Series-Parallel Functions and FPGA Logic Module Design

SHASHIDHAR THAKUR and D.F. WONG
University of Texas at Austin

The need for a two-way interaction between logic synthesis and FPGA logic module design has been stressed recently. Having a logic module that can implement many functions is a good idea only if one can also give a synthesis strategy that makes efficient use of this functionality. Traditionally, technology mapping algorithms have been developed after the logic architecture has been designed. We follow a dual approach, by focusing on a specific technology mapping algorithm, namely, the structural tree-based mapping algorithm, and designing a logic module that can be mapped efficiently by this algorithm. It is known that the tree-based mapping algorithm makes optimal use of a library of functions, each of which can be represented by a tree of AND, OR, and NOT gates (series-parallel or SP functions). We show how to design a SP function with a minimum number of inputs that can implement all possible SP functions with a specified number of inputs. For instance, we demonstrate a seven-input SP function that can implement all four-input SP functions. Mapping results show that, on an average, the number blocks of this function needed to map benchmark circuits are 12% less than those for Actel's ACT1 logic modules. Our logic modules show a 4% improvement over ACT1, if the block count is scaled to take into account the number of transistors needed to implement different logic modules.

Categories and Subject Descriptors: B.6.1 [**Logic Design**]: Design Styles; B.6.3 [**Logic Design**]: Design Aids, B.7.1 [**Integrated Circuits**]: Types and Design Styles; J.6. [**Computer Applications**]: Computer-Aided Engineering

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Field Programmable gate arrays, series-parallel technology mapping, tree-based technology mapping algorithm, universal logic modules

1. INTRODUCTION

FPGAs are composed of programmable logic and routing resources. Arbitrary circuits are implemented by appropriately configuring these resources. This article concentrates on the problem of designing the combina-

This work was partially supported by the Texas Advanced Research Program under grant 003658459, by a DAC Design Automation Scholarship, and by a grant from the AT&T Bell Laboratories.
Authors' address: S. Thakur and D.F. Wong, Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712-1188.

tional portion of logic modules. The combinational logic in most existing commercial FPGAs can be broadly classified as lookup-table based [AT&T 1993; Xilinx Corp. 1994], PLA based [Altera Corp. 1993], or macro-cell based [Actel Corp. 1994]. We design logic modules that fall in the third category.

The technology mapping step in CAD is responsible for mapping circuits to the FPGA architecture. This step involves identifying clusters in the circuit that can be implemented by a single logic module, and covering the circuit by such clusters. Designing a complex logic module that can implement many different functions is a good idea only if one can come up with a mapping algorithm that can utilize the functionality so as to offset the silicon costs associated with the complexity of the logic module. Typically the best mapping algorithms for logic modules are discovered *after* the architecture design has been done.[1] The issue of having a two-way interaction between logic module design and synthesis was stressed at the recent FPGA Symposium [ACM 1995]. This approach is expected to lead to the design of logic modules that can be synthesized efficiently. Our approach presents the first instance of logic module design, where the mapping algorithm is the starting point.

The structural tree-based mapping algorithm [Detjens et al. 1987; Rudell 1989] is very popular, and forms the core of many academic and commercial technology mapping tools. This algorithm decomposes the unmapped logic network into a collection of trees. The library is represented as a set of graphs. Each component tree is then optimally mapped to the library using graph matching techniques. Due to the decomposition into trees, matches are identified only for those library functions that have a representation in the form of a tree of gates (we call such functions series-parallel or SP functions). The algorithm is optimal for libraries restricted to such functions. For a FPGA logic module, the library is the set of functions that can be implemented using one such logic module. We show how to design logic modules that provide a large library of SP functions. Thus such logic modules are mapped efficiently using the tree-based mapping algorithm. We refer the reader to Rudell [1989] for a discussion on mapping using a library of SP functions.

SP functions have the added advantage of having simple and compact CMOS implementations. Hence we aim at designing logic modules that are SP.

There has been recent work on designing FPGA logic modules based on the concept of universal logic modules (ULMs) [Lin et al. 1994, Thakur and Wong 1995]. A function is said to be a ULM for m-input functions if it can implement all m-input functions [Patt 1973; Preparata 1971; Preparata and Muller 1970]. In particular, Lin et al. [1994] give a BDD-based computational procedure to derive ULMs. In contrast, the procedure given by Thakur and Wong [1995] is algebraic, and works for arbitrary values of m. Both methods use a synthesis strategy based on existing mappers for

[1] See Cong and Ding [1994]; Karplus [1991a, 1991b]; and Murgai et al. [1992, 1991].

Table I. Number of Nonequivalent Functions

Inputs	Total number of functions	Number of SP functions
2	4	1
3	14	2
4	222	5
5	616,126	12

lookup-tables. Targeting only the SP functions reduces the functionality, and hence the complexity, of the logic module. Table I compares the total number of *NPN* nonequivalent[2] functions with the number of *NPN* nonequivalent SP functions for various input sizes. It can be seen that the number of SP functions is only a small fraction of the total number of functions.

We show how to implement the logic modules that we design, so as to further optimize their utilization. This is done by decomposing the logic module into two smaller functions. Thus each logic module can implement two functions, allowing multiple nodes in the mapped network to be packed into one logic module.

In particular, we demonstrate a seven-input SP function that can implement all four-input SP functions. A logic module based on this function requires two more transistors than Actel's ACT1 logic module. Mapping results show that, on an average, the number of blocks of this logic module needed to map benchmark circuits are 12% less than those for ACT1. Thus a more efficient use of the silicon area available is made by our logic module.

The rest of the article is organized as follows. We introduce the notation and specify the problem in Section 2. We study the properties of SP functions in Section 3, and formulate the problem of deriving a SP function that can implement all SP functions with a specified number of inputs as a problem on graphs. We solve this problem for practical cases. In Section 4 we show how to use the tree-based mapping algorithm for mapping to our logic modules. We also show how to implement these logic modules to further optimize their utilization. We describe the experiments and their results in Section 5.

2. DEFINITIONS AND PROBLEM SPECIFICATION

We denote the set of m-input Boolean functions by $\mathcal{F}_m$. These functions are assumed to have all m inputs in their supports. The complement of a Boolean function f is denoted by f'. We denote a literal of a variable x by x^* when the specific phase of it is irrelevant.

