BOOLEAN MATCHING IN LOGIC SYNTHESIS

by

Hamid Savoj, Mário J. Silva, Robert K. Brayton,
Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M92/15

7 February 1992

COVER PAGE

BOOLEAN MATCHING IN LOGIC SYNTHESIS

by

Hamid Savoj, Mario J. Silva, Robert K. Brayton,
and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M92/15

7 February 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

Boolean Matching in Logic Synthesis *

Hamid Savoj Mário J. Silva
Robert K. Brayton Alberto Sangiovanni-Vincentelli
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, CA 94720

Abstract

A new formulation for finding the existence of a Boolean match between two functions with don't cares is presented. An algorithm for Boolean matching is developed based on this new formulation and is used within a technology mapper as a substitute for tree matching algorithms. The new algorithm is fast and uses symmetries of the gates in the library to speed up the matching process. Local don't cares are computed for each function being mapped in terms of its inputs. To reduce the frequency in which Boolean matching is used, the gates in the library are grouped into classes such that it is sufficient to try to match a function with the class representative. Experimental results show significant improvement in the final area of the mapped circuits.

*This project was supported in part by NSF/DARPA under contract number MIP-8719546 and MIP-9002962 and grants from IBM and DEC.

1 Introduction

Detection of equivalence of Boolean functions, also called *matching*, is a problem arising in logic synthesis when a boolean network is to be implemented in terms of reusable building blocks. Many solutions have been proposed for this problem almost since the introduction of packaged logic gates. In [4], a tree matching algorithm is used to implement a network in terms of the gates in a library which is similar to the one found in programming language compilers for generation of optimal code for expression trees. Mailhot and DeMicheli present Boolean methods for technology mapping in a more recent paper [5]. Unlike tree matching, the Boolean matching techniques allow the use of don't care information. This can result in better quality circuits because some matches that could not be detected with tree matching techniques are now found. Additionally, there is no need to add inverters to the circuit because both input phases of a function being matched are considered at the same time.

Mailhot and DeMicheli [5] proposed two different algorithms for Boolean matching: one that uses don't cares and another that does not. When matching without don't cares, symmetries are used to speed up the matching process. This technique was the basis of the algorithm of Dietmeyer and Schneider [3] and has also been applied recently by Morrison et al. [6] in a technology mapping algorithm based on a covering approach. Dietmeyer and Schneider computed symmetries in the presence of don't cares, but computation of larger symmetries becomes expensive, because in this situation symmetry sets do not form an equivalence class. For matching with don't cares the algorithm of Mailhot and DeMicheli uses a matching compatibility graph which is built during the setup phase. Each node of this graph corresponds to an NPN-equivalent [7] function. The size of this graph grows exponentially with the size of the variable support of the functions, and has limited the usage of don't cares to the matching of functions with at most 4 inputs. Symmetries are not used with this algorithm. The algorithms in [5] and [3] compute the symmetries of both functions being matched. The computation of symmetries is relatively expensive, as it requires $O(n^2)$ cofactor computations for a completely specified function (no don't cares) with n inputs. Consequently, cofactor computations dominate the cost of finding a boolean match in such algorithms.

We present a new Boolean matching algorithm which uses both symmetries and don't cares at the same time, and requires only the computation of the symmetries of one of the functions. This Boolean matching technique is used both for technology mapping and organizing the gate libraries. In technology mapping, one of the functions to be considered for the matching is a library function whose symmetries can be computed in a setup phase. This speeds up the matching considerably and has enabled the application of Boolean matching techniques to much larger circuits with run times comparable to those obtained using tree matching while obtaining circuits with smaller area.

We use this new Boolean matching technique within a technology mapper which uses the principles previously developed by Keutzer[4], Detjens et al.[2] and Rudell[10].

This is based on finding a tree decomposition of the graph associated with the network and using dynamic programming to map each of the trees. The dynamic programming algorithm visits the nodes of each tree in depth first order and for each node, finds the best match of a subtree, rooted at the node, with the tree representation of a gate in a given library.

In [12], efficient ways are given for computing local don't cares which are applied to node simplification in multi-level logic networks. We use the same don't care computation techniques to find local don't cares at the leaves of each cluster function being mapped to a library gate. These are then used to find the best match for the cluster function. Although we have only used this Boolean matching technique to find the best area for the circuit, the algorithm is very general and can be used in other contexts such as delay optimization, or layout driven technology mapping [9].

