CA 94305
huanghui@cs.stanford.edu

## ABSTRACT

In this paper, we study the incremental technology mapping problem for lookup-table (LUT) based Field Programmable Gate Arrays (FPGAs) under incremental changes. Given a gate-level network, a mapping solution associated with it, and a sequence of changes to the original network, we compute a new mapping solution by modifying the existing one. Moreover, we assume that the given mapping solution is depth-optimal and we are required to come up with a modified mapping solution that maintains the depth optimality. The objective of our incremental mapper is to maintain depth-optimality with very high efficiency while minimizes the modifications to the existing mapping solution. We revealed a set of sufficient conditions for maintaining depth optimal mapping solution after a sequence of incremental changes. Based on these results, we developed a very fast incremental technology mapping algorithm, called IncFlow, that runs up to 300 x faster than the well-known depth-optimal FlowMap algorithm [1](with an average of 14x speedup) while achieves the same depth-optimal mapping quality.

## 1. INTRODUCTION

Similar to all software development efforts, the complete process of system design usually goes through many incremental changes before the design is completed. As a result, it may require many iterations of synthesis, placement and routing, simulation, and timing analysis to complete a complex design. FPGAs show great advantage in supporting such incremental, iterative design methodology due to its re-programmability and high flexibility. On the other hand, traditional compilation techniques do not take advantages of incremental changes and will re-do the whole thing after every iteration. They are not suitable for supporting large designs with possible multiple design iterations. Fast incremental compilation techniques are especially important for supporting such applications as well as runtime-configuration applications [9]. This work on incremental technology mapping is a part of an overall effort at UCLA in developing a highly efficient *incremental* compilation system for FPGAs.

The incremental technology mapping system should consider the following three important objectives:

1. "*Preservability*". The incremental mapping system should preserve as much information as possible from the existing mapping solution. This will lead to fast convergence in the design process.
2. *Efficiency*. A faster mapping system will enable more design iterations and shorten the overall design time.
3. *Quality* of the mapping solution (such as the delay or area) should be as close as possible to that by complete re-maping.

Many technology mapping algorithms for FPGAs have been published in recent years, for example, tree-based Chortle-family algorithms by Francis *et al.* [4][5], the depth-optimal FlowMap algorithm by Cong and Ding [1], the synthesis-based MIS-pga family by Murgai *et at.* [7][8], and the area-minimal mapping algorithm Praetor by Cong *et al* [2]. (See [3] for a more comprehensive survey.) However, none of these algorithms is designed specifically for incremental changes.

Very few papers addressed the incremental design issues for FPGAs. Kukimoto *et al*. presented a redesign technique for FPGAs [6]. However, [6] focused on completely keeping the network structure. Limited by only changing the functionality of lookup-tables with all routing perserved, it will fail on some circuits and cannot handle all types of incremental changes.

We focus our study on fast incremental technology mapping algorithm for lookup table (LUT) based FPGAs under delay constraints. Given a gate-level network, a depth optimal mapping solution associated with it, and a sequence of changes to the original network, we compute a new depth optimal mapping solution by modifying the existing one. The objective of our incremental mapper is to maintain depth-optimality with very high efficiency (i.e. much shorter runtime compared to complete re-mapping) while minimizes the modifications to the existing mapping solution. We will discuss a set of sufficient conditions for maintaining depth optimal mapping solution after incremental changes, and then we will show a fast incremental mapping algorithm, called IncFlow, which can achieve the same mapping quality as FlowMap while runs 14x faster on average.

Sections 2 formulates the problem; Section 3 discusses the properties of incremental mapping and outlines IncFlow algorithm; Section 4 shows the experimental results of IncFlow; Section 5 concludes the paper and discusses the future work. Due to page limitation, proofs of the theorems and other details are left out and available from [10]

## 2. DEFINITIONS AND PRELIMINARIES

### 2.1 Basics

A Boolean network $N$ can be represented as a directed acyclic graph (DAG) where each node represents a logic gate. A directed edge $(i,j)$ exists if the output of gate $i$ is an input of gate $j$. A *primary input* (*PI*) node has no incoming edge and a *primary*


DAC 2000, Los Angeles, California
(c) 2000 ACM 1-58113-188-7/00/0006..$5.00

1. This work was done when the second author was with University of California, Los Angeles

*output* (PO) node has no outgoing edge. We use *input*($v$) to denote the set of nodes which are the fan-ins of gate $v$. Similarly, *output*($v$) is used to denote the set of $v$'s fan-outs. When used with a subscript, e.g. $input_N(v)$ or $output_N(v)$, we refer to the *input* of $v$ or the *output* of $v$ in a specific network $N$. We assume the network is *k-bounded*, that is, for any node $v$ in $N$, $| input_N(v) | \le k$.

