Universal Logic Modules for Series-Parallel Functions *

Shashidhar Thakur and D.F. Wong
Department of Computer Sciences, University of Texas at Austin

Abstract

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 7-input SP function that can implement all 4-input SP functions. Mapping results show that, on an average, the number blocks of this function needed to map benchmark circuits is 12% less than that 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.

1 Introduction

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, 5, 6]. Exploration of logic architectures revolves around established designs (e.g., [2]), with incremental modifications. The issue of having a two-way interaction between logic module design and technology mapping was stressed at the previous FPGA Symposium. To our knowledge, this paper presents the first approach to logic module design, where the mapping algorithm is the starting point. This approach is expected to lead to the design of logic modules for which the chosen mapping algorithm can perform well.

The structural tree-based mapping algorithm forms the basis of most academic and industrial technology mappers. Due to the decomposition of the unmapped logic network into trees, matches will be identified only for those library cells that have a representation in the form a tree of gates. The algorithm is optimal for libraries restricted to such functions. We call such functions series-parallel or SP functions. For a FPGA logic module, the library is the set of functions that can be implemented using one such logic module.

*This work was partially supported by the Texas Advanced Research Program under Grant No. 003658459, by a DAC Design Automation Scholarship, and by a grant from the AT&T Bell Laboratories.

Number of Inputs	Total number of classes	Number of classes with SP functions
2	2	1
3	10	2
4	208	5
5	615,904	12

Table 1: Number of NPN classes of m-input functions.

We will show how to design logic modules which provide a large library of SP functions. Thus, a good use of such logic modules can be done by the tree-based mapping algorithm. We refer the reader to [9] 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 will 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) [4, 11]. A function is said to a $ULM.m$ if it can implement all m-input functions [7, 8]. In particular, Lin et. al. [4] give a BDD based computational procedure to derive ULMs. In contrast, the procedure given by Thakur et. al. [11] was algebraic, and worked for arbitrary values of m. Targeting only the SP functions reduces the functionality, and hence the complexity, of the logic module. The disadvantage of ULM-based logic modules lies in the inability of current technology mappers to exploit all the functionality offered. Both the above mentioned methods used a synthesis strategy based on existing mappers for lookup-tables. Table 1 compares the total number of NPN classes of m-input functions (with exactly m variables in their support) with the number of NPN classes containing SP functions, for different values of m. It can be seen that the number of NPN classes containing SP functions is only a small fraction of the total number of classes.

We will demonstrate a 7-input SP function that can implement all 4-input SP functions. A logic module based on this function requires two more transistors than Actel's ACT1 logic module. Mapping results show that the number blocks of this logic module needed to map benchmark circuits is 12% less than that for ACT1, on an average. The improvement is about 4%, if the block count is scaled by the number of transistors in the implementation. Thus, a more efficient use of the silicon area available is made by our logic module.

We will omit proofs for theorems. Please refer to [10] for details.

2 Definitions

We denote the set of m-input Boolean functions by $\mathcal{F}_m$. The set of functions which have *exactly* m inputs in their support will be denoted by $\mathcal{F}_m^E$. 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 x is irrelevant. The transformation by which a logic module can implement different functions is formally defined below.

Definition 1: 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 $z_1, z_2, \ldots, z_n$ and
2. optionally complementing the output of u.

This transformation is called the *specialization* of u. □

We now formally define *series-parallel (SP) functions.*

Definition 2: Any function with zero or one input is a SP function. If f and g are two SP functions with a disjoint support then $f + g$ and $f * g$ are SP functions. □

The representation we use for a function $f(x_1, x_2, \ldots, x_m)$ is 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 at most once as a literal in the expression. The term SP arises from the fact that such a function can be implemented by a series-parallel network of transistors.

It is an obvious fact that the the logic expression $F(*, +, x_1, x_2, \ldots, x_m)$, for a SP function f, is equivalent to a *labeled tree* with the following properties:

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 an element of the set $\{x_1, x_1', \ldots, x_m, x_m'\}$, such that each variable appears as a label exactly once, in some phase.