As demonstrated in [10], the inclusion of complex CMOS gates in the library is useful because it may lead to a significant reduction in the required area for implementing some combinatorial functions. However, larger cell libraries require more matchings and imply the use of functions with more inputs, making technology mapping with very large libraries computationally expensive. Mailhot and deMicheli [5] proposed a technique for speeding-up the matching by grouping gates in the library in such a way that after finding a match with a representative gate the match with all gates in the group is determined. We have also developed a technique for grouping gates, using our matching algorithm. The gates in the library are matched with each other and the ones that match with inverted inputs or output are stored in the same data structure.

The rest of this paper is organized as follows. Section 2 introduces the notation and terminology used throughout the paper. In section 3, we introduce techniques for simplifying the Boolean matching problem. Section 4 presents a Boolean matching algorithm used for technology mapping. We also explain the significance of symmetries and how to generate different supports of a Boolean function in this section. Section 5 explains the computation of local don't cares to be used for Boolean matching. Section 6 discusses the organization of the library. Finally we present the experimental results in section 7 and conclusions in section 8.

2 Terminology and Notation

Let $(x_1, x_2, \ldots, x_n)$ be the variables in the Boolean space B^n. A *literal* is a variable in its true or complement form (e.g. x_i, or $\overline{x_i}$). A *product term or cube* is the conjunction of some set of literals (e.g. $x_1 x_2 \overline{x_3}$). A *Boolean network* $\mathcal{N}$, is a directed acyclic graph (DAG) such that for each node in $\mathcal{N}$ there is an associated representation of a Boolean function f_i, and a Boolean variable y_i, where $y_i = f_i$. There is a directed edge e_{ij} from y_i to y_j if f_j depends explicitly on y_i or $\overline{y}_i$. A node y_i is a *fanin* of a node y_j if there is a directed edge e_{ij} and a *fanout* if there is a directed edge e_{ji}. A node y_i is a *transitive* *fanin* of a node y_j if there is a directed path from y_i to y_j and a *transitive fanout* if there is a directed path from y_j to y_i. *Primary inputs* $\mathbf{x} = (x_1, \ldots, x_n)$ are inputs of the Boolean network and *primary outputs* $\mathbf{z} = (z_1, \ldots, z_m)$ are its outputs. *Intermediate nodes* of the Boolean network have at least one fanin and one fanout. The *support* of a function f is the set of variables that f explicitly depends on.

The *cofactor* of a sum-of-products f with respect to a literal $x_i(\overline{x_i})$, denoted by $f_{x_i}(f_{\overline{x_i}})$, is a new function obtained by substituting 1(0) for $x_i(\overline{x_i})$ in every cube in f which contains $x_i(\overline{x_i})$.

The Shannon's expansion of a Boolean function f with respect to a variable x is

$$f = x f_x + \overline{x} f_{\overline{x}}.$$

BDD's [1] are compact representations of a recursive Shannon decomposition. They are unique for a given variable ordering and hence are canonical forms for representing Boolean functions. They can be constructed from the Shanon's expansion of a Boolean function by 1) deleting a node whose two child edges point to the same node, and 2) sharing isomorphic subgraphs. Technically the result is a reduced ordered BDD, (ROBDD), which we shall just call BDD.

The *consensus operator or universal abstraction* applied to a function f with respect to a variable x_i is

$$C_{x_i} f \equiv f_{x_i} f_{\overline{x}_i}.$$

This is the largest Boolean function contained in f which is independent of x_i.

The *smoothing operator or existential abstraction* applied to a function f with respect to a variable x_i is

$$S_{x_i} f \equiv f_{x_i} + f_{\overline{x}_i}.$$

This is the smallest Boolean function independent of x_i which contains f.

The *Boolean difference* of a function f with respect to a variable x is defined as

$$\frac{\partial f}{\partial x} \equiv f_x \overline{f}_{\overline{x}} + \overline{f}_x f_{\overline{x}}$$

This function gives all the conditions under which the value of f is influenced by the value of x. Its complement therefore is all the conditions under which f is *insensitive* to x.

