Synthesis Methods for Field Programmable Gate Arrays

ALBERTO SANGIOVANNI-VINCENTELLI, FELLOW, IEEE, ABBAS EL GAMAL, SENIOR MEMBER, IEEE, AND JONATHAN ROSE, MEMBER, IEEE

Invited Paper

Field programmable gate arrays (FPGA's) reduce the turnaround time of application-specific integrated circuits from weeks to minutes. However, the high complexity of their architectures makes manual mapping of designs time consuming and error prone thereby offsetting any turnaround advantage. Consequently, effective design automation tools are needed to reduce design time. Among the most important is logic synthesis. While standard synthesis techniques could be used for FPGA's, the quality of the synthesized designs is often unacceptable. As a result, much recent work has been devoted to developing logic synthesis tools targeted to different FPGA architectures. The paper surveys this work. The three most popular types of FPGA architectures are considered, namely those using logic blocks based on lookup-tables, multiplexers and wide AND/OR arrays. The emphasis is on tools which attempt to minimize the area of the combinational logic part of a design since little work has been done on optimizing performance or routability, or on synthesis of the sequential part of a design. The different tools surveyed are compared using a suite of benchmark designs.

I. Introduction

Synthesis tools that automatically map a design composed of simple gates or described with a hardware description language (HDL) into gates from a given library are becoming widely used. Besides simplifying the design process and reducing design time, these tools have had a major impact on the design methodology for application-specific integrated circuits (ASIC's), allowing designers to select easily among different implementation options, such as between a standard cell and a mask programmable gate array (MPGA) or among different ASIC vendors, based on accurate estimates of performance and area,.

Manuscript received March 23, 1993.

A. El Gamal is with the Depratment of Electrical Engineering, Stanford University, Stanford, CA 94305.

J. Rose is with the Department of Electrical Engineering, University of Toronto, 10 King's College Road, Toronto, Ontario M5S 1A4, Canada.

A. Sangiovanni-Vincentelli is with the Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720.

IEEE Log Number 9210743.

The complexity of field programmable gate array (FPGA) architectures makes manual mapping of designs too difficult and time consuming. Indeed the reduction in turnaround time due to the user programmability of an FPGA may be offset by the time spent to map a design manually. As a result much work has been focused recently on developing synthesis tools targeted to different FPGA architectures. Such tools are now yielding good results and becoming commercially viable.

The most straightforward approach to synthesis for FPGA's is to adapt the synthesis tools developed for MPGA libraries to FPGA's. A design is first mapped into simple gates (such as two input NAND gates), and groups of simple gates are then replaced by logic blocks of the target FPGA. This approach works well for FPGA's with fine-grain blocks such as those from Algotronix, Concurrent Logic, Plessey and Toshiba, since a fine-grain block can only implement one or two simple gates. However, for the more widely used FPGA's with coarse-grain logic blocks such as those from Actel, Altera, and Xilinx, this approach does not in general yield acceptable results. A more promising but challenging approach is to map the design directly into logic blocks. Recently developed FPGA synthesis tools employ both the library mapping approach as well as the direct mapping approach.

In this paper we review the recently developed methods for FPGA synthesis. Even though there is much interest in sequential synthesis for FPGA's, no paper dealing with this topic has been published to date (we are aware of some work that has been submitted for publication [36], [51]). Moreover, most of the developed methods optimize area of a design and only a few optimize performance explicitly. We, therefore, focus our review on combinational synthesis for FPGA's where the registers in the design are explicitly specified by the designer and devote most of the discussion to synthesis methods that optimize area. Since fine-grain FPGA's do not present new challenges to synthesis algorithms we only describe work on synthesis

for coarse-grain FPGA's such as those from Actel, Altera, and Xilinx,

The paper is organized as follows. Basic definitions are given in Section II. In Section III we review the most effective of the known logic minimization and synthesis methods. In Section IV we review several approaches to logic synthesis for FPGA's with "look-up table" logic blocks such as Xilinx's. In Section V we present a similar discussion but for FPGA's with multiplexer-based logic block such as Actel's. In Section VI we briefly discuss synthesis for FPGA's with PLA-based logic blocks.

II. Basic Definitions

A *logic* or *Boolean variable* x takes on one of two values 0 and 1. Denote by x' the complement of the variable x. Both x and x' are referred to as *literals*.

A *Boolean function* $f : \{0,1\}^n \rightarrow \{0,1\}$ is a binary function of logic variables. It is often convenient to represent the n-dimensional Boolean space by an n-dimensional Boolean hypercube. A Boolean hypercube of dimension n contains 2^n vertices. The set of vertices of the hypercube where the function takes on the value 1 is referred to as the *on-set* and the set of vertices where the function takes on the value 0 is referred to as the *off-set*. At times the value of a logic function is not specified for a set of the vertices. In this case, the function is said to be *incompletely specified* and the unspecified set of vertices is referred to as the *don't-care-set* or *dc-set*. The rest of the vertices (i.e., the on-set and the off-set) constitute the *care-set*. The set of inputs on which a function is explicitly defined is referred to as its *support*. In the remainder of the paper we refer to an incompletely specified logic function simply as a logic function unless otherwise stated.

A *cube* of a logic function f is a logic function given by the product of literals whose on-set does not have vertices in the off-set of f. The origin of this name rests on the fact that a product of k literals corresponds to a Boolean hypercube of dimension $n - k$ in the Boolean space of dimension n. A *minterm* is a cube where all the variables are assigned a value 0 or 1. This cube is of dimension 0, and contains only one vertex.

The *Shannon cofactor* or simply the *cofactor* of a logic function f with respect to a variable x, denoted by f_x, is the logic function obtained from f by setting the variable x to the constant value 1. The cofactor of f with respect to x', denoted by $f_{x'}$, is the logic function obtained by setting the variable x in f to the constant value 0.

A logic function has several *representations*, e.g., the set of its minterms (which is equivalent to the truth table representation), the *sum-of-product form*, the *factored form* and the *Binary Decision Diagram*.

A sum-of-product expression for f is a set of cubes that contains all the vertices of the on-set of f and none of the off-set.

A factored form is defined recursively as follows:

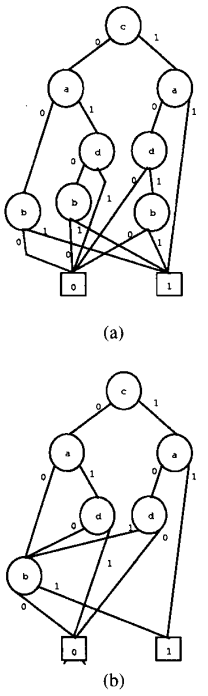
- a literal is a factored form;
- the sum of factored forms is a factored form;
- the product of factored forms is a factored form.

Thus for example, $a+b$, $(a+b)(c+(e'(f+g')))$, where e', g' denote the complement of the variables e, g, are factored forms.

An important characteristic of factored forms is that they may be thought of as representing both a function and its complement, since, by De Morgan's laws, the factored form of the complement of a function can be simply obtained from the factored form of the function by interchanging the logic addition and the logic product operations as well as the phases of the variables. Note that in contrast the sum-of-products form of the complement of a function can be drastically different from the sum-of-product form of the function.

A binary decision diagram (BDD) is a simple yet efficient representation of a completely specified logic function. BDD's were proposed many years ago by Akers but their use in logic manipulations has only recently been made practical and effective by Bryant [10]. A BDD is a directed acyclic graph (DAG) where a logic function is associated with each node. The completely specified logic function f represented by the BDD is associated with the root node. Every node has two fan-out nodes representing the function obtained by cofactoring the logic function represented at the node with respect to a variable and its complement. This variable indexes the node. Let x be the variable indexing node i and f_i the function associated with this node. The *high-node* corresponds to the cofactor f_{ix}, and the *low-node* corresponds to $f_{ix'}$. The leaf nodes are the constant functions 0 and 1. Note that this representation has an exponential number of nodes and is canonical in the sense that given a logic function and an ordering of the variables corresponding to the sequence of cofactoring operations along a path from the root to the leaf nodes, the representation is *unique*. In fact this representation is equivalent to the truth table representation of the function. As such it is not too interesting. However, if the nodes associated with the same logic function are merged, the complexity of the representation can be reduced. The resulting BDD is referred to as reduced BDD or RBDD. The number of nodes in an RBDD can be dramatically lower than for the unreduced BDD. This fact makes RBDD's quite appealing for a number of applications. A further useful simplification of RBDD's, proposed by Bryant, is to choose the ordering of the variable for all paths from the root to the leaf nodes to be the same. This representation is referred to as the reduced ordered BDD (ROBDD) and is canonical. Figure 1 (a) shows an ordered BDD for the function $f = ac + a'bd + bc'd'$ with the order c, a, d and b. The root node is indexed by c. Now, we reduce it by seeing that all nodes indexed by b represent the same function, namely b. We merge them all in one node, and get an ROBDD in Fig. 1(b).

Many operations on this representation are linear in the size of the graph. In addition, verifying whether two logic functions are logically equivalent, amounts to an easy isomorphism check on their ROBDD's, which can be carried out efficiently. Although most functions have an ROBDD representation that is still exponential in the number of



(a)

(b)

Fig. 1. Example of a BDD and an RBDD of a function.

variables, many functions appearing in practice have a low complexity ROBDD representation. The complexity of an ROBDD representation is, however, dependent on variable ordering and finding a good ordering is in general not tractable.

The *if-then-else DAG* representation is a close relative of the BDD. The *if-then-else* DAG is a set of nodes each with three children: each node is a two-to-one selector where the first child is connected to the control input of the selector and the second and third children are connected to the signal inputs of the selector. The behavior function of the node is that if the expression that corresponds to the control input is TRUE $then$ the second child is selected $else$ the third child is selected. In the case of the BDD, the nodes can be regarded as two-to-one selectors as well but with the control input connected directly to the variable associated with the node. Thus an *if-then-else* DAG is more general than a BDD and consequently can yield more compact representations.

An advantage of the *if-then-else DAG* over BDD's appears when converting from a sum-of-products form. Select one variable, say v_1. Let the cubes of the function associated with a node of the Boolean network be partitioned into three sets C_1, C_2, C_3 with respect to v_1: C_1 corresponding to the cubes that do not depend on the selected variable v_1, C_2 containing all the cubes that depend on v_1, and C_3 containing all the cubes that depend on v_1', the complement of v_1. The corresponding *if-then-else DAG* implements *if* C_1 *then TRUE, else (if* v_1 *then* C_2 *else* C_3*)* and contains two nodes as shown in Fig. 2. The first node has the function determined by the cubes in C_1 connected to its control input, the constant TRUE connected to its second child,

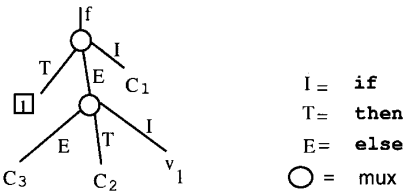


Fig. 2. Realization of a function with ITE's.

and the output of the second node connected to its third child. The second node has its control input connected to the variable v_1, its second child to the cubes in C_2 and its third child to the cubes in C_3. Note that this may be a smaller representation than the BDD for the same function since the expressions connected to the high and low children of the BDD node contain duplicate cubes (the ones that are in C_1). In the *if-then-else* DAG these cubes appear only once.

III. Logic Synthesis

A. Introduction

There are several approaches to logic optimization [9]. The most commonly used approach is to break the synthesis process into two phases: a **technology independent phase**, followed by a **technology mapping phase**. The technology independent phase attempts to generate an optimal *abstract representation* of the logic circuit. The technology mapping phase selects a set of gates from a library[1] to implement the abstract representation while optimizing area, delay or a combination of the two.

For combinational logic, the abstract representation chosen in MIS [8] and in many other university and industrial tools, is the *Boolean network*, a directed acyclic graph $G(V, E)$ where each of the nodes $v \in V$ represents an arbitrarily complex single-output logic function. There is an arc from node j to node i if the function represented by node i depends explicitly on the function represented by j. Node j is said to be a *fan-in* of node i and node i is said to be a *fan-out* of node j. There are two sets of special nodes: input nodes with no incoming arcs which represent *primary inputs*, and output nodes with no outgoing arcs which represent *primary outputs*. An example of a Boolean network is shown in Fig. 3. The network has four primary inputs a, b, c and d, and one primary output z.

Each node of the network may represent an arbitrary logic function (*general node*) or a simple logic function such as a two-input NAND or NOR (*generic node*). The *support* of a node is the set of variables that the corresponding logic function explicitly depends on. During optimization, the nodes of the network may be mapped from a general form to a generic form as will be seen later. A general node can be represented in a sum-of-products form, a factored form, or as a BDD.

Node representation may change from one form to another according to the operations performed. The sum-of-

[1] A library can be given either explicitly as a list of gates, or implicitly with equations or other means of representing a class of logic functions.

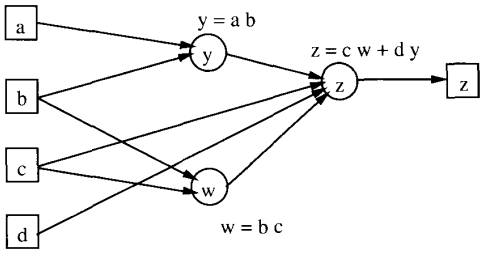


Fig. 3. A Boolean network.

products form is convenient in node minimization where a two-level logic minimizer (e.g., Espresso [7]) is used. The factored form representation is useful since it corresponds to a possible implementation of the function in dynamic CMOS logic where each literal corresponds to a transistor [6]. Moreover, when static CMOS logic is used there is a correspondence between the number of literals in an optimized factored form and the area occupied by its physical implementation. As a result the total number of literals in an optimized factored form is the most commonly used cost function in logic minimization.

The problem of finding an optimum factored form for a given logic function is, however, very complex and exact algorithms are not practical for functions of more than six variables. Heuristics are, therefore, used to compute an optimized factored form. Moreover, minimizing the number of literals does not explicitly consider wiring area which is particularly important for FPGA synthesis. This represents a major challenge in adapting existing and well-proven synthesis approaches to FPGA's.

B. Technology Independent Optimization

The operations performed in the technology independent phase are classified into two classes: network restructuring operations and node minimization. The former includes operations that modify the structure of the Boolean network by introducing new nodes, eliminating others, and by adding and removing arcs, while the latter includes operations that simplify the logic equations associated with nodes [9].

1) Restructuring Operations: Network restructuring operations include decomposition, extraction, factoring, resubstitution, and collapsing.

- *Decomposition* is the process of expressing a given logic function in terms of a number of new functions. For example, let

$$F = abcef + abdef + a'c'd' + b'c'd' ; \quad (1)$$

then a decomposition of F is

$$F = XYef + X'Y' ; \quad (2)$$

where $X = ab$ and $Y = c + d$.
Note that while the expression representing F before decomposition depends explicitly on six variables, the one after decomposition depends explicitly on four variables only. Decomposition is an essential step in logic optimization for FPGA's.