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 will be obvious from context. The *unlabeled tree* corresponding to the SP function f is the tree obtained by eliminating the node labels on $F(*, +, x_1, \ldots, x_m)$.

Two unlabeled trees, $T_1(V_1, E_1)$ and $T_1(V_2, E_2)$, are said to *isomorphic* if there exists a bijection, $\mathcal{B} : V_1 \rightarrow V_2$, such that, for any $v, w \in V_1$, $(v, w) \in E_1$ if and only if $(\mathcal{B}(v), \mathcal{B}(w)) \in E_2$. If T_1 and T_2 are labeled trees then isomorphism between them is defined similarly, with the added restriction that, for every $v \in V_1$, the labels on v and $\mathcal{B}(v)$ are the same. In this paper, whenever we say that two trees are identical we mean they are isomorphic, and we will not distinguish between two such trees.

Example 1: The multiplexer function $g(x_1, x_2, x_3) = x_1' x_2 + x_1 x_3$ 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 shown in Figure 1(b). Labeling the unlabeled tree differently yields the function $x_1 + x_2 x_3$. □

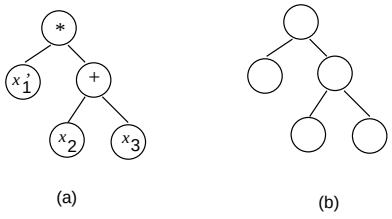

Figure 1: Trees representing $x_1'(x_2 + x_3)$ (a) Labeled tree (b) Unlabeled tree.

Definition 3: A function is said to be a *SP Universal Logic Module* for m-input functions if it covers every SP function with at most m inputs. We say that such a function is *SPULM.m.* □

Note that the above definition does not constrain a *SPULM.m* function to be SP itself. But, as explained in the previous section, a logic module that itself is SP can be efficiently implemented in silicon. Hence, we will aim at designing a *SPULM.m* that is a SP function. We will address the following problem in this paper:

Problem 1 (SP Universal Logic Module Design Problem): For a given value of m, find a SP function f that is *SPULM.m*, and has a minimum number of inputs. □

We state the following result:

Lemma 1: *If u covers all SP functions with exactly m inputs, then it covers all SP functions with at most m inputs.*

Our approach will be to concentrate on designing a SP function that covers all SP functions from $\mathcal{F}_m^E$. By the above lemma, this is enough to guarantee that a function is *SPULM.m.*

3 Deriving SPULM Functions

In this section, we will establish a one-to-one correspondence between the set of unlabeled trees with m leaves and the set of NPN classes containing m-input SP functions. We will characterize the set of m-input SP functions covered by an n-input SP function, for $n \geq m$ and formulate the problem of designing a *SPULM.m* function as a problem on unlabeled trees. We will give solutions to this problem for practical cases.

3.1 SP Functions and Unlabeled Trees

Our first result establishes the fact that, to every SP function there corresponds a *unique* (up to isomorphism) labeled tree. This is stated in the following lemma.

Lemma 2: *The labeled tree corresponding to any SP function is unique, up to isomorphism.*

Thus, the labeled tree and the logic expression are a canonical representations of a SP function. The following corollary follows easily from the above lemma.

Corollary 2.1: *The unlabeled tree corresponding to any SP function is unique, up to isomorphism.*

The following discussion leads to the conclusion that there is a one-to-one correspondence between the set of unlabeled trees with m leaves and the set of NPN classes containing m-input SP functions. The unlabeled tree corresponding to a NPN class is then a canonical representation for the class.

Consider some function $f(x_1, x_2, \ldots, x_m) \in \mathcal{F}_m^E$ represented by the logic expression $F(*, +, x_1, x_2, \ldots, x_m)$. 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 g defined as:

$$g(x_1, x_2, \ldots, x_m) = f(x^*_{\sigma(1)}, x^*_{\sigma(2)}, \ldots, x^*_{\sigma(m)}),$$

is represented by the expression $F(*, +, x^*_{\sigma(1)}, x^*_{\sigma(2)}, \ldots, x^*_{\sigma(m)})$.

From the two observations above, we can conclude that 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, f and

g can be represented by the same unlabeled tree. This implies that the entire NPN class of f can be represented by a single unlabeled tree with m leaves. This tree is unique. This conclusion is stated in Lemma 3.

