# Maple: A Simultaneous Technology Mapping, Placement, and Global Routing Algorithm for Field-Programmable Gate Arrays

Nozomu Togawa Masao Sato Tatsuo Ohtsuki

Dept. of Electronics and Communication Engineering, Waseda University
3-4-1 Okubo, Shinjuku, Tokyo 169, Japan

**Abstract**

*Technology mapping algorithms for LUT (Look Up Table) based FPGAs have been proposed to transfer a Boolean network into logic-blocks. However, since those algorithms take no layout information into account, they do not always lead to excellent results. In this paper, a simultaneous technology mapping, placement and global routing algorithm for FPGAs,* `Maple`*, is presented.* `Maple` *is an extended version of a simultaneous placement and global routing algorithm for FPGAs, which is based on recursive partition of layout regions and block sets.* `Maple` *inherits its basic process and executes the technology mapping simultaneously in each recursive process. Therefore, the mapping can be done with the placement and global routing information. Experimental results for some benchmark circuits demonstrate its efficiency and effectiveness.*

## 1 Introduction

FPGAs (Field-Programmable Gate Arrays) have been attracting public attention as new ASIC devices which are programmable and of high density. Especially, FPGAs which use SRAM as programming elements [10],[11] are wide-spread to make rapid-prototyping systems because of their reprogrammability. SRAM based FPGAs make use of LUTs (Look Up Tables) or similar circuits as LUTs [10],[11] as their basic blocks (logic-blocks). Each LUT can realize any combinational function with up to $k$ inputs (typically, $k = 4$ or 5).

Research on technology mapping for LUT-based FPGAs is very attractive and many algorithms have been proposed. Most of them are classified into two categories according to their objective functions. One aims at minimizing the number of logic-blocks [7],[8],[14],[15]. The other aims at minimizing the level of logic-blocks [5],[6],[9],[16]. Even if those objective functions can be minimized, however, we cannot always produce good layout (placement and routing) results since the objective functions take no layout information into consideration.

Since logic-blocks and routing resources are predefined in an FPGA chip, the layout of each circuit needs to be done within them. Especially, the limitation of routing resources is crucial, and conventional FPGA design which executes technology mapping and layout separately often causes unrouted nets in the layout phase because of a shortage of routing resources. To obtain good layout, i.e., less congested layout, FPGA design is required to consider layout from higher design phases such as technology mapping.

Since a few years ago, technology mapping algorithms which use layout information have been proposed [1],[4],[17]. The algorithm proposed in [17] considers "routability" as an objective, but it does not reflect layout precisely. The algorithm proposed in [1] is based on SA (Simulated Annealing). It executes placement and global routing during each iteration of SA and makes use of its result. The algorithm must be much time-consuming and difficult to optimize a result globally in a short time. The algorithm proposed in [4] executes both technology mapping and placement simultaneously, but does not evaluate the layout including routing.

In this paper, we consider technology mapping for LUT-based FPGAs as the design including placement and routing, and propose a simultaneous technology mapping, placement, and global routing algorithm, which is called `Maple` (MAPping with Layout Execution). `Maple` is based on the simultaneous placement and global routing algorithm [19]. This algorithm [19] has the following advantages:

(1) The global route of each net is represented by a sequence of pseudo-blocks. Since the pseudo-blocks are treated in the same way as logic-blocks, placement and global routing can be executed simultaneously.

(2) As the algorithm is based on recursive process of simple top-down bi-partitioning, it runs fast and is easy to implement.

(3) The algorithm can handle blocks with functionally equivalent terminals.

(4) The algorithm can handle multi-terminal nets without dividing them into two-terminal nets.

`Maple` inherits all of the above advantages and has further ones as follows:

(5) Since `Maple` executes technology mapping at each recursive level, it generates logic-blocks based on placement and global routing information of the level. This leads to less congested layout, i.e., the less number of routing tracks per channel.

(6) `Maple` can execute "replication" in a natural way, which is one of the greatest concern of technology mapping [3].

(7) By making good use of the maximum flow technique, the mapping process itself runs fast, and thus the whole algorithm including placement and global routing is executed very fast.

This paper is organized as follows: Section 2 summarizes existing technology mapping algorithms for LUT-based FPGAs; Section 3 defines the technology mapping, placement, and global routing problem; Section 4 shows the `Maple` algorithm; Section 5 demonstrates experimental results compared with some conventional approaches; and Section 6 gives concluding remarks.

## 2 Technology Mapping for LUT-based FPGAs

Input of technology mapping for LUT-based FPGAs is a Boolean network [3]. The Boolean network is represented by DAG (Directed Acyclic Graph), where each node represents a logic function and each directed edge represents data (logic value) flow from one node to the other. Basic operations of technology mapping algorithms for LUT-based FPGAs are the following (1) and (2) (e.g. [15]) or (2) (e.g. [6]).

(1) *Decomposition* decomposes the input network and makes the number of fanins of each node less than or equal to a given number $k$ (Fig. 1 (a)). Kernel extraction or Roth-Karp decomposition can be applied for this operation [14],[15].

(2) *Covering* covers several nodes of a Boolean network obtained in (1) and generates one logic-block instead of them (Fig. 1 (b)). The objective is to minimize the number of generated logic-blocks or to minimize the logic level. It is not necessary to consider logic functions of nodes explicitly [6],[17].

In technology mapping, the decomposition, which transforms logic functions directly, itself is a difficult problem and have been studied for many years from the viewpoint of logic minimization. It is still difficult even though the decomposition is restricted to FPGAs. On the other hand, the covering is peculiar to LUT-based FPGAs and considered to be a graph covering problem. Since it excludes direct transformation of logic functions, it can be combined with layout design algorithms effectively. From the above, we consider technology mapping for FPGAs as covering of a Boolean network as in [6],[17] and this covering is called technology mapping in the rest of this paper.

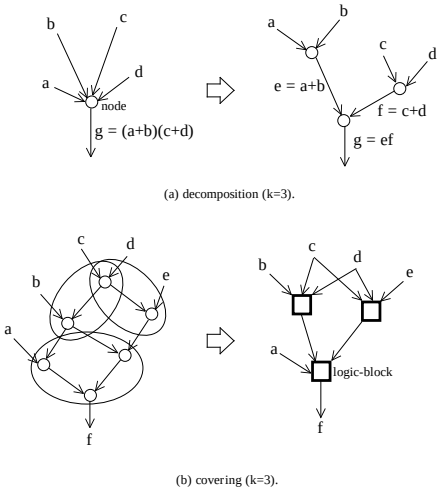

Fig. 1: Technology mapping.