Two functions, f, $g \in \mathcal{F}_m$, are said to be P equivalent if f can be transformed to g by applying a permutation to the inputs of f. Similarly, f and g are *NPN* equivalent if f can be transformed to g by some combination

[2] Two functions are *NPN* equivalent if one can be transformed to the other by a combination of input permutation and input/output complementation.

of input permutations, input complementation, and output complementation. These function equivalences induce equivalence relations on $\mathcal{F}_m$. These equivalence relations partition $\mathcal{F}_m$ into equivalence classes. These classes are called the equivalence classes under P and NPN, respectively. Thus the set of functions that are P equivalent to f are called the P equivalence class of f, and so on.

For $n \geq m$, we say that a function $u(z_1, z_2, \ldots, z_n)$ *covers* a function $f(x_1, x_2, \ldots, x_m)$ if u can be transformed to f by:

(1) Assigning a value from $\{0, 1, x_1, x_1', \ldots, x_m, x_m'\}$ to each of $z_1, z_2, \ldots, z_n$.
(2) Optionally complementing the output of u.

This transformation is called the *specialization* of u. Thus if inverters are available for free, then the function u can implement exactly those functions that it covers.

The following claim follows easily from the definitions:

LEMMA 2.1 *If a function $u(z_1, z_2, \ldots, z_n)$ covers $f(x_1, x_2, \ldots, x_m)$ then it covers every function that is NPN equivalent to f.*

We now formally define *series-parallel* (SP) *functions*. Any function of at most one input is SP. If f and g are two SP functions with a disjoint support, then $f + g$ and $f * g$ are SP functions. We represent a function $f(x_1, x_2, \ldots, x_m)$ by a *logic expression* $F(*, +, x_1, x_2, \ldots, x_m)$. The logic expression for a SP function has the property that every input variable appears exactly once, in some phase, in the expression. The term SP arises from the fact that such functions can be implemented by series-parallel networks of transistors.

It is an obvious fact that the logic expression $F(*, +, x_1, x_2, \ldots, x_m)$, for a SP function f, can be represented by a *labeled tree* such that:

(1) The internal nodes are labeled AND(*) or OR(+) and have at least two children each. The node labels alternate between AND and OR on any path from the root to the leaves.
(2) The tree has m leaf nodes. Each leaf node is labeled by one of $\{x_1, x_1', \ldots, x_m, x_m'\}$ such that each variable, in some phase, appears as a label exactly once.

For a given SP function this labeled tree is unique (up to isomorphism). Due to the exact correspondence with the logic expression for f, we denote the tree corresponding to f by $F(*, +, x_1, x_2, \ldots, x_m)$ too. The meaning is obvious from context. The *unlabeled tree* corresponding to the SP function f is the tree obtained by eliminating the node labels on $F(*, +, x_1, \ldots, x_m)$. Clearly any SP function yields a unique (up to isomorphism) unlabeled tree, but an unlabeled tree may yield different SP functions (by differently labeling the nodes).

Example 2.1 The multiplexer function $g(x_1, x_2, x_3) = x_1'x_2 + x_1x_3$ is not SP. But the function $f(x_1, x_2, x_3) = x_1'(x_2 + x_3)$ is SP and represented by the tree $F(*, +, x_1, x_2, x_3)$ in Figure 1(a). The corresponding unlabeled tree is

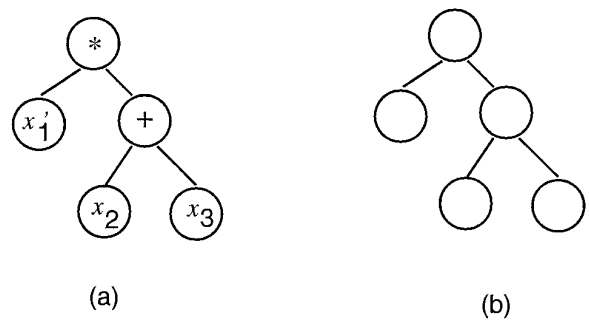

Fig. 1. Trees repesenting $x_1'(x_2 + x_3)$: (a) labeled tree; (b) unlabeled tree.

shown in Figure 1(b). Labeling the unlabeled tree differently yields the function $x_1 + x_2x_3$.

A function $u(z_1, z_2, \ldots, z_n)$ is said to be a *Universal Logic Module* for m-input SP functions if f covers every SP function with at most m inputs. We say that such a function is *SP-ULM.m*.[3] We shall address the following problem:

Logic Module Design Problem. For a given value of m, find a SP function f that is *SP-ULM.m*, and has a minimum number of inputs.

3. DERIVING *SP-ULM* FUNCTIONS

We first establish certain crucial properties of the *NPN* classes containing SP functions. We show that any such class can be uniquely specified by one unlabeled tree. Next we characterize the set of m-input SP functions covered by an n-input SP function, for $n > m$. Together, these two results let us formulate the problem of designing a *SP-ULM.m* function as a problem on unlabeled trees. We give solutions to this problem for practical cases.

The following claim is easy to see.

LEMMA 3.1 *If $u(z_1, z_2, \ldots, z_n)$ covers all* SP *functions with m inputs, then it covers all* SP *functions with at most m inputs.*

PROOF. Consider a function $h(x_1, x_2, \ldots, x_{m-1})$ with $m - 1$ inputs. The function $f(x_1, x_2, \ldots, x_m) = h(x_1, x_2, \ldots, x_{m-1}) + x_m$ is clearly a SP function with m inputs and hence is covered by u. Consider the specialization of u done to cover f. Replacing assignments of any of the variables $z_1, z_2, \ldots, z_n$ to x_m by 0 and to x_m' by 1 will result in h. Hence u covers h. The result follows by induction. □

Hence we concentrate on designing a SP function that covers all functions with m inputs. By the preceding lemma, this is enough to guarantee that a function is *SP-ULM.m*.

[3] This particular notation arises from classical work on universal logic modules in which a function is called *ULM.m* if it covers *all m* input functions.

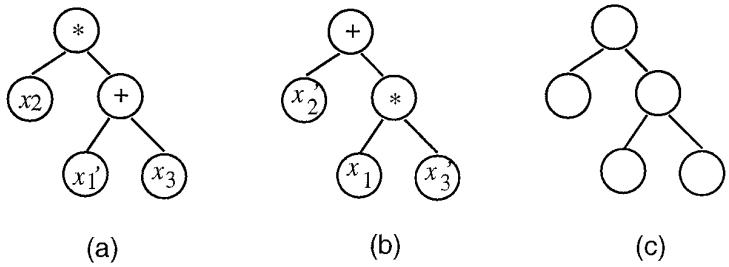

Fig. 2. Trees representing *NPN* equivalent functions (a) tree for $g = x_2(x_1' + x_3)$; (b) tree for g'; (c) unlabeled tree for *NPN* class.

