Algorithm

for Mapping Boolean Network

to LUT Based FPGAs

a dissertation submitted in partial fulfilment of the
requirements for the M.Tech.(Computer Science)
degree of the Indian Statistical Institute

By
Jayasri Bhattacharyya(mtc9910)

under the supervision of
Dr. Sushmita Sur-Kolay
Adavnced Computing and Microelectronics Unit,
Indian Statistical Institute,
Kolkata- 700 035

Indian Statistical Institute
203, Barrackpore Trunk Road
Kolkata 700035

Certificate of Approval

This is to certify that this thesis titled **Algorithms for Mapping Boolean Network to LUT Based FPGAs** submitted by **Jayasri Bhattacharyya** towards partial fulfillment of requirements for the degree of M.Tech. in Computer Science at Indian Statistical Institute, Kolkata embodies the work done under my supervision.

Susmita Sur-Kolay
2.8.2001

Susmita Sur-Kolay,
Associate Professor,
ACM Unit,
Indian Statistical Institute,
Kolkata-700 035.

P. K. Nandi 2/8/01
(External Examiner)

Acknowledgement

I pay my sincerest gratitude to Dr. Susmita Sur-Kolay, for her guidance, advice, enthusiasm and support throughout the course of this dissertation.

I would also like to thank Subhasis Bhattacharyya for his valuable support during the course of this project.

My special thanks to the ACM unit for providing me with computing facilities.

I thank all of my classmates, who gave me numerous suggestions during my project, and also a friendly atmosphere during my two years at ISI, Calcutta.

Finally, I express my heartiest thanks to the members of the M.Tech. Dissertation Committee.

Date: 02.08.2001

Jayasri Bhattacharyya

(Jayasri Bhattacharyya)

Contents

		Page No.
1	**Introduction**	**1**
	1.1 Introduction	1
	1.2 FPGA programming technologies	2
	1.3 Designing with FPGAs	2
	1.4 Scope	4
	1.5 Organization of the report	4
2	**Background**	**5**
3	**Problem formulation**	**7**
4	**The DAG-Map algorithm**	**8**
	4.1 Transformation of arbitrary networks into two input networks	8
	4.2 Technology mapping for delay minimization	10
	4.3 Postprocessing	12
5	**Our approach**	**15**
	5.1 Algorithm 1	15
	5.2 Algorithm 2	22
	5.3 Implementation details	23
6	**Results and conclusion**	**25**
References		27
Appendix		29

Chapter 1

Introduction

1.1 Introduction

Field Programmable Gate Arrays (FPGAs) have become one of the most popular implementation media for digital circuits. The key to FPGAs popularity is the ability to implement any circuit simply by appropriately programming an FPGA. Other circuit implementation options, such as Standard Cells or Mask Programmed Gate Arrays (MPGAs), require that a different VLSI chip be newly fabricated for each design. The use of a standard FPGAs has two key benefits: lower non-recurring engineering (NRE) cost and faster time to market.

All FPGAs are composed of three fundamental components: logic blocks, I/O blocks and programmable routing. A circuit is implemented in an FPGA by programming each of the logic blocks to implement a small portion of the logic required by the circuit, and each of the I/O blocks to act as either an input pad or an output pad, as required by the circuit. The programmable routing is configured to make all necessary connections between logic blocks and from logic blocks to I/O blocks.

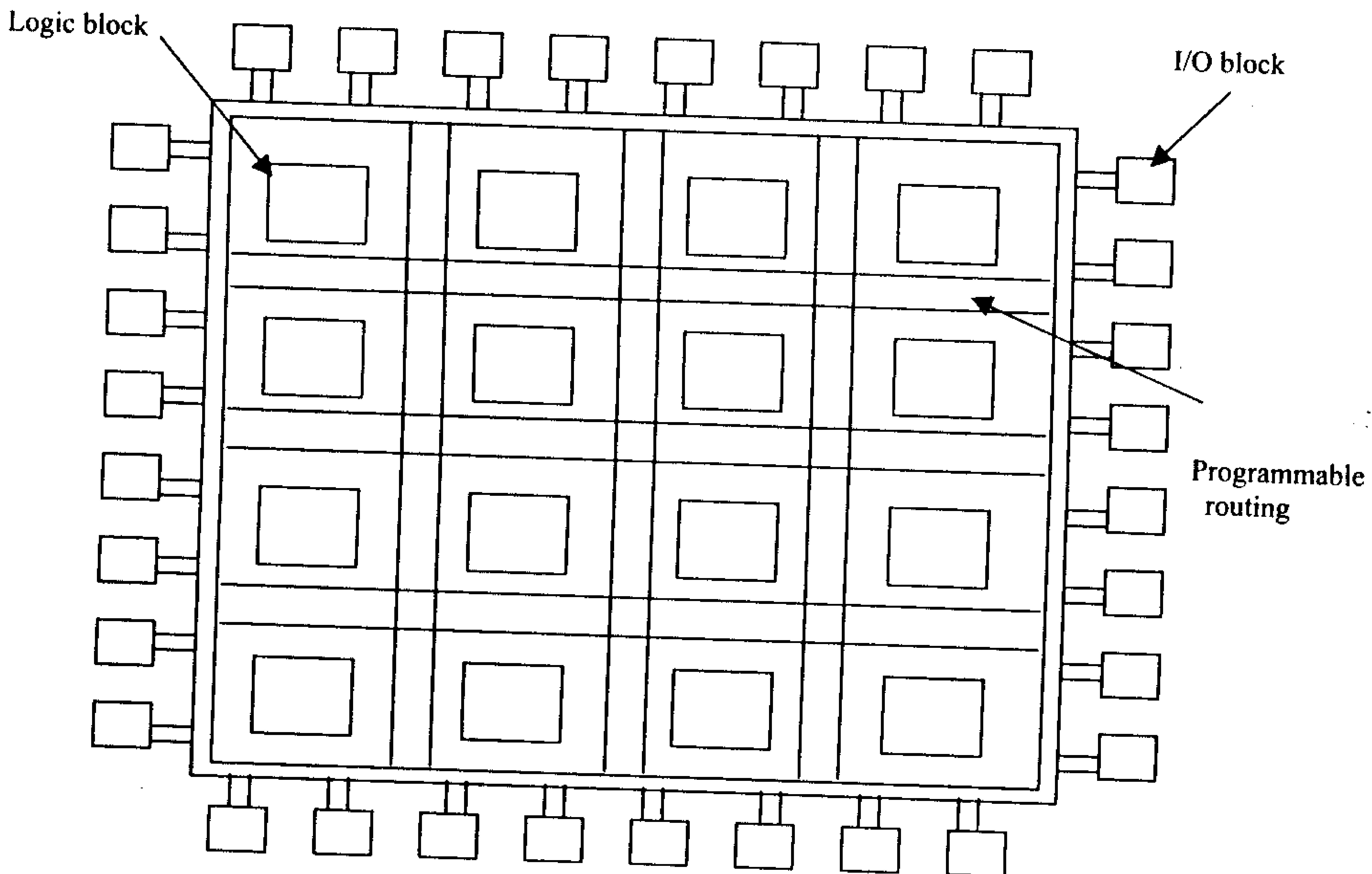


Figure 1.1: An FPGA

1.2 FPGA programming technologies

There are three different approaches of making FPGAs programmable. The most popular technology today uses SRAM cells to control pass transistors, multiplexers, and tri-state buffers in order to configure the programmable routing and logic blocks as required. Pass gates are implemented with nMOS pass transistors because this results in higher speed due to higher carrier mobility in nMOS transistors. Alternative programming technologies are antifuses and floating gate devices.

The logic blocks used in an FPGA strongly influences the FPGA speed and area efficiency. While many different logic blocks have been used in FPGAs, most current commercial FPGAs use logic blocks based on look-up-tables (LUTs). Figure 1.2 shows how a 2-input LUT can be implemented in an SRAM based FPGA -- a K-input LUT requires 2^K SRAM cells and a 2^K -input multiplexer. A K-input LUT can implement any function of K-inputs; one simply programs the 2^K SRAM cells to be the truth table of the desired function.

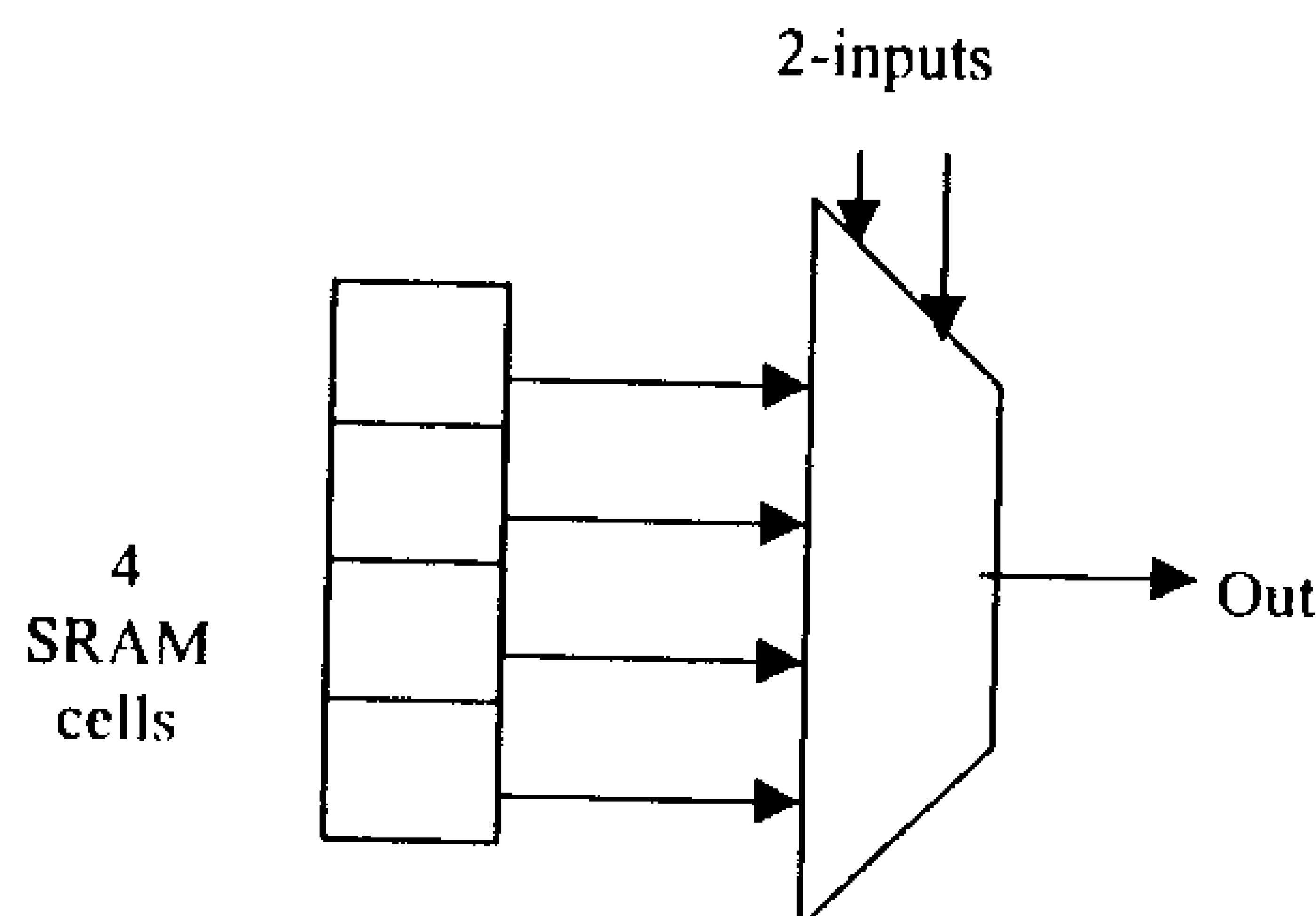


Figure 1.2: A 2-input LUT implemented in an SRAM-based FPGA

1.3 Designing with FPGAs

The problem of determining how to map a circuit into an FPGA is normally broken down into the steps given in Figure 1.3

The step for synthesis to logic block first converts the circuit description into a netlist of basic gates. Then this list of basic gates is converted to a netlist of FPGA logic blocks such that the number of logic blocks needed is minimum and/or circuit speed is maximum. The conversion from a net list of basic gates to a net list of logic blocks can be divided into the steps given in Figure 1.4