2.1 Don't Cares in a Boolean Network

If y_i is the variable at a node and f_i its logic function, then $y_i = f_i$; therefore, we don't care if $y_i \neq f_i$. The expression $\sum_i (y_i \neq f_i)$ is called the *satisfiability don't care* set (SDC) of N. SDC is defined in the extended space B^{n+m} which is composed of both input and intermediate variables. The *observability don't cares (ODC's)* at each intermediate node of a multi-level network are a set of input minterms under which the function at the node can be either 1 or 0 while the functions generated at each primary output z_i remain unchanged. If $\mathbf{z} = (z_1, \ldots, z_l)$ are the primary outputs, then ODC at node y_o is

$$ODC_o = \{m \in B^n | \mathbf{z}_{y_o}(m) \equiv \mathbf{z}_{\overline{y}_o}(m)\}.$$

In practice, it is computationally expensive to compute the full observability don't care at each node. Thus subsets of observability don't cares are used. Subsets of observability don't cares are *compatible* [8] if the function at each node can be changed (as allowed by its observability don't care subset) independent of allowable changes in the functions at other nodes in the network. These compatible subsets can be computed for all the nodes by traversing the Boolean network once [11].

In general, we don't care about the value of every single output for every single input combination. The external don't care for each output z_i of the network N is composed of vertices in B^n which correspond to the input combinations that a) never happen or b) the value of z_i for that particular input combination is not important.

The *local don't cares* [12] at y_i are don't cares computed for the node in terms of its immediate fanins. The local don't cares can also be computed in terms of any cutset of nodes that can express the function f_i.

3 Boolean Matching

We address the Boolean matching problem for two functions $f(x_1, \ldots, x_m)$ and $g(y_1, \ldots, y_m)$ with the same number of inputs and don't care sets $d_f(x_1, \ldots, x_m)$ and $d_g(y_1, \ldots, y_m)$ in this section. The objective is to find an assignment of variables x to y such that there exists a function that is a cover of both f and g.

A particular assignment of variables of g to f $(y_{i_1} = x_{j_1}, y_{i_2} = \overline{x}_{j_2}, \ldots, y_{i_m} = x_{j_m})$ can be represented by a new function

$$\mathcal{A}_k(x,y) = (y_{i_1} \overline{\oplus} x_{j_1})(y_{i_2} \oplus x_{j_2}) \ldots (y_{i_m} \overline{\oplus} x_{j_m}).$$

In general, both $(y_{i_1} \overline{\oplus} x_{j_1})$ and $(y_{i_1} \oplus x_{j_1})$ are possible assignments. The first sets $y_{i_1} = x_{j_1}$; the second sets $y_{i_1} = \overline{x}_{j_1}$. The function $\hat{g}_k$ under variable assignment $\mathcal{A}_k$ is simply $\hat{g}_k(x) = \mathcal{S}_y \mathcal{A}_k(x,y) g(y)$.

Lemma 3.1 *Let $\hat{g}$ and $\hat{d}_g$ represent the new function obtained from g and d_g by switching y's with the corresponding x's for a particular assignment of y's to x's. A matching under this assignment exists if and only if $\hat{g} - \hat{d}_g \leq f + d_f$ and $f - d_f \leq \hat{g} + \hat{d}_g$.*

Proof Assume a matching exists under the given variable assignment and let h represent the function for which the matching exists. $\hat{g}-\hat{d}_g \leq h \leq \hat{g}+\hat{d}_g$ and $f-d_f \leq h \leq f+d_f$; therefore, $\hat{g}-\hat{d}_g \leq f+d_f$ and $f-d_f \leq \hat{g}+\hat{d}_g$. On the other hand, if $\hat{g}-\hat{d}_g \leq f+d_f$ and $f-d_f \leq \hat{g}+\hat{d}_g$, we let $h = (f-d_f)+(\hat{g}-\hat{d}_g)$. Clearly, $\hat{g}-\hat{d}_g \leq h \leq \hat{g}+\hat{d}_g$ and $f-d_f \leq h \leq f+d_f$. ■

Lemma 3.2 *The matching under variable assignment $\mathcal{A}_k$ exists if and only if*

$$\mathcal{M}_k = \mathcal{C}_x(\mathcal{S}_y(\mathcal{A}_k.(d_f + d_g + f\overline{\oplus}g))) \equiv 1 \qquad (1)$$

(The significance of consensus operation is shown in the next Lemma).