Consider some function $f(x_1, x_2, \ldots, x_m) \in \mathcal{F}_m$ and let $F(*, +, x_1, x_2, \ldots, x_m)$ denote the logic expression representing f. The expression obtained by changing all the + operators to $*$ and vice versa is $F(+, *, x_1, x_2, \ldots, x_m)$. This corresponds to the function $d_{f(x_1, x_2, \ldots, x_m)}$—the *dual* of f. By De Morgan's law, the complement of f is given by:

$$f'(x_1, x_2, \ldots, x_m) = d_{f(x_1', x_2', \ldots, x_m')},$$

and is represented by the expression $F(+, *, x_1', x_2', \ldots, x_m')$.

Let $(x^*_{\sigma(1)}, x^*_{\sigma(2)}, \ldots, x^*_{\sigma(m)})$ be obtained by applying a permutation σ to $(x_1, x_2, \ldots, x_m)$, followed by a complementation of some of the variables. Then the function $f(x^*_{\sigma(1)}, x^*_{\sigma(2)}, \ldots, x^*_{\sigma(m)})$ is represented by the expression $F(*, +, x^*_{\sigma(1)}, x^*_{\sigma(2)}, \ldots, x^*_{\sigma(m)})$.

From the two preceding observations we can conclude that, for a SP function f, all the members of the *NPN* class of f are SP. From the manipulation of logic expressions, we can see that the labeled trees corresponding to a function *NPN* equivalent to f can be obtained from that of f by swapping the + and $*$ labels and/or relabeling the leaf nodes by a permutation of $x_1, x_2, \ldots, x_m$ in arbitrary phases. Thus the unlabeled trees for f and g are identical. This implies that the entire *NPN* class of f can be represented by a single unlabeled tree with m leaves. This conclusion is stated in Lemma 3.2.

LEMMA 3.2 *For two* SP *functions, $f, g \in \mathcal{F}_m$, f is NPN equivalent to g if and only if the unlabeled trees corresponding to f and g are identical.*

Example 3.1 Consider the function $f(x_1, x_2, x_3) = x_1'(x_2 + x_3)$. It is represented by the labeled tree in Figure 1. The functions $g = x_2(x_1' + x_3)$ and $g' = x_2' + x_1x_3'$ are SP also and represented by the labeled trees in Figures 2(a) and (b), respectively. The corresponding unlabeled tree, from which the tree representing any member of the *NPN* equivalence class of f can be derived, is shown in Figure 2(c).

An immediate implication of the preceding lemma is that the number of *NPN* equivalence classes of SP functions equals the number of unlabeled trees with m leaves. A formula for this was computed in Knuth [1973, problem 2.3.4.4.5] and was used in determining the entries of Table I.

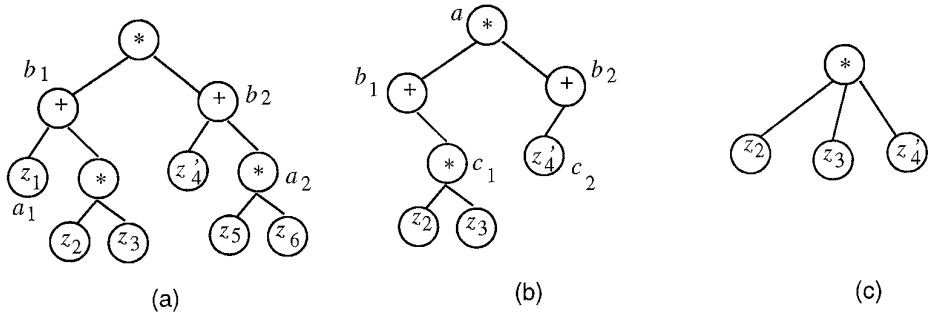

Fig. 3. Collapsing a labeled tree: (a) label tree for u; (b) result of chopping; (c) result of contraction.

Next we characterize the set of functions that can be covered by a SP function. Recall that we want to derive a logic module which is itself a SP function. We first show a simple lemma that asserts that any SP function can always be specialized to cover the functions 0 and 1.

LEMMA 3.3 *Any* SP *function covers the functions 0 and 1.*

PROOF. Consider some SP function $h(z_1, z_2, \ldots, z_k) \in \mathcal{F}_k$. Suppose the input variables appear in the phases $z_1^*, z_2^*, \ldots, z_k^*$ in the logic expression for h. Let D be some indeterminate. Setting $z_1^* = z_2^* = \cdots = z_k^* = D$ results in $h(z_1, z_2, \ldots, z_k) = D$. Choosing $D = 0$ or 1 results in a specialization of h that covers the functions 0 and 1, respectively. □

We define the operation of *collapsing* a tree (labeled or unlabeled) as a composition of an arbitrary number of applications of the following operations:

(1) *Chopping:* Two nodes a and b in the tree, such that a is a child of b, are selected and the entire subtree rooted at a and the edge between a and b are eliminated.
(2) *Contraction:* An internal node b, which has parent a and a single child c, is selected. Node b is eliminated from the tree. If c is an internal node, then the children of c are made children of a and c is eliminated. If c is a leaf, then it becomes a child of a.

The preceding definitions are illustrated by the following example:

Example 3.2 Consider the SP function:

$$u(z_1, z_2, \ldots, z_6) = (z_1 + z_2 z_3)(z_4' + z_5 z_6).$$

The labeled tree for this function is shown in Figure 3(a). The result of chopping the subtrees rooted at a_1 and a_2 is shown in Figure 3(b). Two contractions at nodes b_1 and b_2 result in the tree in Figure 3(c).

Note that the contraction operation is not possible unless preceded by a chopping operation because all the internal nodes in the tree initially have at least two children.

Consider a SP function $u(z_1, z_2, \ldots, z_n)$ and let $U(*, +, z_1, z_2, \ldots, z_n)$ be the labeled tree representing it. In the description of chopping, if the node a is a leaf node with label z_i^*, then chopping the subtree rooted at a is equivalent to:

—assigning 0 to z_i^*, if b is an OR node.

—assigning 1 to z_i^*, if b is an AND node.

If a is an internal node, then the chopping can be done by specializing the function represented by the subtree rooted at a to:

—0 if a is labeled AND (and b is labeled OR).

—1 if a is labeled OR (and b is labeled AND).