- *Extraction* is related to decomposition but operates on a number of given functions. With extraction, the given functions are expressed in terms of newly created intermediate functions and variables. For example, extraction when applied to the following functions

$$F = (a'b + ab')cd \quad (3)$$
$$G = (a'b' + ab) + e + f \quad (4)$$

gives

$$F = Xcd \quad (5)$$
$$G = X' + e + f \quad (6)$$
$$X = a'b + ab' \quad (7)$$

Common subexpressions are identified and extracted in order to minimize the total number of literals by sharing expressions among logic functions. However, the number of arcs in the resulting Boolean network increases which may increase wiring area.
- *Factoring* transforms the sum-of-products form of a logic function into a factored form. For example, F of Eq. (1) can be factored as $abef(c+d) + (ab)'(c+d)'$.
- *Substitution* or *resubstitution* is the process of expressing a given logic function F in terms of another given function G. For example, let $G = abc$ then $F = Gef + abdef + (G + abd)'$.
- *Collapsing*, also called *elimination* or *flattening*, is the inverse operation of substitution. If G is a fan-in node of F, collapsing "pushes" G into F so that F is expressed only in terms of its fan-in nodes which also include the fan-in nodes of G.

All these operations make use of operations analogous to conventional multiplication and division. In fact, decomposition, extraction and factoring depend on finding subexpressions which are "divisors" or "factors" of the representation of the function. The number of divisors and factors of a given Boolean expression, however, can be so large that it is practically impossible to search the space to find one which is optimum with respect to the cost function used in the logic synthesis. As a result in most logic synthesis systems divisors and factors are selected from a restricted space so that the search is much faster and the quality of the result is acceptable.

2) Algebraic Operations: The restricted space is the space of *algebraic expressions*. An algebraic expression is a set of cubes such that no cube contains another, i.e., no cube contains all of the vertices of any other cube. A *Boolean product of two cubes* is the product of the literals of the cubes if no literal appears complemented in one cube and uncomplemented in the other and is zero otherwise. The *product of two expressions* is the set of products of the cubes of the two expressions. A product of two expressions is an *algebraic product* if they are algebraic expressions and if the two expressions have no input variables in common. The basic task in decomposition, extraction, factoring and resubstitution is the operation of *division*: given two functions $FandP$, find Q and R such

that $F = PQ + R$. The *division is algebraic* if PQ is an algebraic product.

Algebraic division can be carried out very quickly. An algorithm exists which can compute the operation in linear time in the number of cubes in the expressions. To perform an effective restructuring of the network with decomposition, factoring and extraction, it remains to find an effective procedure to determine good algebraic divisors, i.e., given F, we wish to find P so that P, Q and R can be expressed with the smallest number of literals. Since the number of divisors is very large, the optimization problem looks hopelessly complex. Kernels, introduced by Brayton and McMullen, are a subset of all algebraic divisors of an expression that can be computed effectively with a number of fast algorithms. It can be proven that optimum algebraic divisors and common factors must be kernels and/or kernel intersections. In MIS there are a number of kerneling operations with different speed-quality trade-offs.

Thus the restructuring operations can be performed quickly and the space searched effectively, but at the expense of the optimality of the solution. Boolean operations such as node minimization can be interspersed with algebraic operations in an attempt to find a better solution.

3) Node Minimization: Node minimization attempts to reduce the complexity of a given network by using Boolean minimization techniques on its nodes. The nodes of the network are Boolean functions that can be minimized using two-level techniques such as the ones used in Espresso. However, considering the functions at the nodes as independent, much optimization is potentially lost. In fact, the inputs of the Boolean functions are related to each other by the nodes of the network that precede the node under consideration and hence are not free to take any combination of values. In addition, for some values of the primary inputs of the network, the output of the node may not be observable at the primary outputs of the network. In both cases the values of the inputs that can never occur at the input of the function and the values of the primary inputs for which the outputs of the nodes are not observable at the primary outputs of the network are *don't cares* for the two-level minimization of the node. The first kind of don't cares is called *Satisfiability Don't Care (SDC)* set, while the second is called *Observability Don't Care (ODC)* set.

An example of SDC is as follows. If node i of the network carries the Boolean function $f(x, y)$, where $x = a + b$, $y = ab + c$ and a, b, c are primary inputs of the network, then $x(a + b)' + x'(a + b)$ and $y(ab + c)' + y'(ab + c)$ are SDC's. In other words, the SDCs represent combinations of variables of the Boolean network that can never occur because of the structure of the network itself.

Unfortunately the SDC's and the ODC's may be very large and it may be impossible to compute them. Hence node minimization in [8] optimizes the two-level representation of a node using a suitably chosen subset of SDC's and ODC's when they are too big.

Another method for node minimization, [4] does not use two-level minimization techniques with don't cares, but rather it simplifies the node function using a *tautology checker.* Tautology checking determines whether a function is identically equal to 1. It can also be used to determine if two Boolean networks are equivalent by taking the corresponding primary outputs and forming their exclusive NOR. If the two Boolean networks are equivalent, the output of the exclusive NOR will be always 1. In [4], a node is tentatively simplified by deleting either literals or cubes from the node representation. The resulting network is checked for equivalence against the original network. If equivalent, the deletion is performed and a simpler representation is obtained. The problem with this method is CPU time since many equivalence checkings need to be performed. On the other hand the previous approach suffers from problems stemming from the size of the SDC and ODC. In most available logic optimization programs, the first minimization technique is adopted using an approximation to the SDC and ODC.

Node minimization has been proven to be very effective for a wide variety of cases. Node minimization is very often the only Boolean operation that is performed during a network optimization run.

C. Technology Mapping

After optimizing the network, the technology mapping phase begins. Here the optimized Boolean network is *mapped* into a network whose nodes are primitive logic functions implemented by the available library gates. In this phase the cost function can be more accurate since the area of the primitive gates is known exactly. However, wiring area is not used as part of the cost function in most of the synthesis systems in use today, even though approaches have been proposed that take wiring into account [1], [40], [41].

The algorithms that are used in technology mapping fall into two main categories:

1. algorithmic approaches (e.g., [29], [8], [31]);
2. rule-based techniques (e.g., [13], [24]).

In the first approach, the Boolean network is mapped into a *subject graph* which is a network consisting of two-input NAND gates. All the gates in the library are also expressed as networks (called *pattern graphs*) in terms of two-input NAND gates, thus yielding a consistent representation between the network and the gates in the library. The problem is now transformed into a covering problem: find the minimum cost cover of the subject graph by the pattern graphs. Since both the subject graph and the pattern graphs are directed acyclic graphs (DAG's), the problem is called *DAG covering by DAG's.* Unfortunately the problem is NP-hard, and since there is no exact algorithm that yields practical results even for relatively small networks [44], heuristics are used.

The first heuristic to be proposed [29] was inspired by the work on optimizing compilers by Aho *et al.* [2]. This heuristic is optimal if the network to be mapped is a tree and the library gates are represented by trees. However, in

general, the optimized Boolean network is not a tree. For this reason, the network is decomposed into trees. Since most of the gates in widely available commercial libraries can be expressed in terms of trees of two-input NAND gates the mapping problem is transformed into a tree-covering-by-trees problem which is easily solved by covering each of the trees separately. This is an efficient heuristic since it is based on proven optimality properties, the running time of the procedure is linear in the size of the trees, and the quality of the results are quite good.

An alternative approach was proposed in [31]. The two-input NAND-gate network is decomposed into subnetworks that are not necessarily trees; the only requirement in common with tree decomposition is that the connection to the rest of the circuit or to a primary output be a node of fanout one (*the sink node*). A dynamic programming approach is used to find the optimum matching of the subnetworks in terms of a given set of primitives (library gates). In this approach, Boolean operations are used to find whether a subnetwork is logically equivalent to one of the library functions (*Boolean matching*). First, a set of *cluster functions* is defined as the set of functions that correspond to connected subgraphs of the subnetwork rooted in the sink node. The leaves of these DAG's are the support variables of the cluster functions. The multilevel structure of the subgraphs is flattened obtaining a two-level representation of the cluster function. The cluster function is then checked against all the library gates to identify those gates that are logically equivalent on the care set of the cluster function. This is done by solving a tautology problem, i.e., the exclusive NOR of the cluster function and of the library gate is taken and checked to determine whether the output of the exclusive NOR is identically equal to one on the care set of the cluster function. The minimum cost match is selected and the procedure is repeated for all the functions which are rooted in the nodes that define the support variables for the cluster function. This defines the basic step for the dynamic programming procedure.

Among the advantages that can be claimed for this approach, we identify:

- the decomposition of the subject graph is not restricted to be a forest of trees;
- don't cares can be naturally incorporated to obtain matches that could not have been obtained with a purely structural approach such as the tree-covering-by-trees approach.

These advantages did not offer substantial improvements over the tree-covering approach when applied to standard libraries on a set of benchmarks. However, as we shall see in Section V-C5), better results were achieved for libraries containing XOR's, multiplexers and majority functions that are notoriously difficult to handle with the tree approach.

A drawback of this approach is the high computational requirement; each match attempt requires the solution of a tautology problem. In [18] and [47] clever methods have been proposed to minimize the number of tautology operations performed.

In both approaches, the original DAG has to be mapped into a network of two-input NAND gates. Note that there is potentially a very large number of possible mappings of the original network in terms of two-input NAND gates. Simple heuristics are used to preserve as much of the structure obtained during the technology independent optimization step as possible, while using a small number of NAND gates. The library gates can also have different representations in terms of two-input NAND gates. However, the number of possible two-input NAND gate representations is rather small in most cases. In the tree-covering-by-trees approach, all possible representations of a given gate in terms of two-input NAND gates are enumerated, thus providing a larger number of matches between the covering trees and the tree to be covered. One limitation of this approach is that it can only be applied to single-output cells. No work has so far been done to address mapping for cells with multiple outputs.

Rule-based techniques traverse the Boolean network and replace subnetworks with patterns representing the gates in the library that match the function of the subnetwork. Rule-based techniques are slower but could yield better final results since detailed information about the gates in the libraries can be captured, and electrical considerations can be taken into account easily.

The present trend in industry is to use a mixed approach, where a tree covering approach is followed by a rule-based clean-up phase.

Timing optimization is carried out using the same approaches but with more difficulty. In the technology independent optimization phase some simple timing model of the network based on the number of levels and the degree of each node can be used to restructure the network to minimize the critical path [49]. In the technology mapping phase, gate delays are known with good approximation and the mapping can be guided to yield a fast implementation.

IV. Synthesis for LUT-Based FPGAs

A. Introduction

LUT-based logic blocks such as the Xilinx configurable logic block (CLB) can implement any logic function of no more than a fixed number of variables. Additional functions can also be implemented depending on the details of the block. For example, the LUT section of the Xilinx series 3000 architecture (Fig. 4) can implement any logic function F with up to five inputs a, b, c, d, e, or any two logic functions F and G with up to four inputs each and five overall variables. In addition, each block has two embedded flip-flops with outputs QX and QY for use in sequential design.[2]

All existing approaches to synthesis for LUT-based FPGA's begin with a network that has been optimized using a technology independent method and, hence, could be

[2] If the internal flip-flops and the feedback paths from them are considered, the Xilinx 3000 architecture allows up to a total of seven different inputs to the two look-up tables.

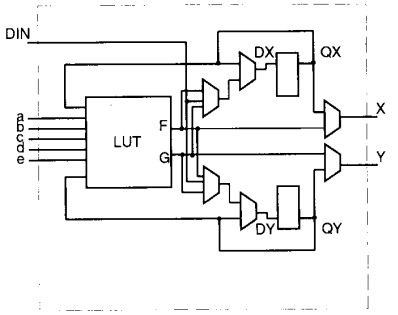


Fig. 4. A CLB of Xilinx 3000.

classified as technology mapping even though some drastic restructuring of the network could result during synthesis.

This section is organized as follows. The most straightforward adaptation of technology mapping approaches to LUT FPGA's is reviewed first. Special algorithms are then presented which take into account practical LUT-based FPGA architectures. Although most approaches published to date deal with area minimization, new techniques which optimize performance are surfacing. These new techniques are reviewed in the last sub-section.

B. Library-Based Technology Mapping

In the tree-covering-by-trees approach to technology mapping, the gates in the library have to be expressed as trees. To use this approach an LUT is viewed as a *collection* of gates. For example in MIS, all the nonequivalent functions[3] are explicitly described in terms of two-input NAND gates. While the nonequivalent gates are fewer than all the possible gates, their number still grows superexponentially. For $k = 2, 3$ the number of nonequivalent functions is reasonable (10 and 78 respectively), but already for $k = 4$ the number of nonequivalent functions is 9014 [19]. In addition some of these functions have a large number of possible two-input NAND gate representations (some have more than 700) and MIS cannot handle the resulting complexity in the library. Thus the number of logic gates represented by the covering trees is restricted. In [19], only a relatively small subset of the functions was included in the library. The subset was selected based on the observation of the behavior of the algorithm for $k = 3$ and the knowledge of the inner operations of MIS. Note that the cost of mapping into any of these functions is constant since all of them can be implemented by a single LUT.

Even after restricting the set of gates to be included, the time needed to perform the mapping is long and is dominated by the time needed to parse and process the library. In [19] it was observed that as k increases, the quality of results deteriorates (not surprisingly since the number of basic functions eliminated from considerations grows quickly). Thus this approach seems inadequate.

[3] An LUT with k inputs can implement 2^{2^k} functions. A function f is equivalent to another g if it can be obtained from g by renaming inputs.

C. Direct Approaches

Direct approaches deal with the functionality of the logic block directly and do not require the explicit construction of a library of gates.

Two direct approaches have been considered:

- Modification of the tree-covering-by-trees algorithm for technology mapping to significantly reduce the CPU time required by the standard technology mapping algorithms [19, 20];
- A two-step approach where:
  - Starting with a technology-independent-optimized network, the nodes of the network are decomposed so that each depends on no more than k variables. The decomposition operation yields a network that is *feasible* since each node can now be implemented directly using a single LUT.
  - The number of nodes is reduced by combining some of them taking into account the particular features of the LUT's [37], [18], [28], [38], [50].

1) Modifying the Tree-Covering Approach:Chortle [19] and its extension *Chortle-crf* [20] use the first direct approach to the technology mapping problem for LUT's. Chortle begins with an AND/OR representation of the optimized Boolean network. This representation is obtained in a straightforward way from the sum-of-products representation of MIS by representing each product and each sum as a separate node. Inversions are represented by labels on the edges.

The network is first decomposed into a forest of trees by clipping the multiple-fan-out nodes. An optimal mapping of each tree into LUT's is then performed using dynamic programming, and the resulting implementations are assembled together according to the interconnection patterns of the forest. These steps are essentially the same as the standard technology mapping algorithm implemented in DAGON and MIS. The main difference is in the way the optimal mapping is done. Note that in the case of LUT's it is not the structure of the logic function that matters in the matching but only the number of variables that the function depends on: given a tree, every subtree that has at most k leaf nodes can be implemented by a single LUT.

Chortle and Dynamic Programming for LUT's: The dynamic programming approach to technology mapping is as follows. The minimum cost implementation of a tree rooted at node i is obtained as the implementation of the subtree T_i rooted at i combined with the minimum cost implementation of the subtrees rooted at the leaf nodes of T_i which yields the minimum overall cost among all such implementations. Thus the optimum technology mapping problem for a tree can be solved recursively starting at its leaf nodes and working towards its root.

In the case of LUT's, when the mapping extends towards the root of the tree, all subtrees rooted at a node that have a number of leaf nodes less than or equal to k must be

considered to make sure that all applicable solutions are searched. Note that all these subtrees have the same cost of 1.

In line with the technology mapping algorithms of DAGON and MIS, Chortle's approach guarantees that an optimum solution is found for every tree but cannot guarantee that the technology mapping for the entire network is optimum.