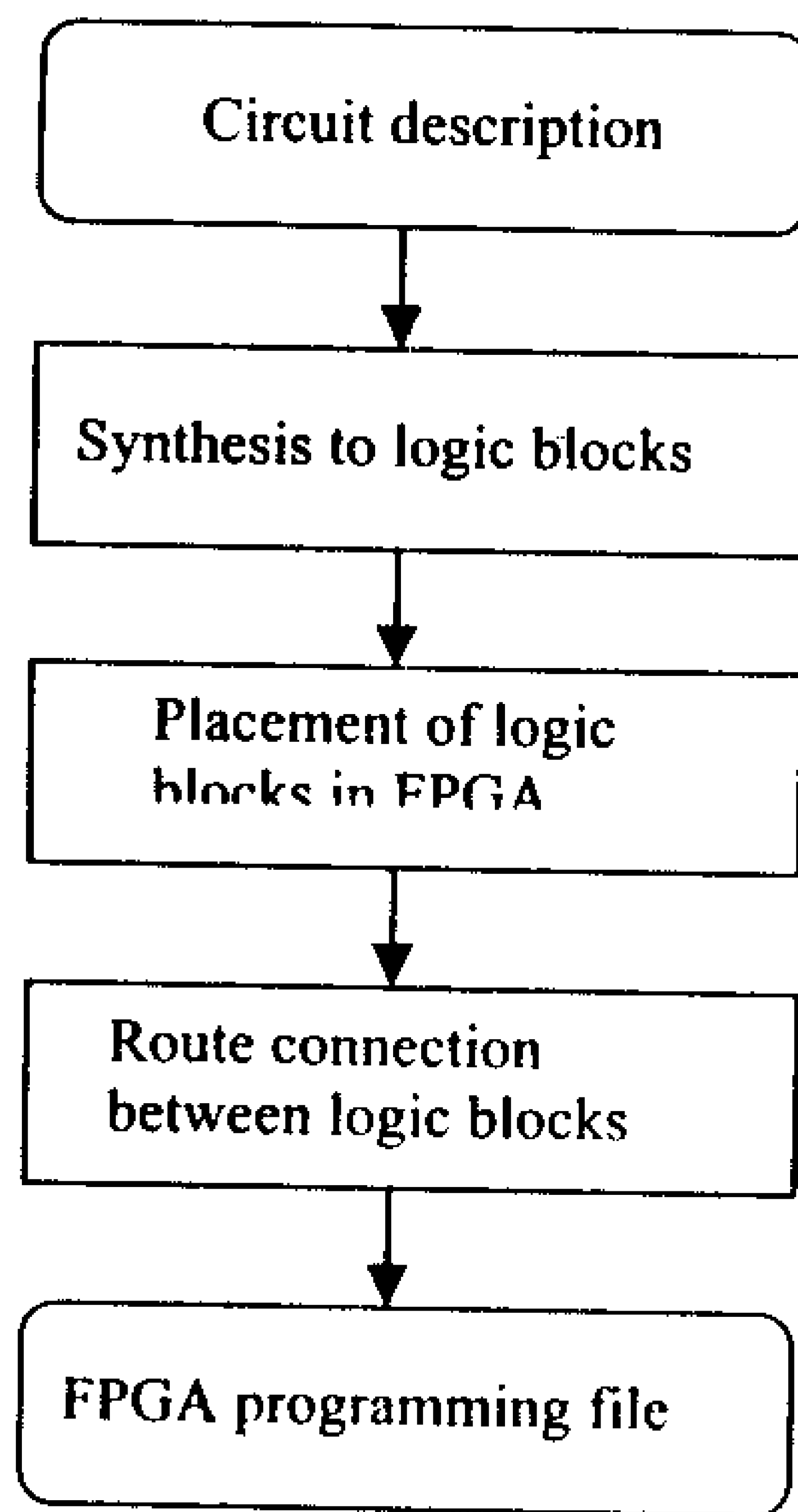


Figure 1.3: A brief description on FPGA

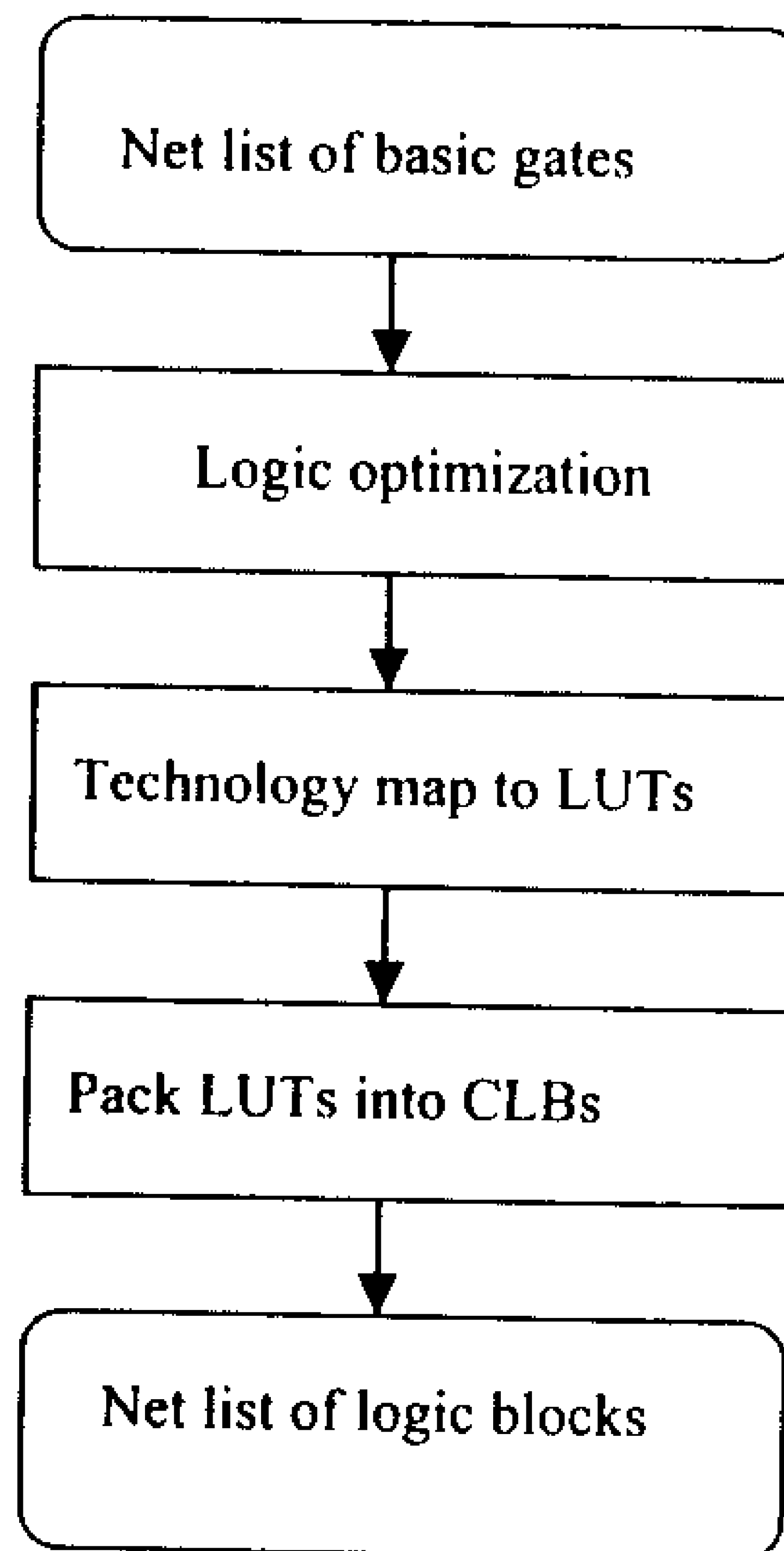


Figure 1.4: Details of synthesis procedure.

Logic optimization step removes redundant logic and simplifies logic whenever possible. This step does not consider the type of elements by which the final circuit is implemented, so it is called technology independent logic optimization.

In the technology mapping step respecting the limitations of the target FPGA, the optimized net list of basic gates is mapped to LUTs. In this step optimization is done to reduce the number of LUTs and/or the number of levels.

1.4 Scope

In this report we are dealing with some aspects of technology mapping problem. The salient criteria that characterizes the problem are minimization of the number of LUTs, delay minimization and routability. Here we are primarily concerned with delay optimization and then with area minimization .

The goal is to compute a mapping solution with small delay and small chip area. The delay of an FPGA circuit is determined as: delay in interconnection paths and delay in K-LUTs. Circuit-delay is approximated by the depth in S, as layout information is not available at this stage and the access time of K-LUT is independent of the function implemented. Therefore, the objective of our algorithm is to determine a mapping solution S with minimum depth and small chip area .

1.5 Organization of the report

The remainder of this report is organized as follows: chapter 2 gives the existing approaches, chapter 3 gives the precise problem formulation, in chapter 4 there is a description of DAG-Map algorithm, in chapter 5 we have talked about our approaches. Chapter 6 presents our experimental results and the conclusion .

Chapter 2

Background

The LUT based FPGA mapping algorithms can be roughly divided into three classes. The algorithms of the first class emphasize on minimizing the number of LUTs in the mapping solutions. This class includes MIS-pga and its enhancement, MIS-pga –new ,by Murgai et al. based on several synthesis techniques[1,2], Chortle and Chortle-crf by Francis et al. based on tree decomposition and bin packing techniques[3,4], Xmap by Karplus based on the if-then-else DAG representation[5], the algorithm by Woo based on the notion of edge visibility[6], and the work by Swakar and Thomas based on clique partitioning approach[7].The MIS-pga program first decomposes a given Boolean network into a feasible network using Roth-Karp decomposition and kernel extraction so that the number of inputs at each node is bounded. MIS-pga then enumerates all possible realizations of each network node and solves the binate covering problem to get a mapping solution using the least number of look-up-tables. In the improved MIS-pga more decomposition techniques are incorporated, including bin-packing, co-factoring, and AND-OR decomposition. The covering problem is solved more efficiently via certain preprocessing operations. The Chortle program and its successor Chortle-crf, decomposes a given Boolean network into a set of fanout-free trees and then carries out technology mapping on each tree using the dynamic programming approach. Bin-packing heuristics are used in Chortle-crf for gate-level decomposition, yielding significant improvement over its predecessor in the quality of solutions and the running time. The Xmap program transforms a given Boolean network into an if-then-else DAG representation and then goes through a simple marking process to determine the final mapping. The technology mapping problem proposed by Woo, introduces the notion of invisible edges to denote the edges which do not appear in the resulting network after mapping.

The algorithms in the second class emphasizes on minimizing the delay of the mapping solution. This class includes MIS-pga-delay[8], Chortle-d[9], DAG-Map[14], and FlowMap[10]. MIS-pga-delay[8] is an extension of MIS-pga, developed by Murgai et al. It contains two phases, mapping and placement/routing. The mapping phase first computes a delay-optimized two-input network, then traverses the network from the primary inputs, collapsing the nodes in the longest paths into their fanouts to reduce the network depth. During this procedure various decomposition techniques are used to dynamically resynthesize the network. The basic approach used in Chortle-d, developed by Francis et al. , is similar to that in Chortle-crf, i.e., decompose the network into fanout-free trees and then use dynamic programming and bin-packing heuristics to map each tree independently, minimizing at each step the depth of the node being processed. DAG-Map

algorithm is a graph-based algorithm that carries out technology mapping and delay optimization on the entire Boolean network, instead of decomposing it into fanout free trees. It is optimal for trees for any K-LUTs and Chortle-d is optimal for trees only when K is no more than six. The FlowMap algorithm solves LUT-based FPGA technology mapping problem for depth minimization optimally in polynomial time. A key step in this algorithm is to compute a minimum height K- feasible cut in a network, which is solved optimally in polynomial time based on network flow computation. This algorithm also effectively minimizes the number of LUTs by maximizing the volume of each cut by several post-processing operations.

There are other algorithms like FlowMap-r [11] and CutMap [12] that minimize area and depth simultaneously.

The mapping algorithms in the third class have the objective of maximizing the routability of the mapping solutions. Although many existing mapping methods showed encouraging results, these methods are heuristic in nature, and it is difficult to determine how far the mapping solutions of these algorithms are away from the optimal solution.

Chapter 3

Problem Formulation

A Boolean network can be represented as a directed acyclic graph (DAG) where each node represents a logic gate and there is a directed edge *(i,j)* if the output of gate *i* is an input of gate *j*. A primary input (PI) node has no incoming edge and primary output (PO) node has no outgoing edge. *input(v)* denotes the set of gates which supply input to *v*. Given a subgraph H of the Boolean network, *input(*H*)* denotes the set of distinct nodes that supply inputs to the gates in H. Let *v* be a node in the network, a K-feasible cone at *v*, denoted by C_v, is a subgraph consisting of *v* and predecessors of *v* such that any path connecting *v* and a node in C_v lies entirely in C_v, and $|input\ (C_v)| \leq K$. *left(v)* denotes the left fanin of node *v* and *right(v)* denotes the right fanin of node *v* where *v* is two input node The level of a node *v* is the longest path from any PI node to *v*. The level of a PI node is zero. The depth of a network is the largest node level in the network.

We assume that each programmable logic block in an FPGA is a K-input lookup table that can implement any Boolean function. Thus a K-LUT can implement any K-feasible cone of a Boolean network. The technology mapping problem is to cover a Boolean network with K-feasible cones. A technology mapping solution S is a DAG where each node is a K-feasible cone and an edge between (C_u, C_v) exists if u is in $input(C_v)$. LUT_v denotes the K-LUT rooted at *v*. *output(*LUT_v*)* is the set of LUTs which has LUT_v as input.

Chapter 4

The DAG-Map Algorithm

The DAG-Map algorithm consists of three major steps. The first step transforms an arbitrary Boolean network to two -input network. The second step maps the two input network into a K-LUT FPGA network with minimum delay. The third step performs a post processing area optimization of the network without increasing the network delay.