Because the subtree rooted at a represents a SP function, Lemma 3.3 implies that these specializations can always be done. The sets of variables appearing as labels of disjoint subtrees of U are disjoint, hence chopping two disjoint subtrees can be done without any conflicting assignments to variables. The contraction at b is equivalent to replacing a single input AND or OR node by a wire. If node c is an internal node, then nodes a and c lie on adjacent levels of the new tree and have the same label. To repair this, the inputs of a and c are merged to create one node. In Example 3.2 the inputs of node c_1 become the inputs of node a after the contractions at b_1. Hence we conclude that any collapsing operation, on a labeled tree U, can be done by assigning 0 or 1 to the literals at some of the leaves of U. For example, the collapsing described in Example 3.2 is equivalent to the assignments $z_1 = 0$ and $z_5 = z_6 = 0$.

We observe that both preceding operations preserve the property of node labels alternating between AND and OR along any path from the root to the leaves. Thus if all the internal nodes in the tree obtained by some collapsing operation have at least two children each, then the resultant labeled tree represents a SP function.

We now establish a connection between the operation of collapsing the labeled tree, corresponding to a SP function $u(z_1, z_2, \ldots, z_m)$, and the set of functions covered by u. Consider a tree T with m leaves obtained by collapsing $U(*, +, z_1, z_2, \ldots, z_n)$. Suppose all internal nodes of T have at least two children each and that the leaves of T are labeled $z_{i_1}^*, z_{i_2}^*, \ldots, z_{i_m}^*$. Let $F(*, +, x_1, x_2, \ldots, x_m)$ be the tree obtained by relabeling the leaves of T by doing the assignments $z_{i_1} = x_1^*, z_{i_2} = x_2^*, \ldots, z_{i_m} = x_m^*$. This represents a SP function, say $f(x_1, x_2, \ldots, x_m)$. We now prove that u covers f.

LEMMA 3.4 *If u and f are as defined above then u covers f.*

PROOF. Because all the internal nodes of T have at least two children each, the tree T represents a SP function of the m variables $z_{i_1}, z_{i_2}, \ldots, z_{i_m}$. As described earlier, the collapsed tree T can be obtained by assigning 0 or

1 to the rest of the inputs of u. Thus u can be specialized to f by assigning an element of the set $\{0, 1, x_1, x_1', \ldots, x_m, x_m'\}$ to each of $z_1, z_2, \ldots, z_n$. Hence u covers f. □

Thus the function $u(z_1, z_2, \ldots, z_n)$ covers all functions derived by collapsing U and relabeling the leaves of the collapsed tree. Hence a way to ensure that a function u is *SP-ULM.m* is to make sure that the labeled tree corresponding to any SP function of m inputs can be obtained by suitably collapsing U and relabeling its leaves. We can simplify this result by using Lemma 3.2. Consider an unlabeled tree S with n leaves, such that every internal node of S has at least two children. Assume that any unlabeled tree with m leaves can be obtained by collapsing S. Label S with AND and *OR* at internal nodes such that nodes on adjacent levels have different labels. Label the leaves $z_1, z_2, \ldots, z_n$ from left to right. The resulting labeled tree, say $U(*, +, z_1, z_2, \ldots, z_n)$, represents a SP function $u(z_1, z_2, \ldots, z_n)$. We claim that this function is *SP-ULM.m*. It is stated in the following lemma.

LEMMA 3.5 *The function $u(z_1, z_2, \ldots, z_n)$ defined above is SP-ULM.m.*

PROOF. Let C be any *NPN* equivalence class consisting of m-input SP functions. By Lemma 3.2, C corresponds to a unique unlabeled tree of m leaves, say S_c. By construction, S_c can be obtained by collapsing S. This, with Lemmas 3.2 and 3.4, implies that u covers some member of the class C. Hence by Lemma 2.1, u covers every member of C. Inasmuch as C was an arbitrary *NPN* class consisting of m input SP functions, we conclude that u covers all m input SP functions. By Lemma 3.1, u is a *SP-ULM.m* function. □

Thus we have reduced the problem of designing a *SP-ULM.m* to the following problem:

Universal Tree Design Problem. For a given value of m, find an unlabeled tree S with a minimum number of leaves, such that S can be collapsed to any unlabeled tree of m leaves.

We do not have an optimal solution to the above problem for a general value of m. But the reduction of the original problem to the one above does aid in understanding the structure of SP functions and the requirements of a *SP-ULM.m* function. In the following we give a greedy procedure that helps construct the tree S for practical values of m.

```
Algorithm universal_tree(m){
/* Returns an unlabeled tree that can be collapsed to any unlabeled tree with m
   leaves. */
   Create a list L of all unlabeled trees with m leaves;
   S = head(L); delete_head(L);
   while (|L| ≠ 0){
      T = head(L); delete_head(L);
      if (S cannot be collapsed to obtain T){
         for i = m − 1 downto 1{
```

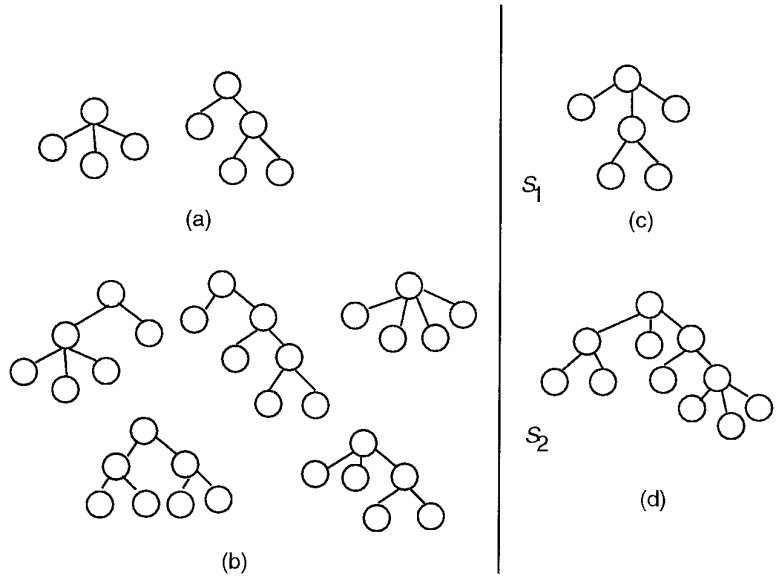

Fig. 4. Constructing unlabled trees used to derive *SP-ULM* functions.

```
7          for each tree T′ s.t. S can be collapsed to T′{
8            if T′ is a subtree of T goto 11;
9          }
10       }
11     Augment S by adding new nodes so that it can be collapsed to obtain T;
12     }
13   }
14   return(S);   }
```

The details of how line 11 is implemented are excluded from the algorithm for simplicity. But the fact that T' is a subtree of T, together with the knowledge of the sequence of operations performed to obtain T' by collapsing T, is used to determine which nodes are to be added. Note that the solution determined by this procedure is sensitive to the particular order in which the unlabeled trees with m leaves are enumerated in the list L. Table I shows that the number of unlabeled trees with m leaves (indicated by the column showing the number of *NPN* classes of m input SP functions) is very small for $m \leq 5$. For these cases one can find the tree S by trying different orders for trees in L.