When i has degree significantly larger than k, the number of subtrees to examine is very large. Since all possible combinations of nodes connected to i of cardinality less than or equal to k must be considered to guarantee an optimal solution, Chortle would spend an inordinate amount of time searching the space of subtrees.[4]

To avoid the explosion of CPU time, Chortle predecomposes the nodes of the network that have degree larger than a limit $l, l > k$. This is done by splitting the nodes into two nodes with nearly the same degree. By doing this the optimality of the solution is not guaranteed any longer but according to [19] the quality of the final solution is hardly affected.

Several factors limit the quality of the solution, however:

- the search for a mapping is artificially limited to the tree boundaries;
- possible duplication of nodes in the network is not considered;
- some of the special features of the LUT-based FPGA's are not considered, e.g.. the fact that two functions can be mapped onto the same LUT in the Xilinx array.

Chortle-crf and Bin-Packing: Chortle-crf [20] extends Chortle by considering node duplication and reconvergent fan-outs. In addition, a key contribution of this work is recognizing that the decomposition problem for an LUT-based FPGA could be approximated as a simple variant of the *bin-packing problem* [23]. The bin-packing problem is to pack a set of objects of given sizes into the minimum number of bins of fixed capacity. The bin-packing problem is NP-hard but simple and very fast heuristics have been used effectively for its solution [23]. Furthermore, these heuristics can be guaranteed to find the optimum solution in some special cases, and are in any case within 22% of the optimum.

Bin-packing heuristics are used in Chortle when the best solution to the mapping problem is sought for a node in a tree during dynamic programming. A two-level representation of a logic function f is considered in this case. The cubes are the set of objects to be packed. The size of an object is given by the number of variables that appear in the corresponding cube. Any set of cubes whose overall size is less than or equal to k is packed into an LUT. In case a cube contains more than k variables, it is considered as a combination of two or more cubes each of which has less than k variables. Note that an LUT implements the OR of the cubes packed into it. Finding the minimum number of bins which contain all the cubes is equivalent to solving the bin-packing problem, but does not yield a solution to the decomposition problem. For example, given the function $F = abcd + efg + hi$ if we pack $abcd$ into one LUT, and the remaining cubes into another, we would still have to build the OR of the two subfunctions $abcd$ and $efg + hi$ to implement the original function resulting in a three LUT implementation. If instead we replace $efg+hi$ with a single literal cube z and we pack z together with $abcd$, we obtain the following decomposition of F:

$$F = abcd + z; \tag{8}$$

$$z = efg + hi \tag{9}$$

The function can now be implemented using only two LUT's; one LUT implementing $z = efg + hi$ feeding into a second LUT implementing $F = z + abcd$.

Chortle uses the first fit decreasing algorithm to solve the resulting bin-packing problem. The algorithm selects the largest object, i.e., the cube with the largest number of variables, and finds the first bin (LUT) where it fits. If no existing bin (LUT) has enough capacity a new bin is created and the cube is placed there (recall that this can always be done, since all cubes have a number of variables that is at most k). When all the cubes have been placed in an LUT, the LUT with the fewest unused inputs is selected and closed. A new variable is created and the corresponding one-variable cube is placed in the first LUT where it can be accommodated. If none is found a new one is created. The procedure is repeated until only one LUT remains open. This last LUT is closed but no new variable is created. This last LUT is the one that provides the output corresponding to the original function.

This algorithm has a remarkable property. It can be proved that if the cubes of the given function are disjoint, i.e., they have disjoint support, then the algorithm generates a tree of LUT's of minimum size that implements the given function for $k \leq 6$ [20], [38].

In Chortle-crf, the algorithm is applied to a tree of AND and OR functions. Hence all the cubes indeed have disjoint support and the solution to the technology mapping problem is optimum as in Chortle but it can be obtained much more quickly because of the speed of the packing algorithm (experimentally it has been observed to run up to 28 times faster than the Chortle algorithm).

The speed of the bin-packing algorithm is the key to addressing two shortcomings of the original Chortle approach, namely optimization across tree boundaries and duplication of logic. The results obtained by Chortle can be improved if local reconvergence is considered in the optimization (see Fig. 5).

If the cubes are not disjoint, then the optimization problem is no longer similar to the bin-packing problem, since now the capacity needed to pack a set of cubes into the same LUT is not the sum of the size of each cube as in the case of the standard bin-packing problem. In fact, if two cubes c_1 and c_2 share p variables, where c_1 has p_1 variables and c_2 has p_2 variables, then the capacity needed is $p_1 + p_2 - p$ and not $p_1 + p_2$. Thus we could have a

[4]This is equivalent to considering all possible decompositions of node i so that the resulting decomposition is implementable by an LUT.

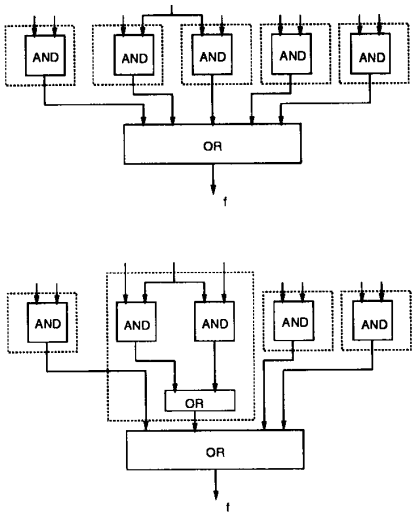


Fig. 5. Local reconvergence.

more effective optimization if we were to pack cubes with shared variables in the same LUT as shown in the following example.

Suppose that the function $F = ab+cd+de+fg+hi$ is to be implemented. Using the First Fit Decreasing algorithm with $k = 5$, we could have the following decomposition:

$$z_1 = ab + cd; \tag{10}$$
$$z_2 = de + fg + z_1; \tag{11}$$
$$F = hi + z_2. \tag{12}$$

If cubes cd and de are placed into the same LUT we would have:

$$z_1 = ab + cd + de; \tag{13}$$
$$F = fg + hi + z_1. \tag{14}$$

However, placing cubes with shared variables in the same LUT *a priori* may not always yield the optimum solution. Furthermore, there may be several cubes that share variables and they may not fit in the same LUT. In this case, the question of which groups of cubes should be "merged" into a single LUT arises. Since the bin-packing algorithm is very fast, the solution chosen by Chortle-crf is to run the algorithm exhaustively on all possible cases, i.e., no merging (equivalent to considering all cubes as disjoint), and all possible mergings of cubes with shared variables. Note that, if the number of cubes with shared variables is large, this approach would be too expensive even if the analysis of each case could be carried out very fast. A heuristic has been added recently which searches for maximum sharing in an LUT [20].

A similar approach is taken to optimize across the fan-out points of the network. Suppose that the network to implement has two outputs given by:

$$f_1 = de + z; \tag{15}$$
$$f_2 = fg + z; z = abc. \tag{16}$$

In this case, the decomposition of the network into a forest of trees would force the implementation of z as the output of an LUT, and would require two more LUT's to implement f_1 and f_2. However, we could merge z into f_1 and f_2 to yield:

$$f_1 = de + abc; \tag{17}$$
$$f_2 = fg + abc. \tag{18}$$

In this case, some logic is "duplicated" (the cube abc would appear in two LUT's) but the number of LUT's is reduced.

In Chortle-crf the following approach is taken. Every path starting from a fan-out point has to reach either another fan-out point or a primary output v. The node that produces v as output is called a *visible node*. The optimization process considers all the visible nodes as functions to be implemented. Two possible implementations are then examined, one with the fan-out variable considered as an input variable (corresponding to the standard tree decomposition approach), the other with the fan-out variable replaced by its expression in terms of its fan-ins. The best solution is then selected. If there is more than one reconvergent fan-out at a visible node, the process considers all the possible combinations of choices for each of the fan-out points that reconverges to the visible node.

If there are many reconvergent paths terminating at a node, the optimization may take a long time because of the very large number of cases to be checked. A possible remedy to this situation is to preprocess the network using a decomposition step (e.g., with one of the *kerneling* algorithms of MIS). It can be shown that the number of reconvergent paths after preprocessing is always less than or at worst equal to the number of reconvergent paths in the network at the end of Chortle's AND-OR decomposition.

2) The Two-Step Approach: This approach, proposed first in [37] and followed also in [18], [28], [50] begins as in the Chortle case with a network that has already been optimized via technology-independent transformations. The first step in the two step approach is to use decomposition to obtain a feasible network. In the second step, the network is manipulated to reduce the number of LUT's used by exploiting the characteristics of the particular LUT architecture considered.

Today, many designs are entered directly in a form which guarantees a feasible implementation in an LUT-based FPGA (for example, an XNF description of Xilinx). In this case, the first step is not needed.

First step: Decomposition: All nodes of the network that have more than k inputs, are decomposed to yield a feasible network.

MIS-pga1 decomposition In the first version of MIS-pga, two decomposition techniques are used:

- kernel decomposition;
- the Roth-Karp decomposition [26].

In kernel decomposition, kernels of the logic function of an infeasible node n_O are extracted and evaluated with a cost function which attempts to consider not only the number of LUT's but also the wiring resources that may be

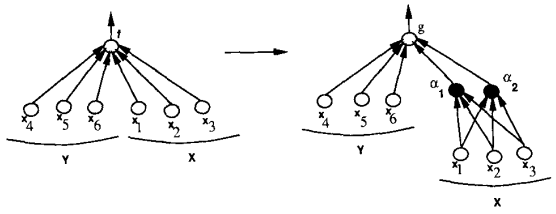


Fig. 6. Application of Roth-Karp decomposition.

needed in the implementation. When a kernel is extracted, a new node is created. Its output is then fed into the original node. After the kernel is extracted, input variables have to be provided to the corresponding node. If the original node shares variables with the kernel, then new edges are added to the network. If the two nodes are implemented in separate LUT's, the new edges correspond to signals to be routed in the FPGA. Hence, it makes sense to select for extraction the kernel which creates the minimum number of new edges. This decomposition is referred to as *split* decomposition.

Note that, following this procedure, the kernel extracted and node n_0 may have more than k variables each. Thus the procedure should be applied recursively until all nodes are feasible. However, the recursive decomposition may fail to produce a feasible network if there are no kernels for an infeasible node except itself (e.g., $abcefm$ or $ab+c+ghp$) and a different technique must be used. In MIS-pga1, an AND-OR decomposition is applied until all nodes are feasible. For example, $abcefm$ is split into $z = abc$ and $zefm$; $ab + c + gph$ is split in $z = ab + c$ and $z + gph$.

The Roth-Karp decomposition [26] is an efficient algorithm which implements the classical decomposition theory of Ashenhurst and Curtis [3], [12].

Ashenhurst gave necessary and sufficient conditions for the existence of a *simple disjoint decomposition* of a function f of n variables. A simple disjoint decomposition of f is of the form:

$$f(x_1, x_2, \ldots, x_s, x_{s+1}, \ldots, x_n) = g(\alpha(x_1, x_2, \ldots, x_s), x_{s+1}, \ldots, x_n) \quad (19)$$

Curtis [12] extended the result to a generalized decomposition of the form:

$$f(x_1, x_2, \ldots, x_s, x_{s+1}, \ldots, x_n) = g(\alpha_1(x_1, x_2, \ldots, x_s), \ldots, \alpha_t(x_1, x_2, \ldots, x_s), x_{s+1}, \ldots, x_n) \quad (20)$$

The set $X = \{x_1, x_2, \ldots, x_s\}$ is called the *bound set*. The set $Y = \{x_{s+1}, \ldots, x_n\}$ is called the *free set*. Figure 6 shows the structure of the decomposition obtained (for $k = 5$).

We denote by a decomposition chart the truth-table of f where minterms of $B^n = \{0,1\}^n$ are arranged as follows. The minterms in the space B^s correspond to the columns of the chart and those in B^{n-s} to the rows. The entries in the chart are the values that f takes for all the possible combinations. For example, if $f(a,b,c) = abc + a'b'c$, the decomposition chart for f for partition $ab|c$ is

$\frac{ab}{c}$	00	01	10	11
0	0	0	0	0
0	1	0	0	1

The necessary and sufficient conditions were given in terms of the *decomposition chart* for f for the partition $x_1x_2 \ldots x_s|x_{s+1} \ldots x_n$ (also represented as $\frac{x_1x_2 \ldots x_s}{x_{s+1} \ldots x_n}$). Curtis showed that the decomposition (20) exists if and only if the corresponding decomposition chart has at most 2^t distinct column patterns (or its *column multiplicity* is at most 2^t). To get the functions α_i, *equivalence classes* of minterms in B^s are formed. Two minterms in B^s are equivalent if they have the same column patterns. If M is the column multiplicity, there will be M equivalence classes. Each class is then assigned a binary code. The minimum code length is $\lceil log_2 M \rceil = t$. Bit i of the binary code corresponds to the function α_i. The function g can then be determined by considering each minterm in the on-set of f and replacing its bound part by the binary code for the corresponding equivalence class.

We illustrate the decomposition technique using previous example. There are two distinct column patterns, resulting in the equivalence classes $c_1 = \{00, 11\}$ and $c_2 = \{01, 10\}$. $M = 2 \Rightarrow t = 1$. Let c_1 be assigned the code 1 and c_2 0. Then $\alpha_1(a,b) = ab + a'b'$. Since $f = abc + a'b'c$, $g = \alpha_1 c + \alpha_1 c = \alpha_1 c$.

The Roth-Karp decomposition is based on the same theory but avoids building decomposition charts, which always require exponential space, by using a cube representation.

In order to make an infeasible node feasible, $|X|$ should be at most k. This ensures that $\alpha_1, \ldots, \alpha_t$ are feasible. However if $t+|Y|$ is greater than k, g has to be decomposed further and the procedure is applied recursively, until all nodes involved are feasible. Since a nontrivial disjoint decomposition may not exist, an AND/OR decomposition is used as a last resort.

The choice of the bound set affects the form of g and t so that different bound sets may yield different decompositions. Since the procedure is computationally expensive, attempting several choices of bound sets to obtain good results is out of the question. The strategy used in MIS-pga1, instead, is to simply pick as bound set the first k variables of the function. More research is needed to find whether a more intelligent choice of bound set would yield significantly better results. It is important to point out that for symmetric functions all bound sets of a given cardinality produce the same g and hence the simple minded heuristic used in MIS-pga1 does not compromise the quality of the final result for this class of functions.

It is not possible to prove that the Karp-Roth decomposition strategy is always better than the split decomposition.[5] Experimental results indicate that the Roth-Karp decomposition is most effective when the node to be decomposed is a symmetric function. However, lacking a general theory, MIS-pga1 uses both decompositions and selects the best result among the two.

Hydra decomposition: Hydra [18] is a program specifically targeted to multiple output LUT's.

In Hydra the decomposition step to make the Boolean network feasible consists of two operations applied in sequence. The first is a simple-disjoint decomposition. The second is an AND-OR decomposition which is applied only if the nodes of the network are still infeasible after the application of simple-disjoint decomposition.