Proof Let $\hat{g} = \mathcal{S}_y\mathcal{A}_k g$ and $\hat{d}_g = \mathcal{S}_y\mathcal{A}_k d_g$ then

$$\begin{aligned}
\mathcal{M}_k &= \mathcal{C}_x(\mathcal{S}_y(\mathcal{A}_k(d_f + d_g + f\overline{\oplus}g))) \\
&= \mathcal{C}_x(\mathcal{S}_y\mathcal{A}_k d_f + \mathcal{S}_y\mathcal{A}_k d_g + \mathcal{S}_y\mathcal{A}_k fg + \mathcal{S}_y\mathcal{A}_k\overline{f}\overline{g}) \\
&= \mathcal{C}_x(d_f\mathcal{S}_y\mathcal{A}_k + \hat{d}_g + f\mathcal{S}_y\mathcal{A}_k g + \overline{f}\mathcal{S}_y\mathcal{A}_k\overline{g}) \\
&= \mathcal{C}_x(d_f + \hat{d}_g + f\overline{\oplus}\hat{g})
\end{aligned}$$

$\mathcal{C}_x(d_f+\hat{d}_g+f\overline{\oplus}\hat{g}) = 1$ if and only if $(d_f+\hat{d}_g+f\overline{\oplus}\hat{g}) = 1$. Assume $(d_f+\hat{d}_g+f\overline{\oplus}\hat{g}) = 1$. Let m be a minterm in $\hat{g}-\hat{d}_g$. Then $m \in f+d_f$, otherwise $d_f+\hat{d}_g+f\overline{\oplus}\hat{g} \neq 1$. As a result, $\hat{g}-\hat{d}_g \leq f+d_f$. In the same way, $(d_f+\hat{d}_g+f\overline{\oplus}\hat{g}) = 1$ implies $f-d_f \leq \hat{g}+\hat{d}_g$. Therefore, if $\mathcal{C}_x(d_f + \hat{d}_g + f\overline{\oplus}\hat{g}) = 1$, a match exists. If $f - d_f \leq \hat{g} + \hat{d}_g$ and $\hat{g} - \hat{d}_g \leq d_f + f$, then $\hat{g}f + d_f + \hat{d}_g = \hat{g} + d_f + \hat{d}_g$ and $\overline{f}\ \overline{\hat{g}} + d_f + \hat{d}_g = \overline{\hat{g}} + d_f + \hat{d}_g$. Therefore $(d_f + \hat{d}_g + f\overline{\oplus}\hat{g}) = 1$. ■

We can organize equation (1) in a more computationally efficient way by using the result of the following lemma.

Lemma 3.3 *If $i \neq j$, $\mathcal{C}_{x_i}\mathcal{S}_{y_j}(x_j\overline{\oplus}y_j)h(x,y) = \mathcal{S}_{y_j}(x_j\overline{\oplus}y_j)\mathcal{C}_{x_i}h(x,y)$.*

Proof

$$\begin{aligned}
\mathcal{C}_{x_i}\mathcal{S}_{y_j}(x_j\overline{\oplus}y_j)h &= \mathcal{C}_{x_i}\mathcal{S}_{y_j}(x_jy_jh_{y_j} + \overline{x}_j\overline{y}_jh_{\overline{y}_j}) \\
&= \mathcal{C}_{x_i}(x_jh_{y_j} + \overline{x}_jh_{\overline{y}_j}) \\
&= (x_j(\mathcal{C}_{x_i}h)_{y_j} + \overline{x}_j(\mathcal{C}_{x_i}h)_{\overline{y}_j}) \\
&= \mathcal{S}_{y_j}(x_j\overline{\oplus}y_j)\mathcal{C}_{x_i}h.
\end{aligned}$$

■

Lemma 3.4 *Let $\mathcal{A}_k = (y_1\overline{\oplus}x_1)(y_2\overline{\oplus}x_2)\ldots(y_m\overline{\oplus}x_m)$. Then $\mathcal{M}_k = \mathcal{C}_x(\mathcal{S}_y(\mathcal{A}_k.(d_f + d_g + f\overline{\oplus}g)))$ can be expressed as*

$$\mathcal{M}_k = (\mathcal{C}_{x_m}\mathcal{S}_{y_m}(x_m\overline{\oplus}y_m)\ldots\mathcal{C}_{x_1}\mathcal{S}_{y_1}(x_1\overline{\oplus}y_1)(d_f + d_g + f\overline{\oplus}g))$$