Figures 4(a) and (b) depict all the unlabeled trees with three and four leaves, respectively. Figures 4(c) and (d) show unlabeled trees S_1 and S_2 that can be collapsed to any unlabeled tree with three and four leaves, respectively. It can be shown that S_1 indeed has a minimum number of leaves among all trees with the property that they can be collapsed to any tree with three leaves. Because there are two unlabeled trees with three leaves, any tree with this property has to have at least four leaves. Inasmuch as S_1 has four leaves it has to have a minimum number of leaves among such trees. A similar but more complicated argument establishes

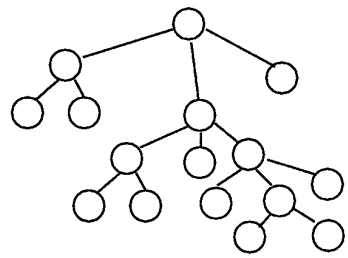

Fig. 5. Tree that can be collapsed to any unlabled tree with five leaves.

the minimality of the number of leaves of S_2. These unlabeled trees are labeled to obtain the following functions:

$$u = z_1(z_2 + z_3)z_4, \tag{1}$$

$$v = (z_1 + z_2)z_3(z_4 + z_5z_6z_7). \tag{2}$$

By Lemma 3.5, u is *SP-ULM.3* and v is *SP-ULM.4*.

We used Algorithm *universal_tree* to obtain an unlabeled tree that can be collapsed to any unlabeled tree with five leaves. This tree has ten leaves, and is shown in Figure 5.

4. IMPLEMENTATION ISSUES

This section shows how the *SP-ULM* functions for the practical cases, derived in the previous section, can be efficiently implemented and mapped. In Section 4.1 we show how the technology mapping library of a SP function is generated. A tree-based mapper is used on this library to do the mapping. Note that, from the discussion in the previous section, we can conclude that any function *NPN* equivalent to a *SP-ULM.m* function is a *SP-ULM.m* function as well. In Section 4.2 we argue that the specific function, *NPN* equivalent to a *SP-ULM.m* function, chosen in the implementation does make a difference. We give a heuristic to choose this function. Section 4.3 shows how the *SP-ULM* functions can actually be implemented as a composition of two functions of a smaller number of inputs. Thus each logic module, based on such a function, can be used to implement two functions of less than m inputs or one of m inputs. We give reasons to justify this.

4.1 Mapping to the *SP-ULM* Functions

By definition, any SP function $f(x_1, x_2, \ldots, x_m)$ can be implemented using one module of a *SP-ULM.m* function $u(z_1, z_2, \ldots, z_n)$, if each of $x_1, x_2, \ldots, x_m$ is available in both phases. But, in reality, the negative phases of the variables need to be explicitly generated using logic modules specialized to implement inverters. Hence the library of the logic module, that is, the set of functions that can be implemented using one logic module, contains only those functions that are covered by u by assigning one of $\{0, 1, x_1,$

$x_2, \ldots, x_m\}$ to $z_1, z_2, \ldots, z_n$. The set of functions of m inputs that u can cover, using these assignments, can be done by the procedure described before Lemma 3.4. But, because the negative phases of $x_1, x_2, \ldots, x_m$ are not allowed, the relabeling of leaves can use only these variables in the positive phase. Note that functions that are P equivalent need be represented by one library function only.

A *SP-ULM* function covers some nonSP functions too. As the tree-based mapper cannot make use of these, we filter out these functions while creating the library. A m-input SP function in the library of u is derived as follows: The tree $U(*, +, z_1, z_2, \ldots, z_n)$ is collapsed to get a tree T with m leaves, labeled $z^*_{i_1}, z^*_{i_2}, \ldots, z^*_{i_m}$. The assignments $z_{i_1} = z_1, z_{i_2} = x_2, \ldots, z_{i_m} = x_m$ are used to relabel the leaves of T. The resulting labeled tree corresponds to an m-input function in the library of u. Iterating over all the possible ways to collapse U gives the entire library. The library of an n-input function contains functions of up to n inputs. This library is given as input to the tree-based technology mapper, along with the optimized logic network.

An immediate implication of the preceding discussion is that at least one complemented literal has to occur in the implementation of a *SP-ULM* function (in order for an inverter to be in the library). Hence we derive the following two functions from those in Equations (1) and (2), by complementing an arbitrarily chosen literal:

$$u_1 = z_1'(z_2 + z_3)z_4, \tag{3}$$

$$v_1 = (z_1 + z_2)z_3'(z_4 + z_5z_6z_7). \tag{4}$$

As an example, the library of the function u_1 is shown in the following:

$$\{x_1', x_1x_2, x_1'x_2, x_1 + x_2, x_1'x_2x_3, x_1(x_2 + x_3), x_1'(x_2 + x_3), x_1'x_2(x_3 + x_4)\}.$$

Note that the functions $x_1(x_2 + x_3)$ and $x_1'(x_2 + x_3)$ are *NPN* equivalent to each other, yet they are both provided in the library so that the technology mapper can reduce the number of inverters in the mapped circuit by choosing the appropriate library function.

4.2 Choosing a *NPN* Equivalent Implementation for a *SP-ULM* Function

The preceding discussion reveals the fact that, for a given *SP-ULM.m* function, choosing the correct function *NPN* equivalent to it is important. For example the function u_1 in Equation (3) is *NPN* equivalent to the function:

$$u_2 = z_1'(z_2' + z_3)z_4. \tag{5}$$

Of the four functions *NPN* equivalent to the three-input AND, u_2 can implement $x_1'x_2x_3$ and $x_1'x_2'x_3$ whereas u_1 can only implement $x_1'x_2x_3$. Some thought reveals that the lower functionality of u_1, as compared to u_2, is due

to the presence of symmetric inputs[4] in u_1. It can be seen that z_2 and z_3 are symmetric in u_1. Thus the assignment $z_1 = x_1$, $z_2 = x_2$, $z_3 = 0$, $z_4 = x_3$ yields the same function as $z_1 = x_1$, $z_2 = 0$, $z_3 = x_2$, $z_4 = x_3$ for u_1, namely, $x_1'x_2x_3$. On the other hand, the same pair of assignments for u_2 yields the functions $x_1'x_2'x_3$ and $x_1'x_2x_3$, respectively. Clearly, having many different members of the same *NPN* equivalence class in the library is beneficial to the quality of the technology mapping, as the technology mapper can then use these to optimize the number of inverters in the mapped circuit.