Among all possible choices of variables to place in X[6], Hydra considers only the ones that can be shared with other functions. The rationale for this choice is best explained with an example. Let

$$F_1 = F_1(a, b, c, d, e, f), \tag{21}$$

$$F_2 = F_2(c, d, e, f, g), \tag{22}$$

be the network to be implemented with a Xilinx series 3000 FPGA. If the two functions are decomposed independently their implementation would require at least three single-output Xilinx CLB's since F_1 has support larger than five. However, if the following decomposition is applied,

$$z_1 = h_1(c, d, e, f), \tag{23}$$

$$z_2 = h_2(c, d, e, f), \tag{24}$$

then

$$F_1 = F_1(a, b, z_1), \tag{25}$$

$$F_2 = F_2(z_2, g), \tag{26}$$

and two multiple-output Xilinx CLB's would suffice since h_1, h_2, F_1 and F_2 have support less than or equal to four and the pairs $(h_1, h_2), (F_1, F_2)$ have joint support less than or equal to five.

In Hydra, the choice of the set X'is guided by the construction of the *shared input graph*. This graph has as many nodes as the Boolean network and there is an arc between node i and node j if f_i and f_j, the functions associated with the nodes, share some inputs. A weight equal to the cardinality of the set of shared variables is assigned to each arc. The graph is traversed searching for arcs with largest weight. The set of variables identified by the arcs are tested to see whether a simple disjoint decomposition of both functions that share that set of variables is possible. Note that testing for disjoint decomposition is expensive. It is exponential in the cardinality of the set S.

Given the cost of testing whether a given function has a simple disjoint decomposition, Hydra performs an AND-OR decomposition preprocessing step on the network after technology independent optimization so that the number of variables in the support of all the nodes is no larger than a given limit l. In [18], the best results were obtained with $l = 9$.

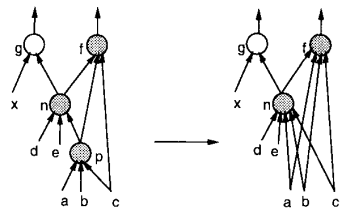


Fig. 7. Node elimination.

Xmap decomposition: Xmap [28] uses the *if-then-else DAG* representation discussed in Section II. Once the Boolean network is converted into an *if-then-else* DAG, all the nodes are feasible if $k > 2$ since they have three inputs.[7] The conversion of the Boolean network into an *if-then-else* DAG can be considered as a decomposition technique which makes a general network feasible.

Second step: Node elimination: After obtaining a feasible network, the number of nodes (and hence LUT's) can be reduced substantially by combining some of them. An example is shown in Fig. 7. Here $k = 5$. Node p can be collapsed into f and n without making them infeasible. This decreases the number of nodes in the feasible network by 1.

The following node elimination techniques which can be applied to any LUT-based architecture have been proposed:

- *Local elimination* [8], [18] also called *partitioning* [37], where nodes are eliminated by examining only node-fan-out pairs.
- *Covering* [37], [18], [28], [50], where nodes are eliminated by considering the overall structure of the network.
- A third technique referred to as *Merging* is implementation-dependent algorithm that exploits the particular LUT-based architecture [37], [18], [28], [50].

Local elimination: The basic idea of local elimination is to examine pairs (i, j) of nodes where node i is a fan-in to node j. If the node obtained by collapsing node i into node j is feasible, i.e., the new support set of j has cardinality less than or equal to k, then the new combined node can be implemented by a single LUT. However, creating this new node may substantially increase the number of connections among LUT's and hence make the wiring problem more difficult. While Hydra accepts local eliminations as soon as they are found, MIS-pga1 orders all possible local eliminations as a function of the increase in the number of interconnections resulting from each elimination. The best local eliminations are then selected greedily.

Covering: While local elimination can be used successfully in reducing the number of LUT's, its myopic view of the structure of the network causes it to miss better

[5] As is often the case in many steps of logic optimization even for standard libraries.

[6] There is a large number of possible choices: $O(|S|!)$.

[7] For $k = 2$, the *if-then-else* triple is converted into three nodes with two inputs. Let $z = \textit{if } a \textit{ then } b \textit{ else } c$ be the triple to be converted. Then the three nodes can be constructed as $z_1 = ab$, $z_2 = a'c$ and $z = z_1 + z_2$.

solutions. Covering takes a global view of the network. It identifies clusters of nodes that could be combined into a single LUT.

The most general formulation of the covering problem for LUT's is given in Mis-pga1 [37]. Let a *supernode* of a node i, S_i, be a cluster of nodes consisting of i and some nodes in the transitive fan-in of i, such that the maximum number of inputs to S_i is k and if a node $j \in S_i$, then all the nodes on some path from j to i are in the supernode as well. Note that each supernode is a feasible node (the number of inputs is less than or equal to k by definition), and all its nodes could be implemented by a single LUT.

There may be several supernodes associated to a node i. The covering algorithm of [37] generates all of them.

Repeating this procedure for all nodes generates a potentially large set of supernodes that can be used to cover the original network. The optimum covering problem is to find the smallest set of supernodes that covers all the nodes of the network. If we did not have any other constraints to satisfy, this problem would be a standard NP-hard *set covering problem* for which good heuristics as well as relatively fast exact algorithms are known [23]. This is not the case, however. We have to make sure that each input to the optimum supernode set is an output of some other supernode in the set or be a primary input. This constraint poses a limitation in the way we choose supernodes; choosing a particular supernode may exclude several others from consideration.

This constraint makes the covering problem much harder: it becomes a *binate covering problem* [44] for which no generally effective heuristic or relatively fast exact algorithms have been found. As a result, the computation time for Mis-pga1, which employs both an exact algorithm [33] and a heuristic, is excessive.

In [18], [50], [28] a variety of greedy heuristics are proposed to solve the covering problem. It is interesting to note that the computation time for these heuristics is very short and that the quality of the final solution does not seem to suffer too much with respect to the optimum solution given the same initial network.

Hydra [18] examines the nodes of the network by ordering them by decreasing number of inputs. The nodes with k inputs are assigned to an LUT (note that the node may have some reconvergent path terminating in it and that by collapsing a number of predecessors, the number of inputs may actually decrease and allow a number of nodes to be mapped into the same LUT; Hydra will miss this). An unassigned node with maximum number of inputs is chosen out of the other nodes. A second node is then chosen so that the two nodes can be merged into the same LUT and a cost function maximized. The cost function is a linear combination of the number of shared inputs and the total number of inputs. The emphasis on shared inputs is aimed at improving the result of the subsequent merging step, as described below. This greedy procedure stops when all unexamined nodes have been considered.

The procedure used by Xmap [28] traverses in a breadth first fashion the *if-then-else* DAG from inputs to outputs and keeps a log of the number of inputs that are seen in the paths that connect the primary inputs to the node under consideration. If the node has more than k inputs some of its predecessors have to be placed in a different LUT. These predecessors are chosen according to the number of their inputs. The more inputs they can isolate from the node under consideration the better. This algorithm is very fast because of the lack of any backtracking in the search strategy. It is also in general more powerful than Hydra's since it considers reconvergence. However, it does not consider the possibility of packing two different functions in one LUT while Hydra does.

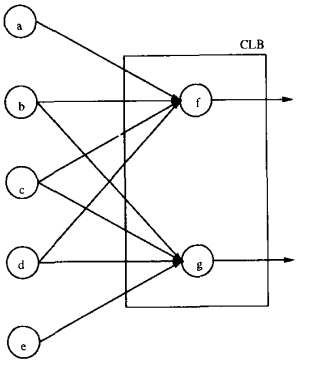


Fig. 8. Two functions in a CLB.

The heuristics used in VISMAP [50] consist of three basic steps. In the first, the network is traversed from inputs to outputs and supernodes are greedily identified as they are encountered in the traversal. The network is then traversed again and all possible clusterings in the supernodes are examined to determine the best. This procedure identifies fewer supernodes as compared to MIS-pga1 but solves the covering problem exhaustively and hence optimally. However, if the number of nodes to be considered is large, the exhaustive procedure would be too slow. The network is therefore partitioned into subnetworks before the covering procedure is carried out.

Merging: In all approaches, except Hydra, single output functions are considered in the decomposition, local elimination and covering steps. However, when industrial FPGA's are considered, the particular features of the architectures must be taken into consideration. The purpose of the merging step is to combine nodes that share some inputs. Figure 8 shows two functions f and g which can be put on the same CLB of a Xilinx 3000 FPGA.

The approaches presented in [37, 28, 50] perform a post-processing step to merge pairs of nodes after covering. The problem is formulated as a *maximum cardinality matching* problem [37]: let $G(X, E)$ be a graph where the set of nodes X are nodes of the original network and where the pairs of nodes that can be merged in one Xilinx CLB, i.e., that have support size no larger than four and combined support no larger than five, are adjacent. The maximum reduction in the number of CLB's needed to implement the network is achieved when the largest set of disjoint adjacent pairs are combined. This is the maximum cardinality matching

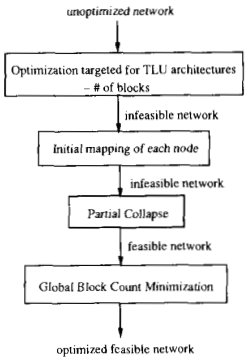


Fig. 9. Flow: MIS-pga2.

problem in a nonbipartite graph. Even though a polynomial time algorithm (in fact $O(n^{2.5})$) exists for the solution of this problem, the algorithm is fairly difficult to implement and its running time can be long for large networks. As a result, heuristics are often used; [28], [50] use greedy matching algorithms which are not guaranteed to find an optimal solution but are very fast.

MIS-pga2: A framework for LUT-logic optimization: Since most of the algorithms used in LUT-based synthesis are heuristic it is very difficult, or even impossible, to compare them in a rigorous way. Extensive experimentation is therefore used.

Acceptable results over a fairly large number of designs can be obtained using most of the approaches presented. However, no single heuristic can find the best results consistently across all designs. Hence, a system encompassing several different algorithms which can be run sequentially or independently would allow the user to customize the synthesis approach to any particular design or architecture.

This is the approach of MIS-pga2. Figure 9 shows the flow-chart of where an initial optimization phase is followed by a sequence of technology mapping algorithms.

In MIS-pga2, the technology independent phase is not strictly independent of the technology and uses a cost function that is different from the one used in MIS. The reason is that unlike gate arrays and standard cells the number of literals in the factored form may not approximate well the actual implementation cost for FPGA's. A good estimate for the cost of a particular decomposition for an FPGA is produced by the bin-packing algorithm applied to the nodes that are modified during the technology independent optimization. This is practical given the speed of the bin-packing algorithm.

In MIS, the nodes of the Boolean network are represented both in a sum-of-products form and in a factored form. Starting the technology-based optimization with one representation or the other does make a difference in the final cost of the implementation. Since there is no theory which can predict the outcome of the choice, MIS-pga2 optimizes both representations and selects the best result. A similar brute-force approach is followed in decomposition where no single algorithm can outperform all others in all benchmarks.

MIS-pga2 offers four decomposition options in addition to the two offered in MIS-pga1 [Roth-Karp and split decomposition described in Section IV-C2)]. These are:

1) **Bin-packing.** The algorithm for bin-packing used in MIS-pga2 is the Best-Fit Decreasing heuristic that selects the bin which has the maximum leftover capacity after the cube has been assigned to it.[8] If the cubes have disjoint support then for $k \leq 5$ an optimum tree implementation is found (a similar result was proved independently for First-Fit Decreasing in [20]).
2) **Co-factoring decomposition.** This approach, applied only if $k \geq 3$, is particularly effective for functions where cubes share several variables. Each node is decomposed by computing the Shannon cofactor $af_a + a'f_{a'}$ until the leaf nodes have support that is no larger than k. All nodes of the network after the decomposition (except possibly the leaf nodes) have at most three inputs. If $k \geq 4$, a simple post-processing elimination step similar to the approach proposed in Xmap may be tried to reduce the number of nodes in the network. It is possible to give an upper bound on the number of CLB's needed to implement the network obtained by this simple decomposition [38]. However, this bound is exponential in the number of inputs t of the function and hence this procedure may not be good if $t \gg k$.
3) **AND/OR decomposition.** This decomposition breaks up the nodes of the network so that the resulting network has nodes that are either inverters, two-input AND gates or two-input OR gates which can be packed by the covering step.
4) **Disjoint decomposition.** This decomposition is the *decomp -d* option in MIS. It partitions the cubes of the function into a set of cubes with disjoint support and then creates a node for each partition and a node that is the OR of the outputs of the partition nodes. Note that since the nodes of the partition have cubes of disjoint support, the bin-packing heuristic when applied to the result will provide a locally optimum decomposition. Thus disjoint decomposition could be an effective preprocessing step for bin-packing.

In MIS-pga2, local elimination is applied not only to the nodes in a feasible network while maintaining feasibility, but also to nodes in an infeasible network. This algorithm, called *partial collapse*, is shown in Figure 10. It collapses nodes of a possibly infeasible network into their fan-outs and recomputes the cost of the network using the bin-packing algorithm. The candidate nodes for collapsing are chosen according to the number of inputs. The list of nodes that result in a gain when individually collapsed is formed and an integer programming problem is solved to select the subset that gives the best overall gain.

[8] Note that MIS-pga2 does not extract a forest of trees to perform the mapping as Chortle-crf does. Instead, it uses the heuristic to decompose infeasible nodes of the Boolean network.

```
partial_collapse(η)
{
    L = list_of_candidate_nodes_for_collapsing(η);
    foreach_node_in_L {
        collapse node into its fanouts;
        recompute cost of fanouts;
        if ∑ new_cost(fanout) < ∑ old_cost(fanout) + cost (node) save node
    }
    select_subset_of_saved_nodes_for_max_gain();
    collapse_selected_nodes();
}
```

Fig. 10. Partial collapse.

The integer programming problem is computationally expensive to solve but provides the best set of nodes to collapse. If the number of nodes that yield a gain if collapsed is large, this approach becomes computationally infeasible. An alternate greedy approach to the problem selects at each stage the node whose collapsing would yield the best gain.

In MIS-pga2, covering is performed either with the exact binate-covering algorithm if the network to cover is not too large or otherwise with heuristics. As in MIS-pga1, merging is carried out using the max-cardinality matching algorithm on the covered network. Instead of applying successive covering and merging which may yield suboptimal results a new formulation of a combined covering-merging step as a single, binate covering problem was suggested in [38].

Since MIS-pga is in the public domain, many researchers have been able to use the framework and some of the algorithms to develop their own novel approaches, thus adding to the library of algorithms available to the FPGA designer and tool developer.

Modifying the optimization steps: Several attempts have been made to target the optimization steps to LUT-based FPGA architectures.

- In MIS-pga2 the cost function in kernel extraction was changed.
- Fujita and Matsunaga [22] modified the simplification step to better suit LUT-based architectures. Whereas in the standard simplification step, a minimal representation of the function at each node is sought, in the modified simplification step in [22], the target is to minimize the support of each node of the network. Each node n is now simplified as follows. First, candidate nodes are selected which may be used for fan-ins of n. Characteristic functions of n and of the candidate nodes are computed. From these, sets of minimal supports for n are computed using the algorithm of Halatsis and Gaintans [25]. Finally, the irredundant cover for n is computed using a minimal support. The algorithm allows use of don't care sets. After this step, any LUT technology mapper may be used.