4.1 Transformation of arbitrary networks into Two-Input Networks

For transforming an arbitrary Boolean network into a two input Boolean network an algorithm DMIG is used.

algorithm: decompose-multi-input gate(DMIG)
Let V = $input(v) = \{u_1, u_2, u_3 \ldots\ldots u_m\}$;
    while |V |> 2 do
        let u_i and u_j be two nodes of V with smallest levels;
        introduce a new node x ;
        $input(x) = \{u_i, u_j\}$;
        $level(x) = \max(level(u_i), level(u_j)) + 1$;
        $V = (V - \{u_i, u_j\}) \cup \{x\}$
    end-while ;
    connect only two nodes left in V to v as its inputs;
    return the binary tree T(v) rooted at v;
end-algorithm.

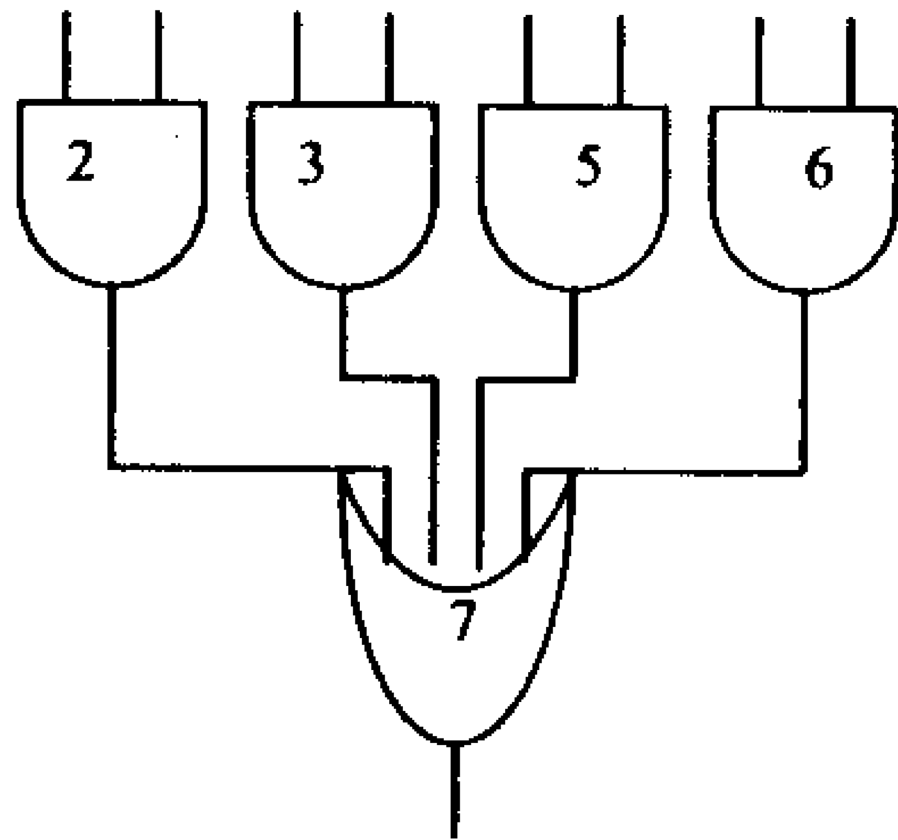
A straightforward way to transform an n-node arbitrary network into a two-input network is to replace each m-input gate $(m \geq 3)$ by a balanced binary tree. But this straight forward transformation may increase the network depth by as much as an $O(\log n)$ factor. It can be shown that the DMIG algorithm increases the network depth by at most a small constant factor.

Theorem 1: [14] For any arbitrary Boolean network G of simple gates, let G' be the network obtained by applying the DMIG algorithm to each multi-input gate in topological order starting from the PI nodes. Then

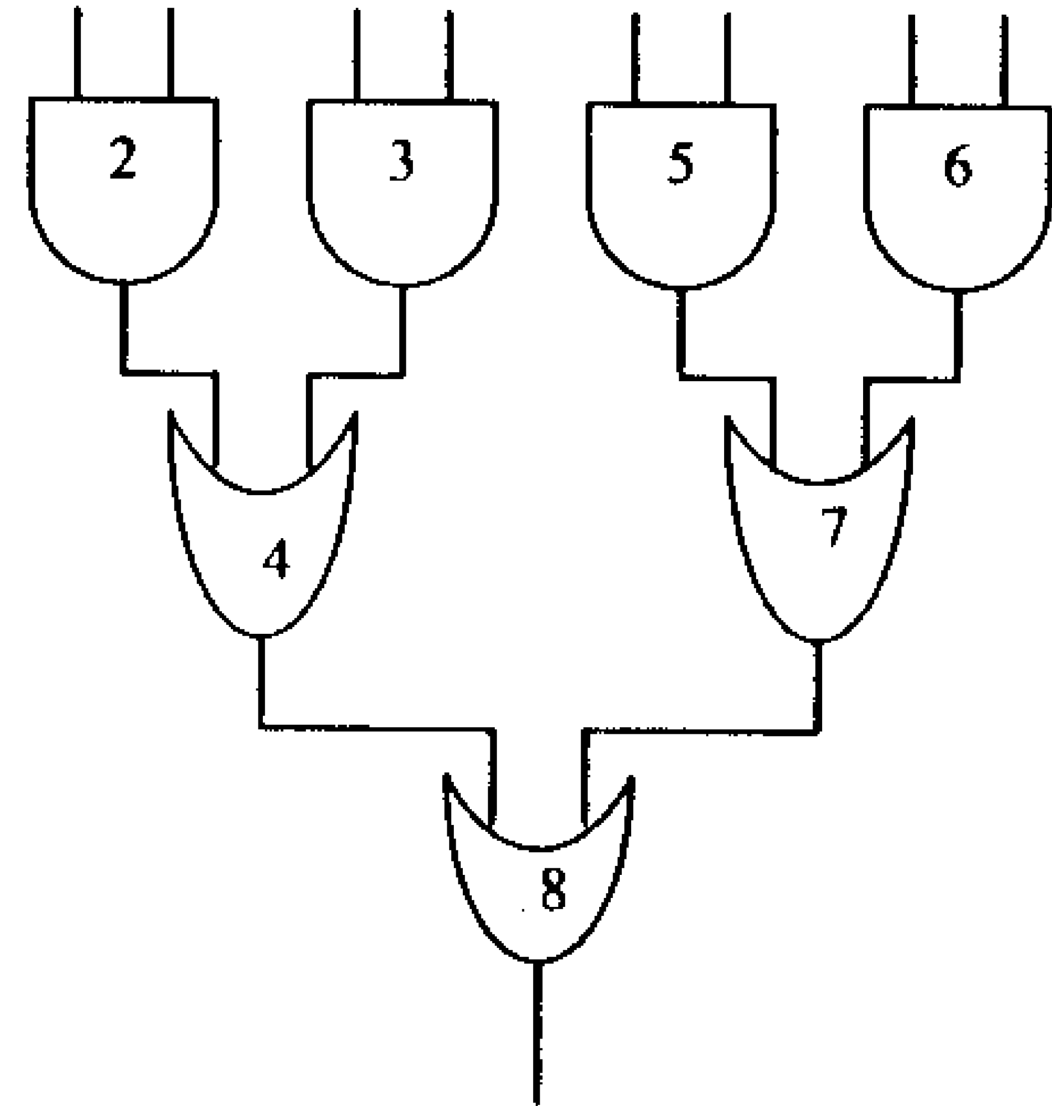
$depth(G') \leq \log 2d \cdot depth(G) + \log I,$ d=max degree of fanout in G;
I= number of PI nodes in G;

Now since d is bounded by a constant, so the depth of the two input network G' is increased by only a constant factor *log 2d*. Again for most networks in practice *depth(G) = O(log I)*. Hence the *depth (G')* is only a constant factor time of *depth(G)*.

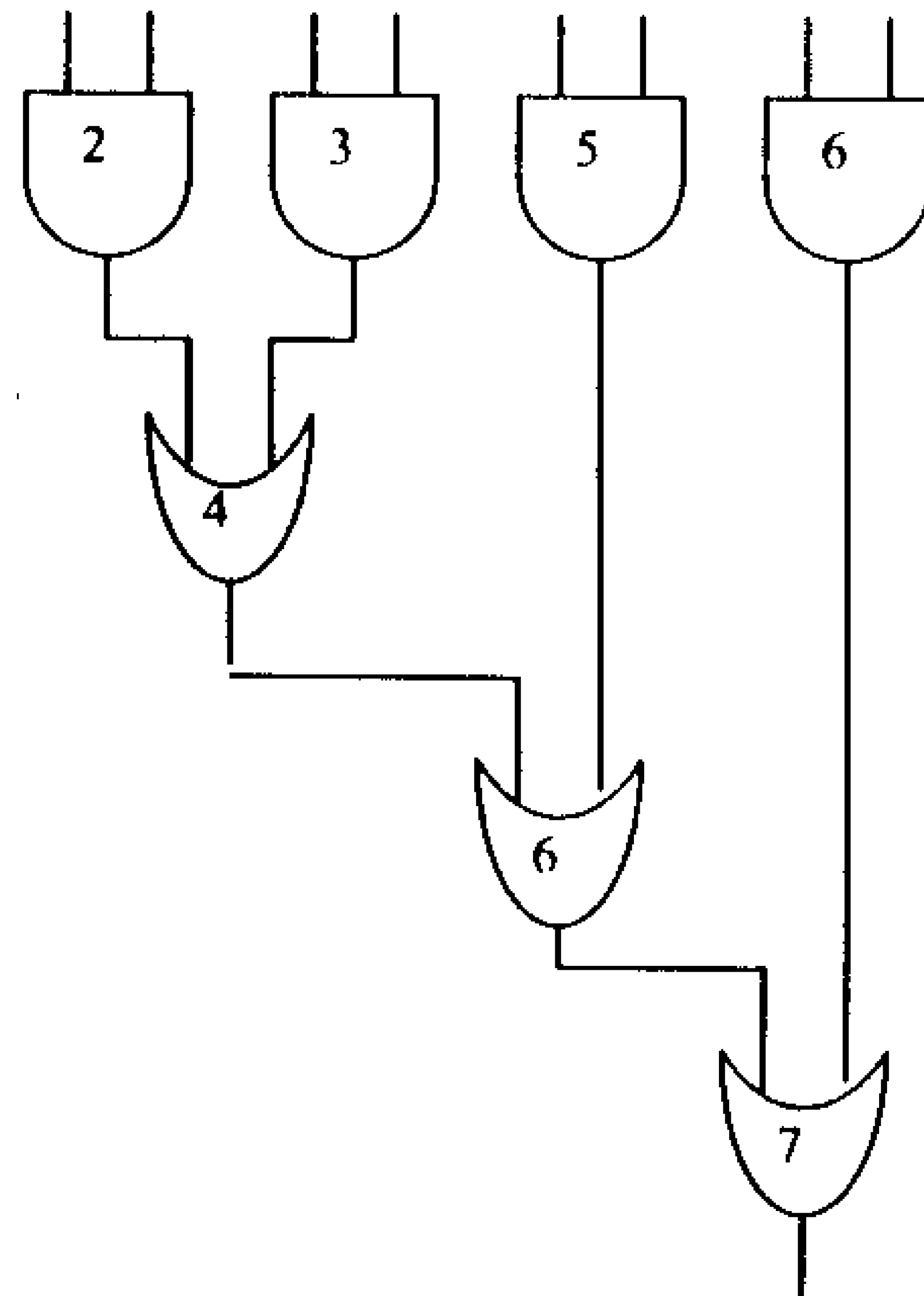
Fig. 4.1(a) shows a 4-input gate v (where the numbers inside the node indicate their levels) and Fig. 4.1(b) shows the result of replacing it by a balanced binary tree. However if we replace it by the binary tree in Fig. 4.1(c), the level of v remains 7.



(a) A four input gate



(b) Transformation using Balanced tree



(c) Transformation using DMIG

Figure 4.1: Transforming a multi-input network into two input network

4.2 Technology mapping for delay minimization

This step maps the two input network into a K-LUT FPGA network with mininmum delay.DAG-Map algorithm consists of two steps.

In the first step a level h(v) is assigned to each node v of the two-input network,such that h(v) is equal to the level of of the K-LUT in the final mapping solution. In the second step K-LUTs in the final mapping solution are generated.