A *cone* at node $v$, denoted as $C_v$, is a subgraph consisting of $v$ and its predecessors such that any path connecting a node in $C_v$ and $v$ lies entirely in $C_v$. The notation of $input(C_v)$ or $input_N(C_v)$ is also used to represent the set of distinct nodes outside $C_v$ which supply inputs to the gates in $C_v$. A *maximum cone* at $v$, also known as the *transitive fanin network* of $v$, denoted as $N_v$, is a cone consisting of $v$ and *all* of its predecessors. A cone $C_v$ is said to be *k-feasible* if and only if $|input(C_v)| \le k$.

Several concepts about *cuts* in a network will be used in our discussion. Given a network $N$ with a source $s$ and a sink $t$, a cut (X, X') is a partition of the nodes in the network such that $s \in$ X, $t \in$ X', and no nodes in X' provide input to any node in X. In particular, when a cut is computed for the transitive fanin network of node $t$, X' may be considered as a cone rooted at $t$ inside network $N$. Therefore, we can apply the previous definitions on $k$-feasibility to cuts. A cut (X, X') is said to be *k-feasible* if and only if X' is a *k-feasible* cone, otherwise it is *k-infeasible*. For every node $v$ and its fanin network $N_v$, a cut (X,X') in $N_v$ is a partition of the nodes such that all PI nodes belong to X and $v$ belongs to X'. It is clear that every cone rooted at $v$ corresponds to a cut in $N_v$.

*The technology mapping problem for k-LUT based FPGAs is to cover a given k-bounded Boolean network with k-feasible cones.* Note that cones can overlap and possible gate duplication is allowed. Gate duplication may help delay minimization as it increases the parallelism in circuit [1].

Consistently in the later part of this paper, we will use letter $N$ to represent the original *unmapped* network while using letter $M$ to represent the corresponding *mapped* $k$-LUT network. We use the notation *LUT*($v$) to represent the $k$-LUT rooted at $v$. The minimum possible depth of $M$ is called the *minimum mapping depth* of $N$. We use the notation *label*($v$) to denote the *minimum mapping depth* of $N_v$ for a given node $v$. *label*($v$) is also called the *minimum mapping depth* of $v$. If used with a subscript, e.g. $label_N(v)$, it is referred to the *label* of $v$ in network $N$.

## 2.2 Primitive Changes

To study the property of incremental modification, we break down an arbitrary modification into primitive changes. Given a network $N$, any modification to $N$ can be decomposed into a sequence of following primitive changes:

1. adding a degree-0[1] node to the network,
2. deleting a degree-0 node from the network,
3. adding an edge between two existing nodes,
4. deleting an edge between two existing nodes,
5. changing the function of a single node.

It is easy to show that an arbitrary modification made by the designer can be represented by a list of primitive changes, denoted as $L$, such that applying $L$ on the original network $N$ will result in the modified network $N'$.

## 2.3 Problem Formulation

Given a gate-level network $N$, a corresponding $k$-LUT mapping solution $M$ and a list of primitive changes, denoted as $L$, the incremental technology mapping problem is to compute a new mapping solution, denoted as $M'$, for the *modified* network, denoted as $N'$, which results from applying $L$ to $N$. The goal of incremental mapping is to minimize the runtime and the changes in the mapping solution while optimizing certain design metric. In this paper, we focus on the depth optimal mapping problem which requires that both $M$ and $M'$ are depth optimal mapping solutions. Given a depth optimal mapping solution $M$, we try to obtain the depth optimal mapping solution $M'$ for $N'$.