C. Comparisons and Observations In Table 1, we present results of MIS-pga2, Chortle-crf, and Xmap. The starting networks are the same, except that we had to run *decomp -g* on the starting network before running Chortle-crf; otherwise Chortle-crf does not complete on many examples in reasonable time. These networks were obtained by repeatedly running several MIS scripts until no improvement was obtained and then picking the best result. MIS-pga2 and Chortle-crf were run on a DEC5500 (a 28 mips machine). Xmap was run on a SUN4/370 (a 12.5 mips machine). The table shows the number of *five-input single-output LUT's*

Table 1 Number of Five-Input Single-Output LUT Blocks: n Number of Five-Input LUT Blocks; t Run Time in Seconds.

example	MIS-pga2		Chortle-crf		Xmap	
	n	t	n	t	n	t
z4ml	5	5.0	7	0.1	9	0.2
misex1	11	2.7	11	0.1	11	0.2
vg2	20	7.4	21	0.1	24	0.2
5xp1	18	22.4	28	0.4	31	0.3
count	31	5.8	31	0.3	31	0.2
9symml	7	127.2	44	6.4	55	0.4
9sym	7	339.7	59	12.8	73	0.5
apex7	60	18.7	60	0.6	65	0.5
rd84	10	73.7	35	1.3	36	0.4
e64	80	14.7	80	0.3	80	0.5
C880	82	546.8	88	2.2	103	0.8
apex2	67	388.5	64	2.9	81	0.7
alu2	109	773.8	116	7.1	126	0.9
duke2	110	203.7	111	1.7	127	0.8
C499	68	1074.4	89	2.6	75	0.5
rot	181	282.1	188	2.7	212	1.4
apex6	182	243.9	198	2.9	231	1.6
alu4	55	887.5	70	2.5	98	0.7
apex4	412[1]	198.7	579	98.9	664	6.4
des	904	3186.3	927	35.4	1042	6.8
sao2	28	41.9	27	0.5	37	0.4
rd73	6	24.0	16	0.3	21	0.1
misex2	28	3.4	28	0.1	28	0.2
f51m	17	14.4	27	0.4	33	0.3
clip	28	58.4	31	0.7	38	0.3
bw	28	17.3	39	0.3	43	0.3
b9	39	27.6	41	0.4	48	0.4

[1]Modified kernel extraction and partial collapse could not finish so a faster script was used.

Table 2 Number of Two-Output Xilinx CLB's

example	MIS-pga2	Chortle-crf	Xmap	Hydra	VISMAP[1]	xnfopt
z4ml	4	4	7	4	4	5
misex1	9	9	9	9	9	10
vg2	18	20	19	21	20	19
5xp1	13	22	20	22	19	22
count	30	31	22	24	-	25
9symml	7	38	38	37	46	50
9sym	7	51	52	66	50	54
apex7	43	47	50	44	46	49
rd84	9	28	28	29	42	36
e64	56	48	55	47	-	62
C880	72	75	79	72	76	113
apex2	60	53	60	69	-	94
alu2	96	94	89	88	-	85
duke2	94	83	83	81	93	107
C499	66	83	57	63	49	81
rot	143	133	146	145	137	230
apex6	165	161	182	165	155	159
alu4	49	63	76	145	-	161
apex4	371	479	475	503	-	560
des	-[2]	707	674	701	-	971
sao2	28	26	28	37	32	50
rd73	5	15	13	13	14	23
misex2	25	21	23	21	22	24
f51m	15	25	22	17	-	25
clip	23	25	24	27	-	35
bw	27	31	29	29	-	38
b9	32	30	39	36	27	29

[1]The starting networks were obtained by running the MIS script once and may differ from those used for other systems.
[2]Merge could not finish.

needed to implement the benchmark and the time taken (in sec.) in columns n and t respectively.

The following must be noted before comparing the results:

1) The results are sensitive to the starting networks used. Hence some of the results cannot be directly compared. However, some systems such as MIS-pga2 attempt to target the optimization to LUT-based architectures.
2) An implementation with smaller number of LUT's is not necessarily more routable.

Interestingly, on benchmarks *5xp1, 9sym, rd84, C499, apex4, rd73, f51m* and *bw*, MIS-pga2 performs much better than other systems. Part of the reason is that some of these circuits are symmetric and, therefore, the Roth-Karp decomposition works very well. Also, exact covering techniques can be applied on some of the small benchmarks to obtain significant improvements. However, the time taken by Chortle-crf and Xmap is much less than MIS-pga2.

The results for two-output Xilinx 3000 CLB's are presented in Table 2. The results for MIS-pga2, Chortle-crf, Xmap, Hydra, VISMAP and xnfopt (the proprietary system from Xilinx [52]) are compared. A "-" in the VISMAP column indicates that the results were not available. The starting networks for all systems (except VISMAP) are the same. For xnfopt, the number of passes for each example was set to 10. However, for *C499, rot, des, C5315* and *C880*, an interrupt was externally generated after 8 passes since xnfopt was taking too much time. *MIS-pga2* outperforms *Chortle-crf*, *Xmap* and *Hydra* by 15.6 %, 16.9% and 16.9% respectively.[9]

Note that MIS-pga2, Chortle-crf, Xmap and VISMAP exploit the two-output feature in a post-processing step, whereas Hydra targets mapping from the very beginning for two outputs.

Comparing the number of two-output LUT's with the number of single output LUT's for each system, Xmap gets significant improvements. One reason is that Xmap uses a cofactoring technique which generates nodes with at most three fan-ins. The possibilities for merging are much higher. MIS-pga2 does not do as well, because it works too hard on minimizing the number of single output LUT's, which may not be good for merging.

D. Synthesis for Routability The algorithms presented in the previous sections are primarily concerned with minimizing the number of LUT's needed to implement a given logic function. In the most popular LUT-based FPGA, wiring resources are scarce and as a result a logic function requiring far fewer blocks than available on a single FPGA

[9]We did not consider *des* in this set of results, since *merge* could not finish.

may not be routable.[10] Hence routability should be carefully considered as a cost function in optimization. However, it is difficult to predict at the logic synthesis stage what routing resources will be needed.

To establish a link between logic synthesis and layout, a correspondence between nodes in the Boolean network and cells in the layout is assumed [1], [40]. In this case, the physical implementation has a close resemblance to the topology of the Boolean network in that inputs to the nodes are signals to be routed on the final chip. If the signal sources are fixed (for example, if the pad positions are predetermined) then it makes sense to manipulate the network so that signals arrive to the nodes in an order that is consistent with their positions on the chip. This order is called *lexicographical order* in [1].

In [41], the nodes of the Boolean network are placed in the two dimensional plane with algorithms that are approximate versions of the ones used in actual placement. In this case, wiring area and length can be estimated with better accuracy by considering the nodes of the Boolean networks as physical blocks and the edges in the Boolean network as interconnections.

However, the results reported in [40] for technology independent logic synthesis and technology mapping with layout considerations are not as good as one would expect. The average improvements are of the order of a few percents over the standard approach that does not take layout into direct consideration. Contrastingly, in [48] it is claimed that significantly better area is achieved after layout than with other approaches where the link with layout is not as explicit.

In the case of LUT-based FPGA's, several algorithms attempt to minimize the needed wiring. For example, the cost functions used by MIS-pga and VISMAP take wiring into account by penalizing the creation of additional signals while operating on the network.

Synthesis for routability is an area where more research is needed both for FPGA's as well as for more conventional ASICs.

E. Performance Optimization Given the high performance requirements of system designs and the added delays due to the programmability of FPGA's, timing optimization is a very important goal of logic synthesis. It is important to note that minimizing area, which is the most common goal of today's synthesis tools, may result in slow implementations. Much research has been done on logic synthesis for timing optimization and its relationship with testability (e.g., [14], [49], [45]).

Delay in a circuit is due to delays in gates and interconnects. For mask-programmed design styles implemented in older technologies (above 1 micron), delay is mostly due to logic gates while interconnect delay is negligible. However, for submicron technologies, and for FPGA's, interconnect delays are at least as large as the delays in the logic blocks. For example, for LUT-based FPGA's that use pass transistors as switching elements, the delay of a signal through general purpose interconnect could be much larger than that through one logic block.

There are three basic approaches to synthesis for performance optimization:

- Delay optimization is equated to minimizing the depth in the network [21], [11].
- The delay of the circuit is approximated by a combination of block delay and interconnect delay. Interconnect delay is estimated as a function of number of levels, nodes, and edges in the network [39].
- Critical path analysis with a (possibly simplified) delay model [39] is performed on a placed circuit. In this approach, logic optimization and actual layout are performed in concert.

The first approach is certainly faster. The number of levels in a circuit correlates with the performance of the circuit particularly well when the delay is mostly due to block delays. The third approach is more accurate but is potentially computationally inefficient due to the complexity of the mixed layout-synthesis algorithms used. The second approach is a compromise between the need for better accuracy and compute-time requirements.

Reducing the number of levels: The approach developed in [21] is a variation on the basic algorithm of Chortle-crf. The bin-packing algorithm is still used but here the cost function optimized is not the number of LUT's but the number of levels of logic. Assuming that the delay is entirely due to the logic blocks, minimizing this cost function corresponds to minimizing the delays of all paths in the circuit. Unfortunately, the number of LUT's tends to grow large when minimizing all paths. For this reason a post processing step that reduces the number of LUT's without increasing the delay of the circuit has been proposed in [21]. Minimizing the delay of all paths is in general an overkill since the performance of the circuit depends only on the critical paths.[11] The critical paths are usually not known *a priori*, however.

The procedure starts as in Chortle-crf with a network that has been AND/OR decomposed and then split into trees. For each tree, the nodes are grouped according to their levels. Primary inputs are assigned level 0. A node is at level D if the highest level node among its fan-ins is at level $D-1$. Nodes at the same level are grouped into a set called *stratum*. The first-fit decreasing algorithm is then applied to minimize the number of LUT's in each stratum.

After all strata have been processed, the outputs of the LUT's at level D are connected to the available inputs of LUT's at level $D+1$. If the number of available inputs is not sufficient, a new LUT is added at level $D+1$. This algorithm is guaranteed to find the minimum depth implementation of the tree if $k \leq 6$. Otherwise, it may produce a suboptimal solution [21].

[10] Taking layout into account while performing logic synthesis is important also in other ASIC technologies, since for large sea-of-gates and standard cell designs wiring area is often larger than the area occupied by logic macros.

[11] A critical path in a directed acyclic graph (DAG) is a path from the primary inputs to the primary outputs of maximum length, i.e., maximum number of levels.

Reconvergent fan-outs are taken into account in the optimization process with a heuristic that is essentially the same as the one proposed in Chortle-crf.

A postprocessor finds the critical paths of the circuit and the network is processed to minimize the number of LUT's but with the constraint that the length of the critical paths must remain the same. First the area minimization of Chortle-crf is applied to the network. The algorithm may change the length of the critical path. To avoid an increase in estimated delay, all paths that have length larger than the critical path in the original network are re-processed using the algorithm for the number of levels described above. The procedure is iterated until all paths meet the target delay constraint.

The procedure has been observed to yield circuits that have 35% fewer levels than Chortle-crf but with 59% more LUT's.

Another approach to timing optimization, dag-map [11], follows the clustering algorithm by Lawler *et al.* [30]. The network is first mapped into a network of two-input NAND gates as in MIS, but with an improved algorithm that guarantees that the number of levels in the transformed circuit is within a constant of the number of levels in the original circuit. A clustering algorithm is then applied which labels nodes of the network beginning with the primary inputs and ending with the primary outputs. Primary inputs are labeled 0. A label is assigned to a node, v, after all its inputs have been labeled. Let $input(V)$ be the set of input nodes to the set of nodes V. Let $N_p(v)$ be the set of predecessors of v with label p. Then if

$$|input(N_p(v)) \cup \{v\}| \leq k \tag{27}$$

the node v is labeled p, otherwise $p+1$.

After all nodes are labeled, a backtrack phase begins where nodes are assigned to k-input LU's. This phase begins with the primary outputs which are placed in a queue. For each node in the queue, the node and all nodes that have the same label are assigned to an LUT. The node is then deleted from the queue and the set of nodes that are inputs to the nodes placed in the LUT are now added to the queue. The phase ends when only primary inputs remain in the queue.

Dag-map operates on the network without decomposing it into trees. If the starting network happens to be a tree, it is optimal. Also it may replicate nodes in order to achieve a lower number of levels. However, the replication in many cases could be too much.

Approximating the delay with layout information: MIS-pga2 [38] attempts to find an implementation that meets a set of timing constraints and uses the minimum number of LUT's. The timing constraints are given in terms of required arrival times at the primary outputs. Arrival times are provided for the primary inputs. Given delays on the LUT's and an estimate of interconnect delays, the network can be traced to determine the critical paths. The trace has a forward pass where the arrival times of all the signals are found and a backward pass where the required times of all signals are computed. The difference between the required time and the arrival time is the slack of a node. In this formulation of the delay optimization problem, a negative slack corresponds to a circuit that does not satisfy the timing requirements. Hence, the delay reducing operations are applied to the path where negative slacks are found. Note that if all slacks are nonnegative the circuit meets the timing constraints and no timing optimization is needed. If indeed the fastest circuit is desired, then the required times at the outputs can be tightened until no feasible solution is found. This strategy can be implemented by optimizing the path with minimum slack.

The starting point for MIS-pga2 in delay mode is a network where technology independent timing optimization has been carried out using the standard MIS script. Note that the network is in terms of two-input gates and hence is feasible.

The optimization in MIS-pga2 is divided into two basic approaches:

- A placement independent approach, where the optimizations are all performed at the logic level and a rough estimate of the interconnect delay is used;
- A placement dependent approach where synthesis driven placement using simulated annealing is performed. Here the interconnect delay estimate is accurate.

Placement independent (PI) optimization: In this approach, the placement and routing phase is considered to be a stochastic process. An LUT-based FPGA is modeled as a square grid where the nodes of the grid correspond to the LUT locations, and the classical results of Donath [15] on average wiring length L_{av} as a function of number of blocks to be placed on the grid and their interconnections can be used. Donath's theory estimated the average wiring length to be

$$L_{av} = \frac{2}{3}\sqrt{|V|}|E| \tag{28}$$

where V is the set of LUT's (nodes of the Boolean network) to be placed and E is the set of edges in the network. The empirical delay formula is then given as:

$$Delay = \lambda\ell + (aL^2 + bL + c) \tag{29}$$

where λ is the delay of a CLB, ℓ is the number of levels in the network and $L = log(L_{av})$. This formula has been derived empirically by mapping a fairly large number of examples with Xilinx placement and routing tools and then fitting the data. The parameters in the delay equation, a, b, and c, are used to tune the equation. The delay equation is used to evaluate the performance of a circuit in place of the cruder estimate based on the number of logic levels.

The overall algorithm has the following form. For each node in the critical path, it tries to collapse the node into its fan-outs. The elimination is then accepted if the node so obtained is feasible, or if it is not feasible, but can be redecomposed so that the delay estimate decreases.

Placement driven (PD) logic resynthesis: A more accurate estimate of the delay of the circuit can be achieved

after placement. However, in the standard flow of synthesis-based design, placement is performed after logic synthesis is completed and hence there is no feedback from placement to logic synthesis. The placement-dependent approach proposed in MIS-pga2 starts from an optimized feasible network obtained by the previous placement-independent approach.

The placement problem is formulated as assigning locations to point modules on an n by n grid (in the Xilinx XC3000 series, n can take values from eight to 18). This problem is solved using simulated annealing. The difference from the standard simulated annealing algorithm is in the resynthesis step. At the end of the iterations at each temperature below a threshold, *critical* sections are identified. Logic synthesis and force directed placement techniques are used to restructure and reposition these sections. The logic synthesis techniques used are *decomposition* and *partial collapse*. These techniques are local; i.e., only the neighborhood of a critical section is explored for a better solution. The algorithm is summarized by the pseudo-code in Fig. 11.