Lemma 3: *For two SP functions, f and g, f is NPN-equivalent to g if and only if the unlabeled trees corresponding to f and g are identical.*

An immediate implication of the above lemma is that the number of NPN-equivalence classes of SP functions in $\mathcal{F}_m^E$ equals the number of unlabeled trees with m leaves. A formula for this was computed in [3, problem 2.3.4.4.5], and was used in determining the entries of Table 1.

3.2 Functions Covered by SP Functions

In this section we will 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 state a simple lemma that asserts that any SP function can always be specialized to cover the functions 0 and 1.

Lemma 4: *Any SP function covers the functions* 0 *and* 1.

We define the operation of collapsing a tree (labeled or unlabeled) as follows:

Definition 4: The operation of *collapsing* a tree is an application of an arbitrary sequence of the following two operations:

1. *Chopping:* Two nodes a and b in the tree, such that a is a child of b, are selected. 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 above definitions are illustrated by the following example:

Example 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 is Figure 2(a). The result of chopping the subtrees rooted at a_1 and a_2 is shown in Figure 2(b) . Two contractions at nodes b_1 and b_2 result in the tree in Figure 2(c). □

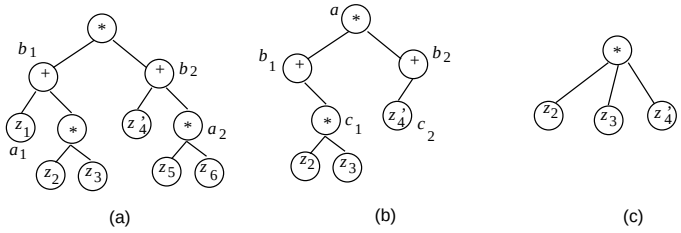

Figure 2: Collapsing a labeled tree (a) Labeled tree for u (b) Result of chopping (c) Result of contraction.

Note that the contraction operation is not possible unless preceded by a chopping operation, since all the internal nodes in the tree have at least two children to begin with.

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

Since the subtree rooted at a represents a SP function, Lemma 4 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 nodes a and c are merged to create one node. In Example 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 2 is equivalent to the assignments $z_1 = 0$ and $z_5 = z_6 = 0$.

We observe that both the above 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 will now establish a connection between the operation of collapsing the labeled tree, corresponding to a SP function $u(z_1, z_2, \ldots, z_n)$, 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 claim that u covers f. This is stated in the following lemma:

Lemma 5: *If u and f are as defined above, then u covers f.*

3.3 Constructing SPULM Functions

By the result of Lemma 5, the function u 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 *SPULM.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. 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 *SPULM.m*. This is the result of the following lemma:

Lemma 6: *The function u, defined above, is SPULM.m.*

Thus, we have reduced the problem of designing a *SPULM.m* to the following problem:

Problem 2 (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 with m leaves. □