DAG-Map Algorithm:

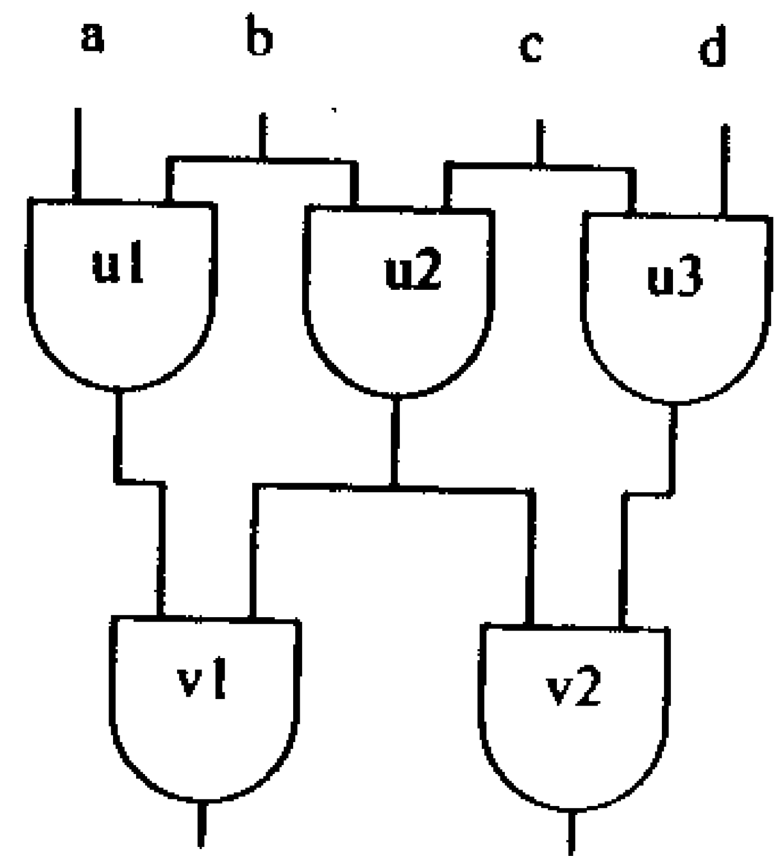
```
algorithm DAG-Map
/*step 1: labelling the network*/
for each PI node v do
        h(v) = 0 ;
T= list of non-PI nodes in topological order;
while T is not empty do
        remove the first node v from T;
        let p = max{h(u) | u ∈ input (v)} ;
        if | input (N_p(v) ∪{v}) | ≤ K
        then h(v) = p
        else h(v) = p+1
end-while ;

/*step 2: generate K-LUTs*/
L = list of PO nodes;
 while L contains non-PI nodes do
        remove a non-PI node v from L, i.e. L=L – {v} ;
        introduce a K-LUT v' to implement the function of v such that
                input (v') = input(N_h(v) (v)) ;
        L= L∪ input(v')
 end-while ;
end-algorithm .
```

In the step 1 of the algorithm we want to make $h(v)$ as small as possible to minimise delay. The nodes are labelled in topological order. The label of each PI node is zero. If v is not PI node and if the p is the maximum label of the nodes in $input(v)$, then v is assigned label p or $p+1$. $N_p(v)$ denotes the set of predecessors of v with label p. If $| input (N_p (v) \cup \{v\}) | \leq K$, $h(v)$ is assigned p else it is assigned $p+1$.

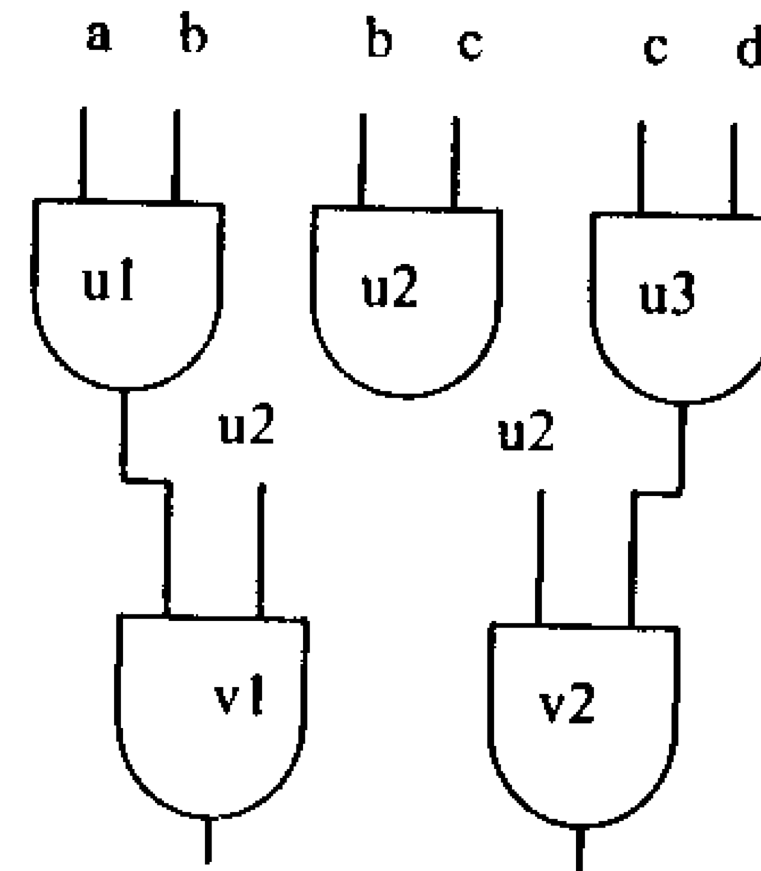
In step 2 L is the set of outputs which are implemented using K-LUTs. Initially it contains all the PO nodes. For each node v in L v is removed from L , a K-LUT v' is generated to implement v then L is updated to $L \cup input(v')$ where $input(v')=input(N_{h(v)}(v))$. This step ends when L consists of only PI nodes of the original network.

Advantages of DAG-Map

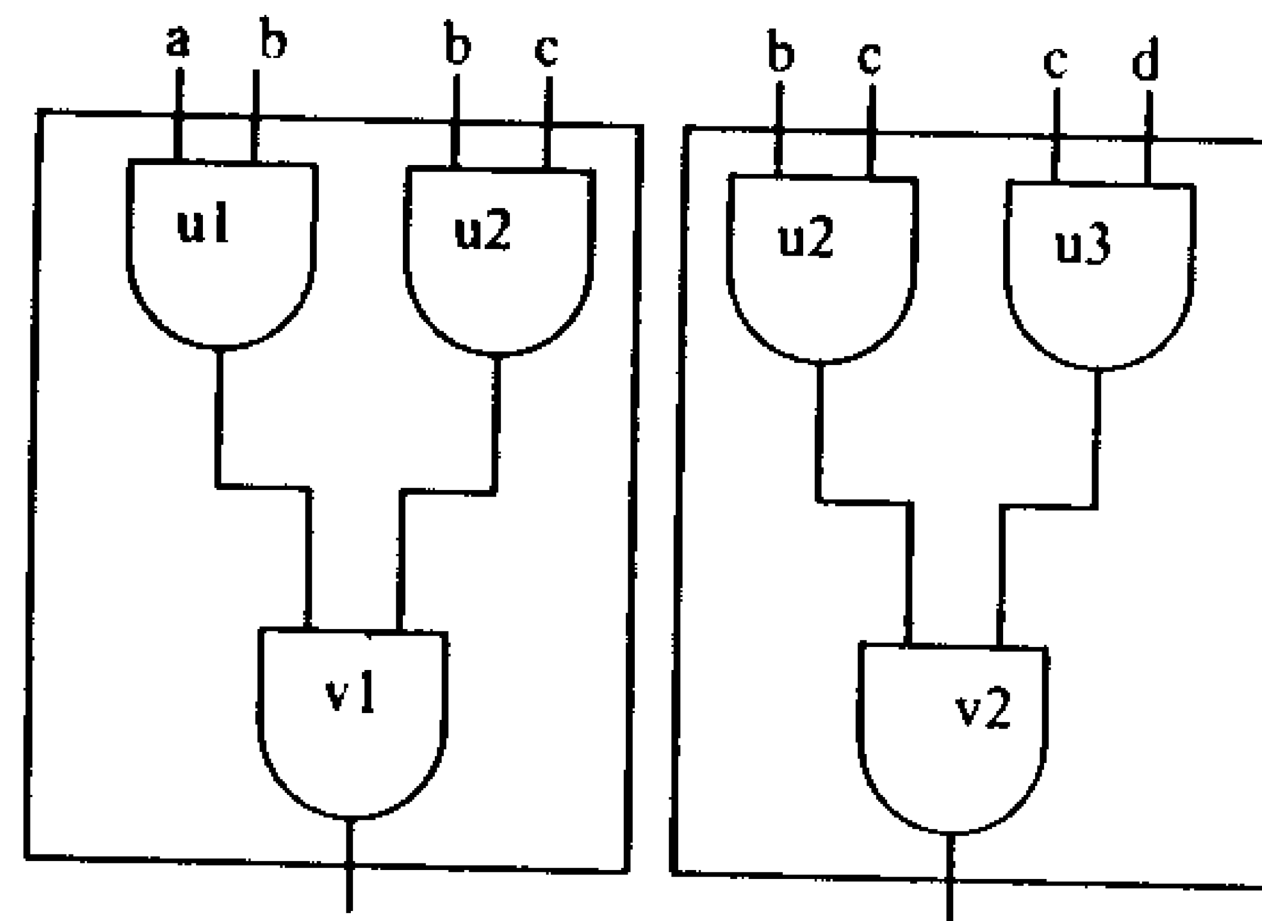
1) DAG-Map does not decompose the network into fanout free trees, which gives better mapping solution. As for example consider the network in 4.2(a), if we decompose into fanout free trees then we have 2 level mapping solution with 3 LUTs. But using DAG-Map we have a 1-level mapping solution with only 2 LUT's.



(a) Original network



(b) Decomposition into fanout free trees



(c) Mapping solution using DAG-Map

Figure 4.2: A mapping example for K=3

2) In DAG-Map algorithm nodes are replicated when necessary to reduce network delay. In the above fig we find that replication of node u_2 results into a one level mapping solution. But if that node is not replicated, the depth would be at least two.

Disadvantages of DAG-Map

1) The DAG-Map algorithm is optimal when the original network is a tree. When the original network is not a tree this algorithm may not give optimal solution.

2) While giving the mapping solution this algorithm does not take care of the fact that if an LUT contains some other LUT then the nodes of the bigger LUT which lies outside the LUT contained in it could be packed in the higher level LUTs .But if this fact is taken under consideration then there is a chance of reduction in number of LUTs in the mapping solution without any increase in delay.

Theorem: [14] For any integer K, if the Boolean network is a tree with fanin no more than K then at each node, the DAG-Map algorithm produces a minimum depth mapping solution for K-LUT based FPGAs.

4.3 Post-Processing:--Area optimization without increasing delay.

In DAG-Map the main objective is delay minimization. After applying DAG-Map two post-processing operations are done which reduces the number of K-LUTs without increasing the depth in the mapping solutions.

Two processing operations are done, they are namely gate-decomposition and predecessor packing.

The basic idea of gate decomposition is as follows. If node v is a simple gate of multiple inputs in the mapping solution, for any two of its inputs u_i and u_j, if u_i and u_j are single fanout nodes, we can decompose v into two nodes v_{ij} and v' such that $input(v_{ij}) = \{u_i,u_j\}$ and $input(v') = input(v) \cup \{v_{ij}\} - \{u_i, u_j\}$. Such a decomposition produces a logically equivalent network because of the associativity of the simple functions. Now in this case if $| input(u_i) \cup input(u_j) | \leq K$, then we can implement u_i , u_j and v using one K-LUT. So number of K-LUTs are reduced by one (without any increase in delay) moreover the decomposed node v has one fewer inputs (beneficial for further post-processing). When v implements a complex function Roth-Karp decomposition is used to determine if the node can be feasibly decomposed to v' and v_{ij} as described above. Given a Boolean function F(X,Y), where X and Y are Boolean vectors, the Roth-Karp decomposition determines if there is a pair of Boolean functions G and H such that F(X,Y)=G(H(X),Y),and generates such G and H if they exist. In this case, F is the function implemented by v, $X=(u_i, u_j)$, and Y consists of the remaining inputs of v. If the Roth-Karp decomposition succeeds on a pair of inputs u_i and u_j of node v, and $|input(u_i) \cup input(u_j)| \leq K$, then the gate decomposition is applicable.

When corresponding to a decomposable node v there exists u_i and u_j with the properties as described above then u_i and u_j are said to be mergeable nodes. In general Roth-Karp decomposition run in exponential time, but here since the number of fanins of a K-LUT is bounded by a small constant K, so here Roth-Karp takes only constant time.

Another post processing operation for area optimization is called predecessor packing. The idea behind this method is simple. For each node v, we examine all of its input nodes. If $| input(v) \cup input(u_i) | \leq K$ for some input node u_i , and u_i has only a single fan-out, then v and u_i are merged into a single K-LUT . In this case we also say that node

u_i and v are mergeable ,and call v the base of merge. This operation reduces the number of LUT by one.

There are many pairs of mergeable nodes in a network, but not all of these operations can be performed at the same time. For this an undirected graph G=(V,E) is constructed, where the vertex set V represents the nodes of the K-LUT network, an edge (v, w) is in E if and only if v and w are mergeable. A maximum matching in G is found and merge operation is applied, then G is reconstructed and the above procedure is repeated until E is empty.

If n is the total number of nodes in the actual Boolen network DAG-Map takes $O(n)$ time and post-processing takes $O(n^3)$ time ,the complexity of maximum matching algorithm being $O(n^3)$.