The cost function is also particularly tuned for the problem at hand,

$$\Delta c = (1 - \beta(T))\Delta l + \beta(T)\Delta d, \tag{30}$$

where l is the total estimated net length, d is the estimated delay and $\beta(T) \in [0, 1]$ is a temperature-varying parameter monotonically decreasing with T, which gives more weight to total net length at higher temperatures and more weight to delay at lower temperatures. The form of $\beta(T)$ was determined experimentally. The delay estimate is performed using two models, the Elmore model [16] and the Penfield-Rubinstein model [43]. The choice of the model to use is left to the user. The Penfield-Rubinstein model is in general more accurate but more expensive to compute. In any case, since many moves are in general attempted by simulated annealing, the actual delay calculation when a move is evaluated is carried out with the Elmore model. The delay calculation is performed with the Penfield-Rubinstein model only if the move is accepted.

Before entering a new simulated annealing inner loop, a placement of the resynthesized part of the network is carried out. The placement algorithm used is a simple force-directed algorithm that finds a good position for the blocks of the circuit affected by the local resynthesis procedure. The positions of all the other blocks of the network are not changed. Note that the number of blocks may increase as a result of the resynthesis step. However, the capacity of the chip is never exceeded.

Comparisons and observations: Results on the use of MIS-pga2, Chortle-d and dag-map to optimize performance are reported in this section. First the benchmarks were optimized for area. Then a delay reduction script was used to obtain delay optimized networks in terms of two-input NAND gates. In Table 3, results after the placement independent optimization phase of MIS-pga2 (column PI)

```
/* a = temp factor (a < 1); T = current temperature;
T_l = starting temperature for logic synthesis;
m = number of moves per temperature; */
{
    T = start_temp;
    while (T > final_temp) {
        j = 0;
        while (j < m) {
            get two random locations for swap;
            evaluate Δc, change in cost;
            accept swap with probability e^(-Δc/T);
            if swap accepted, do delay trace;
            j++;
        }
        if (T < T_l) do logic resynthesis and
                replacement for delay;
        T = T * a;
    }
}
```

Fig. 11. Simulated annealing for placement and resynthesis.

and Chortle-d are reported, using the same starting networks for both programs. We set k to 5. We are restricting ourselves to single output LUT's. The results for dag-map are taken from [11] and the starting networks are not the same as those for the other two systems. For each example, we report the number of levels, nodes, edges and the CPU time (in sec.) on a DEC5500 (a 28 mips machine) in the columns *lev, nodes, edges* and t respectively. Out of 27 benchmarks, MIS-pga2 generates fewer levels on 9 and more on 13. On average (computed as the arithmetic mean of the percentage improvements for each example), MIS-pga2 needed 2.9% more levels. The number of blocks and the number of edges it needed are 58.7% and 66.2% respectively, of those for Chortle-d.[12] As shown later, the number of nodes and edges may play a significant role in determining delay of a network. However, Chortle-d is much faster than MIS-pga2.

A direct comparison with dag-map is not possible since it uses different starting networks. However, dag-map produces fewer levels on many circuits, sometimes at the expense of higher LUT count.

The starting networks for placement are obtained from the level reduction algorithms. Two sets of experiments were reported in [38]:

1) **map**: Place and route the network using *apr*, the Xilinx placement and routing system. This is done for the networks obtained after MIS-pga2 PI phase and for those after Chortle-d.

2) **xln_p**: On the networks obtained after MIS-pga2 PI phase, perform a timing-driven placement using the

[12] A more recent version of Chortle-d has a post-processing stage to reduce the number of blocks without increasing the number of levels in the circuit.

Table 3 Results for Level Reduction: lev Number of Levels in the Feasible Network; nodes Number of Nodes in the Feasible Network; edges number of Edges in the Feasible Network; and t Run Time in Seconds

example	MIS-pga2 - PI				Chortle-d				dag-map[1]	
	lev	*nodes*	*edges*	*t*	*lev*	*nodes*	*edges*	*t*	*lev*	*nodes*
z4ml	2	10	42	2.1	3	20	74	0.1	3	17
misex1	2	17	71	1.7	3	25	99	0.1	2	17
vg2	4	39	165	1.7	3	54	206	0.1	3	42
5xp1	2	21	88	3.5	4	29	115	0.1	3	28
count	4	81	336	5.1	3	102	368	0.1	3	87
9symml	3	7	35	9.9	4	76	273	0.1	5	61
9sym	3	7	35	15.2	5	130	477	0.2	5	63
apex7	4	95	383	8.4	4	131	452	0.2	5	94
rd84	3	13	61	9.8	4	69	268	0.2	4	48
e64	5	212	857	15.7	4	356	1236	0.6	3	167
C880	9	259	1070	39.0	7	383	1437	0.9	8	246
apex2	6	116	481	9.8	5	165	578	0.2	5	164
alu2	6	122	543	42.6	8	316	1189	0.7	9	199
duke2	6	164	685	16.4	4	248	863	0.4	4	195
C499	8	199	896	58.8	6	436	1736	1.8	5	204
rot	7	322	1312	50.0	6	439	1608	1.0	6	328
apex6	5	274	1209	60.0	5	361	1360	0.8	5	284
alu4	11	155	648	15.4	8	194	710	0.3	10	303
sao2	5	45	189	9.5	4	58	220	0.1	-	-
rd73	2	8	36	4.4	4	52	183	0.1	-	-
misex2	3	37	160	1.4	2	52	188	0.1	-	-
f51m	4	23	100	5.9	5	65	237	0.1	-	-
clip	4	54	219	3.7	4	83	281	0.1	-	-
bw	1	28	138	8.3	1	28	138	2.6	-	-
b9	3	47	199	2.3	3	62	225	0.1	-	-
des	11	1397	6159	937.8	9	3024	10928	9.2	6	1480
C5315	10	643	2826	282.2	9	1221	4509	3.6	-	-

[1]The starting networks were obtained by running an MIS script and then `speed_up` and (may) differ from those used for the other two systems.

placement-dependent algorithm. The logic synthesis phase is entered once at each temperature. The resulting placement is routed using *apr* (with its placement phase disabled). The routing tool is instructed to route more critical nets first, as determined by the slacks computed for each edge.

The results of these experiments for placement, resynthesis and routing are shown in Table 4. The table shows the delay through the circuits in nanoseconds after placement and routing. Only benchmarks that were successfully placed and routed on a Xilinx FPGA are shown. The second and third columns give the delays for the designs synthesized using MIS-pga2 PI and Chortle-d respectively. The fourth column refers to the set of experiments *xln_p* for MIS-pga2 PI. The delay numbers in the table are computed from the placement and routing information generated by *apr*. This information gives the length of each net in the layout. *map* (MIS-pga2) gives lower delay than *map* (Chortle-d) on the majority of the examples. More interestingly, we can study the effect of the number of nodes and edges on the delay. For example, although the number of levels in *count* is 3 for Chortle-d and 4 for MIS-pga2 (Table 3), the *map* delay through the circuit for Chortle-d is 3 ns more than MIS-pga2. Note that the block delay and hence delay of a level is 9 ns.[12] Smaller numbers of nodes and edges obtained by MIS-pga2 (PI) help in offsetting the level advantage of Chortle-d by 6 ns. In fact, the *xln_p* option makes the difference even larger. For *vg2, duke2* and *misex2*, the *map* delays for MIS-pga2 are higher than those for Chortle-d, but the difference in delays is less than 9 (*difference in levels*).

Experimental results involving synthesis with placement and routing for the Xilinx FPGA show that the delay optimization performed with the statistical approach gives good results, while surprisingly, the placement-dependent algorithm only occasionally improves the results and by no more than 10%. In addition, the results demonstrate that the number of levels is not an accurate measure of the delay of the circuit (although it is important in reducing the delay). In fact, at the expense of extra CPU time, MIS-pga2 in general achieves better delay than Chortle-d with fewer CLB's and edges but overall more levels.

The disappointing results of the placement-dependent approach are consistent with the unexciting results obtained by combined synthesis and placement in more traditional ASIC styles [41], [40]. These results are counter-intuitive and leads us to believe that much work remains to be done to couple synthesis and layout in a more effective way.

[12]Speed grade -70 is used.

Table 4 Delays of Placed and Routed Designs: map (MIS-pga2) Using *apr* after Running MIS-pga2 PI Phase; map (Chortle-d) Using *apr* after Running Chortle-d; and xln_p Using Just the Routing of *apr* after Running MIS-pga2 PI and PD phases

example	map (MIS-pga2)	map (Chortle-d)	xln_p
z4ml	33.60	56.00	31.00
misex1	33.10	58.00	36.20
vg2	82.90	76.40	76.30
5xp1	33.60	77.40	35.90
count	88.40	91.88	79.02
9symml	54.00	84.10	53.50
9sym	53.70	110.40	53.50
apex7	97.75	108.00	93.90
rd84	50.70	77.80	54.30
apex2	147.43	134.30	142.50
duke2	125.13	114.70	151.83
alu4	256.35	230.68	-[1]
sao2	104.00	82.30	96.00
rd73	33.60	85.00	31.00
misex2	53.80	47.80	53.70
f51m	72.60	107.50	76.60
clip	81.10	84.10	84.60

[1]Two nets could not be routed.

V. Logic Synthesis for Multiplexer-Based Architectures

A. Introduction

Multiplexer-based (MUX-based) FPGA architectures use logic blocks that are combinations of a number of multiplexers and possibly a few additional logic gates such as AND's and/or ORss. Programming is achieved by programmable switches that may connect the inputs of the block to signals coming from other blocks or to the constants 0 or 1, or that may bridge together some of these inputs. In Fig. 12, the ACT-1 and ACT-2 logic modules are illustrated. Note that there are three multiplexers and an OR gate in ACT-1 and three multiplexers, an OR gate and an AND gate in ACT-2.

These logic blocks can implement a fairly large number of logic functions. For example, for the ACT-1 module, shown in Fig. 12, all two-input functions, most three-input functions [27] and several functions with more inputs (the maximum number of inputs to the logic block is eight) can be implemented. However, some of these functions are equivalent in the sense that they only differ by permutation of their inputs. In [34], 702 unique functions for ACT-1 and 766 for ACT-2 were counted.

The recently introduced QuickLogic architecture uses a more complex logic block allowing the inputs to the first level multiplexers to come from AND gates with inverted and noninverted inputs, thereby providing programmable inversion for the multiplexer inputs [42]. Since this architecture has been only recently disclosed an analysis of its power in terms of the number of functions that can be generated is not available at this time.

As in the case of LUT-based architectures, the number of blocks, the logic functions that these blocks can implement and the wiring resources are the main constraints. And similarly, the architecture-specific mapping also starts with a network that has been optimized by the technology-independent operations.

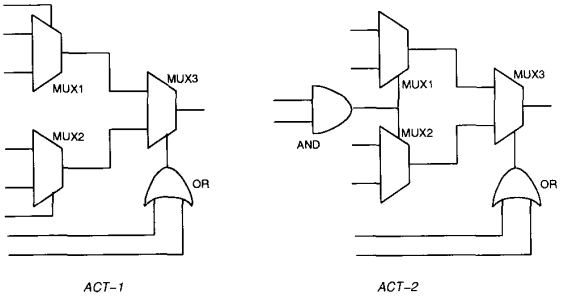


Fig. 12. Actel architectures.

We first present the most straightforward library-based approach and then review the more complex architecture-specific approaches.

B. Library-Based Technology Mapping

A library is created which has gates that represent all the functions obtained from the multiplexer-based logic block either by tying inputs to constants or by bridging some of them. Efficient algorithms that use BDD's can produce all nonequivalent functions[13] implemented by a MUX-based block in a fairly short time. However, the number of library functions may be large, although not as large as in the case of LUT-based architectures (706 for the ACT-1 as compared with 9014 for a four-input–one-output LUT).

Technology mapping algorithms based on dynamic programming are quite effective for libraries with one or two hundred gates, but are considered too slow for significantly larger libraries. Library reduction techniques are therefore

[13]The number of functions that can be implemented is very large, but several are equivalent and hence they need not be enumerated.

applied to reduce the number of gates. The least frequently used gates are removed. At Texas Instruments, the 766 gates of ACT-2 were reduced to 115 [34]. Experimental results showed a certain insensitivity with respect to the size of the library [34]. However, we believe that such reduction may impede significant optimizations.

An advantage of library-based mapping is that it is completely insensitive to changes in the logic block architecture. The only change that needs to be made is the creation of a new library. In addition, the same tool could be used for other target technologies.

C. Direct Approaches

In direct approaches no library is generated. The mapping is performed directly onto the logic blocks.

All proposed direct approaches for MUX-based architecture are quite similar [37], [17], [27], [35] and are not as different from the standard library-based approach as the direct approaches discussed earlier for LUT-based architectures. Since a BDD is simply a network of multiplexers and given the wealth of existing algorithms for manipulating and optimizing BDD's, BDD's have been used as the basis of most proposed direct approaches.

1) Using BDD's: The MIS-pga1 Approach

Overview: In MIS-pga1 [37], the dynamic programming approach to technology mapping is extended to pattern graphs and subject graphs described in terms of two-to-one multiplexers (BDD's) instead of two-input NAND gates.

The first step in this procedure is to represent each node function of the network with a BDD. As in standard technology mapping, the BDD is reduced to a forest of trees and then each tree is mapped.

If the structure of the logic block consists only of two-to-one multiplexers, then only a few pattern graphs are needed to characterize the "library" fully. For example only four patterns suffice for describing the simplified structure of the ACT-1 logic block where the OR feeding the output multiplexer has been removed. For more complex blocks, taking into consideration the nonhomogeneous structure can yield a larger set of pattern graphs (though never as large as the ones needed for the standard library approach described above), some of which are unusable by the dynamic programming approach since they are not trees. Thus a reduced set of patterns is used; in MIS-pga1 only eight patterns are considered. Covering of the subject-graph by patterns is done using dynamic programming. After an initial mapping, an iterative improvement phase is used. It consists of three main operations: partial collapsing, decomposition and phase assignment.

Building the BDD for the Subject Graph: It would be possible to build a BDD for the entire network in terms of its primary inputs. However, such a BDD may be very inefficient as a starting point for implementation since the structure of the initial network obtained by technology independent optimization would be lost. In addition, there may be cases for which a BDD is too large. For these reasons, the BDD's are only built for the functions stored at each node.

Two representations are actually used for each node: the reduced-ordered BDD and the BDD. It is well known that the size of the ROBDD for a function depends strongly on the ordering of the inputs. Since the problem of finding the optimum ordering is NP-complete, heuristics are used for this task. However, if a node function has only a few variables, it is worthwhile to generate exhaustively all orderings and choose the best, since the size of the representation is directly related to the size of the implementation.

To allow the implementation of this strategy, a decomposition step is performed first on the network to force all nodes to have at most k inputs (in MIS-pga1, the best value of k is found to be between 3 and 6). Note that this is similar to the LUT problem and in fact a similar sequence of algorithms for decomposition is tried.

There are some drawbacks for this procedure:

1) The limitation on the number of inputs is artificial.
2) The input ordering constraint imposed by the ROBDD may be too severe and may yield a poor result.
3) Nodes in the ROBDD may have multiple parents, and so the tree decomposition may yield many small trees thus reducing the power of the dynamic programming approach.