# 3 Problem Formulation

## 3.1 FPGA Layout Model

An FPGA layout model is defined as in Fig. 2, based on the commercial FPGA architecture proposed in [11]. Each *logic-block* is connected to *switch-blocks* through input and/or output terminals. Switch-blocks are connected to each other by horizontal or vertical *tracks*. A set of tracks which connects two switch-blocks is called a *channel*. A position for placing a logic-block is called a *slot*.

Logic-blocks on the periphery of the FPGA layout model are connected to I/O pads and called *I/O blocks*. Each I/O block has input and output terminals for all of the adjacent switch-blocks. Other logic-blocks realize LUTs with four inputs and one output ($k = 4$) by programming. Each of them has an input terminal for each of four adjacent switch-blocks and an output terminal for the lower-left adjacent switch-block (Fig. 3).

Switch-blocks are considered as switch boxes which make a possible connection of terminals and tracks by programming.

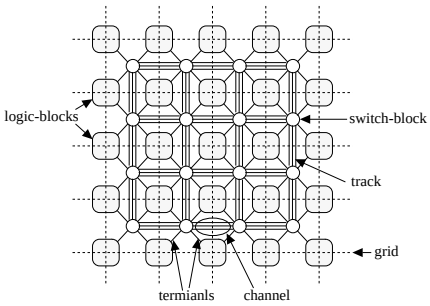

Fig. 2: FPGA layout model.

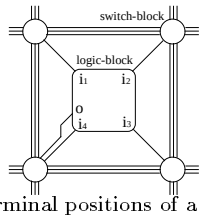

Fig. 3: Terminal positions of a logic-block.
($i_1, i_2, i_3, i_4$: input terminals, $O$: output terminal)

## 3.2 Technology Mapping, Placement, and Global Routing Problem

In the same way as other technology mapping algorithms, input of our algorithm is a Boolean network. Among nodes of a Boolean network, nodes which have no incoming edges are called *primary inputs* and nodes which have no outgoing edges are called *primary outputs*. Other nodes are called *intermediate nodes*. If the number of incoming edges to each node of a Boolean network is less than or equal to $k$, the Boolean network is called *feasible*. An input Boolean network is assumed to be feasible in this paper.

For two nodes $s, t$ of a Boolean network, $s$ is a *transitive fanin* of $t$ if there exists a directed path from $s$ to $t$, and $t$ is a *transitive fanout* of $s$. For a node $s$ of a Boolean network, a set of $s$ and transitive fanins of $s$ is called a *cover* of $s$. A cover of $s$ outputs a signal from only $s$. If the number of input nodes for a cover of $s$ is less than or equal to $k$, the cover is called *feasible*. A feasible cover of $s$ can be realized by one logic-block.

Technology mapping of a Boolean network is to satisfy the following (a) - (c).

(a) Each of primary inputs and outputs corresponds to one logic-block.

(b) Each of intermediate nodes is included in at least one cover.

(c) Inputs of any cover are outputs from other covers or primary inputs.

As described earlier, routing congestion, i.e., the maximum number of tracks per channel to route all nets, should be as small as possible for FPGA design. So the following problem whose objective is minimization of routing congestion is focused in this paper.

**[Definition]** Technology mapping, placement, and global routing problem is for given

(1) a feasible Boolean network and

(2) a set of slots and terminal positions of logic-blocks,

to determine

(a) covers of a Boolean network (technology mapping),

(b) a slot position where each logic-block is placed, with considering each cover to be one logic-block (placement),

(c) a terminal position of each net (pin assignment) and
(d) a sequence of channels through which each net passes (global routing)
within given slots so as to minimize the maximum number of tracks per channel. Note that I/O blocks need to be placed on the outermost slots.

Fig. 4 shows an example of a Boolean network and its layout.

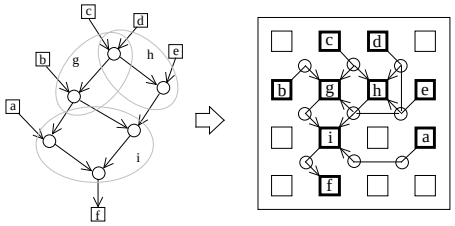


Fig. 4: Boolean network and its layout.

# 4 The Maple Algorithm

## 4.1 Definitions

The following terminology is defined based on [19].

*Unit-cell*: a room partitioned by the grid (broken lines) of Fig. 2.

*Subregion*: a rectangular region composed of adjacent unit-cells.

*Cut-line*: a horizontal or vertical line partitioning one subregion into two pieces. It is drawn along the grid of Fig. 2.

*Pseudo-block*: a fictitious block placed on a cut-line to maintain the connection of a net divided by the cut-line. Every pseudo-block is assigned in a channel.

*Block set*: a set of logic-blocks and pseudo-blocks. Each block set is assigned on a subregion boundary.

*Node set*: a set of nodes of a Boolean network. Each node set exists inside a subregion.

## 4.2 Simultaneous Placement and Global Routing Algorithm [19]

The proposed algorithm is an extended version of a simultaneous placement and global routing algorithm [19], of which outline is explained in this section. The simultaneous placement and global routing algorithm is based on recursive bi-partition of a layout region like min-cut placement [2]. When bi-partitioning a region by a cut-line, a block set assigned to the region is partitioned into three block sets, such as one assigned to slots in the upper (or left) side of the cut-line, one assigned to slots in the lower (or right) side of the cut-line, and one assigned to slots on the cut-line. At that time, if there exists a connection between the block sets partitioned by the cut-line, a pseudo-block is generated on the cut-line. Since pseudo-blocks are treated in the same way as logic-blocks, placement and global routing can be executed simultaneously. Repeating this process recursively, a position of each logic-block and a global route of each net represented by a sequence of pseudo-blocks are obtained.

## 4.3 Maple: A Simultaneous Technology Mapping, Placement, and Global Routing Algorithm

To obtain less congested layout, a technology mapper should generate each logic-block for those nodes which are close together if all the nodes of a Boolean network are assumed to be placed evenly on an FPGA chip based on connections among them. This process is considered to be technology mapping which takes layout requirements into account.