Proof The statement of the lemma follows by induction and lemma 3.3. ■

All the possible assignmets of variables y to x are not required to check whether a matching exists. First we express necessary conditions for a matching to exist. Let $|f|$ represent the number of minterms in the function f. Once BDD's are built for functions f and g, then $|f|$ and $|g|$ can be easily found by traversing the corresponding BDD's only once. Given node n in the BDD of f with children nl and nr, the number of minterms in the function represented by n in the ordered BDD of f can be found if this number is known at nl and nr. We represent the difference between the variables n and nl in the variable ordering by l (if n appears right before nl, $l = 1$) and the difference between the variable of n and nr in the variable ordering by r. The number of onset points for the function at n is $|n| = 2^{l-1}|nl| + 2^{r-1}|nr|$. Initially, the number of minterms at *node 1* is set to 1 and *node 0* is set to 0. Also, if the root of the BDD is not the first variable in the ordering we multiply the count at the root node by 2^k where k is the difference between the root node and the first variable in the ordering.

Theorem 3.5 *A matching between f and g exists under any variable assignment only if $|f - d_f| \leq |g + d_g|$, $|\overline{f} - d_f| \leq |\overline{g} + d_g|$, $|g - d_g| \leq |f + d_f|$, and $|\overline{g} - d_g| \leq |\overline{f} + d_f|$. In particular, if $d_f = 0$ and $d_g = 0$, $|f| = |g|$.*

Proof Each onset point of f must be mapped to an onset point or don't care point of g and each offset point of f must be mapped to an offset point or don't care point of g. If $|f - d_f| > |g + d_g|$ some onset points in f cannot be mapped to any onset or don't care point of g. The proof is similar for other cases. ■

Lemma 3.6 *A matching under the assignment $x_i = y_j$ exists only if $|f_{x_i} - d_{f_{x_i}}| \leq |g_{y_j} + d_{g_{y_j}}|$, $|\overline{f}_{x_i} - d_{f_{x_i}}| \leq |\overline{g}_{y_j} + d_{g_{y_j}}|$, $|g_{y_j} - d_{g_{y_j}}| \leq |f_{x_i} + d_{f_{x_i}}|$, $|\overline{g}_{y_j} - d_{g_{y_j}}| \leq |\overline{f}_{x_i} + d_{f_{x_i}}|$, $|f_{\overline{x}_i} - d_{f_{\overline{x}_i}}| \leq |g_{\overline{y}_j} + d_{g_{\overline{y}_j}}|$, $|\overline{f}_{\overline{x}_i} - d_{f_{\overline{x}_i}}| \leq |\overline{g}_{\overline{y}_j} + d_{g_{\overline{y}_j}}|$, $|g_{\overline{y}_j} - d_{g_{\overline{y}_j}}| \leq |f_{\overline{x}_i} + d_{f_{\overline{x}_i}}|$, and $|\overline{g}_{\overline{y}_j} - d_{g_{\overline{y}_j}}| \leq |\overline{f}_{\overline{x}_i} + d_{f_{\overline{x}_i}}|$. In particular, if $d_f = 0$ and $d_g = 0$, $|f_{x_i}| = |g_{y_j}|$ and $|f_{\overline{x}_i}| = |g_{\overline{y}_j}|$.*

Proof If $x_i = y_j$, each onset point of $(f_{x_i} - d_{f_{x_i}})$ must be mapped to a point in $(g_{y_j} + d_{g_{y_j}})$, therefore, $|f_{x_i} - d_{f_{x_i}}| \leq |g_{y_j} - d_{g_{y_j}}|$. Other cases can be proved in the same way. ■

Corollary 3.7 *A matching under the assignment $\overline{x}_i = y_j$ exists only if $|f_{x_i} - d_{f_{x_i}}| \leq |g_{\overline{y}_j} + d_{g_{\overline{y}_j}}|$, $|\overline{f}_{x_i} - d_{f_{x_i}}| \leq |\overline{g}_{\overline{y}_j} + d_{g_{\overline{y}_j}}|$, $|g_{\overline{y}_j} - d_{g_{\overline{y}_j}}| \leq |f_{x_i} + d_{f_{x_i}}|$, $|\overline{g}_{\overline{y}_j} - d_{g_{\overline{y}_j}}| \leq |\overline{f}_{x_i} + d_{f_{x_i}}|$, $|f_{\overline{x}_i} - d_{f_{\overline{x}_i}}| \leq |g_{y_j} + d_{g_{y_j}}|$, $|\overline{f}_{\overline{x}_i} - d_{f_{\overline{x}_i}}| \leq |\overline{g}_{y_j} + d_{g_{y_j}}|$, $|g_{y_j} - d_{g_{y_j}}| \leq |f_{\overline{x}_i} + d_{f_{\overline{x}_i}}|$, and $|\overline{g}_{y_j} - d_{g_{y_j}}| \leq |\overline{f}_{\overline{x}_i} + d_{f_{\overline{x}_i}}|$.*

From now on, we concentrate on the use of Boolean matching in technology mapping where we try to match a cluster function having some local don't cares with a library function which has no don't cares ($d_g = 0$).