# 3. IncFlow ALGORITHM

## 3.1 Brief Review of FlowMap Algorithm

FlowMap ([1]) is a depth optimal mapping algorithm for $k$-LUT based FPGAs. In FlowMap, every node $v$ in the network $N$ has a *label*, denoted as *label*($v$). FlowMap formulates the problem of finding *LUT*($v$) as computing a *minimum height k-feasible* cut (X,X') in $N_v$, where the height, denoted as $h$(X, X'), is the largest *label* of nodes in $X$. Under the labeling rule of FlowMap, *label*($v$) is 0 for primary inputs and $h$(X,X')+1 for non-PI nodes. It was shown that *label*($v$) equals to the minimum mapping depth of $N_v$ for any node $v$. After every node's *label* has been calculated, the mapping phase will generate the mapping solution from PO to PI in the reverse topological order. *LUT*($v$) is generated according to the *minimum height k-feasible* cut (X,X') of $v$. The key step of FlowMap is to compute the *minimum height k-feasible* cut for each node. It is converted into finding the *max-volume-min-cut* in the *induced network* of $N_v$. FlowMap can find the depth optimal mapping solution for a $k$-bounded network with $n$ nodes and $m$ edges in O($kmn$) time.

## 3.2 Overview of IncFlow Algorithm

IncFlow has a similar flavor as FlowMap. It is constituted of two major steps: incremental label update and incremental mapping solution generation.

Given the original network $N$, its depth optimal mapping solution $M$, and a list $L$ of primitive changes, IncFlow assumes every node $v$ in $N$ already has a *label*, denoted as $label_N(v)$, which is the minimum mapping depth of $N_v$.

In the label update phase, IncFlow tries to determine if a node's *label* needs to be re-calculated. If the *label* of $v$ needs to be re-calculated, IncFlow computes the minimum height $k$-feasible cut for $N'_v$ in the *modified* network using flow computation as in [1] and update $v$'s *label* by setting $label_{N'}(v)$ to the new value. If node $v$'s *label* does not need to be re-calculated, IncFlow will let $label_{N'}(v)=label_N(v)$. IncFlow will enter the incremental mapping phase if there is no more node whose *label* needs update.

In the mapping phase, IncFlow will determine if it needs to re-generate *LUT*($v$) for each possibly affected node $v$. If so, IncFlow generates *LUT*($v$) according to the *minimum height k-feasible* cut of $N'_v$. Finally, IncFlow will eliminate redundant LUTs in the new solution.

## 3.3 Incremental Label Update

The purpose of labeling phase is to find the minimum mapping depth for each node. Under incremental label update, $label_{N'}(v)$ may be obtained by copying the old value from $label_N(v)$ or by re-calculation. Therefore, the fundamental of incremental label update is to identify those nodes whose $label_{N'}$ *might* be different from $label_N$ and only update *labels* for such nodes.

A *transitive fan-out graph* of node $n$, denoted as TF($n$), is a subgraph of $N$ such that:

1) $n \in$ TF($n$),

2) $v \in$ TF($n$) if and only if at least one of its fan-in belongs to TF($n$).

Given $Q$ is a list of nodes, a *transitive fan-out graph* of $Q$, denoted as TF($Q$), is the union of TF($n$) for each node $n$ in the list $Q$.

[1] A degree-0 node is a node without fan-in and fan-out.

A node $v$ in $N'$ is said to be *modified* if either $v \notin N$, or either $input_{N'}(v) \neq input_N(v)$ or $output_{N'}(v) \neq output_N(v)$.

Suppose the list $Q$ contains all the *modified* nodes in $N'$, it is not difficult to show that:

**Lemma 1** For any node $v$, if $v \notin$ TF($Q$), $label_{N'}(v) = label_N(v)$.

Lemma 1 suggests that only the *label* of nodes inside TF($Q$) need to be updated. Based on this result, the following Algorithm is a straightforward implementation of the label update phase:

**Algorithm 1** Incremental Update – Complete update of TF($Q$)

**foreach** node $n$ in TF($Q$) in topological order **do**
  re-calculate $label(n) = h$(X,X')+1

Algorithm 1 will re-label all nodes in TF($Q$). However, in practice, it is likely only a small portion of TF($Q$) really needs to be updated. It is more desirable if we could stop re-labeling a node's fan-outs under certain conditions.

Intuitively, one might think if $n$ keeps the old *label* after re-calculation, i.e. $label_{N'}(n) = label_N(n)$, then $n$'s fan-outs do not need to be re-labeled. Unfortunately, it is not always true. Instead, we have the following:

**Theorem 1** If the modification did not remove any edge from $N$, we do not need to further re-label $n$'s fan-outs if:

1. $label_{N'}(n) = label_N(n)$, *and*,
2. $n$ is *not* covered by any $LUT(v)$ in $M$ where $v \neq n$

Therefore, if the modification does not include deleting an edge, we can use a more efficient algorithm to update *label*s.