For example, consider technology mapping of a Boolean network of Fig. 5. When each logic-block is an LUT with three inputs ($k = 3$), covering which minimizes the number of logic-blocks is shown in Fig. 5 (a). However, this mapping does not always lead to a good layout result. It sometimes leads to routing congestion and requires more tracks per channel (Fig. 5 (b)). In Fig. 5 (b), bold boxes show the slots on which logic-blocks are placed. On the other hand, if we do mapping so as to avoid routing congestion based on layout requirements, we can obtain a less congested result as in Fig. 5 (c), (d). This is our motivation to propose an algorithm, Maple, which executes technology mapping with placement and global routing.

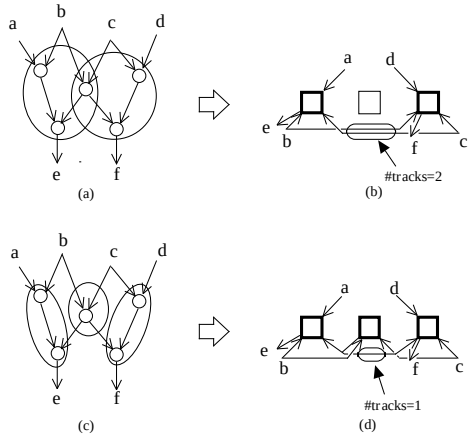


Fig. 5: Effect of technology mapping to its layout ($k = 3$).

The basic process of Maple is to bi-partition regions recursively and to partition nodes of a Boolean network into three sets as in [19] (Fig. 6). When partitioning the node set $B$ in Fig. 6, nodes which are expected to be assigned to slots on the cut-line are considered as candidates for technology mapping. This has the following advantages:

(1) Nodes on the cut-line are determined so as to strongly reflect the result of placement and global routing at each recursive level, and their positions are limited to slots on the cut-line. Therefore, mapping which takes layout requirements into account is realized for those nodes. As a result, less congested layout can be generated as in Fig. 5.

(2) After mapping, each cover of nodes becomes one logic-block. Those logic-blocks are on the boundary of partitioned regions, and the recursive process of [19] can be continued. Therefore, Maple inherits the advantages of [19].

For example, in a Boolean network and a region of three slots as in Fig. 5 a vertical cut-line is drawn through the center slot. From the positions of $a$ - $f$ which are inputs or outputs of the Boolean network, the center node of the Boolean network is selected as a node which is better to be assigned on the cut-line, i.e., a candidate for technology mapping (see Section 4.4 for details). This center node is covered with a logic-block on the cut-line. The remaining nodes in the Boolean network is assigned on the other slots in the subsequent recursive process. Note that, in case there exist only two slots, Maple generates at most two logic-blocks as in Fig. 5 (a).

Maple is realized by extending the algorithm in [19] and doing the covering process for nodes on cut-lines. The outline of the algorithm is as follows.

**[Algorithm Maple]**

*Step 1.* Assume each of primary inputs and outputs of an input Boolean network to be an I/O block. I/O blocks are placed

on four sides and four corners of the layout region. The slot positions for them are determined gradually in recursive process. In case where the slot positions of I/O blocks are specified as input, they are placed on the positions. Otherwise, primary inputs and primary outputs are placed separately with a reasonable balance. Put the entire layout region as a subregion into a queue $Q$.

*Step 2.* Pick a subregion $R$ from $Q$. If $Q$ is empty, halt.

*Step 3.* Partition $R$ into two subregions $R_1$ and $R_2$ by a cut-line. The cut-line is drawn in such a way that the longer sides of the subregion boundary of $R$ are partitioned so as to make the partitioned subregions closer to a square.

*Step 4.* Corresponding to the partition of $R$, the block sets on the subregion boundary ($A$ and $C$ in Fig. 6) are partitioned into three sets respectively.

*Step 5.* For nodes which are expected to be assigned on the cut-line, execute technology mapping. This step will be explained in details in Sections 4.4 - 4.7.

*Step 6.* Assign covers which include nodes generated by technology mapping in Step 5 (which are regarded as logic-blocks) to slots on the cut-line. Assign other nodes to slots of the upper or lower (left or right) side of the cut-line. This step will be explained in Section 4.8.

*Step 7.* Assign nets to terminals for logic-blocks on the cut-line.

*Step 8.* Generate a pseudo-block on the cut-line for each net crossing it.

*Step 9.* If $R_1$ and/or $R_2$ can be further partitioned, put them (it) into $Q$. Return to Step 2. ■

Steps 2, 3, 8, and 9 must be trivial. Steps 1, 4, and 7 are done in the same way as [19]. So, Steps 5 and 6 are described in detail in the following sections, with respect to a horizontal cut-line.

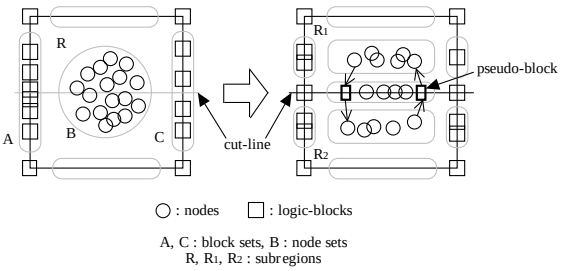


Fig. 6: Recursive process of partitioning a region.

### 4.4 Candidate Nodes for Technology Mapping

Technology mapping algorithm consists of three main components: candidate selection, covering, and replication. They are explained in Sections 4.4 - 4.6 respectively, and summarized in Section 4.7. In this section, we determine which nodes in the node set inside a subregion (nodes in $B$ of Fig. 6) should be assigned to slots on the cut-line.

Let $G$ be a directed graph which is generated in such a way that its nodes are associated with blocks (logic-blocks and pseudo-blocks) on the boundary and the nodes inside the subregion. Edges and their direction of $G$ are the same as the input Boolean network. Note that each block set on the subregion boundary ($A$ and $C$ in Fig. 6) has been already partitioned into three sets in Step 4. Then, an initial label is given to each node $v$ corresponding to a block on the subregion boundary according to its assigned position as follows:

$$label(v) = \begin{cases} 1 & \text{(upper side of the cut-line)} \\ 0 & \text{(lower side of the cut-line)} \\ 0.5 & \text{(on the cut-line)} \end{cases}$$

The initial labels are propagated along directed edges of $G$, and each node $v$ is given a label $label_f(v)$ ($f$ of $label_f(v)$ stands for the forward direction) by the following expression.

$$label_f(v) = \sum_{u \in I(v)} label_f(u) / \mid I(v) \mid$$

where $I(v)$ denotes fanin nodes of $v$. $label_f(v)$ is the mean value of labels of $I(v)$. Therefore, those labels are calculated with taking connections between nodes into account.