The solution to the above problem is called a Universal Tree for the given value of m. 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 above does aid in understanding the structure of SP functions and the requirements for a function to be *SPULM.m*. Below we give a greedy algorithm that constructs the tree S, for practical values of m. This algorithm inspects each tree with m leaves, and greedily adds nodes to S to ensure that it can be collapsed to all trees with m leaves.

```
Algorithm universal_tree(m){
1    Create a list L of all unlabeled trees with m leaves;
2    S = head(L); delete_head(L);
3    while (|L| ≠ 0){
4       T = head(L); delete_head(L);
5       if (S cannot be collapsed to obtain T){
6          for i = m − 1 downto 1{
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 above 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.

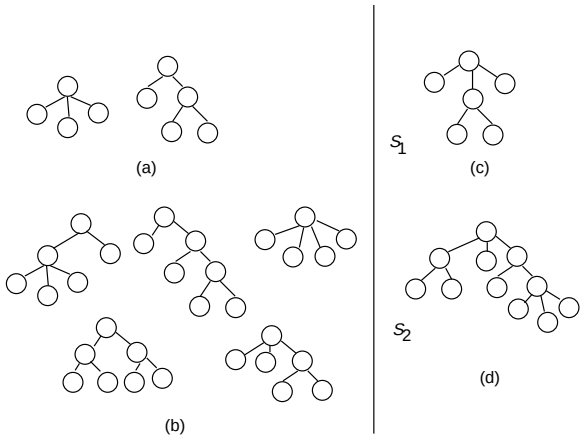

Figure 3: Example Universal Trees for $m = 3$ and 4.

Figures 3(a) and (b) depict all the unlabeled trees with 3 and 4 leaves, respectively. Figures 3(c) and (d) show trees S_1 and S_2 that are Universal Trees for $m = 3$ and 4, 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 3 leaves. Since there are two unlabeled trees with 3 leaves, any tree with this property has to have at least 4 leaves. Since S_1 has 4 leaves, it has to have a minimum number of leaves among such trees. A similar but more complicated argument establishes the minimality of the number of leaves of S_2. These unlabeled trees are labeled to obtain the functions u and v defined as follows:

$$u = z_1(z_2 + z_3)z_4 \quad (1)$$

$$v = (z_1 + z_2)z_3(z_4 + z_5z_6z_7) \quad (2)$$

By Lemma 6 u is *SPULM.3* and v is *SPULM.4*. A Universal Tree for $m = 5$ is shown in Figure 4.

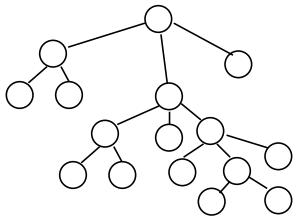

Figure 4: Universal Tree for $m = 5$.

We used Algorithm *universal_tree* to obtain an unlabeled tree that can be collapsed to any unlabeled tree with 5 leaves. This tree has 10 leaves, and is shown in Figure 4.

4 Implementation Issues

In this section, we will show how the SPULM functions derived in the previous section can be efficiently implemented and mapped. We will show how the library of the SPULM functions is generated. We will also show how choosing a particular NPN-equivalent implementation and expressing the SPULM function as a composition of two functions with fewer inputs leads to good implementations.

4.1 Creating the Library

By definition, any SP function $f(x_1, x_2, \ldots, x_m)$ can be implemented using one module of a *SPULM.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, since we assume that the logic module is not dual-output, 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 contains only those functions that are covered by u by assigning an element of $\{0, 1, x_1, x_2, \ldots, x_m\}$ to each of $z_1, z_2, \ldots, z_n$. Note that functions that are P-equivalent need to be represented by one library function only. A SPULM function covers some non-SP functions too. Since the tree-based mapper cannot make use of these, we filter out these functions while creating the library.

An immediate implication of the above discussion is that at least one complemented literal has to occur in the implementation of a SPULM 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:

$$u_1 = z_1'(z_2 + z_3)z_4 \quad (3)$$

$$v_1 = (z_1 + z_2)z_3'(z_4 + z_5z_6z_7) \quad (4)$$

As an example, the library of the function u_1 consists of the functions $x_1', x_1x_2, x_1'x_2, x_1 + x_2, x_1'x_2x_3, x_1(x_2 + x_3), x_1'(x_2 + x_3)$, and $x_1'x_2(x_3 + x_4)$. 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 Form

The discussion in the previous section reveals the fact that, for a given *SPULM.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 defined as follows:

$$u_2 = z_1'(z_2' + z_3)z_4 \quad (5)$$

Of the four functions NPN-equivalent to the three-input AND, u_2 can implement $x_1'x_2x_3$ and $x_1'x_2'x_3$, while 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 in

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

Table 2: Statistics of mapped networks.

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 the assignment $z_1 = x_1, z_2 = 0, z_3 = x_2, z_4 = x_3$ for u_1, namely $x_1' x_2 x_3$. On the other hand, the same pair of assignments for u_2 yields the functions $x_1' x_2' x_3$ and $x_1' x_2 x_3$, respectively. Clearly, having 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 on the number of inverters in the mapped circuit.