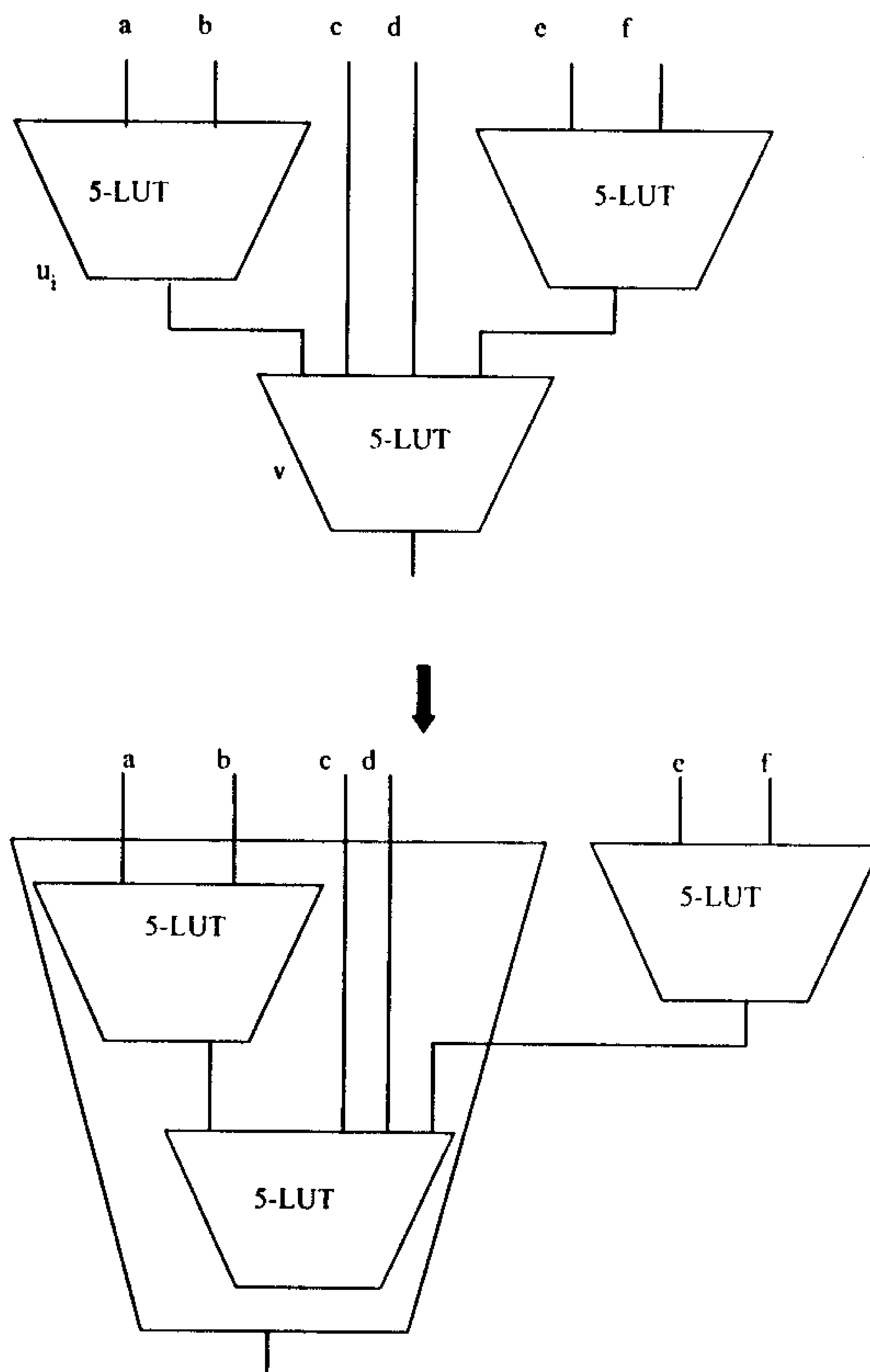


Figure 4.3: Predecessor packing for area minimization

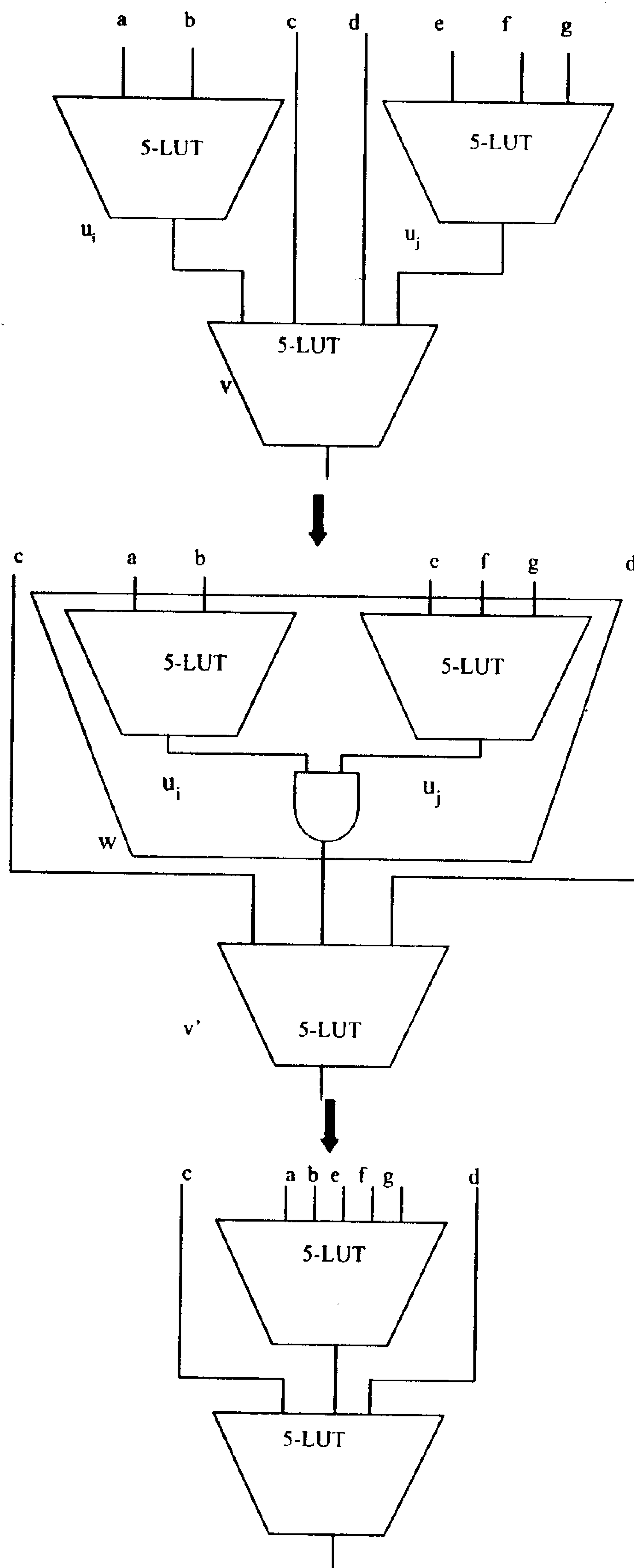


Figure 4.4: Gate decomposition for area optimization (assume K=5)

Chapter 5

Our approach

5.1 Algorithm 1

In the labeling phase of DAG-Map algorithm a label is assigned to each node which reflects the label of the LUT in which it will be packed in the mapping solution. In order to minimize the delay in DAG-Map each node v is assigned a label h(v) as small as possible. In some cases it is possible that by packing an internal node, say v, in a LUT of higher level the final delay of each output remains the same as in DAG-Map. If it happens that both the fanins of v are output node of some LUTs then this packing results in reduction of number of LUTs in the mapping solution.

Let us consider the two input network in Figure 3(a). The numbers within each node denotes the label as assigned by DAG-Map algorithm and the numbers on the left side of the node denotes the number of the node. In the mapping phase of the algorithm we see that 12 LUTs are needed in the mapping solution by using DAG-Map (assuming K=4).

The LUTs are as follows:

LUT rooted at 33	with inputs 18,25,28
LUT rooted at 32	with inputs 17,30
LUT rooted at 26	with inputs 21 ,23
LUT rooted at 25	with inputs 18 ,19,20
LUT rooted at 18	with inputs 0,1,2,3
LUT rooted at17	with inputs 0,1,3
LUT rooted at 19	with inputs 1,2,3
LUT rooted at 21	with inputs 7,8,9
LUT rooted at 20	with inputs 4,5,6,7
LUT rooted at 30	with inputs 23,24
LUT rooted at 28	with inputs 19,20,23,21
LUT rooted at 23	with inputs 3,5,6,8

Now we see that node 18 can be packed with the LUT rooted at 33. Again node 30 can be packed with the LUT rooted at 32. With this packing the LUTs rooted at 18 and at 30 are no more needed. So by packing a node in an LUT in higher level there is a reduction in the number of LUTs by 2 keeping the delay same as in DAG-Map.

All the PI nodes, PO nodes, and all those internal nodes that are output of an LUT (in the mapping solution of DAG-Map) are called here temporary output nodes. Corresponding

to each node v we have a function t(v) such that t(v) is set to 1 when v becomes a temporary output node, otherwise it is set to 0. Here when node 18 is packed with the LUT rooted at 33 there is a reduction in the number of LUTs by one, the reason behind this is that 18 is a temporary output node and both the fanins of node 18 are temporary output nodes.

If they were not then the number of LUTs would not have decreased. Same is the reason behind the reduction of number of LUTs when 30 is packed with the LUT rooted at 32.

In our algorithm (Quick-Map-Spack) we have traced the temporary outputs in the labelling phase of the algorithm. Then in the mapping phase when we have a temporary output node v, we first check whether both the fanins of v are temporary output nodes or not. Then we check whether this node could be packed in all the LUTs in output(LUT_v).If all these conditions hold true then we do the required packing and LUT count reduces by 1.

While packing a node we are not considering the fact that whether the node is on a critical path or not. In our example we have packed node 18 and node 30 in higher level LUTs.18 is on a non-critical path but node 30 is on the critical path. The packing of a node on critical path is possible because a node, say v, becomes eligible for being packed in the higher level LUT if and only if at least one of its two fanins becomes the root of an LUT which is totally contained in the LUT rooted at v. i.e. this packing reduces the number of LUTs but not the level. So while packing we do not have to see whether the node is on a critical path or not.

algorithm 1: Quick-Map-Spack

```
/*step 1:labelling  the network */
for each node v do
        t(v) = 0;
for each PI node v do
        h(v)=0;
        t(v)=1;
T= list of non-PI nodes in topological order;
while T is not empty do
        remove the first node v from T;
        let p= max{h(u) | u ∈input(v) };
        if |input (N_p(v) ∪{v})| ≤ K
        then h(v) = p
        else  h(v) = p+1
                ∀u ∈input(v)  t(v)=1
end-while;


/*step 2:generate K-LUTs*/
L = list of PO nodes;
while L contains non-PI nodes do
        remove a non-PI node v from L, i.e. L=L – {v} ;
        if t(v)=1 and v is not a PO node
        then introduce a K-LUT LUT_V to implement the function of v such that
                input (LUT_V) = input(N_h(v) (v)) ;
                L= L∪ input(LUT_V)
                If t(left(v))=1 and t(right(v))=1
                 and if ∀ LUT_u ∈ output(LUT_V)  |input N_h(u)(u) |≤K - 1
                then  ∀LUT_u ∈ output(LUT_V) modify the K-LUT LUT_u such that
                             input(LUT_u) = input(N_h(u)(u)) ∪{left(v),right(v)} - v
                           remove  K-LUT LUT_V;
                              L= L ∪ { left(v),right(v)} - input(LUT_V);
                end-if;
        end-if;
 end-while ;
end-algorithm .
```

Using our algorithm (Quick-Map-Spack)the mapping solution of the two input network in figure 3 needs 10 LUTs. The LUTs are as follows:

LUT rooted at 33	with inputs 17,2 ,25.28
LUT rooted at 32	with inputs 28,23,17
LUT rooted at 26	with inputs 21 ,23
LUT rooted at 25	with inputs 18 ,19,20
LUT rooted at17	with inputs 0,1,3
LUT rooted at 19	with inputs 1,2,3
LUT rooted at 21	with inputs 7,8,9
LUT rooted at 20	with inputs 4,5,6,7
LUT rooted at 28	with inputs 19,20,23,21
LUT rooted at 23	with inputs 3,5,6,8