An alternative is to use BDD's where the sequence of variables in the graph is not forced to be the same for all vertices of the BDD. The goal in constructing these BDD's is to minimize the number of nodes as well as the number of nodes with multiple parents. This second goal is important to offer the maximum degree of freedom to the dynamic programming approach.

The algorithm for building the BDDs uses Shannon cofactoring repeatedly until all leaf functions are unate.[14] A minimum cover problem is then solved to find a good factored form representation of the unate function with respect to the architecture. This is in tune with the general strategy followed in logic minimization [7] where a generic function is decomposed with the Shannon cofactoring operation until a unate function is reached. In both cases, the variables for the cofactoring operations are chosen so that the leaves become unate quickly.

While this procedure remedies some of the drawbacks for ROBDD's mentioned above, it too has drawbacks. For example, there may be duplications in the branches of the BDD that would not have appeared in the ROBDD. Since it is not possible to tell *a priori* which representation yields the best result, in MIS-pga1 both are tried and the best result is selected.

Covering is performed on the forest of trees using dynamic programming. If the logic block is the simplified version of ACT-1 shown in Fig. 13, it can be proved that the four pattern graphs shown in Fig. 14 yield the optimum matching in a selected number of cases [37].

[14] A logic function is unate in a variable x if it depends only on x or its complement but not both. A function is unate if it is unate in all its variables.

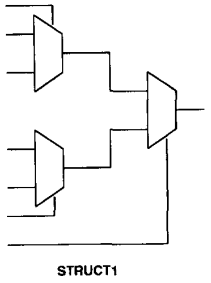


Fig. 13. Simplified ACT-1 architecture.

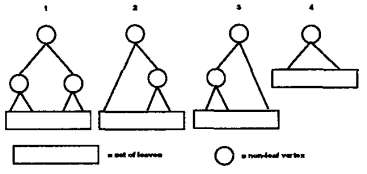


Fig. 14. Pattern graphs for MIS-pga1.

For the ACT-1 module, a set of eight pattern-graphs is sufficient, given that the subject-graph has no restriction on the number of times a variable may appear on the path from the root to a leaf node, and the covering procedure is exact. Note that if we use the cofactoring technique in constructing the subject-graph (which is the case with BDD's), this set of pattern-graphs is not sufficient. In fact, there are functions which can be realized with one ACT-1 module, but a BDD-based procedure will always use more than one module. One such function is $f = (a+b)(a'c+ab)+a'b'(kl+k'm)$ [35].

Iterative improvement: Since the algorithms used in the dynamic programming algorithm and in building the BDD representation are local in nature (the subject graph is broken into a forest of trees), an iterative improvement phase is used to improve the final results. The strategy used in MIS-pga1 is shown in Fig. 15.

The algorithm used in partial collapse is the same as in the case of LUT-based architectures in MIS-pga2 (see Fig. 10).

The decomposition phase selects nodes that have a fairly large number of fan-ins and decomposes them using the same approach as for LUT-based FPGA's. Only decompositions that reduce the cost are accepted.

The phase assignment algorithm operates on one node at a time and greedily selects the least cost polarity of the function associated with the node.

2) The Amap Approach: Karplus [27] proposed a quick algorithm that carries out a technology mapping into MUX-based architectures by mapping the network into *if-then-else* DAG's.

Here, the selector function at each vertex can be a function of inputs, rather than being only an input. When compared to BDDs this results in more freedom in the mapping phase. In addition, when this representation is built from a sum-of-products form by the cofactoring procedure, duplicate cubes are avoided. In the ITE representation, the

```
iterative_improvement(η) /*  η is a network */
{
     repeat {
          partial_collapse(η);
          decompose_nodes(η);
     } (until satisfied or no further improvement);
     phase_assignment(η);
}
```

Fig. 15. Iterative improvement.

if vertex corresponds to the select line of the multiplexer. The *then* and *else* children correspond to the branches taken when the *if* child evaluates to 1 and 0 respectively and are mapped to the inputs of the multiplexer.

Amap creates the ITE DAG and then preprocesses it to find an initial good local form for the mapping. In particular, single literal inputs are commuted to bring them to the *if* part so that the OR function of the ACT-1 module is better utilized. A quick phase assignment is also performed.

After this preprocessing, the final covering is carried out in a single pass with a greedy procedure. The procedure has to tradeoff the advantage of placing as much of the subject DAG as possible into a block and the disadvantage of hiding a signal that feeds several blocks. In the latter case, logic must be replicated and a larger implementation may result. In fact, if a signal that is shared by a number of vertices is not hidden in all the fan-outs, it has to be the output of a block. Thus pushing logic into a block would not provide any saving in this case.

The nodes of the ITE DAG are processed in a top-down fashion, starting from the primary outputs. Each node is then mapped into the output multiplexer of the ACT-1 architecture. In doing so, the use of the OR gate in the select input is made as efficient as possible. The *then* and the *else* children are then mapped to the input multiplexers. These multiplexers may not be fully utilized and may in fact be used merely as buffers if the variables corresponding to the *then* and the *else* children are already implemented or if they have high fan-out (in Amap a high fan-out is a fan-out of three or more).

After the mapping to the input multiplexers has been done, the output multiplexer is revisited to see whether a more compact representation exists by exploiting the actual function implemented by the block.

The entire procedure is recursively applied until all nodes are either primary inputs or they have been implemented in some block.

Since only a single pass is performed on the ITE DAG and the mapping is carried out locally, the algorithm is fast. The experimental results presented in [27] show that not much is lost with respect to the more complex optimization procedures of MIS-pga1 in terms of quality.

3) The Proserpine Approach: This approach follows the same general structure of the technology mapping algorithms of MIS and Ceres [31]. First, the network is partitioned into multiple trees, and the nodes of the network

are decomposed into two-input AND/OR gates to maximize the granularity of the network and to offer more freedom to the dynamic programming algorithm.

The basic difference lies in the way matching is performed. The algorithm does not require the explicit representation of the pattern graphs. Instead, it requires the representation of the "largest" logic function implementable by the basic block, i.e., the function computed by the structure with each input connected to a separate variable. The algorithm customizes the block with the correct operation during the matching process.

The set of functions that can be implemented by a MUX-based logic block corresponds to the set of functions that result from stuck-at and bridging faults. An input connected to a constant corresponds to a stuck-at fault and bridged inputs correspond to a bridging fault.

The stuck-at inputs belong to the set S, the ones that are bridged to the set B. Then the problem to be solved is: Given a function $F(y_1, ..., y_m)$ and the module function $G(x_1, ..., x_n)$ with $m \leq n$, find a stuck-at set S, a bridging set B and an ordering of the variables Ω such that F and $G_{SB\Omega}$ are functionally equivalent, i.e., there is a match for F in G.

The function F to match against the module function G is obtained by examining the nodes of the AND/OR network and collapsing them recursively. A number of different functions are created that are called *cluster* functions. For each cluster function matching is performed and the dynamic programming algorithm is used to minimize the block count.

Solving the matching problem is not easy especially when bridging is allowed. We will not review this case and refer the reader to the original papers on the subject [17], [5].

For the stuck-at faults, an ROBDD is built for the module function and for the cluster function. A sufficient condition for a match is that the ROBDD of the cluster function be isomorphic to a sub-graph of the module ROBDD. It is obvious that the part of the module ROBDD that does not correspond to the cluster function representation can be reduced by setting an appropriate set of inputs to 0 and/or 1. However, there are cases where a match exists but the cluster function ROBDD is not isomorphic to any sub-graph of the module function ROBDD. This is due to the fact that the orderings of the variables used to build the ROBDDs may not be compatible. Hence, to discover if a function matches, all possible variable orderings of the module function should be considered and the corresponding ROBDDs should be checked for isomorphism. Of course, this may be quite expensive and identical subgraphs in separate ROBDDs corresponding to different orderings may end up being checked a large number of times. In [17], a new structure called the *Global Boolean Decision Diagram*, GBDD, is proposed to make the matching algorithm faster. This structure is built by combining the BDDs corresponding to all the orderings. Combining the BDDs in an appropriate way removes all the duplications making the sub-isomorphism check much faster.

Given the complication of dealing with bridging faults, Proserpine first attempts to find a match with the GBDD as described above. If no matching is found, bridging is then considered. In [5], new bridging algorithms are described.

A few interesting observations were made after running Proserpine on a benchmark set. It was found that the bridging contributed insignificantly to improving the mapping (using the Ceres framework), and at most a single bridge is needed in the vast majority of cases.

The Proserpine approach is powerful in that it can consider any logic block where the programming mechanism allows the inputs to be connected to constants and/or bridged. As such, it is useful as an architecture-exploration tool.

4) The MIS-pga2 Approach MIS-pga2 [35] is based on the same general flow of MIS-pga1 but with some key modifications, some of which are borrowed from the other approaches presented above.

The overall algorithm is as follows:

1) Each node of the network is first mapped. If the function f at the node can be implemented by one block (using the matching algorithm described in [35]), the corresponding match is saved. Otherwise, an ITE of f is constructed. The ITE is covered by the pattern graphs of Figure 16 using dynamic programming [29]. Both the ITE and its cover are saved.
2) An iterative improvement phase using partial collapse and decomposition follows after the initial mapping. Partial collapse tries to collapse a node into all its immediate fanouts. If the sum of the new costs of the fanouts is less than the sum of the old cost of the node and the fanouts, the collapse is accepted. The new cost of a fanout is determined by remapping it using step 1. This process is repeated for all nodes of the network. Using `decomp -g` of MIS [8] a node is decomposed and the decomposed nodes are mapped. If the cost improves, the original node is replaced by its decomposition. Partial collapse and decomposition are repeated for some number of iterations.
3) Each node of the network is replaced by a set of nodes, each of which can be implemented using one basic block of the architecture. This is done by using the cover of the ITE at each node.
4) If the number of primary inputs of the network is small (say less than 10), an ROBDD is constructed for the network. This ROBDD is then mapped using the method described in [37]. If this mapping is better, it is accepted. Construction of ROBDDs helps when the circuit is symmetric.

The algorithm is applicable for both ACT-1 and ACT-2 modules. However, some architecture-specific changes have to be made. The main differences are in the matching algorithm and the pattern graph construction.

5) Comparisons and Observations: We present the results obtained using these approaches for a set of benchmark examples. These examples were optimized as in Section

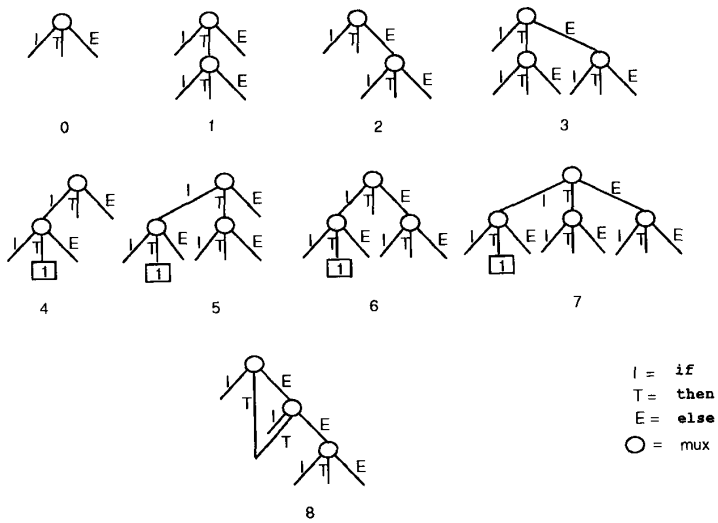


Fig. 16. Pattern graphs for MIS-pga2.

4.3.3. In Table 5, results are presented for MIS-pga2, MIS, Ceres, Amap and MIS-pga1 with the same starting networks. We also present results for Proserpine, although we do not know if the starting points are the same.[15] For MIS, a library containing around 90 gates was used, whereas for Ceres, a complete library for ACT-1 was used and depth of the search while covering was set to 5. MIS-pga2, MIS and MIS-pga1 were run on a DEC5500 (a 28 mips machine), whereas Amap was run on a SUN4/370 (a 12.5 mips machine). The option used for Amap was -an3. The options used for MIS-pga1 is act_map − h3 − n1 − q − d4 − f3 − M4 − l − g0.001. This means that both an ROBDD and an unordered BDD are constructed for each function. An optimum ROBDD is constructed for any function with at most four inputs. One iteration of the iterative improvement phase is executed. In the partial collapse routine, only the nodes with fanin no greater than three are considered for collapsing. In the decomposition routine, all nodes with fanin of four or more are considered. The phase assignment algorithm is also executed. Finally, a last-gasp routine is entered at the end [37]. This routine builds a network $\eta(n)$ from each node n of the final network η, where $\eta(n)$ has one internal node, one primary output and as many primary inputs as the number of fanins of n. It then performs technology decomposition on $\eta(n)$ and then applies mapping and the iterative improvement phase to get a network $\eta'(n)$. If the cost of $\eta'(n)$ is less than that of n, the routine replaces n by $\eta'(n)$ in η. MIS-pga2 used two iterations, performed a last-gasp, but did not do a quick phase.

Table 5 shows the number of ACT-1 modules needed to implement the benchmark, and the time taken in seconds in columns n and t respectively.

[15] Proserpine is not being distributed yet and the corresponding column in Table 5 has been taken from [17].

MIS-pga2 in general performs better than other systems. Some possible explanations are given below.

1. The reason why MIS-pga2 outperforms the MIS technology mapper is because the multiplexer representation of a function used in MIS-pga2 fits nicely into the multiplexer-based architecture.
2. Though Amap also uses ITEs, its way of construction is different. MIS-pga2 runs an iterative improvement phase and uses a matching algorithm prior to the construction of ITEs.
3. MIS-pga1 constructs BDD's and hence could replicate parts of the cubes in the 0 and 1 branches. However, by exhaustively generating all reduced ordered BDD's for a function (if it has at most four inputs), MIS-pga1 is able to achieve better results on many examples. For the same reason, it is many times slower.

However, there are benchmarks (e.g., *C1908, C499*) where Amap gives better results. This is due to the different mapping technique it uses.

VI. Wide-AND/OR Architectures

Wide-AND/OR architectures are extensions of the standard two level PLD architectures. The complexity of the logic blocks which are general PLAs with several inputs (from about 20 to 100) connected together by some kind of bus structure is high. The logic synthesis problem is similar to the logic optimization problems encountered in PLA design. Most of the proprietary systems are based on two-level logic optimization programs with some help for decomposition.

To the best of our knowledge, the only paper that deals with the aspects of Wide-AND/OR arrays with a novel approach is [32].