The heuristic we use is based on the intuition that symmetric inputs cause reduced functionality. Having designed a *SP-ULM.m* function we identify the sets of symmetric inputs. For each set of symmetric inputs we pick half the elements of this set and decide to have the inverted phase literals for these in the final *SP-ULM.m* function. This is best explained using the functions u_1 and v_1, defined in Equations (3) and (4), as examples. The inputs to function u_1 can be partitioned into the sets, $\{z_1\}$, $\{z_2, z_3\}$, and $\{z_4\}$, of symmetric inputs. Using the preceding heuristic we get the function u_2 in Equation (5) that is *NPN* equivalent to u_1, but has no symmetric variables. Similarly the inputs to the function v_1 can be partitioned into the sets, $\{z_1, z_2\}$, $\{z_3\}$, $\{z_4\}$, and $\{z_5, z_6, z_7\}$, of symmetric variables. Using the heuristic yields the function:

$$v_2 = (z_1' + z_2)z_3(z_4 + z_5'z_6z_7). \tag{6}$$

Note that the extra functionality does not come without an overhead. Two extra transistors are needed to implement each inverter needed to provide the negative phase literals. Section 5.1 shows the results of comparing the quality of mapped circuits using libraries corresponding to different *NPN* equivalent *SP-ULM* functions.

4.3 Decomposed Implementation of the *SP-ULM* Functions

A study of the frequencies with which various members of the library of a *SP-ULM* function are used in technology mapping shows that functions with a smaller number of inputs are used more often than those of a larger number of inputs. This behavior has a serious effect on the utilization of the functionality of the logic modules. Table II shows the distribution of the number of inputs in the nodes of the mapped networks, obtained by mapping MCNC circuits. The libraries of functions u_2 and v_2, defined in Equations (5) and (6), were used. It is seen that a large fraction of the nodes in the networks mapped using the library of u_2 had at most two inputs. Similarly, a large fraction of the nodes in the networks mapped using the library of v_2 had at most three inputs. In fact, for v_2, no function with more than five inputs was used. It is indeed wasteful to have a seven-input logic module implementing three or four input functions most of the time. We suggest a remedy to this problem in this section.

[4] Inputs z_1, z_2 for a function $u(z_1, z_2, \ldots, z_n)$ are symmetric if $u(z_1, z_2, \ldots, z_n) = u(z_2, z_1, \ldots, z_n)$.

Table II. Usage of Library Functions of v_2: Statistics of Mapped Networks

Number of Inputs k	Percentage of Nodes with at most k inputs	
	u_2	v_2
1	14	11
2	67	51
3	95	75
4	100	95
5	100	100

We observe that the functions u_2 and v_2 can be decomposed into two functions with disjoint sets of inputs. This is shown by the equations:

$$A = z_1' z_4 \qquad B = z_2' + z_3 \qquad u_2 = A * B \tag{7}$$

$$A = (z_1' + z_2) z_3 \qquad B = z_4 + z_5' z_6 z_7 \qquad v_2 = A * B \tag{8}$$

Now each logic module is implemented as a two-output block such that B appears at one output of the block and either A or $A * B$ appears at the other output. This is illustrated in Figure 6. A programmable multiplexer is used to select between A and $A * B$. The obvious overhead then is that of implementing the multiplexer in each logic module.

To map circuits using this implementation of the logic module we create a library as described in Section 4.1. Each library function is annotated A, B, C, or D depending on whether it needs block A, block B, either A or B, or both A and B to be implemented. For example, for the function u_2 and blocks A and B as defined in Equation (7), the function $x_1' x_2$ is annotated A and $x_1' + x_2$, x_1' and $x_1'(x_2' + x_3)$ are annotated B, C, and D, respectively. After the mapping has been done, a postprocessing step of matching nodes that can be packed into one logic module is done.

The packing is done with the aid of a graph constructed as follows: one node is created for each node of the mapped network that is annotated A, B, or C. This graph will have edges for every pair of nodes that can be implemented using a single logic module. To this end, an edge is added between every pair of nodes annotated differently. An edge is added between each pair of nodes annotated C. A node is created for every node annotated D and an edge is added to any node annotated B or C, if the two nodes can packed into the same logic module. A maximum cardinality matching [Cormen et al. 1990] in this graph leads to an optimal packing decision for the mapped network.

While creating the library for this implementation of a *SP-ULM* function, the functions annotated A, B, or C are assigned an area of 0.5, and those annotated D are assigned an area of 1. This is done to influence the technology mapper to include more nodes annotated A, B, or C in the mapping solution.

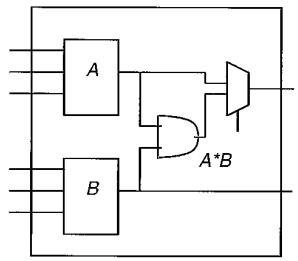

Fig. 6. Implementing the logic module as a composition of two functions.

5. EXPERIMENTAL RESULTS

We now describe the experiments used to test the logic modules we derived. Clearly the best test of the effectiveness of a logic module is the quality of mapped circuits, measured in terms of the number of logic modules needed to implement them. We use a set of MCNC circuits, optimized using the *rugged script* in MIS. Our experiments are divided into two sets. In the first set we study the effect of different implementations of the *SP-ULM* functions on the quality of the mapped circuits. As mentioned in the introduction and in the previous section, there is a tradeoff between the increased functionality obtained by using a particular implementation of a *SP-ULM* function and the extra area needed to do the implementation. We study this tradeoff. The second set of experiments compares the best logic module based on a *SP-ULM* function with Actel and Xilinx logic modules. We use the tree-based mapper for mapping our logic modules and specialized algorithms available in MIS for the Actel and Xilinx modules. Section 5.1 deals with the first set and Section 5.2 with the second set of experiments.

5.1 Comparing Different Implementations of *SP-ULM* Functions

Three different implementations of the *SP-ULM.3* and *SP-ULM.4* functions were discussed in Section 4: initial implementations (u_1, v_1), implementations after inverting some literals in the logic expression (u_2, v_2), and the decomposed implementations (u_3, v_3). The libraries for these six functions were automatically generated from the logic expressions describing them. The optimized networks were mapped using each of these libraries. Table III shows the results of these experiments. We report the block counts after mapping for each of the six libraries. For the implementations of the functions in the decomposed form, the matching-based method for packing two nodes of the mapped networks was used in a postprocessing step. The results for these implementations include two block counts, one for the mapped network before (b.m.) and one for it after this postprocessing (a.m.).