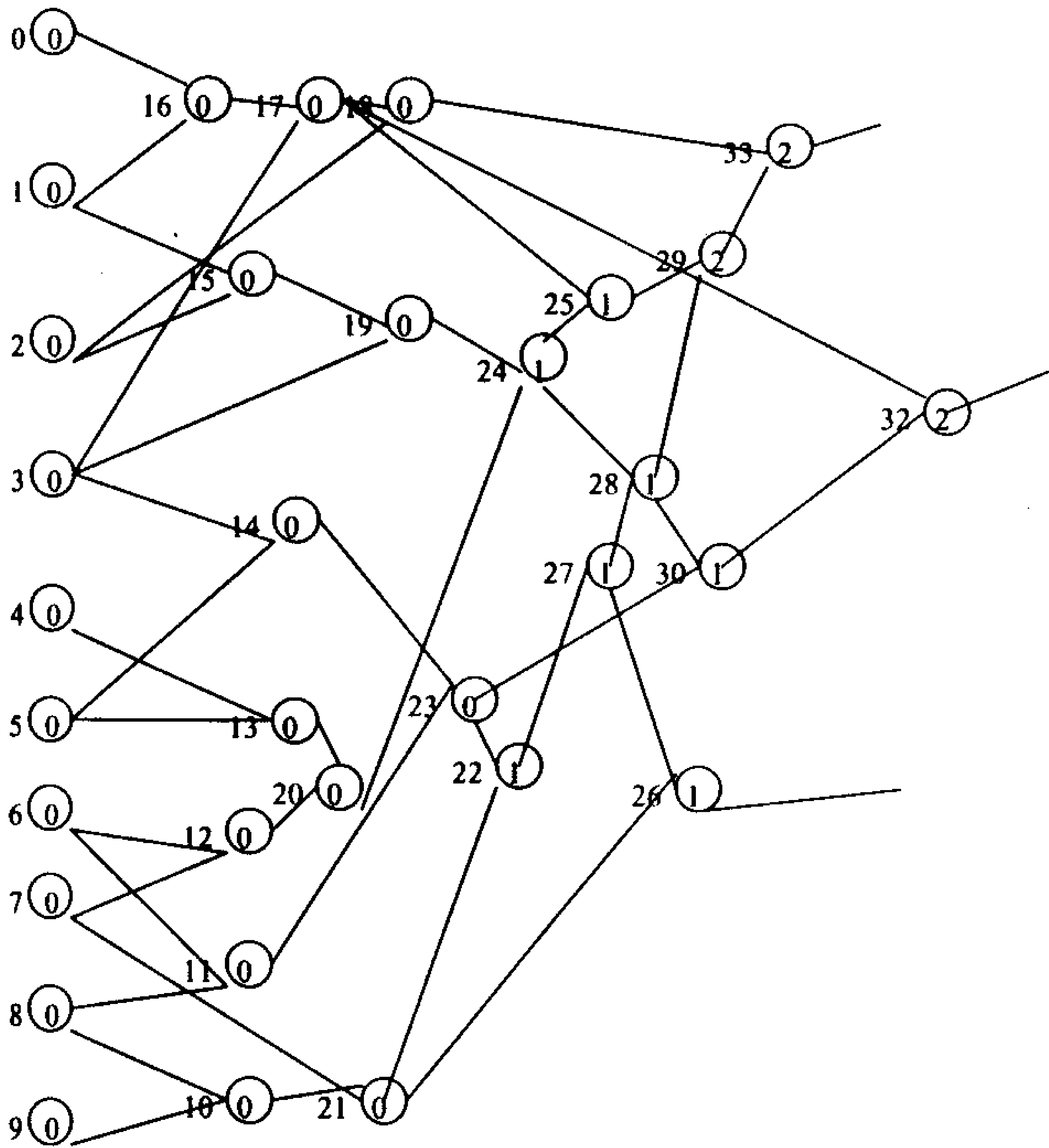


Figure 3(a) :A two input Boolean network. Numbers on the left side of the nodes give the node number and the numbers within the nodes give the label of the node when the network is labeled according to the DAG-Map algorithm.

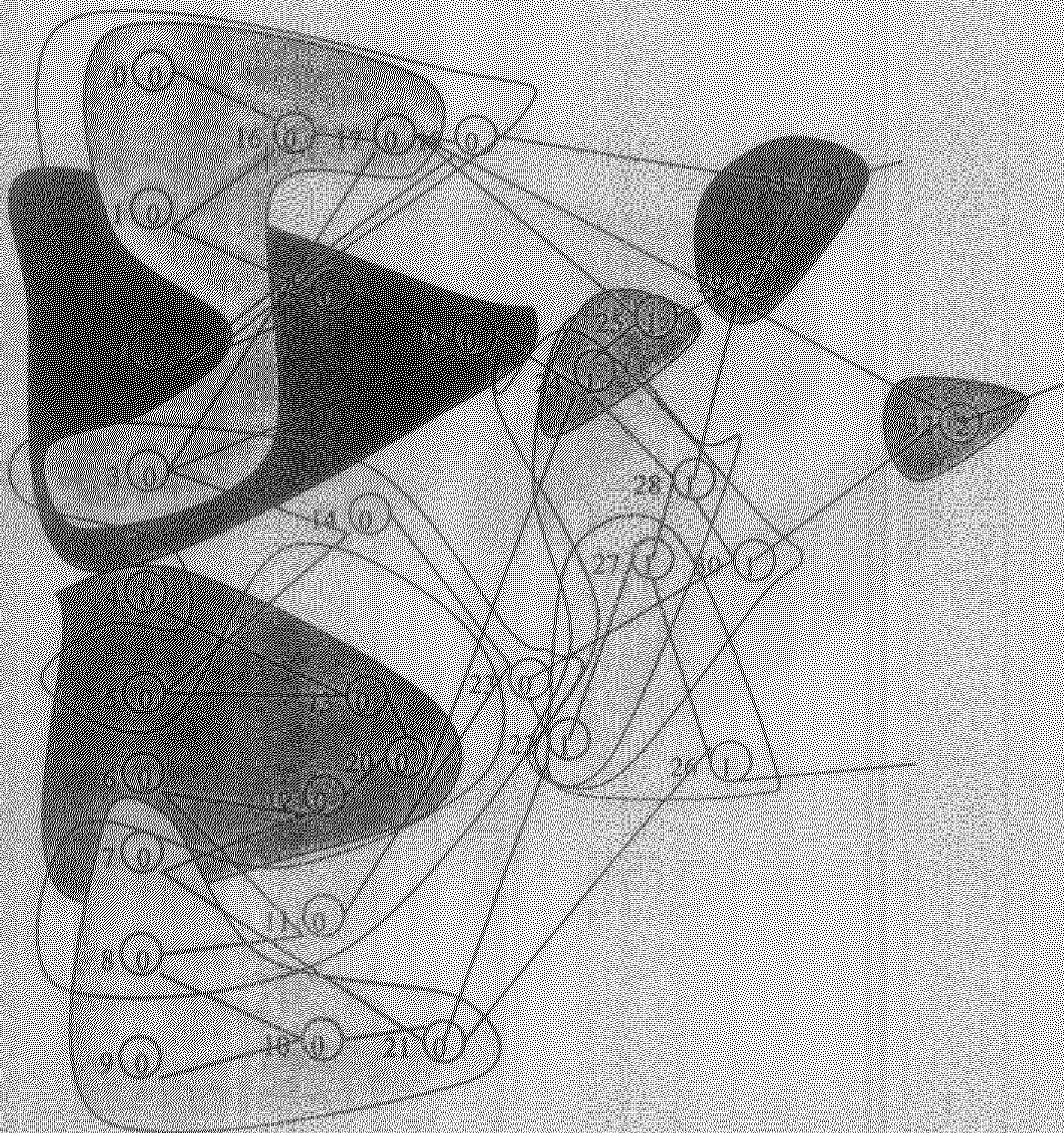


Figure 3(b): The mapping solution using DAG-Map algorithm of the two input network in Figure 3(a). Here K=4, number of LUTs= 12, delay = 3

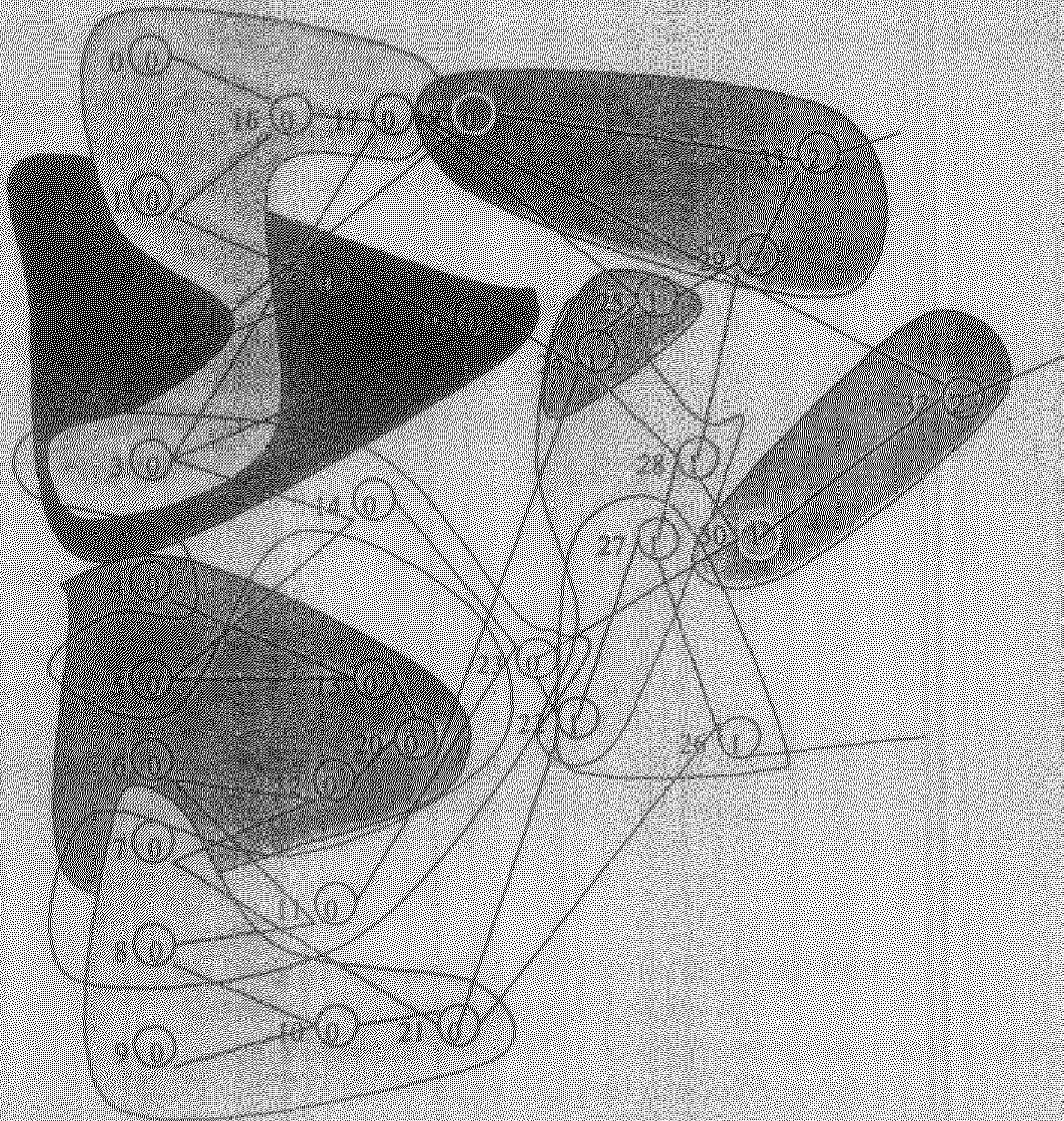


Figure 3(c): The mapping solution using our algorithm (algorithm 1) of the two input network in figure 3.The nodes which are white out lined are the nodes which are being differently packed as they are bring packed in DAG-Map. Here K= 4, number of LUTs = 10, delay = 3

5.2 Algorithm2

In the DAG-Map after executing the DAG-Map algorithm on the two input Boolean network some postprocessing is done to reduce the number of LUTs. In this postprocessing phase nodes are merged using two methods namely, predecessor packing and gate decomposition. In the following algorithm we have introduced the idea of predecessor packing in a greedy manner while getting a mapping solution i.e. while packing nodes in a LUT we have packed an LUT with its predecessor whenever possible.

algorithm2 : Quick-Map-Ppack

```
/*step 1: labelling the network*/
for each PI node v do
        h(v)=0;
T= list of non-PI nodes in topological order;
 while T is not empty do
        remove the first node v from T;
        let p= max{h(u) | u ∈input (v)};
        if |input (N_p(v) ∪{v})| ≤ K
        then h(v) = p
        else  h(v) = p+1
 end-while;

/*step 2:generate K-LUTs*/
flag=0;
L= list of PO nodes;
 while L  contains  non-PI nodes do
        remove a non-PI node v from L, i.e. L=L –{v};
        TMP=input(N_h(v) (v));
        while(TMP is not empty)
        remove u from TMP
        if (|input( N_h(u)(u) ∪N_h(v)(v))| ≤ K+1)and(u is fanout free)
              then L=L∪input(N_h(v)(v) ∪N_h(u)(u)) - u;
                     introduce a K-LUT v' to implement the function of v such that
                     input(v')= input(N_h(v)(v)) ∪input(N_h(u)(u))-u;
                     flag=1;
                     break;
        end-while;

        if flag=0
              then L =L∪input(N_h(v)(v));
                     introduce a K-LUT v' to implement the function of v such that
                     input(v')= input(N_h(v)(v));

  end-while;
 end-algorithm;
```

In the section 6 we have given a table comparing the number of LUTs needed when algorithm 2 is executed on some MCNC benchmark circuit with the number of LUTs needed when in the post processing step of DAG-Map algorithm only predecessor packing is used .

In the postprocessing phase of DAG-Map, so that maximum possible number of nodes can be merged the maximum matching algorithm is used repeatedly. Now maximum matching algorithm takes $O(n^3)$ time while DAG-Map takes O(n) time. Time complexity of our algorithm (algorithm 2) is O(n) ,Table 6.1 in section 6 shows that the number of LUTs in the mapping solution using algorithm 2 is very close to the number of LUTs needed in the mapping solution using DAG-Map and then applying predecessor packing in the postprocessing step, though the former algorithm takes less time.

5.3 Implementation details

Input specification:
We have taken arbitrary Boolean networks in blif or pla format. Then we have processed this circuit upto some extent using the SIS package. If the circuit is in pla(blif) format we have used read_pla (read_blif) command of SIS package to read the circuit. Then tech_decomp -a 2 -o 2 is used which decomposes multi-input Boolean network into two input Boolean network. Then we have used simplify command which performs two level minimization, nodeopt command that removes redundant nodes and reset_name command that renames the nodes in an increasing order. These commands simplify, nodeopt and reset_name are used in this order again and again until there is no reduction in the total number of nodes. Then we have used write_eqn command to write the output on a given file. Our program takes this file as input. Our program takes also the number of PI, number of PO and the value of K as input.