The initial labels are also propagated in the reverse direction of edges in a similar way in our algorithm. Let $label_b(v)$ ($b$ of $label_b(v)$ stands for the backward direction) be a label attached in this direction to each node $v$. $label(v)$ for each node $v$ is calculated by averaging $label_f(v)$ and $label_b(v)$. $label(v)$ is considered to indicate to which side of the cut-line $v$ is better to be assigned. The closer to 1 (resp., 0) $label(v)$ is, the more likely $v$ is better to be assigned to the upper (resp., lower) side.

Based on $label(v)$, each node of the node set $B$ is expected to be assigned to the upper side slots, the lower side slots, or slots on the cut-line so as to balance the slot utilization. So, we sort the nodes based on their labels. Then, we assign them to the upper side slots, slots on the cut-line, and the lower side slots, in the descending order of their labels, in proportion to the numbers of slots of the three parts. Candidates for technology mapping should be nodes which are expected to be assigned to the slots on the cut-line. Let $l_{min}$ and $l_{max}$ be the minimum and maximum labels of those nodes, respectively, and then nodes whose labels satisfy the following inequality are decided to be the candidates for mapping.

$$l_{min} \leq label(v) \leq l_{max}$$

### 4.5 Generation of a Feasible Cover

An algorithm for generating a cover is presented in this section. Since the number of slots on the cut-line is fixed, one logic-block should cover as many candidates as possible. So, some nodes which are not candidates could be covered to obtain feasible covers. Note that the number of inputs of each cover for candidate nodes must not exceed $k$, since the number of inputs for each logic-block is up to $k$. In the following, we propose an algorithm which generates such covers by using the network flow theorem. We use the terminology of [18].

A $level$ (depth) of each node of the directed graph $G$ is defined in such a way that the level of nodes which have no incoming edges is set to be 0, and the level of the other nodes is the maximum level of the fanin nodes plus one. Let $t$ be a node which is the candidate for mapping and has the largest level. A cover of $t$ will include $t$ and transitive fanins of $t$. So, If we start the covering process from the node of the largest level, the cover is expected to include the most candidate nodes.

Based on the directed graph $G$, we generate a graph $G(t)$ whose nodes are the node $t$, the transitive fanins of $t$ which are inside the subregion or on the subregion boundary, and an additional node $s$ which is connected to nodes corresponding to blocks on the subregion boundary, as in Fig. 7 (b). Fig. 7 shows an example of $k = 3$ and bold circles indicate the candidates. To obtain a feasible cover of $t$, we further generate $G_R(t)$ with capacities from $G(t)$ in the following way (see Fig. 8 (a)). Each edge of $G(t)$ corresponds to that of $G_R(t)$ with the capacity of infinity. The nodes $s$ and $t$ of $G_R(t)$ are the same as those of $G(t)$. For each node $v$ of $G(t)$ besides $s$ and $t$, we generate two nodes, $v_1$ and $v_2$, and connect them by an edge from $v_1$ to $v_2$ with the capacity of one in $G_R(t)$. Incoming edges of $v$ correspond to those of $v_1$ and outgoing edges of $v$ correspond to those of $v_2$.

For $G_R(t)$, we will obtain the maximum flow from $s$ to $t$. When we bi-partition the nodes of $G_R(t)$ into two sets: one including $s$

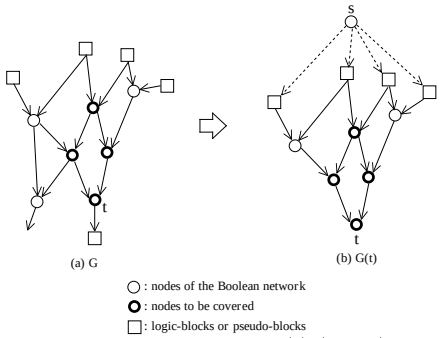


Fig. 7: Generation of $G(t)$ ($k = 3$).

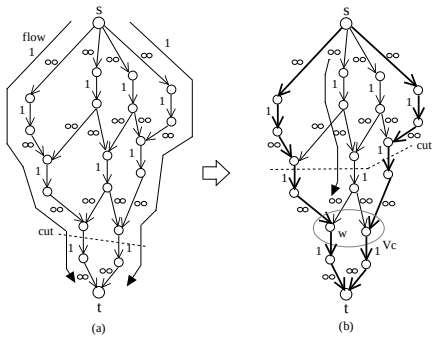


Fig. 8: Generation of a feasible cover of $t$ by means of maximum flow in $G_R(t)$.

and the other including $t$, the bi-partitioning line is called a *cut* and the sum of the capacities of edges which cross the cut in the direction from $s$ to $t$ is called a *size* of the cut. According to the maximum flow minimum cut theorem, if the maximum flow is up to $k$, a cut whose size is at most $k$ exists. It is clear that the edges which cross the cut in such a direction are at most $k$ edges with the capacity of one in $G_R(t)$. All the transitive fanouts for each node in $G(t)$ which is corresponding to one of those edges in $G_R(t)$ form a feasible cover of $t$ because at most $k$ nodes are the inputs of the cover. Since the input Boolean network is feasible, the maximum flow with the size at most $k$ exists.

To explain our procedure which computes the maximum flow, we introduce a *residual graph* [18] first. Nodes of a residual graph are the same as $G_R(t)$. For each edge of $G_R(t)$, if the value of "the capacity of the edge - the flow of the edge" is more than zero, we generate an edge of the same direction with the capacity of the value. If the flow of each edge is more than zero, we generate an edge with the reverse direction and the capacity of that flow. Fig. 9 shows the residual graph of Fig. 8 (a) in which there are two flows. A directed path between two nodes in the residual graph is called an *augmenting path*.

The maximum flow is obtained by means of the augmenting path method [18] as follows. In the residual graph of $G_R(t)$, we find an augmenting path from $s$ to $t$, and then increase the flow by one along the path in $G_R(t)$. We repeat this process until no augmenting paths exist from $s$ to $t$. The flow is the maximum one when no augmenting paths exist.