**Algorithm 2** Incremental Update – Partial Update of TF($Q$)

**foreach** node $n$ in TF($Q$) **do**
  **if** ($n$ is in $Q$) **then** $n$.NEED_RELABEL:=**true**;
  **else** $n$.NEED_RELABEL:=**false**;
**foreach** node $n$ in TF($Q$) in topological order **do** {
  **if** ($n$.NEED_RELABEL) **then**{
    re-calculate $label(n) = h$(X,X')+1
    **if** ($label(n)$ changed **or**
      $n$ is covered by some $LUT(v)$ in $M$ but $v \neq n$)
    **then foreach** $v$ in output($n$) **do** $v$.NEED_RELABEL:=**true**;
}}

However, If the modification did remove some edge from $N$, Algorithm 2 *cannot* guarantee that every node still has a label that equals to its *minimum mapping depth*. In this case, we should follow algorithm 1 to update labels.

The label update phase of IncFlow combines Algorithm 1 and Algorithm 2. If the modification did not remove any edge, IncFlow follows Algorithm 2 to update labels, otherwise it follows Algorithm 1.

Suppose (X,X') is the minimum height $k$-feasible cut of $N_v$ for node $v$. Cone X' can be used as $LUT(v)$ for $v$ in the mapping solution. This information should be stored, say, as $Cluster(v)$. When a node $v$ is re-labeled, IncFlow will find a new minimum height $k$-feasible cut and update $Cluster(v)$ to reflect the change. In that case, IncFlow will also set a flag variable $v$.NEED_REMAP to be true, which tells the mapping phase that $LUT(v)$ needs to be re-generated.

## 3.4 Incremental Mapping Solution Generation

Labeling of the network follows the topological order from PIs to POs. Generation of the mapping solution is carried out in the reverse direction. Instead of completely regenerating the whole network, incremental mapping solution generation will directly modify the existing mapping solution. This process may include adding additional $k$-LUTs, removing redundant $k$-LUTs and changing connections between $k$-LUTs.

Since the label update phase has prepared the information on whether or not a $LUT(v)$ needs to be re-generated, the incremental mapping phase can use this information: if a node $v$ is marked as NEED_REMAP by the incremental labeling phase, the incremental mapping phase will generate $LUT(v)$ and add it to the mapping solution or use it to replace the old $LUT(v)$.

When $LUT(v)$ has been re-generated or added to the mapping solution, one of its inputs, for instance, $u$, may not exist in the original mapping solution. $u$ may not even belong to TF($Q$) and therefore can not be marked as NEED_RELABEL in the labeling phase. In this case, $LUT(u)$ must be generated and added into the mapping solution. We should use the cut information stored in $Cluster(u)$ to form $LUT(u)$.

The incremental mapping phase in IncFlow is like this:

**Algorithm 3** Incremental Mapping;

**foreach** node $v$ in $N'$ in reverse topological order from PO to PI **do** {
  **if** $v$.NEED_REMAP=**true then** {
    let *lut*=node with the same function and input as $Cluster(v)$;
    add *lut* into $M$;
    $v$.NEED_REREMAP:=false;
    **foreach** node $u \in input(lut)$ **do**
      **if** ($u \notin M$) **then** $u$.NEED_REMAP:=**true**;
}}
remove redundant nodes in M; let M'=M;

# 4. EXPERIMENT RESULTS

We tested IncFlow on 12 MCNC benchmark circuits for LUT size k=5 after 50 *independent* incremental changes. We first do a full mapping on each benchmark (using FlowMap) and then make 50 independent incremental changes. After each change, both IncFlow and FlowMap will be applied on the modified network to generate the new depth optimal mapping solution. We collect the data on the number of new/removed LUTs, the number of new/removed edges and the CPU runtime after each iteration, then we compare the *final* mapping solution and *total* runtime of IncFlow and FlowMap. Also shown in the table are the *average* number of new/removed LUTs and edges per iteration of IncFlow. The result is shown in Table 1 and 2.

In Table 1, we randomly add a new 2-input simple gate $n$ to the network $N$ per iteration. One of the two inputs of $n$ is a new primary input and the other input $x$ is randomly picked from $N$. We use one of $x$'s output as $n$'s output. The function of $n$ is either OR or AND, determined randomly. In this case, the modification does not remove any edge from $N$.