Output specification:
Our program gives the total number of LUTs needed. It also gives the input (nodes) and output (node) corresponding to each LUT .

Data Structures:
Three structures are used: dagnode, st_node, list_node.

dagnode- After reading the input a directed acyclic graph is formed. Corresponding to each node in the dag we have a dagnode. It has the following important fields: left, right, parent, name.
left points to the dagnode corresponding to the left fanin of this node, right points to the dagnode corresponding to the right fanin. parent points to an array of dagnodes corresponding to the fanouts of this particular node. name contains the name of the node.

list_node --It contains a pointer ptr to a dagnode, and a pointer next to a list_node.

st_node -- It contains a pointer ptr to a dagnode , and a pointer lptr to a linked-list of list_node. In fact lptr of a node say, v ,points to a linked-list which contains pointer to the dagnodes corresponding to the nodes in $input(N_{h(v)}(v))$.

Important Modules:
Some important modules in our implementation:

find_dag : The task of this function is to form the directed acyclic graph .

level_dag : this module performs the labelling part of our algorithms.

apply_map : this module performs the mapping part of our algorithms.

Chapter 6

Results and Conclusion

In this report, we have presented two graph based technology mapping algorithm for delay optimization in look-up-table based FPGA design. Both the algorithm Quick-Map-Spack and Quick-Map-Ppack carries out technology mapping on the entire Boolean network. We have implemented Quick-Map-Spack and Quick-Map-Ppack algorithms using the C language. We have taken arbitrary Boolean networks in blif or pla format. Then we have processed this circuit upto some extent using the SIS package. If the circuit is in pla(blif) format we have used read_pla (read_blif) command of SIS package to read the circuit. Then tech_decomp -a 2 -o 2 is used which decomposes multi-input Boolean network into two input Boolean network. Then we have used simplify command which performs two level minimization, nodeopt command that removes redundant nodes and reset_name command that renames the nodes in an increasing order. These commands simplify, nodeopt and reset_name are used in this order again and again until there is no reduction in the total number of nodes. On this resulting two input network we ran our algorithm Quick-Map-Spack on that same two input Boolean network we apply the DAG-Map as available in SIS package. The comparison of these two results is given in the table 6.1.

Table 6.1 gives the reduction of the number of LUTs compared to the number of LUTs in DAG-Map when Quick-Map-Spack is run on the MCNC benchmark circuits (given in the leftmost column of the table), for K=3, 4 and 5.

Table 6.1: Comparison of Quick-Map-Spack with DAG-Map

	# nodes in two input Boolean network	# LUTs in DAG-Map - # LUTs in Quick-Map-Spack		
		K=3	K=4	K=5
count	395	12	11	5
9sym	331	24	0	0
9symml	307	16	0	0
vg2	558	66	5	5
misex1	60	4	3	1
rd84	3284	511	0	30
duke2	2145	145	10	14
alu4	3267	360	10	7
sao2	246	33	0	1

Before running Quick-Map-Ppack on a circuit we do the same preprocessing with SIS package as mentioned above. In table 6.2 we showed the reductions in number of LUTs for a range of values of K=3 to 5 by applying Predecessor Packing over DAG-Map outcomes (column 2, 3 and 4) and Quick-Map-Ppack algorithm just after node labeling (column 5, 6 and 7). In Column 8, 9 and 10 we have showed the difference between the reductions in the # LUTs by applying predecessor packing and QMPpack.
We observe that for K=3, 4 and 5 the differences are (on an average) 5, 3 and 2 respectively. We find that using a linear algorithm QMPpack we are getting a mapping solution with number of LUTs very close to the number of LUTs needed in a mapping solution with time complexity $O(n^3)$

Table 6.2: Compares the result of Quick-Map-Ppack with DAG-Map + predecessor packing

	# of nodes in two input Boolean network	# LUTs reduced by applying predecessor packing on DAG-Map algorithm.			# LUTs in DAG-Map (without post processing) - # LUTs in QMPpack			#LUTs in QMPpack - #LUTs in DAG-Map (with Predecessor Packing)		
		K=3	K=4	K=5	K=3	K=4	K=5	K=3	K=4	K=5
count	395	16	17	15	16	14	12	0	3	3
9sym	331	24	15	8	24	15	7	0	0	1
9symml	307	16	14	8	16	14	8	0	0	0
vg2	558	107	34	19	91	20	15	16	14	4
misex1	60	12	2	0	12	2	0	0	0	0
alu4	3267	380	117	54	359	109	50	21	8	4
rd84	3284	1008	30	140	1008	30	140	0	0	0
duke2	2145	273	71	70	272	68	65	1	3	5
sao2	246	40	8	5	36	7	5	4	1	0
Total								42	29	17

References:

[1] Murgai, R., Y. Nishizaki, , N.Shenoy , R. K.Brayton, and A. Sangiovanni-Vincentelli, "Logic Synthesis Algorithms for Programmable Gate Arrays," *Proc.* 27th ACM/IEEE *Design Automation Conf.*, pp.620-625,1990

[2] Murgai, R.,N.Shenoy, R. K.Brayton, and A. Sangiovanni-Vincentelli, "Improved Logic Synthesis Algorithms for Table Look Up Architectures, " *Proc.*IEEE *Int'l Conf. on Computer-Aided Design*, pp.564-567,Nov.1991.

[3] Francis,R.J., J. Rose, and K. Chung, "Chortle: A Technology Mapping Program for Lookup Table Based Field Programmable Gate Arrays," *Proc.* 27th ACM/IEEE *Design Automation Conf.*, pp.613-619,1990

[4] Francis,R.J., J. Rose, and Z.Vranesic, "Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs ," *Proc.* 28th ACM/IEEE *Design Automation Conf.*,pp. 613-619, June 1991.

[5] Karplus,K.,"Xmap: A Technology Mapper for Table-Lookup Field Programmable Gate Arrays," *Proc.* 28th ACM/IEEE *Design Automation Conf.*,pp. 240-243, June 1991.

[6] Woo,N.S.,"A Heuristic Method for FPGA Technology Mapping Based on the Edge Visibility," *Proc.* 28th ACM/IEEE *Design Automation Conf.*,pp. 248-251, June 1991.

[7] Sawkar,P. and D.Thomas, "Technology Mapping for Table-Look-Up Based Field Programmable Gate Arrays," ACM/SIGDA *Workshop on Field Programmable Gate Arrays*, pp. 83-88,Feb.1992

[8] Murgai, R.,N.Shenoy, R. K.Brayton, and A. Sangiovanni-Vincentelli, "Performance Directed Synthesis for Table Look Up Programmable Gate Arrays," *Proc.*IEEE *Int'l Conf. on Computer-Aided Design*, pp.572-575,Nov.1991.

[9] Francis,R.J., J. Rose, and Z.Vranesic, "Technology Mapping for Dely Optimization of Lookup Table Based FPGAs," *MCNC logic synthesis workshop*,1991.

[10] Cong,J., A.Kahng, P.Trajmar,and K.C.Chen,"Graph Based FPGA Technology Mapping for Delay Optimisation," ACM *Int'l Workshop on Field Programmable Gate Arrays*, pp.77-82,Feb.1992.

[11]J. Cong and Y. Ding, " *On Area/Depth Trade- off in LUT-Based FPGA Technology Mapping,* "*IEEE Trans. on VLSI Systems*, vol. 2, no. 2, pp. 137-148, June 1994.

[12]Cong, J. and Y.-Y. Hwang, "Simultaneous Depth and Area Minimization in LUT-Based FPGA Mapping," *Proc. ACM 3rd Int'l Symp. on Field Programmable Gate Arrays, pp. 68- 74, Feb. 1995.*

[13] Roth, J. P. and R. M. Karp, "Minimization Over Boolean Graphs," *IBM Journal of research and development,* pp.227-238, April 1962.