Table 5 number of ACT-1 Blocks: n Number of ACT-1 Blocks; and t Run Time in Seconds

example	MIS-pga2		MIS		Ceres	Amap		MIS-pga1		proserpine[2]
	n	t	n	t	n	n	t	n	t	n
z4ml	19	9.4	20	2.0	17	20	0.8	16	19.6	-
misex1	18	6.0	22	1.9	22	25	1.1	20	11.5	25
vg2	35	3.1	47	3.9	42	44	1.5	36	18.4	46
5xp1	40	10.4	51	4.4	47	42	1.7	45	60.0	53
count	39	8.6	63	6.7	62	41	1.5	46	13.2	-
9symml	26	50.3	73	10.5	-[3]	74	2.7	80	3123.8	-
9sym	26	55.8	99	14.1	136	106	4.1	119	17582.8	-
apex7	95	30.7	113	10.6	106	104	3.8	96	42.1	121
C1908	168	121.5	188	19.3	192	158	7.6	175	646.5	-
rd84	50	30.7	62	6.5	61	62	2.6	61	151.4	70
e64	94	7.1	95	7.9	95	105	3.3	94	3.9	-
C880	169	36.1	175	18.2	177	190	7.2	171	77.2	-
apex2	112	36.7	106	11.8	175	122	5.0	124[1]	8.3	-
alu2	185	48.9	193	24.2	173	188	8.3	208	824.5	-
duke2	165	29.3	176	17.9	172	175	6.9	166	403.5	177
C499	166	55.5	174	17.8	166	136	7.0	166	35.3	170
rot	285	67.2	313	28.9	418	335	11.9	288	1071.8	465
apex6	282	76.6	360	34.4	441	392	15.0	289	255.5	396
alu4	121	19.8	149	31.6	326	160	6.1	132	145.9	350
des	1351	357.3	1571	756.6	1638	1634	67.2	1749[1]	762.9	-
sao2	51	24.4	52	8.0	86	56	2.0	62	54.2	-
rd73	30	12.1	32	2.7	32	32	1.6	31	37.2	-
misex2	40	3.5	46	4.3	42	47	1.5	41	6.1	45
f51m	44	14.4	52	5.2	54	56	2.2	48	36.6	63
clip	48	25.1	57	4.9	62	60	2.5	51	91.9	73
bw	60	11.7	81	7.1	61	83	3.6	65	20.1	67
b9	66	19.7	64	5.9	101	81	2.7	65	31.6	-
C5315	591	359.8	704	88.0	725	653	26.9	656	673.1	-

[1]Used `"act_map-n2-h2-d4-f3-g0.001"` and `"act_map-h3-M4"` since the default command timed out.
[2]Starting networks differ from other systems.
[3]Segmentation fault.

This approach is targeted to a general architecture that has as a logic block an AND plane of a PLA whose outputs (ORs of rows of the AND plane) are fed into a set of simple gates and hence implements three-level logic.[16]

The approach was applied to the architecture offered by PlusLogic, where the simple gates are two-input gates that can implement any logic function of two inputs.

The basic algorithm of [32] restricts the use of the two-input logic to AND gates. The simplified optimization problem solved by [32] is as follows. Given a logic function F, find two PLAs, PLA_1 and PLA_2, so that if g_1 and g_2 are the sum-of-products form of the logic implemented by the PLA's, then g_1g_2 covers F and the total number of cubes is minimized.

If F is incompletely specified, then g_1 and g_2 must satisfy the following conditions to be valid:

- let f be the on-set of F. Then, $f \subseteq g_1, f \subseteq g_2$.
- Let r be the off-set of F. Then $rg_1g_2 = \emptyset$.

[16]There is some evidence that three-level logic has advantages over two-level logic from the point of view of compactness of representation and over multilevel logic from the point of view of speed. Sasao [46] argued that little is gained by going to more than three levels even though this view is not universally shared.

The algorithm proposed in [32] is a heuristic that produces g_1, g_2 so that the conditions above are satisfied.

The algorithm structure is as follows:

1) Choose an initial g_1, g_1^0 that contains f.
2) Using g_1^0, obtain a minimal g_2 so that $g_1^0g_2$ satisfies the conditions.
3) Using g_2, obtain a minimal g_1 that satisfies the conditions.
4) Iterate for several choices of g_1^0 and pick the pair that has minimum size.

The key point in the algorithm is the minimization step for g_2 given g_1 (or *vice versa*). This minimization is carried out with Espresso [7] on appropriate incompletely specified functions that are obtained exploiting the degree of freedom offered by the structure of the implementation.

To extend the result to the more powerful output logic offered by PlusLogic, the algorithm goes through a phase assignment procedure that optimizes the use of the output logic for all cases except for the exclusive OR and exclusive NOR functions that cannot be reduced to the optimal phase selection.

The algorithm has been shown to produce very good results as compared with the implementation of the logic using only two-level PLA's.

A final remark is that here also the algorithm is able to cope well with structures that are simpler than those offered by the commercially available architectures. Logic synthesis algorithms are more effective if the logic is uniform and simple.

VII. Conclusions

We have reviewed logic synthesis algorithms and methosa for FPGA's. The paper focused on FPGA's with large-granularity logic blocks, since these yield design problems that are sufficiently different from the standard logic synthesis problems.

We believe that logic synthesis is an essential step in the design of FPGA's. The commercially available architectures offer difficult challenges for algorithm designers. The algorithms developed so far are mainly targeted towards the minimization of the number of logic blocks used. Only a few deal with the optimization of performance and routability.

We expect that in the future more powerful algorithms will emerge that can also effectively take into consideration performance constraints and the scarcity of interconnect resources. While FPGA and tool vendors offer some limited logic synthesis capabilities now, they will ultimately offer more sophisticated logic synthesis tools for the most commonly available architectures. We expect that tool vendors will offer logic synthesis environments where it will be easy to go from one FPGA architecture to another and from one ASIC style to another. We also expect to see new architectures that are designed with logic synthesis in mind so that optimization algorithms can be more effective.

Much work remains to be done to deal with sequential logic synthesis. All architectures offer a number of sequential elements. It is important to evaluate whether the number and type of sequential elements offered is good especially in view of the use of sequential logic synthesis. Timing and retiming of sequential circuits in the presence of a fixed (and possibly large) number of sequential elements is an interesting problem.

Ultimately logic synthesis will be extended to multiple chip systems. The problem of partitioning logic into multiple chips is a general problem for all ASIC styles but it is particularly relevant for FPGA's given the constraints on resources. While netlist partitioning algorithms are available not much is available to partition a design at a higher level of abstraction. We expect to see a number of partitioning approaches of this kind to appear shortly.

Acknowledgment

The authors would like to thank R. Murgai, N. Shenoy and Dana How for editorial help and useful comments on an earlier version of the paper. This work is partially supported by DARPA under contract numbers J-FBI-90-073 (for the first author) and J-FBI-89-101 (for the second author).

References

[1] P. Abouzeid, K. Sakoutan, G. Saucier,. and F. Poirot, "Multi-level synthesis minimizing the routing factor," in *Proc. Design Automation Conference*, ACM-IEEE, June 1990, pp. 365–368.

[2] A. V. Aho and J. D. Ullman, *Principles of Compiler Design.* NewYork: Addison-Wesley, 1977.

[3] R. L. Ashenhurst, "The decomposition of switching functions," in *Proc. Int. Symp. Theory of Switching Functions*, 1959.

[4] K. Bartlett, D. Bostick, G. Hachtel, R. Jacoby, M. Lightner, M. Moceyunas, C. Morrison, and D. Ravenscroft, "Bold: A multi-level logic optimization system," in *Proc. Int. Conf. Computer-Aided Design*, 1987.

[5] A. Bedarida, S. Ercolani, and G. DeMicheli, "A new technology mapping algorithm for the design and evaluation of electrically programmable gate arrays," in *1st Int. ACM/SIGDA Workshop on FPGAs*, 1992.

[6] R. K. Brayton, N. Brenner, C. Chen, G. Hachtel, C. McMullen, and R. Otten, "The Yorktown silicon compiler," in *Proc. Int. Symp. Circ. Syst. (ISCAS-85)*, 1985, pp. 391–394.

[7] R. K. Brayton, C. McMullen, G. D. Hachtel, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[8] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A multiple-level logic optimization system," in *IEEE Trans. Computer-Aided Design*, pp 1062–1081, 1987.

[9] R. K. Brayton, A. Sangiovanni-Vincentelli, and G. D. Hachtel, "Multi-level logic synthesis," *Proc. IEEE*, pp. 264–300, Feb. 1990.

[10] R. Bryant, "Graph based algorithms for. Boolean function manipulation," in *IEEE Trans. Computers*, pp. 667–691, 1986.

[11] J. Cong, A. Kahng, and P. Trajmar, "Graph based fpga technology mapping for delay optimization," *1st Int. ACM/SIGDA Workshop on FPGAs*, 1992.

[12] H. A. Curtis, "A generalized tree circuit," *J. ACM*, 1961.

[13] J. Darringer, W. Joyner, J. Gerbi, L. Berman, and L. Trevillyan, "LSS: A System for Production Logic Synthesis," *IBM J. Res. Development*, pp. 537–545, 1984.

[14] G. DeMicheli, "Performance oriented synthesis of large scale domino CMOS circuits," *IEEE Trans. Computer-Aided Design*, pp. 751–765, 1987.

[15] W. E. Donath, "Statistical properties of the placement of a graph," *SIAM J.*, vol. 16, no. 2, pp. 439–457, Apr. 1968.

[16] W. C. Elmore, "The transient response of damped linear networks with particular regard to wideband amplifiers," *J. Appl. Phys.*, pp. 55–63, 1948.

[17] S. Ercolani and G. DeMicheli, "Technology mapping electrically programmable gate arrays," in *Proc. Design Automation Conf.*, 1991, pp. 234–239.

[18] D. Filo, J. C. Yang, F. Mailhot, and G. D. Micheli, "Technology mapping for a two-output ram-based field-programmable gate arrays," in *European Design Automation Conf.*, 1991, pp. 534–538.

[19] R. J. Francis, J. Rose, and K. Chung, "Chortle: A technology mapping program for lookup table-based field. programmable gate arrays," in *Proc. Design Automation Conf.*, 1990, pp. 613–619.

[20] R. J. Francis, J. Rose, and Z. Vranesic, "Chortle-crf: Fast technology mapping for lookup table-based fpgas," In *Proc. Design Automation Conf.*, 1991, pp. 227–233.

[21] R. J. Francis, J. Rose, and Z. Vranesic, "Technology mapping of lookup table-based fpgas for performance," in *Proc. Int. Conf. Computer-Aided Design*, 1991, pp. 568–571.

[22] M. Fujita, and Y. Matsunaga, "Multi-level logic minimization based on minimal support and its application to the minimization of look-up table type fpgas," in *Proc. Int. Conf. Computer-Aided Design*, 1991.

[23] M. R. Garey and D. S. Johnson, *Computers and Intractability.* New York: W. H. Freeman and Co., 1979.

[24] D. Gregory, K. Bartlett, A. deGeuss, and G. Hachtel, "Socrates: A system for automatically synthesizing and optimizing combinational logic," in *Proc. Design Automation Conference*, 1986, pp. 79–85.

[25] C. Halatsis and N. Gaitanis, "Irredundant normal forms and minimal dependence sets of a boolean function," *IEEE Trans. Computers*, Nov. 1978, pp. 1064–1068,.

[26] R. M. Karp and J. P. Roth, "Minimization over Boolean graphs," in *IBM J. Res. and Development*, Apr. 1962.

[27] K. Karplus, "Amap: A technology mapper for selector-based field-prorammable gate arrays," *Proc. Design Automation Conf.*, pp. 244–247, 1991.

[28] K. Karplus, "Xmap: A technology mapper for table-lookup field-programmable gate arrays," in *Proc. Design Automation*

Conference, 1991, pp. 240–243.
[29] K. Keutzer, "Dagon: Technology binding and local optimization by DAG matching," in *Proc. Design Automation Conference*, 1987, pp. 341–347.
[30] E. L. Lawler, K. N. Levitt, and J. Turner, "Clustering to minimize delay in digital networks," in *IEEE Trans. Comput.*, pp. 47–57, Jan. 1969.
[31] F. Mailhot and G. DeMicheli, "Technology mapping using boolean matching and don't care sets," in *European Design Automation Conf.*, 1990, pp. 2120–216.
[32] A. A. Malik, D. Harrison, and R. K. Brayton, "Three-level decomposition with application to plds," *Proc. Int. Conf. Computer Design*, pp. 628–633, 1991.
[33] M. J. Mathony, "Universal logic design algorithm and its application to the synthesis of two-level switching circuits," in *IEE Proc.*, 1989.
[34] M. Mehendale, C. H. Shaw, and D. Wilmoth, "ALFA: Automatic library generation for logic module based FPGA's," in *1st Int. ACM/SIGDA Workshop on FPGAs*, 1992.
[35] R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli, "An improved synthesis algorithm for multiplexor-based PGAs," in *1st Int. ACM/SIGDA Workshop on FPGAs*, 1992.
[36] R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Sequential synthesis for table look up PGA's," in *Euro ASIC*, 1992.
[37] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton, and A.Sangiovanni-Vincentelli, "Logic synthesis for programmable gate arrays," in*Proc. Design Automation Conf.*, 1990, pp. 620–625.
[38] R. Murgai, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Improved logic synthesis algorithms for table look up architectures," in *Proc. Int. Conf. Computer-Aided Design*, 1991, pp. 564–567.
[39] R. Murgai, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Performance directed synthesis for table look up pragrammable gate arrays," in *Proc. Int. Conf. Computer-Aided Design*, 1991, pages 572–575.
[40] M. Pedram, and N. Bhat, "Layout Driven Logic Restructuring/Decomposition," in *Proc. Int. Conf. Computer-Aided Design*, 1991, pp. 134–137.
[41] M. Pedram and N. Bhat, "Layout driven technology mapping," in *Proc. Design Automation Conf.*, 1991, pp. 99–105.
[42] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli, "A classification and survey of field-programmable gate array architectures," in *Proc. IEEE*,vol. 81, no 7, July 1993.
[43] J. Rubinsein, P. Penfield, and M. A. Horowitz, "signal delay in rc tree networks," *IEEE Trans. CAD*, pp. 119–127, July 1983.
[44] R. Rudell, *Logic Synthesis for VLSI Design.* PhD thesis, Univ. California, Berkeley, 1989.
[45] A. Saldanha, R. K. Brayton, and A. Sangiovanni-Vincentelli, and C. Kwang-Ting, "Timing optimization with testability considerations," in *Proc. Int. Conf. Computer-Aided Design*, pp. 460–463, 1990.
[46] T. Sasao, "On the complexity of Three-Level Logic Circuits," in *Proc. MCNC Int. Workshap on Logic Synthesis*, Research Triangle Park, NC, May 1989.
[47] H. Savoj, M. Silva, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Boolean Matching in Logic Synthesis," In *European Design Automation Conf.*, pp. 168–174, 1992.
[48] P. Sicard, M. Crastes, K. Sakouti, and G. Saucier, "Automatic synthesis of boolean functions on Xilinx. and actel programmable devices," in *Euro ASIC*, pp. 142–145, May 1991.
[49] K. J. Singh, A. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Timing optimization of combinational logic," in *Proc. Int. Conf. Computer-Aided Design*, 1988, pp. 282–285.
[50] N-S. Woo, "A heuristic method for FPGA technology mapping based on edge visibility," in *Proc. Design Automation Conf.*, ACM-IEEE, June 1991, pp. 248–251.
[51] N-S. Woo, "ATOM: Technology Mapping of Sequential Circuits for Lookup Table-based FPGAs," In *Design Automation Conf.*, 1992, submitted for publication.
[52] *The Programmable Gate Array Book*, Xilinx Inc., 2069, Hamilton Ave., San Jose, CA-95125.

Alberto Sangiovanni-Vincentilli (Fellow, IEEE), for a photograph and biography please see this issue of the PROCEEDINGS.

Abbas El Gamal (Senior Member, IEEE), for photograph and biography please see the Prolog to the Special Section in this issue of the PROCEEDINGS.

Jonathan Rose (Member, IEEE), for a photograph and biography please see this issue of the PROCEEDINGS.