The heuristic we use is based on the intuition that symmetric inputs to the logic module cause reduced functionality. Having designed a *SPULM.m* function, we identify the sets of symmetric inputs. For each set of symmetric inputs, we pick half of the elements of this set and decide to have the inverted phase literals for these in the final *SPULM.m* function. Using this heuristic on functions u_1 and u_2 yields the function u_2 defined in Equation(5) and the function v_2 defined as follows:

$$v_2 = (z_1' + z_2) z_3 (z_4 + z_5' z_6 z_7) \tag{6}$$

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.

4.3 Decomposed Implementation

A study of the frequencies with which various members of the library of a SPULM 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 2 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 can be 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, 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 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 modules 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 5. 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 a post-processing step, of identifying pairs of nodes in the mapped circuit that can be packed into one logic module, is done. Algorithms for maximum cardinality matching on graphs can be used

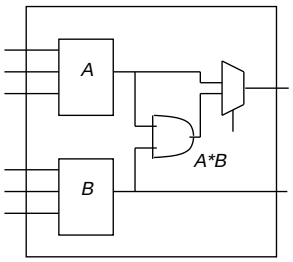

Figure 5: Decomposed implementation.

to implement this step. While creating the library for the decomposed implementation of a SPULM function the functions that can be implemented using either submodule A or B are assigned an area of 0.5, while those requiring the entire logic module are assigned an area of 1. This is done to influence the technology mapper to include more nodes in the mapping solution, that can be implemented using only a part of the logic module in the mapping solution.

5 Experimental Results

We will now describe the experiments used to test the logic modules that 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 will compare different implementations of SPULMs, and also commercial logic modules.

Three different implementations of the *SPULM.3* and *SPULM.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). Optimized networks for MCNC benchmarks were mapped using each of these libraries. Table 3 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 into one logic module, was used in a post-processing step. The results for these implementations include two block counts, one for the mapped network before (b.m.) and one for it after this post-processing (a.m.).

We observe that the decomposed implementations of u_2 and v_2 result in a reduction in the block count of about 27% and 45%, respectively, over the non-decomposed implementations. This is a direct consequence of the fact that the technology mapper makes a better use of library functions with a smaller number of inputs.

By themselves, the block counts above 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 4. The number of transistors needed equals twice the number of literals in the logic expression, with two extra 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:

1. Among the *SPULM.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 over-ridden by the area penalty imposed.

2. Among the *SPULM.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, routing resources will limit the utilization of the *SPULM.3* functions.

Hence, we expect the decomposed form of v_2 to be the most effective SPULM-based logic module.

We will now compare the decomposed implementation of v_2 with Actel's ACT1 and Xilinx's XC2000 (four-input LUT-based) logic modules. The *act_map* algorithm [6] implemented in MIS was used to map using ACT1. The block counts reported here are the same as those reported in [6]. The LUT mapping algorithm in MIS [5] 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 *xl_merge* in MIS does the necessary matching step. We used this as a post-processing step after mapping for the Xilinx logic modules. The results of these experiments are reported in the last two columns of Table 3. 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 5. 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 3 in this table for ease of comparison. The normalized block count for each implementation is determined by normalizing the product of total block count and transistor count by this product for ACT1.

We can make the following remarks based on the above 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 above. 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. Also, the the LUT-based logic modules are re-programmable.