[14] Cong,J., A.Kahng,Y.Ding, P.Trajmar,and K.C.Chen,"DAG-Map: Graph Based FPGA Technology Mapping for Delay Optimisation," IEEE *Design and Test of Computers*, pp.7-20, Sep.1992

```
/******************************************************/
/******* Source code for algorithm 1*****************/
/******************************************************/

# include<stdio.h>
# include<ctype.h>
# define max(a,b) (a>b ? a : b)

int t_nodes;        /*t_nodes= total no. of nodes except the PI nodes*/
int lut_count;      /*lut_count= total no. of LUTs needed in the mapping soln*/
int g_depth=0;

typedef struct node1
      {
      int if_in;
      int if_out;
      int tmp_out;
      int name;
      int label;
      int lut;
      int depth;
      int slack;
      struct node1 *left;
      struct node1 *right;
      struct node1 **parent;
      }dagnode;

typedef struct node3
      {
      int name;
      dagnode *ptr;
      struct node3 *next;
      }list_node;

typedef struct node2
      {
      int if_lut;
      int slack;
      int is_in;
      int name;
      int left;
      int right;
      int t_nodes;
      int p_count;
      dagnode *ptr;
      list_node *lptr;
      }st_node;


typedef struct node4
      {
      int name;
      int **chlid;
      }lut_dagnode;


/*************************************************************************/
```

```
st_node **read_input(int n)
{                              /*this fn reads the input from the file*/
                               /*which contains the two input Boolean */
                               /*fns in eqn form*/

int j,i,flag,c,lno,flag1,d[10],val;int p;
st_node **a;
FILE *fp;
flag=0;
flag1=0;
lno=0;
p='0';
t_nodes=0;
fp=fopen("input filename","r");
if(fp==NULL)
      {
      exit(0);
      }
while((c=fgetc(fp))!=EOF)
      {
      if(c=='\n')
            {
            t_nodes=t_nodes+1;
            }
      }

fclose(fp);
a=(st_node **)calloc(t_nodes+2*n,sizeof(st_node *));

for(i=0;i<t_nodes+2*n;i++)
      {
      a[i]=(st_node *)calloc(1,sizeof(st_node));
      a[i]->name=i;
      a[i]->left=-1;
      a[i]->right=-1;
      }

fp=fopen("input filename","r");
if(fp==NULL)
      {
      exit(0);
      }
while((c=fgetc(fp))!=EOF)
            {
             if(c=='\n')
                  lno++;
             if(c=='=')
                  flag=1;
             if(c=='!')
                  flag1=1;
             if((c=='[')&&(flag==1))
                  {


                        {i=0;
                        if(isdigit(c=fgetc(fp)))
                              {
```

```
				d[i++]=c;
				flag=3;

				}
			else
				flag=4;
			while((c=fgetc(fp))!=']')
				{
				if(isdigit(c))
					d[i++]=c;
				}
			val=d[0]-p;

			for(j=1;j<i;j++)
				val=val*10 + (d[j]-p);

			if(flag==3)
				{

				a[lno]->left=val;
				if(flag1==1)
					{
					exit(0);
					}
				}
			if(flag==4)
				{
				if(flag1==1)
					{

					a[lno]->left=2*val+t_nodes;
					flag1=0;
					}
				else
					a[lno]->left=2*val+t_nodes+1;

				}
			}

				flag=2;

		}
	if((c=='[')&&(flag==2))

		{


			{i=0;
			if(isdigit(c=fgetc(fp)))
				{
				d[i++]=c;
				flag=3;

				}
			else
				flag=4;
			while((c=fgetc(fp))!=']')
```

```
                                {
                                if(isdigit(c))
                                        d[i++]=c;
                                }
                        val=d[0]-p;

                        for(j=1;j<i;j++)
                                val=val*10 + (d[j]-p);

                        if(flag==3)
                                {

                                a[lno]->right=val;

                                }
                        if(flag==4)
                                {
                                if(flag1==1)
                                        {

                                        a[lno]->right=2*val+t_nodes;
                                        flag1=0;
                                        }
                                else
                                        a[lno]->right=2*val+t_nodes+1;

                                }
                        }

                flag=0;
                }
        }
return a;
}

/***********************************************************/

dagnode *creat_dagnode(int name)
                        /*allocates space for dagnode*/
{
dagnode *p;
p=(dagnode *)calloc(1,sizeof(dagnode));

if(p==NULL)
      {
      printf("no space available\n");
      exit(0);
      }
p->name=name;
p->lut=-1;
return p;
}


/**************************************************/

void insert_leftlink(st_node **a,int i,int j)
```

```
                                /*inserts link between a dagnode*/
{                               /*and its left fanin dagnode*/
if(a[i]->ptr==NULL)
      {
      printf("left link cannot be inserted due to non_existence of upper
node\n");
      exit(0);
      }
a[i]->ptr->left=a[j]->ptr;
a[j]->p_count=a[j]->p_count + 1;
}

/*************************************************************************/

void insert_rtlink(st_node **a,int i,int j)
                                      /*inserts link between  a dagnode */
{                                     /*and its right fanin dagnode*/
if(a[i]->ptr==NULL)
      {
      printf("right link cannot be inserted due to non_existence of upper
node\n");
      exit(0);
      }
a[i]->ptr->right=a[j]->ptr;
a[j]->p_count=a[j]->p_count + 1;
}


/*************************************************************************/

insert_uplink(st_node **a,int i,int j,int k)
                                /*inserts link between a dagnode and its */
{                               /*fanout nodes*/
a[j]->ptr->parent[(a[j]->t_nodes)++]=a[i]->ptr;
a[k]->ptr->parent[(a[k]->t_nodes)++]=a[i]->ptr;
}

/*************************************************************************/
st_node  **find_dag(st_node **a,int n)
                          /*this function forms the directed acyclic*/
{                         /*graph.insert_leftlink,insert_rtlink,insert_uplink */
int i,j;                  /*are called from this function*/

for(i=0;i<t_nodes+2*n;i++)
      {
      if(a[i]->ptr==NULL)
            a[i]->ptr=creat_dagnode(a[i]->name);
      }
for(i=0;i<t_nodes+2*n;i++)
      {
      if(a[i]->left>=0)
            insert_leftlink(a,i,a[i]->left);
      if(a[i]->right>=0)
            insert_rtlink(a,i,a[i]->right);
      }
for(i=0;i<t_nodes+2*n;i++)
      {
```

```
	if(a[i]->ptr!=NULL)
		a[i]->ptr->parent=(dagnode
**)calloc(a[i]->p_count,sizeof(dagnode *));


	}

for(i=0;i<t_nodes;i++)
	{

	if(a[i]->ptr!=NULL)
		{

			insert_uplink(a,i,a[i]->left,a[i]->right);

		}
	}
return a;
}



/**********************************************************************/
int read_level(st_node **a,int n)
{
int i,flag,c,val;
FILE *fp;
fp=fopen("input level filename","r");
	if(fp==NULL)
		{
		printf("\nerror\n");
		exit(0);
		}
	else
		printf("%x\n",fp);
for(i=t_nodes;i<t_nodes+2*n;i++)
	a[i]->ptr->label=0;
while((c=fgetc(fp))!='\n');
i=1;
while((c=fgetc(fp))!=EOF)
	{
	if(c==':')
		flag=1;
	if(c=='\n')
		{
		i++;
		flag=0;
		}
	if(flag==1)
		{
		if(c=='[')
			{
			val=0;
			while((c=fgetc(fp))!=']')
				{
				val=val*10+(c-'0');
				}
```

```
                    a[val]->ptr->label=i;
                    }
            }



      }
fclose(fp);

return i;
}

/******************************************************************/
dagnode **order_node(st_node **a,int n,int lev)
                  /*orders the dagnodes according to their levels*/
{
int i,j,p;
dagnode **head;
head=(dagnode **)calloc(t_nodes+2*n,sizeof(dagnode *));
p=0;
for(i=0;i<=lev;i++)
      for(j=0;j<t_nodes+2*n;j++)
            {
            if(a[j]->ptr->label==i)
                  head[p++]=a[j]->ptr;

            }

return head;
}

/*******************************************************************/
void mark_out(st_node **a,int m)            /*marks PO dagnodes*/
{
int i;
for(i=0;i<m;i++)
      {
      a[i]->ptr->if_out=1;
      }
}
/********************************************************************/
void mark_in(st_node **a,int n)             /*marks PI dagnodes*/
{
int i;
for(i=t_nodes;i<t_nodes+2*n;i++)
      a[i]->ptr->if_in=1;
}
/********************************************************************/
void insert_node(list_node *head,dagnode *b,st_node **a)
                                            /*inserts node in alist*/
{
int p,q;
list_node *tmp,*prev;
if(head==NULL)
      {
      printf("error in insertion\n");
      exit(0);
```

```
        }
tmp=head;
if(b->if_in==1)
        {
        p=b->name-t_nodes;

        q=p%2;
        p=p-q + t_nodes;
        b=a[p]->ptr;
        }
while(tmp!=NULL)
        {
        if(tmp->ptr==b)
                return;
        prev=tmp;
        tmp=tmp->next;
        }
prev->next=(list_node *)calloc(1,sizeof(list_node));
prev->next->ptr=b;
return;
}

/*****************************************************************/

list_node *input_list(list_node *head,dagnode *b,int p,st_node **a)
{
if(b->left==NULL)
        {
                insert_node(head,b,a);
        }
else
        {
        if(b->left->lut!=p)
                insert_node(head,b->left,a);
        if((b->left!=NULL)&&(b->left->lut==p))
                head=input_list(head,b->left,p,a);
        if(b->right->lut!=p)
                insert_node(head,b->right,a);
        if((b->left!=NULL)&&(b->right->lut==p))
                head=input_list(head,b->right,p,a);
        }
return head;
}

/***************************************************************/
int count_input(list_node *head)
                                /*counts the no. of nodes in a list*/
{
int i;
list_node *tmp;
tmp=head;
i=0;
while(tmp!=NULL)
        {
        i++;
        tmp=tmp->next;
        }
```

```
return i-1;
}

/*****************************************************************/
dagnode **level_dag(dagnode **head,st_node **a,int n,int k)
                                /*this fn label the nodes according */
{                               /* to the labelling phase of DAG-Map*/
int i,p,q;
list_node *l,*m;

for(i=0;i<2*n;i++)
      head[i]->lut=0;
for(i=2*n;i<2*n+t_nodes;i++)
      {
      if((head[i]->left->lut==-1)||(head[i]->right->lut==-1))
            {
            printf("error in level_dag\n");
            exit(0);
            }
      p=max(head[i]->left->lut,head[i]->right->lut);
      l=(list_node *)calloc(1,sizeof(list_node));
      l->ptr=(dagnode *)calloc(1,sizeof(dagnode));
      l->ptr->name=-1;
      q=count_input(input_list(l,head[i],p,a));
      if(q>k)
            {
            head[i]->lut=p+1;
            g_depth=max(g_depth,head[i]->lut);
            head[i]->left->tmp_out=1;
            head[i]->right->tmp_out=1;

            a[head[i]->name]->lptr=l;
            m=l->next=(list_node *)calloc(1,sizeof(list_node));
            m->ptr=head[i]->left;
            m->next=(list_node *)calloc(1,sizeof(list_node));
            m->next->ptr=head[i]->right;
            }
      else
            {

            head[i]->lut=p;
            g_depth=max(g_depth,head[i]->lut);
            a[head[i]->name]->lptr=input_list(l,head[i],p,a);
            }
      }
return head;
}


/******************************************************************/
void insert_list(list_node *head,list_node *new,st_node **a)
                         /*inserts a list of nodes in a list of nodes*/
{
while(new!=NULL)
      {
      if(new->ptr->name!=-1)
```

```
            insert_node(head,new->ptr,a);
      new=new->next;
      }
return;
}


/************************************************************************/
int not_only_pi(list_node *head)

                  /*checks if the list pointed by head contains*/
                  /*any node other than the PI nodes*/
{
list_node *tmp;
if (head->next==NULL)
      return -1;
tmp=head->next;
while(tmp!=NULL)
      {
      if((tmp->ptr->left!=NULL)||(tmp->ptr->right!=NULL))
            return 1;
      tmp=tmp->next;
      }
return 0;
}

/***************************************************************************
**/
void display_lut(list_node *head,dagnode *a)
                              /*displays a LUT i.e. its inputs and
output*/
{
list_node *tmp=head;
printf("output=%d\t",a->name);
printf("inputs are:  ");
while(tmp!=NULL)
      {
      if(tmp->ptr->name!=-1)
      printf(" %d\t",tmp->ptr->name);
      tmp=tmp->next;
      }
printf("\n");
/*printf("\n***************\n");*/
}

/*****************************************************************/
```

```
/*****************************************************************/
int remove(list_node *l,dagnode *p)
                                /*removes a dagnode from alist of
dagnodes*/
{
list_node *tmp,*prev;
tmp=l;
if(l->ptr==p)
      return -1;
      prev=tmp;
      tmp=tmp->next;
while(tmp!=NULL)
      {
      if(tmp->ptr==p)
            {
            prev->next=tmp->next;
            return 1;
            }
      prev=tmp;
      tmp=tmp->next;

      }
return 0;
}

/*****************************************************************/
list_node *apply_map(st_node **a,int m,int k)
                                /*this fn performs the mapping to LUTs*/
{                               /*this fn calls
insert_node,insert_list,remove,*/
int i,flag;                     /*not_only_pi,count_input*/
dagnode *p;
list_node *l,*tmp,*head,*prev,*per_head;
int ll=0;


head=(list_node *)calloc(1,sizeof(list_node));
head->ptr=(dagnode *)calloc(1,sizeof(dagnode));
head->ptr->name=-1;


for(i=0;i<m;i++)
      insert_node(head,a[i]->ptr);

per_head=head->next;
flag=0;
while(not_only_pi(head)==1)
      {
      p=head->next->ptr;
      if((p->left!=NULL)&&(p->right!=NULL))
            {
            if(p->if_out!=1)

                  if((p->left->tmp_out==1)&&(p->right->tmp_out==1))
                        {
                        tmp=per_head;
```

```
                    while(tmp!=NULL)
                        {
                        if(tmp->ptr==p)
                                break;
                        if(find(a[tmp->ptr->name]->lptr,p)==1)
                            if(count_input(a[tmp->ptr->name]-
>lptr)>=k-1)

                                flag=1;
                        tmp=tmp->next;
                        }

                    if (flag==0)
                        {
                        flag=2;
                        ll++;
                        tmp=per_head;
                        while(tmp!=NULL)
                            {
                            if(tmp->ptr==p)
                                break;
                            if(find(a[tmp->ptr->name]-
>lptr,p)==1)

                                {
                                insert_node(a[tmp->ptr-
>name]->lptr,p->left);

                                insert_node(a[tmp->ptr-
>name]->lptr,p->right);

                                if(remove(a[tmp->ptr->name]-
>lptr,p)==-1)

                                    a[tmp->ptr->name]-
>lptr=a[tmp->ptr->name]->lptr->next;

                                if(remove(per_head,p)==-1)
                                    per_head=per_head-

>next;

                                insert_node(per_head,p-
>left);

                                insert_node(per_head,p-
>right);

                                }
                            tmp=tmp->next;
                            }
                        }
                    }

            if(flag!=2)
                {
                l=a[p->name]->lptr;
                insert_list(head,l);
                }
            }
        head->next=head->next->next;
        flag=0;
        }
printf("\n\n\nll==%d\n\n\n",ll);
return per_head;

}
```

```
/*********************************************************************/
main()
{
int n,m,i,p,k,ll;
st_node **a;
dagnode **head;
list_node *tmp,*tmp1,*q;
printf("give the total no of input variables and output variables\n");
scanf("%d %d %d",&n,&m,&k);
printf("%d  %d  %d\n",n,m,k);
a=read_input(n);
a=find_dag(a,n);
display_dag(a,n);
p=read_level(a,n);
head=order_node(a,n,p);
mark_in(a,n);
mark_out(a,m);
head=level_dag(head,a,n,k);
tmp=apply_map(a,m,k);
show_dag(a,tmp);
calculate_slack(a,m,n,p-1);
display1(a,n);
}
```

```
/*****************************************************************/
/******Source code for algorithm 2 ****************************/
/*The code for this algorithm is same as that for algorithm 1*/
/*except the routine apply-map. apply_map routine for*********/
/* algorithm 2 is given below **********************************/
/*****************************************************************/
list_node *apply_map(st_node **a,int m,int k)
                              /*this fn performs the mapping to LUTs*/
{                             /*this fn calls
                              insert_node,insert_list,remove,*/
int i,flag;                   /*not_only_pi,count_input*/
dagnode *p;
list_node *l,*q,*tmp,*head,*prev,*per_head;
int ll=0,mm=0;


head=(list_node *)calloc(1,sizeof(list_node));
head->ptr=(dagnode *)calloc(1,sizeof(dagnode));
head->ptr->name=-1;


for(i=0;i<m;i++)
      insert_node(head,a[i]->ptr,a);

per_head=head->next;
flag=0;
while(not_only_pi(head)==1)
      {
      p=head->next->ptr;

      if((p->left!=NULL)&&(p->right!=NULL))
            {a[p->name]->if_lut=1;
            tmp=q=a[p->name]->lptr;
             while(tmp!=NULL)
                  {
                  if(tmp->ptr->left!=NULL)
                        {
                        if(a[tmp->ptr->name]->lptr==NULL)
                              printf("error in packing\n");
                        /*if(count_input(q)<k)
                              mm++;*/
                        if(a[tmp->ptr->name]->p_count==1)
                        {

if(count_input(q)+count_input(a[tmp->ptr->name]->lptr)<=k+1)
                              {
                              printf("%d this node is packed with
%d\n",tmp->ptr->name,p->name);
                              ll++;
                              flag=2;

insert_list(q,a[tmp->ptr->name]->lptr,a);
                              if(remove(q,tmp->ptr)==-1)
                                    q=q->next;
                              insert_list(per_head,q,a);
                              if(remove(per_head,tmp->ptr)==-1)
                                    per_head=per_head->next;
```

```
                                        break;
                                        }
                                }
                                else
                                        mm++;
                                }

                        tmp=tmp->next;
                        }

                if(flag!=2)
                        {
                        l=a[p->name]->lptr;
                        insert_list(per_head,l,a);
                        }
                }
        head->next=head->next->next;
        flag=0;
        }
printf("\n\n\nll==%d  \n\n\n",ll);
return per_head;
}

/*************************************************************************/
```