4 Boolean Matching for Technology Mapping

The objective of a technology mapper is to map a circuit into a set of gates in the library. The given circuit is first decomposed into a set of 2-input gates and then into a set of disjoint trees. As in [4, 2, 10], we use dynamic programming to map each of the trees into a set of library gates. The trees are mapped in topological order; each tree is mapped after all its fanin trees. Mapping is a two step process. In the first step, called *matching*, we find the minimum cost matching for the root of the tree. In the second step, called *gate assignment*, we implement the logic function of the tree in terms of library gates as determined in the matching phase.

The first phase of technology mapping is to traverse the target tree bottom-up from the primary inputs. At each node, all possible functions up to a given number of inputs having that node as output are considered. These functions are called *cluster functions*; their corresponding subgraphs are called *clusters* [5]. In our formulation, a cluster is represented by a root node and a set of leaf nodes (cutset of nodes) separating the root node from the rest of the network. We use an iterative algorithm for cluster generation, that starts with a cluster consisting only of the root node, and generates new clusters by expanding every cluster. Expansion of a cluster is done by removing each of the nodes of the cutset one at a time and adding its fanin nodes to it. If some of the clusters generated in this process have been generated before, or contain more nodes than the maximum number of inputs in any gate of the library, they are simply discarded. Each iteration expands the clusters generated on the previous iteration only. Cluster generation is stopped after an iteration that does not produce more clusters.

During gate assignment we build a new network that contains the best map at each tree. At each tree, we need to choose the phase of the root node of the tree. The less costly phase in terms of area is currently chosen unless the root node is a primary output where the positive phase is chosen. The penalty for using the phase that is not implemented is the cost of an inverter. After all the trees in the network are mapped, we traverse these trees in reverse order, and check what phase of the root is used in each tree. If the implemented phase in the new network for a particular tree is always inverted before it is used by its fanout trees, we switch to the other phase of that tree to reduce cost.

The matching problem is to find any library function that can be matched with a cluster function. The correspondence between the inputs of the cluster function and the library gate is sought first, then one checks if the functions are equivalent under such condition. In the presence of local don't cares the matching problem can be formulated as follows. Let $f(x_1, x_2, \ldots, x_n)$ be a cluster function with local don't-care $d(x_1, x_2, \ldots, x_n)$, and

$g(y_1, y_2, \ldots, y_m)$ be a library function where $m \leq n$. If $m > n$, some of the inputs of the library gate must be set to 0, 1, or tied together. Such gates can be added to the library in a preprocessing step. For architectures composed of particular types of gates where the case $m > n$ is important, special techniques can be devised to do Boolean matching. If $m < n$, a matching exists only if the support f can be reduced using the given don't care set. This is unlikely in a well-optimized circuit because most redundant connections are already removed. For each cluster function we generate all the possible supports and match each one with a library gate of the same number of variables.

4.1 Generating all Supports

We divide the matching problem into two parts. First, we generate $\bar{f}$ and $\bar{d}$ for each possible support of a cluster function f with don't care d. The circuits given for mapping are usually well optimized and do not have many redundancies; therefore, we expect few possible supports by which f can be represented. Each function $\bar{f}$ is then compared to all the library gates which have the same number of inputs.