After obtaining the maximum flow by the above method, `Maple` searches nodes from $s$ in the residual graph (see Fig. 9). Then, it generates a cut which bi-partitions nodes of $G_R(t)$ into searched ones and unsearched ones. Generally, several minimum cuts exist for one maximum flow. Since `Maple` searches nodes from $s$, the number of searched nodes including $s$ is minimum. Thus,

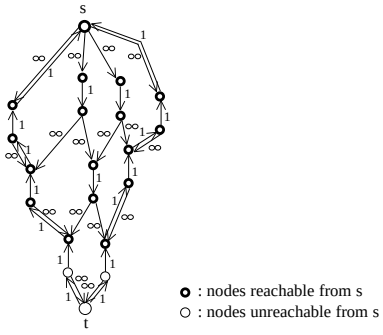


Fig. 9: Residual graph for Fig. 8 (a).

the number of nodes included in the generated cover becomes maximum.

If the maximum flow is less than $k$, more nodes could be covered. Let $V_c$ be a set of nodes which do not correspond to blocks on the subregion boundary and whose outgoing edge crosses the cut. `Maple` tries to search more augmenting paths. Let $w$ be one of nodes of $V_c$ whose label is in the range from $l_{min}$ to $l_{max}$ (in case where there are no such nodes, a node whose label is closest to this range). Then, an augmenting path from $s$ to $w$ is searched (Fig. 8 (b)). When there exist no augmenting paths from $s$ to $w$, it means that the node $w$ is covered. In such a case, we search the residual graph of $G_R(t)$ from $s$ and generate a cut which bi-partitions nodes of $G_R(t)$ in the same way as described above.

The above process is repeated until we finally find $k$ augmenting paths or the generated cover includes all the nodes of $G(t)$ except nodes corresponding to blocks on the subregion boundary.

The generated cover may include some nodes which are not candidates for technology mapping. Such nodes are expected to be excluded from the cover. If there are such nodes, we apply the following operation for each of the nodes in the ascending order of their levels. First, we assume to exclude the node from the cover. If the cover remains feasible, the node is really excluded. Otherwise, the node is still included in the cover.

The above algorithm is summarized as follows.

**[Algorithm for Generating a Feasible Cover]**

*Step 1.* For a node $t$, generate a directed graph $G_R(t)$. Let $V_c = \{t\}$.

*Step 2.* Let $w$ be one of nodes of $V_c$ whose label is in the range from $l_{min}$ to $l_{max}$. If there are no such nodes, a node whose label is closest to this range is chosen.

*Step 3.* In the residual graph of $G_R(t)$, search an augmenting path from $s$ to $w$, and increase the flow by one along the path in $G_R(t)$. Repeat this path search until there exist no such paths or the total number of searched paths becomes $k$. If the total number of searched paths becomes $k$, go to Step 5.

*Step 4.* Search nodes from $s$ in the residual graph of $G_R(t)$. Then, generate a cut which bi-partitions its nodes into searched ones and unsearched ones. Let $V_c$ be a set of nodes which do not correspond to blocks on the subregion boundary and whose outgoing edge crosses the cut. If $V_c$ is not empty, go to Step 2.

*Step 5.* Let a cover of $t$ be nodes in $G(t)$ other than those which generate the nodes reachable from $s$ in the residual graph of $G_R(t)$. For the nodes in the cover, if there are some nodes which are not candidates for technology mapping, try to exclude them from the cover. ■

The above process is terminated when at most $k$ augmenting paths are obtained. An augmenting path is obtained, once all the nodes of $G_R(t)$ are searched. Therefore, the time complexity of

generating a cover of a node $t$ is proportional to the number of nodes in the subregion, i.e., the size of $G$.

### 4.6 Replication and Label Updating

Since each cover corresponds to one logic-block, the number of its output must be one. In a cover of $t$, all the nodes which have outputs other than $t$ need to be replicated (Fig. 10 (a), (b)).

Let $V_t$ be a set of nodes included in the cover of $t$. Let $v$ be a node in $V_t$ other than $t$ whose outgoing edge is connected to an outside node. The node $v$ and its transitive fanins are replicated. Each replicated node has outgoing edges to the original node's fanouts which are not in $V_t$ and incoming edges from the original node's fanins. The replication is executed in the descending order of levels of nodes which need to be replicated. By following this order, we can replicate nodes which have outgoing edges to already replicated nodes efficiently. When we obtain a cover of $t$ and replication for it, the cover, i.e., $V_t$ is represented by one logic-block $t$ as in Fig. 10 (c).

`Maple` utilizes the labels described in Section 4.4. So, we give a label to each of generated logic-blocks and replicated nodes. Labels are given to them by propagating labels of neighbors.

The directed graph obtained by the above process is logically equivalent to the initial Boolean network, and then we repeat the above process for this graph.

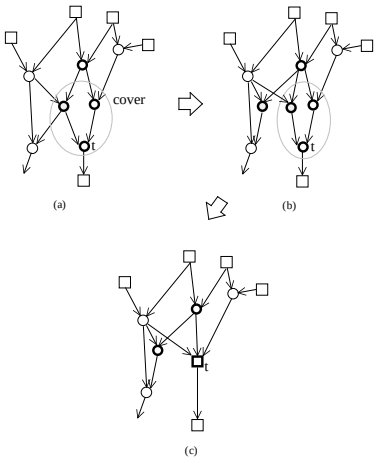

Fig. 10: Replication.

### 4.7 Mapping Algorithm

The mapping algorithm in `Maple` is summarized according to Sections 4.4 - 4.6.

**[Mapping Algorithm]**

*Step 1* (Section 4.4). Sort nodes inside the subregion in the descending order of their labels and determine the range of labels ($l_{min} \leq label \leq l_{max}$) of the nodes which should be assigned to slots on the cut-line.

*Step 2.* For a directed graph $G$ generated by blocks on the subregion boundary and nodes inside the subregion, let $t$ be an uncovered node inside the subregion which has the largest level within the range of Step 1. If there exist no such nodes, halt.

*Step 3* (Section 4.5). Obtain a cover of the node $t$.

*Step 4* (Section 4.6). Replace the cover of $t$ with a logic-block $t$, and make replications. Give labels to the generated logic-block and nodes. Go to Step 2. ■

### 4.8 Node Set Partitioning

Each generated logic-block should be assigned to a slot on the cut-line. However, in case where the number of such logic-blocks is over the number of slots on the cut-line, logic-blocks whose labels are closer to 0.5 are assigned to the slots on the cut-line first. If there are logic-blocks which cannot be assigned, we put such logic-blocks back to uncovered nodes. Then, we assign them with other nodes in $G$ to slots of the upper and lower side of the cut-line, in the descending order of their labels, in proportion to the number of slots of each region. This process achieves a partition such that nets tend not to cross the cut-line, thus leads to less congested (fewer tracks per channel) layout.