We should state here that the area of a logic module is more than that of the logic function in it. Some programming circuitry is 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 quantize this overhead. For example, if the overhead is one transistor per input then the normalized improvement of our logic modules over ACT1 will be about 9%. If the overhead is two transistors per input then the normalized improvement will be about 12%.

6 Remarks

We used our previous techniques based on the theory of universal logic modules [11] to derive SPULM functions. Note that these techniques do not constrain the SPULM 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 SPULM function was 7, the same number of inputs as the function in Equation(2). But that function was more complex than the one in Equation(2). The reason for this is that the technique described in the previous work 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 SPULM function. This is not necessarily a drawback of the previous technique, as that was targeted towards finding a function that covers *all* functions, and not necessarily SP functions alone. Since, in this paper, we restricted ourselves to SPULM functions that were themselves SP, the resulting functions were much simpler. Consequently, the number of non-SP functions covered by the SPULM functions derived here is also smaller.

Finding an algorithm that computes the optimal solution to the Universal Tree Design problem 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 SPULM (i.e., a function that covers only the important SP functions). This will lead to a further reduction in the size of the logic module.

References

[1] J. Cong and Y. Ding. An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Trans. CAD/ICAS*, 13(1):1–12, January 1994.

[2] R. Gopisetty and S. Lan. On exploration of multiplexor based FPGA logic modules. Manuscript, 1994.

[3] D. E. Knuth. *Fundamental Algorithms.* Addison-Wesley Publishing Company, second edition, 1973.

[4] C. Lin, M. Marek-Sadowska, and D. Gatlin. Universal logic gate for FPGA design. In *Proceedings of ICCAD.* IEEE/ACM, 1994.

[5] R. Murgai, , N. Shenoy, R.K. Brayton, and A. L. Sangiovanni-Vincentelli. Improved logic synthesis algorithms for table lookup up architectures. In *Proceedings of ICCAD*, pages 564–567. IEEE/ACM, 1991.

[6] R. Murgai, R.K. Brayton, and A. L. Sangiovanni-Vincentelli. An improved synthesis algorithm for multiplexer-based PGAs. In *Proceedings of DAC*, pages 380–386. IEEE/ACM, 1992.

[7] Y. N. Patt. Optimal and near-optimal universal logic modules with interconnected external terminals. *IEEE Trans. Comp.*, C-22(10):903–907, October 1973.

[8] F. P. Preparata. On the design of universal Boolean functions. *IEEE Trans. Comp.*, C-20(4):418–423, April 1971.

[9] R. L. Rudell. *Logic Synthesis for VLSI Design.* PhD thesis, U.C. Berkeley, April 1989.

[10] S. Thakur. *Logic Module Design and Technology Mapping for Field-Programmable Gate Arrays.* PhD thesis, U.T. Austin, May 1996. In preparation.

[11] S. Thakur and D. F. Wong. On designing ULM-based FPGA logic modules. In *Int'l Symp. on FPGAs*, pages 3–9. ACM SIGDA, 1995.

Circuit	u_1	u_2	u_2 decomp.		v_1	v_2	v_2 decomp.		XC2000	ACT1
			b.m.	a.m.			b.m.	a.m.	4-LUT	
5xp1	75	72	86	52	61	54	59	32	29	35
9sym	92	91	107	70	81	62	77	44	15	17
9symml	125	113	132	95	90	75	95	61	15	17
C1908	339	375	407	260	318	227	312	161	99	159
C499	366	415	449	268	346	225	339	170	86	166
C5315	1186	1203	1309	855	975	762	997	539	380	558
C880	279	267	325	220	233	182	233	119	102	159
alu2	219	220	263	184	183	166	190	108	102	175
alu4	528	523	607	438	405	352	447	290	235	110
apex2	173	168	206	135	151	129	151	89	83	99
apex6	543	492	540	362	436	332	391	208	183	272
apex7	156	153	173	113	129	111	127	69	61	92
b9	84	76	87	63	68	57	66	36	38	60
bw	108	108	118	79	85	74	86	48	46	54
clip	77	74	81	57	60	52	57	36	28	43
count	96	98	115	66	80	66	82	49	32	39
des	2316	2221	2493	1649	1726	1432	1654	957	932	1315
duke2	282	270	316	203	241	208	242	132	132	158
e64	137	168	188	95	137	126	135	71	66	94
f51m	55	59	60	42	48	38	46	25	16	39
misex1	39	37	46	29	29	24	29	15	11	16
misex2	71	65	82	46	58	50	65	35	30	38
rd73	44	51	54	31	37	35	39	20	13	25
rd84	104	97	114	71	83	67	82	50	17	36
rot	451	442	505	308	382	317	373	195	166	255
sao2	90	89	101	68	73	59	68	45	35	49
vg2	63	65	76	50	46	40	47	26	26	30
z4ml	29	32	36	21	26	23	27	14	5	14
Total	8127	8044	9076	5930	6587	5345	6516	3644	2983	4124

Table 3: Results of technology mapping.

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

Table 4: Comparison of different logic modules based on SPULM functions.

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

Table 5: Comparison of different logic modules.