We observe that, for the cases of decomposed implementations of u_2 and v_2, the matching step to pack multiple nodes into one logic module causes a reduction of about 33% in the block count. This is a direct consequence of

Table III. Mapping Results of Different Implementations of *SP-ULM* Functions

Circuit	u_1	u_2	u_2 decomp. b.m.	u_2 decomp. a.m.	v_1	v_2	v_2 decomp. b.m.	v_2 decomp. a.m.
z4ml	29	32	36	21	26	23	27	14
misex1	39	37	46	29	29	24	29	15
vg2	63	65	76	50	46	40	47	26
5xp1	75	72	86	52	61	54	59	32
count	96	98	115	66	80	66	82	49
9symm1	125	113	132	95	90	75	95	61
9sym	92	91	107	70	81	62	77	44
apex7	156	153	173	113	129	111	127	69
C1908	339	375	407	260	318	227	312	161
rd84	104	97	114	71	88	67	82	50
e64	137	168	188	95	137	126	135	71
C880	279	267	325	220	233	182	233	119
apex2	173	168	206	135	151	129	151	89
alu2	219	220	263	184	183	166	190	108
duke2	282	270	316	203	241	208	242	132
C499	366	415	449	268	346	225	339	170
rot	451	442	505	308	382	317	373	195
apex6	543	492	540	362	436	332	391	208
alu4	528	523	607	438	405	352	447	290
des	2316	2221	2493	1649	1726	1432	1654	957
sao2	90	89	101	68	73	59	68	45
rd73	44	51	54	31	37	35	39	20
misex2	71	65	82	46	58	50	65	35
f51m	55	59	60	42	48	38	46	25
clip	77	74	81	57	60	52	57	36
bw	108	108	118	79	85	74	86	48
b9	84	76	87	63	68	57	66	36
C5315	1186	1203	1309	855	975	762	997	539
Total	8127	8044	9076	5930	6587	5345	6516	3644

the fact that the technology mapper uses library functions with a smaller number of inputs most of the time.

By themselves, the preceding block counts are not an adequate comparison of the effectiveness of different logic modules. These have to be weighted by the areas of the logic modules themselves. The transistor counts for the logic modules, assuming a CMOS implementation, are shown in Table IV. The number of transistors needed equals twice the number of literals in the logic expression, plus two transistors for each literal in the negative phase. For the decomposed implementations, six extra transistors are required to implement the multiplexer (assuming a CMOS implementation using pass gates). The total block count for the mapping experiment is reproduced in this table for ease of comparison. The normalized block count for each implementation is determined by normalizing the product of the total block count and the transistor count by this product for u_1.

Based on the results of the experiments we can make the following comments:

Table IV. Comparison of Different Logic Modules Based on *SP-ULM* Functions

Function	u_1	u_2	u_2 decomp.	v_1	v_2	v_2 decomp.
Transistors	10	12	18	16	18	24
Total Block Count	8127	8044	5930	6587	5345	3644
Normalized Block Count	1.0	1.18	1.31	1.30	1.18	1.07

(1) Among the SP-ULM.3 functions, u_1 is the best option. This function being small, the sensitivity of the area of the logic module to the addition of an inverter or a multiplexer is high. Hence, although the block count is reduced by using u_2 or the decomposed implementation of u_2, the benefit of this is overridden by the area penalty imposed.

(2) Among the SP-ULM.4 functions, the decomposed implementation of v_2 is observed to be the best choice. For these functions the relative increase in the size of the logic module, due to the extra inverters and multiplexer, is less than the decrease in block count.

(3) The number of nodes in the mapped network is a good estimate of the number of nets to be routed. An experiment showed that the average number of pins per net is about the same for networks mapped using any of the six libraries. Thus the routing resources will limit the utilization of the SP-ULM.3 functions.

Hence we expect the decomposed form of v_2 to be the most effective logic module.

5.2 Comparing SP-ULM Functions With Commercial Architectures

Now we compare the best SP-ULM logic module with commercial logic modules. We compared the decomposed implementation of v_2 with Actel's ACT1 and Xilinx's XC2000 (four-input LUT-based) logic modules.

We first compare the results of mapping for ACT1 and the four-input LUTs with those for the decomposed implementation of v_2. The *act_map* algorithm [Murgai et al. 1992] implemented in MIS was used to map ACT1. The block counts reported here are the same as those reported in Murgai et al. [1992]. The LUT mapping algorithm in MIS [Murgai et al. 1991] was used to do the mapping for the four-input LUT-based logic modules. The Xilinx logic modules allow two functions to be mapped to the same logic module, provided the input sets of these functions satisfy certain constraints. The function x_merge in MIS does the necessary matching step. We used this as a postprocessing step after mapping for the Xilinx logic modules. The results of these experiments are reported in Table V. We have shown the block counts after mapping (and merging in the case of Xilinx).

A comparison of the number of transistors needed for these logic modules is given in Table VI. The transistor count for the LUT is derived by assuming that six transistors are used per SRAM cell and that the decoder circuitry has a CMOS implementation. We reproduce the total block counts from Table V in this table for ease of comparison. The normalized block

Table V. Mapping Results of Commercial Logic Modules

Circuit Name	XC2000 (4-LUT)	ACT1
5xp1	29	35
9sym	15	17
9symml	15	17
C1908	99	159
C499	86	166
C5315	380	558
C880	102	159
alu2	102	175
alu4	235	110
apex2	83	99
apex6	183	272
apex7	61	92
b9	38	60
bw	46	54
clip	28	43
count	32	39
des	932	1315
duke2	132	158
e64	66	94
f51m	16	39
misex1	11	16
misex2	30	38
rd73	13	25
rd84	17	36
rot	166	255
sao2	35	49
vg2	26	30
z4ml	5	14
Total	2983	4124

count for each implementation is determined by normalizing the product of total block count and transistor count by this product for ACT1.

We make the following comments based on the preceding experiments:

(1) We observe that our logic module has one less input and one more output than ACT1, but mapped circuits need 12% less blocks for our logic module than for ACT1. The improvement is only 4% after the transistor count is taken into consideration. But the nets to be routed are fewer for our logic modules.