In Table 2, we randomly add 25 2-input simple gates in the same way as in Table 1 and undo the changes immediately. This totals 50 iterations of incremental changes. Please note the undoing involves deleting edges. Therefore the labeling part of IncFlow has to follow Algorithm 1 to handle undos.

The data were collected on a Sun Ultra II with 512M memory. IncFlow runs 23.1 times faster than FlowMap in the first case and 8.82 times faster in the second case. Overall, the average speedup is 14.3 (geometric mean). It is clearly shown in the tables that IncFlow achieves depth optimal mapping solution with the same quality as FlowMap in terms of area (#LUT). Besides, on average, the number of new/removed LUTs per iteration of IncFlow is limited to about 2.5% of the total number of LUTs in the original mapping solution in the case without edge removals (Table 1) and 3.5% in the case involving edge removals (Table 2). Similarly the number of new/removed edges per iteration is limited to 1.6% and 2.3% respectively. This implies that IncFlow only affects a very small portion of the mapping solution per iteration.

| Circuits | FlowMap | | | IncFlow | | | | | | | Speedup |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Final Solution | | CPU time (s) | Final Solution | | Average changes made to the existing solution per iteration | | | | CPU time (s) | |
| | depth | #LUT | | depth | #LUT | # new LUT | # removed LUT | # new edges | # removed edges | | |
| 5xp1 | 4 | 64 | 4.10 | 4 | 64 | 4.00 | 3.38 | 9.72 | 7.28 | 0.39 | 10.5 |
| count | 6 | 86 | 5.98 | 6 | 86 | 4.40 | 3.76 | 11.42 | 9.46 | 0.96 | 6.2 |
| C499 | 6 | 112 | 84.39 | 6 | 112 | 5.62 | 4.86 | 14.30 | 11.38 | 26.69 | 3.2 |
| Apex7 | 5 | 125 | 8.94 | 5 | 125 | 3.58 | 2.74 | 8.64 | 5.78 | 1.20 | 7.5 |
| Alu2 | 8 | 197 | 25.46 | 8 | 197 | 5.82 | 5.26 | 14.86 | 12.84 | 3.84 | 6.6 |
| duke2 | 5 | 251 | 18.86 | 5 | 251 | 5.64 | 5.14 | 12.20 | 10.46 | 0.80 | 23.6 |
| C880 | 9 | 246 | 39.68 | 9 | 246 | 4.14 | 3.72 | 9.74 | 8.26 | 8.00 | 5.0 |
| Apex6 | 5 | 373 | 29.62 | 5 | 373 | 3.42 | 2.22 | 9.10 | 5.84 | 0.50 | 59.2 |
| Alu4 | 6 | 1284 | 124.53 | 6 | 1284 | 4.52 | 4.36 | 9.66 | 9.08 | 1.25 | 100.0 |
| Des | 5 | 1422 | 234.74 | 5 | 1422 | 11.66 | 14.10 | 34.34 | 41.32 | 1.92 | 122.3 |
| too_large | 7 | 5091 | 542.11 | 7 | 5091 | 1.46 | 1.32 | 2.88 | 1.70 | 1.70 | 318.9 |
| Clma | 18 | 6297 | 6317.3 | 18 | 6298 | 7.04 | 5.48 | 25.00 | 20.36 | 78.00 | 81.0 |
| | | | | | | | | | | Average Speedup | 23.1 |