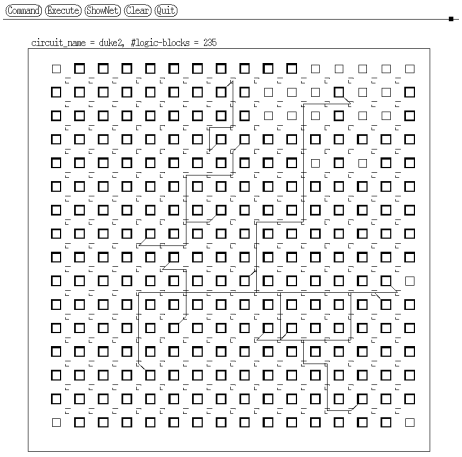

Fig. 11: Layout result for *duke2*.

### 4.9 Computational Complexity

Let the number of slots on an FPGA chip be $L \times L$, the number of nodes of a Boolean network be $N_n$, and the number of pseudo-blocks generated in the whole process be $N_p$. If $T$ is the maximum number of tracks per channel, $N_p$ is at most $O(L^2 \cdot T)$. Let $N$ be $N_n + N_p$.

`Maple` is based on top-down hierarchical bi-partitioning. Each subregion has a *level* of the hierarchy. The level of the entire layout region is one. If the subregion with the level of $i$ is bi-partitioned, each of bi-partitioned subregions has the level of $(i+1)$. It must be clear that the level of unit-cells is $O(\log L)$.

At each level of hierarchy, since the number of candidate nodes for mapping is at most $O(N_n)$ and a cover of each node is obtained in $O(N)$ time, the mapping process requires $O(N_n \cdot N)$ time. Labeling and assignment of nets to terminals require $O(N)$ time and sorting requires $O(N \log N)$ time at each level.

From the above discussion, since the algorithm requires $O(N_n \cdot N + N \log N)$ at each level, the time complexity of `Maple` is $O(N(\log N + N_n) \log L)$.

The space complexity is clearly $O(N)$.

## 5 Experimental Results

`Maple` is implemented on SUN Sparc Station 2 (28.5 MIPS) in C language and applied to MCNC benchmark circuits shown in Table 1. Fig. 11 shows the outcome for the circuit *duke2* obtained by `Maple`. In Fig. 11, logic-blocks and switch-blocks are shown as solid and dashed boxes, respectively. Bold boxes indicate the slots on which logic-blocks are placed. Global routes of some nets are also shown.

We have compared four algorithms, i.e., `Maple` and three conventional algorithms.

**Algorithm 1:** After technology mapping (decomposition and covering) using mis-pga(new) [15], we execute SA based pairwise exchange placement [12] followed by hierarchical global routing [13]. The objective of mis-pga(new) is to minimize the number of logic-blocks. In SA, the objective is to minimize the estimated total wire length, where the

estimated wire length of each net is the half perimeter of the minimum rectangle surrounding all the terminals of the net. The hierarchical global routing [13] aims at minimizing the total wire length and flattening routing congestion.

**Algorithm 2:** After technology mapping using mis-pga(new), we execute min-cut placement followed by hierarchical global routing [13]. For min-cut placement, the algorithm proposed in [2] is applied.

**Algorithm 3:** After technology mapping using mis-pga(new), we execute simultaneous placement and global routing [19].

**Maple:** After generating a feasible Boolean network by the decomposition in mis-pga(new) (we used a command `xl_split`), we execute **Maple**.

For the experiments, the number of slots in the FPGA model is determined in such a way that, after technology mapping (decomposition and covering) by mis-pga(new), the logic-block utilization (except I/O blocks) becomes close to but no more than 80%, as in Table 1, where the utilization is the number of used logic-blocks divided by the number of slots. This is because, if the logic-block utilization is over 80%, the routing congestion is much increased and it is known from experiences that all nets cannot be routed in the prefabricated tracks in an FPGA chip [17].

Tables 2 - 5 show the experimental results. In the tables, $\#lb$ denotes the number of logic-blocks. $\#t$ denotes the maximum number of tracks per channel to require to do global routing for all nets. $wl$ denotes the total number of tracks occupied by the nets. CPU time is shown in the form of "time for technology mapping" + "time for placement" + "time for global routing" in Tables 2 and 3, and "time for technology mapping" + "time for placement and global routing" in Table 4.

From the results for the same chip-size of each circuit, though the total wire length by **Maple** is more than that by conventional algorithms, the maximum number of tracks per channel by **Maple** is 25% less than that of the algorithm 1, 39% less than that of the algorithm 2, and 12% less than that of the algorithm 3. This implies that routing congestion is more reduced by design which takes the layout information into account from higher phases. The reduction of tracks per channel is more outstanding in larger circuits. CPU time of **Maple** is short enough compared with the sum of CPU times of several phases in conventional algorithms.

The numbers of slots and tracks per channel of FPGA chips are predefined. In the conventional design, we need to limit the logic-block utilization to less than 80% in order to route all the nets of a circuit in predefined tracks [3],[17]. Therefore, how much we can reduce routing congestion, i.e., minimizing the maximum number of tracks per channel, with utilizing all the slots on a chip is quite important for routing given nets without leaving any unrouted nets. The experimental results show that the conventional approaches that emphasize only on minimizing the number of logic-blocks cause the increase of routing congestion. On the other hand, **Maple** generates 20% more logic-blocks than the conventional algorithms by technology mapping based on requirements of placement and global routing, and makes good use of those logic-blocks in terms of the decrease in tracks per channel.

From the above discussion, it is shown that **Maple** is efficient and effective.

## 6 Conclusions

We have proposed **Maple**, a simultaneous technology mapping, placement, and global routing algorithm for FPGAs. **Maple** produces layout which is less congested by generating logic-blocks based on placement and global routing information. Experimental results show its efficiency and effectiveness.

For the paths of which timing is critical, **Maple** can reduce their delays by the covering which covers them with fewer logic-blocks and the node partitioning such that they cross fewer cut-lines which lead a reduction of switch-blocks. For the circuits in the above experiments, delay constrains have been imposed on several paths. $\#cp$ and $d_{max}$ of Table 1 denote the number of paths with the constraints and the delay constraints, respectively. Tables 5 and 6 show that **Maple** with path delay constraints has achieved no constraints violations except for two circuits.