(2) The four-input LUT uses silicon area poorly compared to ACT1 and our logic module. But the transistor count is true under the assumptions we made in the foregoing. Also, the LUT has the advantage of having fewer inputs and being fully symmetric in its inputs. This makes it very flexible as far as routing goes. In addition, the LUT can be used to make a reprogrammable logic module.

Table VI. Comparison of Different Logic Modules

Function	v_2 decomp.	4-LUT	ACT1
Inputs	7	4	8
Outputs	2	2	1
Transistors	24	248	22
Total Block Count	3644	2983	4124
Normalized Block Count	0.96	8.16	1.0

We should remark here that the area of a logic module is more than that of the logic function in it. Some programming circuitry is usually required at the inputs of the logic module. Hence we expect the normalized improvement to be more than the 4% reported here. Unfortunately, we do not have information about the relative sizes of programmable switches and transistors to quantify this overhead. For example, if the overhead is one transistor per input, then the normalized improvement of our logic module over ACT1 will be about 9%. If the overload is two transistors per input, then the normalized improvement will be about 12%.

6. COMMENTS

We also used the techniques based on the theory of universal logic modules, described in Thakur and Wong [1995], to derive functions that are SP-ULM.3 and SP-ULM.4. Note that these techniques do not constrain the SP-ULM function to be SP itself. For $m = 3$ we obtained the same function as in Equation (1). For $m = 4$ the smallest number of inputs we could get for the SP-ULM function was 7, the same number of inputs as the function in Equation (2). But this function was much more complex than the one in Equation (2).

The reason for this is that the technique described in Thakur and Wong [1995] imposes a certain structure on the universal function. This results in a constraint on the space of functions that can be chosen as the required SP-ULM function. This is not necessarily a drawback of the previous techniques, as that was targeted towards finding a function that covers *all* functions, and not necessarily SP functions alone. Thus synthesis algorithms for lookup-table FPGAs were applicable to those logic modules. When those techniques were applied to generate SP-ULM functions, the resulting functions also covered many other (nonSP) functions. Naturally these SP-ULM functions are more complex. Because in this article we restricted ourselves to SP-ULM functions that were themselves SP, the resulting functions were much simpler. Consequently, the number of nonSP functions covered by the SP-ULM functions derived in this article is smaller.

7. CONCLUSIONS AND FUTURE WORK

There are many tradeoffs involved in the design of a logic module. Some of the tradeoffs are between silicon area, functionality, synthesizability, and

the effect on routing. We showed how the first three issues can be addressed using the concept of SP functions. The optimality of the tree-based technology mapping algorithm for such functions guarantees the synthesizability of logic modules that provide a complete library of SP functions. We showed how such logic modules can be systematically designed. We demonstrated a logic module that had a comparable complexity to Actel's ACT1 logic module. The number of blocks needed to map benchmarks using our logic module is substantially lower than that using ACT1. We conclude that considering the synthesizability of a logic module at design time can aid in making more efficient use of silicon and in reducing algorithm development effort after the design of the logic module.

We reduced the Logic Module Design problem to the Universal Tree Design Problem. We gave a greedy heuristic that, in general, finds a suboptimal solution to the latter problem. Finding the optimal solution is an interesting open problem.

A statistical analysis of mapped circuits will show exactly which of the SP functions in the library are used more often than others. One could then decide to design a logic module that is approximately SP-ULM, that is, it does not cover all SP functions but only the important ones. This might lead to a further reduction in the size of the logic module. The techniques in Section 3 directly apply to such an approach.

We considered trees of AND and OR gates, as in the standard implementation of a tree-based technology mapper that assumes a logic network decomposed into two-input AND and OR gates. The algorithm being structural in nature, it can be used for logic networks decomposed into AND/OR gates and 2-1 multiplexers. In fact this extension of the mapping algorithm is used to do the mapping for the Actel logic modules in Murgai et al. [1992]. We propose to extend our ideas to design logic modules that can be expressed as a tree of multiplexers and AND/OR gates, and that are universal, in some sense, for a class of such functions. Given that multiplexers have efficient pass gate implementations, this might be an important direction.

REFERENCES

ACM 1995. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. ACM, New York.

ACTEL CORP. 1994. *FPGA Data Book and Design Guide*.

ALTERA CORP. 1993. *Data Book*.

AT&T 1993. *Optimized Reconfigurable Cell Array (ORCA) Series Field Programmable Gate Arrays*.

CONG J. AND DING, Y. 1994. An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Trans. CAD/ICAS 13*, 1 (Jan.), 1–12.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. 1990. *Introduction to Algorithms*. McGraw Hill, New York.

DETJENS, E., GANNOT, G., RUDELL, R., SANGIOVANNI-VINCENTELLI, A., AND WANG, A. 1987. Technology mapping in MIS. In *Proceedings of ICCAD*, IEEE, 116–119.

KARPLUS, K. 1991a. Amap: A technology mapper for selector-based field-programmable gate arrays. In *Proceedings of DAC*, ACM, 244–247.

KARPLUS, K. 1991b. A technology mapper for table-lookup field-programmable gate arrays. In *Proceedings of DAC*, ACM, 240–243.

KNUTH, D. E. 1973. *Fundamental Algorithms*, 2nd ed. Addison-Wesley, Reading, MA.

LIN, C., MAREK-SADOWSKA, M., AND GATLIN, D. 1994. Universal logic gate for FPGA design. In *Proceedings of ICCAD*, IEEE, 164–168.

MURGAI, R., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. L. 1992. An improved synthesis algorithm for multiplexer-based PGAs. In *Proceedings of DAC*, ACM, 380–386.

MURGAI, R., SHENOY, N., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. L. 1991. Improved logic synthesis algorithms for table lookup up architectures. In *Proceedings of ICCAD*, IEEE, 564–567.

PATT, Y. N. 1973. Optimal and near-optimal universal logic modules with interconnected external terminals. *IEEE Trans. Comput. C-22*, 10 (Oct.), 903–907.

PREPARATA, F. P. 1971. On the design of universal Boolean functions. *IEEE Trans. Comput. C-20*, 4 (April), 418–423.

PREPARATA, F. P. AND MULLER, D. E. 1970. Generation of near-optimal universal Boolean functions. *JCCS 4* (April), 93–102.

RUDELL, R. L. 1989. Logic synthesis for VLSI design. U.C. Berkeley, Ph.D. thesis, April.

THAKUR, S. AND WONG, D. F. 1995. On designing ULM-based FPGA logic modules. In *Proceedings of the International Symposium on FPGAs*, ACM, New York, 3–9.

XILINX CORP. 1994. *The Programmable Logic Data Book.*

Received June 1995; revised September 1995; accepted October 1995