**Table 1 Adding 50 simple gates (no edge removals)**

| Circuits | FlowMap | | | IncFlow | | | | | | | Speedup |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Final Solution | | CPU time (s) | Final Solution | | Average changes made to the existing solution per iteration | | | | CPU time (s) | |
| | depth | #LUT | | depth | #LUT | # new LUT | # removed LUT | # new edges | # removed edges | | |
| 5xp1 | 3 | 32 | 3.63 | 3 | 33 | 5.22 | 5.22 | 14.22 | 14.22 | 0.78 | 4.65 |
| count | 5 | 54 | 4.66 | 5 | 54 | 5.26 | 5.26 | 13.5 | 13.5 | 1.31 | 3.56 |
| C499 | 4 | 74 | 75.84 | 4 | 74 | 1.18 | 1.18 | 1.0 | 1.0 | 21.86 | 3.46 |
| Apex7 | 4 | 83 | 7.88 | 4 | 83 | 3.24 | 3.24 | 6.86 | 6.86 | 0.77 | 10.23 |
| Alu2 | 7 | 169 | 23.31 | 7 | 169 | 8.78 | 8.78 | 22.92 | 22.92 | 4.88 | 4.77 |
| duke2 | 4 | 226 | 18.22 | 4 | 226 | 5.58 | 5.58 | 13.26 | 13.26 | 2.49 | 7.32 |
| C880 | 8 | 225 | 38.41 | 8 | 225 | 2.56 | 2.56 | 5.38 | 5.38 | 3.63 | 10.53 |
| Apex6 | 5 | 313 | 29.94 | 5 | 313 | 3.18 | 3.18 | 10.06 | 10.06 | 2.32 | 12.91 |
| Alu4 | 6 | 1276 | 122.78 | 6 | 1276 | 3.58 | 3.58 | 8.68 | 8.68 | 10.31 | 11.91 |
| Des | 5 | 1544 | 238.94 | 5 | 1544 | 7.82 | 7.82 | 22.56 | 22.56 | 12.26 | 19.44 |
| too_large | 7 | 5084 | 543.24 | 7 | 5084 | 1.24 | 1.24 | 1.62 | 1.62 | 44.1 | 12.32 |
| Clma | 18 | 6220 | 6623.5 | 18 | 6220 | 10.82 | 10.82 | 41.6 | 41.6 | 234.4 | 28.25 |
| | | | | | | | | | | Average Speedup | 8.82 |

**Table 2 Adding 25 gates and then undo the changes (*involving* edge removals)**

# 5. CONCLUSION AND FUTURE WORK

In this paper we have studied incremental technology mapping for *k*-LUT based FPGAs. We have classified the type of incremental changes. An incremental mapping algorithm, called IncFlow, was developed and presented in the paper. IncFlow only updates part of the mapping solution as needed, and can significantly reduce the CPU runtime (by over 300x when compared to the FlowMap algorithm) while still achieving depth optimal mapping solution.

In order to make the most use of our incremental mapping result, the subsequent placement and routing should be incrementally too. Other members in our research group are currently working on the incremental placemant and routing algorithms. In addition, we plan to extend the IncFlow algorithm to support non-unit delay models.

# 6. ACKNOWLEDGMENTS

This work is partially supported by Actel, Lucent Technologies, Quickturn and Xilinx under the California MICRO Program.

# 7. REFERENCES


[1] J. Cong and Y. Ding, FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs, IEEE Trans. on Computer-Aided Design, Jan. 1994, Vol. 13, No. 1, pp. 1-12.

[2] J. Cong, C. Wu and Y, Ding, Cut Ranking and Pruning: Enabling A General And Efficient FPGA Mapping Solution, Proc. ACM Int'l. Symp. on FPGA, Monterey, CA, Feb. 1999, pp. 29-35.

[3] J. Cong and Y. Ding, Combinational Logic Synthesis for LUT Based Field Programmable Gate Arrays, ACM Trans. on Design Automation of Electronic Systems, Vol. 1, No. 2, April, 1996, pp. 145-204.

[4] R. J. Francis, J. Rose and Z. G. Vranesic, Technology Mapping of Lookup Table-Based FPGAs for Performance, Proc. IEEE International Conference on Computer-Aided Design, 1991, pp 568-571.

[5] R. J. Francis, J. Rose and Z. G. Vranesic, Chortle-crf: Fast technology mapping for lookup table-based FPGAs, Proc. ACM/IEEE Design Automation Conference 1991, pp 227-233.

[6] Y. Kukimoto and M. Fujita, Rectification Method for Lookup-Table Type FPGA's, ICCAD '92

[7] R. Murgai, R. K. Brayton, N. Shenoy and A. Sangiovanni-Vincentelli , Improved Logic Synthesis Algorithms for Table Look-Up Architectures, Proc. of ICCAD-91, pp. 564-567, 1991

[8] R. Murgai, R. K. Brayton, N. Shenoy and A. Sangioyanni-Vincentelli, Performance directed synthesis for table look up programmable gate arrays, Proc. of ICCAD-91, pp 572-575, 1991

[9] J. G. Eldredge and B. L. Hutchings. Run-time reconfiguration: A method for enhancing the functional density of SRAM-based FPGAs, Journal of VLSI Signal Processing, v.12: pp. 7--86, 1996

[10] J. Cong, H. Huang, Depth Optimal Incremental Technology Mapping for Field Programmable Gate Arrays, UCLA-CSD 200004, Technical Report, March 2000