Table 1: Benchmark circuits.

| circuit | $\#i$ | $\#o$ | #slots | $\#cp$ | $d_{max}$ |
|---|---|---|---|---|---|
| alu2 | 10 | 6 | 16 × 16 | 4 | 177 |
| alupla | 25 | 5 | 9 × 9 | 12 | 48 |
| bw | 5 | 28 | 11 × 11 | 18 | 124 |
| duke2 | 22 | 29 | 16 × 16 | 4 | 87 |
| f51m | 8 | 8 | 9 × 9 | 30 | 101 |
| misex1 | 8 | 7 | 7 × 7 | 4 | 24 |
| misex3 | 14 | 14 | 17 × 17 | 16 | 200 |
| misex3c | 14 | 14 | 17 × 17 | 12 | 218 |
| rd73 | 7 | 3 | 6 × 6 | 20 | 31 |
| rd84 | 8 | 4 | 8 × 8 | 3 | 38 |
| term1 | 34 | 10 | 12 × 12 | 25 | 82 |
| vg2 | 25 | 8 | 10 × 10 | 4 | 29 |

$\#i$: number of primary inputs, $\#o$: number of primary outputs, $\#cp$: number of paths with maximum permissible delay constraints, $d_{max}$: maximum permissible delay for the paths with constraints

Table 2: Algorithm 1 (mis-pga(new) + SA based placement + hierarchical global routing).

| circuit | $\#t$ | $\#lb$ | $wl$ | CPU time [s] |
|---|---|---|---|---|
| alu2 | 15 | 152 | 1537 | 434.1 + 9518.9 + 4.40 |
| alupla | 11 | 68 | 367 | 50.5 + 218.1 + 0.73 |
| bw | 9 | 91 | 524 | 25.1 + 2477.9 + 0.74 |
| duke2 | 20 | 192 | 2244 | 341.2 + 8551.8 + 7.56 |
| f51m | 7 | 51 | 233 | 33.9 + 516.9 + 0.37 |
| misex1 | 4 | 28 | 75 | 7.9 + 86.1 + 0.05 |
| misex3 | 16 | 190 | 2086 | 430.9 + 19712.5 + 7.13 |
| misex3c | 20 | 189 | 2010 | 248.9 + 18173.4 + 9.33 |
| rd73 | 4 | 23 | 72 | 36.9 + 49.7 + 0.07 |
| rd84 | 7 | 33 | 153 | 98.1 + 219.4 + 0.19 |
| term1 | 10 | 110 | 663 | 72.6 + 5773.2 + 1.45 |
| vg2 | 5 | 59 | 193 | 10.9 + 1298.1 + 0.39 |
| total | 128 | 1186 | 10157 | 1791.0 + 66596.0 + 32.41 |
| ratio | 1.00 | 1.00 | 1.00 | – |

$\#t$: maximum number of tracks per channel, $\#lb$: number of logic-blocks, $wl$: total wire length

Table 3: Algorithm 2 (mis-pga(new) + min-cut placement + hierarchical global routing).

| circuit | $\#t$ | $\#lb$ | $wl$ | CPU time [s] |
|---|---|---|---|---|
| alu2 | 17 | 152 | 1607 | 434.1 + 25.41 + 5.95 |
| alupla | 11 | 68 | 369 | 50.5 + 1.52 + 0.38 |
| bw | 14 | 91 | 649 | 25.1 + 2.78 + 0.85 |
| duke2 | 23 | 192 | 2275 | 341.2 + 26.65 + 8.39 |
| f51m | 11 | 51 | 268 | 33.9 + 0.76 + 0.34 |
| misex1 | 4 | 28 | 80 | 7.9 + 0.18 + 0.07 |
| misex3 | 21 | 190 | 2305 | 430.9 + 36.33 + 10.02 |
| misex3c | 22 | 189 | 2214 | 248.9 + 37.18 + 9.96 |
| rd73 | 6 | 23 | 72 | 36.9 + 0.12 + 0.08 |
| rd84 | 9 | 33 | 148 | 98.1 + 0.40 + 0.18 |
| term1 | 12 | 110 | 755 | 72.6 + 6.40 + 1.14 |
| vg2 | 6 | 59 | 267 | 10.9 + 2.12 + 0.31 |
| total | 156 | 1186 | 11007 | 1791.0 + 139.85 + 37.67 |
| ratio | 1.22 | 1.00 | 1.08 | – |

Table 4: Algorithm 3 (mis-pga(new) + simultaneous placement and global routing).

| circuit | $\#t$ | $\#lb$ | $wl$ | CPU time [s] |
|---|---|---|---|---|
| alu2 | 14 | 152 | 1515 | 434.1 + 1.56 |
| alupla | 8 | 68 | 381 | 50.5 + 0.37 |
| bw | 10 | 91 | 609 | 25.1 + 0.64 |
| duke2 | 13 | 192 | 1846 | 341.2 + 1.94 |
| f51m | 7 | 51 | 227 | 33.9 + 0.20 |
| misex1 | 4 | 28 | 75 | 7.9 + 0.14 |
| misex3 | 15 | 190 | 1931 | 430.9 + 1.97 |
| misex3c | 15 | 189 | 1834 | 248.9 + 1.97 |
| rd73 | 4 | 23 | 70 | 36.9 + 0.10 |
| rd84 | 6 | 33 | 136 | 98.1 + 0.22 |
| term1 | 9 | 110 | 761 | 72.6 + 0.83 |
| vg2 | 5 | 59 | 331 | 10.9 + 0.33 |
| total | 110 | 1186 | 9716 | 1791.0 + 10.27 |
| ratio | 0.86 | 1.00 | 0.96 | – |

Table 5: `Maple` without path delay constraints.

| circuit | $\#t$ | $\#lb$ | $wl$ | CPU time [s] | $\#v$ | $d$ |
|---|---|---|---|---|---|---|
| alu2 | 13 | 192 | 1740 | 5.7 + 3.58 | 4 | 209 |
| alupla | 7 | 78 | 386 | 0.8 + 0.82 | 12 | 57 |
| bw | 8 | 105 | 602 | 1.6 + 1.12 | 5 | 145 |
| duke2 | 12 | 235 | 2264 | 4.7 + 4.90 | 4 | 103 |
| f51m | 7 | 63 | 323 | 1.3 + 0.81 | 14 | 119 |
| misex1 | 4 | 34 | 79 | 0.4 + 0.28 | 4 | 28 |
| misex3 | 11 | 236 | 2145 | 6.2 + 4.14 | 7 | 236 |
| misex3c | 11 | 231 | 2101 | 6.0 + 4.86 | 3 | 257 |
| rd73 | 4 | 23 | 60 | 0.6 + 0.16 | 10 | 37 |
| rd84 | 6 | 41 | 156 | 0.7 + 0.38 | 3 | 45 |
| term1 | 9 | 136 | 838 | 1.9 + 1.54 | 10 | 97 |
| vg2 | 5 | 69 | 297 | 0.9 + 0.48 | 4 | 35 |
| total | 96 | 1443 | 10991 | 30.8 + 23.07 | 80 | 1368 |
| ratio | 0.75 | 1.21 | 1.08 | – | – | 1.00 |

CPU time of Maple is shown in the form of "time for xl_split" + "time for Maple," where xl_split is the command in mis-pga which decomposes the input Boolean network and makes the number of the inputs of all nodes no more than $k(=4)$ as preprocessing.

Table 6: `Maple` with path delay constraints.

| circuit | $\#t$ | $\#lb$ | $wl$ | CPU time [s] | $\#v$ | $d$ |
|---|---|---|---|---|---|---|
| alu2 | 13 | 190 | 1945 | 5.7 + 10.92 | 0 | 171 |
| alupla | 7 | 75 | 382 | 0.8 + 1.17 | 1 | 52 |
| bw | 9 | 102 | 633 | 1.6 + 2.97 | 0 | 122 |
| duke2 | 12 | 230 | 2144 | 4.7 + 5.65 | 0 | 87 |
| f51m | 8 | 61 | 319 | 1.3 + 3.07 | 1 | 105 |
| misex1 | 4 | 37 | 90 | 0.4 + 0.35 | 0 | 24 |
| misex3 | 11 | 231 | 2282 | 6.2 + 18.22 | 0 | 200 |
| misex3c | 11 | 235 | 2204 | 6.0 + 15.48 | 0 | 218 |
| rd73 | 4 | 23 | 65 | 0.6 + 0.20 | 0 | 31 |
| rd84 | 6 | 39 | 162 | 0.7 + 0.43 | 0 | 38 |
| term1 | 10 | 137 | 888 | 1.9 + 4.03 | 0 | 82 |
| vg2 | 5 | 67 | 334 | 0.9 + 0.71 | 0 | 29 |
| total | 100 | 1427 | 11448 | 30.8 + 63.20 | 2 | 1159 |
| ratio | 0.78 | 1.20 | 1.13 | – | – | 0.85 |

The delays of one switch-block and one logic-block are set to one and three, respectively. The delay of each path is computed as the sum of the delays by logic-blocks and switch-blocks on the path. The path delay constraint $d_{max}$ of Table 1 is set to be 85% of the maximum delay obtained by Maple without constraints. $v$ is the number of violations for constraints. $d$ is the maximum delay of the paths with constraints.

## References


[1] N. B. Bhat and D. D. Hill, "Routable Technology Mapping for FPGA's," Proc. ACM/SIGDA Int. Workshop on FPGAs (FPGA'92), pp. 143-148, Feb. 1992.

[2] M. A. Breuer, "Min-Cut Placement," J. Design Automation and Fault Tolerant Computing, Vol. 1, No. 4, pp. 343-362, 1977.

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

[4] C. -S. Chen, Y. -W. Tsay, T. T. Hwang, A. C. H. Wu and Y. -L. Lin, "Combining Technology Mapping and Placement for Delay-Optimization in FPGA Designs," Proc. ICCAD-93, pp. 123-127, 1993.

[5] J. Cong, Y. Ding, A. Kahng, P. Trajmar and K. C. Chen, "An Improved Graph-Based FPGA Technology Mapping for Delay Optimization," Proc. 1992 IEEE Int. Conf. on Comput. Design, pp. 154-158, 1992.

[6] J. Cong and Y. Ding, "An Optimal Technology Mapping Algorithm of Delay Optimization in Lookup-Table Based FPGA Designs," Proc. ICCAD-92, pp. 48-53, 1992.

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

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

[9] R. J. Francis, J. Rose and Z. Vranesic, "Technology Mapping of Lookup Table-Based FPGAs for Performance," Proc. ICCAD-91, pp. 568-571, 1991.

[10] H. -C. Hsieh, W. S. Carter, J. Ja, E. Cheung, S. Schreifels, C. Erickson, P. Freidin, L. Tinkey and R. Kanazawa, "Third-Generation Architecture Boosts Speed and Density of Field-Programmable Gate Arrays," Proc. IEEE 1990 Custom Integrated Circuits Conf., pp. 31.2.1-31.2.7, 1990.

[11] K. Kawana, H. Keida, M. Sakamoto, K. Shibata and I. Moriyama, "An Efficient Logic Block Interconnect Architecture for User-Reprogrammable Gate Array," Proc. IEEE 1990 Custom Integrated Circuits Conf., pp. 31.3.1-31.3.4, 1990.

[12] S. Kirkpatric, C. D. Gelatt, Jr. and M. P. Vecchi, "Optimization by Simulated Annealing," Science, Vol. 220, No. 4598, pp. 671-680, 1983.

[13] U. P. Lauther, "Top Down Hierarchical Global Routing for Channelless Gate Arrays Based on Linear Assignment," Proc. VLSI '87, pp. 109-120, 1987.

[14] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton and A. Sangiovanni-Vincentelli, "Logic Synthesis for Programmable Gate Arrays," Proc. 27th DA Conf., pp. 620-625, 1990.

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

[16] R. Murgai, N. Shenoy, R. K. Brayton and A. Sangiovanni-Vincentelli, "Performance Directed Synthesis for Table Look Up Programmable Gate Arrays," Proc. ICCAD-91, pp. 572-575, 1991.

[17] M. Schlag, J. Kong and K. Chan, "Routability-Driven Technology Mapping for Look Up Table-Based FPGAs," Proc. 1992 IEEE Int. Conf. on Comput. Design, pp. 86-90, 1992.

[18] R. E. Tarjan, Data Structures and Network Algorithms, McGraw Hill, 1983.

[19] N. Togawa, M. Sato and T. Ohtsuki, "Simultaneous Placement and Global Routing Algorithm for FPGAs," Proc. ISCAS'94, pp. 483-486, 1994.