Let f be a cluster function with the support X. A support $X_i \subseteq X$ ($\overline{X}_i = X - X_i$) is a valid representation for f if and only if $\mathcal{S}_{\overline{X}_i}(f - d) \leq (f + d)$, or equivalently $(f - d) \leq \mathcal{C}_{\overline{X}_i}(f + d)$ ($\mathcal{S}_{\overline{X}_i}(f - d) \leq (f + d)$ implies that $\mathcal{S}_{\overline{X}_i}(f - d) \leq \mathcal{C}_{\overline{X}_i}(f + d)$). This new function $\bar{f}$ can be represented as $\bar{f} = \mathcal{S}_{\overline{X}_i}(f - d)$ with don't care set $\bar{d} = \mathcal{C}_{\overline{X}_i}(f + d) - \mathcal{S}_{\overline{X}_i}(f - d)$. The algorithm shown in Figure 1 is used to generate all the possible supports for a cluster function f. The original arguments given to *generate_support* are $f_l = f - d$, $f_h = f + d$, *vars* is all the variables in f and d (this is also saved as a possible support for f), and *start* $= 0$.

Other techniques have been recently suggested for generating all possible supports of a function [14]. We are still investigating such techniques.

4.2 Boolean Matching Algorithm

The algorithm for finding the existence of a match between a library gate $g(y_1, \ldots, y_m)$ and a cluster function $f(x_1, \ldots, x_m)$ with don't care set $d(x_1, \ldots, x_m)$ is shown in Figure 2. f and g have the same number of inputs. The argument M is originally set to $M = d + f \overline{\oplus} g$. The argument i shows the variable in f for which a match is sought. i is set to 0 originally. Before calling *boolean_match*, we check the necessary condition given by theorem 3.5. If that condition is not satisfied, f and g cannot be matched. Each input x_i of f must be matched with an input y_j of g.

x_i can be equal to y_j if the necessary conditions as given by lemma 3.6 are satisfied. If they are not satisfied, $\overline{x}_i = y_j$ is tried. If that is not possible either, y_j is not a possible match for x_i and is skipped. If no input of g can be set equal to x_i, f and g cannot be matched.

```
function $generate\_support(f_l, f_h, vars, start)$
begin
    for ($i = start; i < number(vars); i$++) begin
        $x_i = vars(i)$
        if ($\mathcal{S}_{x_i} f_l \le f_h$) begin
            $newvars(k) = vars(k)$ for $k < i$
            $newvars(k) = vars(k+1)$ for $k \ge i$
            save $newvars$ as a possible support
            $newf_l = \mathcal{S}_{x_i} f_l$
            $newf_h = \mathcal{C}_{x_i} f_h$
            $generate\_support(newf_l, newf_h, newvars, i)$
        end
    end
end
```

Figure 1: Generating Supports

4.3 Symmetries

Most gates in the library have many symmetries. We find all such symmetries for all the gates in the library in a preprocessing step. For example, gate g might have two inputs y_k and y_j which are symmetric. If $x_i = y_k$ is not possible, then clearly x_i cannot be set equal to y_j either and is skipped. There is another kind of symmetry which can be used to speed up Boolean matching. Given a library gate $g = y_1y_2 + y_3y_4 + y_5y_6$, y_1 is symmetric with y_2, y_3 is symmetric with y_4, and y_5 is symmetric with y_6. If we switch the variables y_3 and y_4 with y_1 and y_2, we get exactly the same function. In this example, y_1y_2 are *group symmetric* with y_3y_4 and y_5y_6. Therefore if a variable x_i cannot be matched with y_1, it cannot be matched with any other variable in g and no matching exists. On the other hand, if y_1 has been matched with some other variable x_j and x_i cannot be matched with y_2, we still need to try $x_i = y_3$. Symmetries and group symmetries are found for each of the gates in the library.

4.4 Heuristic

Once we find a variable y_j that can be set equal to x_i, we reduce the size of the matching problem at hand by one variable and try to match the rest of the variables in f and g. Using the result of lemma 3.4, we compute $newM = \mathcal{C}_{x_i}\mathcal{S}_{y_j}(x_i\overline{\oplus}y_j)M$. A necessary

```
function $boolean\_match(f, d, g, i, M)$
begin
  if ($M = 0$)
    return match_not_found
  if ($M = 1$)
    return match_found
  $x_i$ = ith variable in $f$
  for each variable $y_j$ of $g$ not matched yet begin

    if $y_j$ is symmetric to a $y_k$ already tested
      continue

    /* check the necessary conditions for $x_i = y_j$ */
    if (($|f_{x_i} - d_{x_i}| \leq |g_{y_j}|$) and ($|\overline{f}_{x_i} - d_{x_i}| \leq |\overline{g}_{y_j}|$) and
      ($|f_{\overline{x}_i} - d_{\overline{x}_i}| \leq |g_{\overline{y}_j}|$) and ($|\overline{f}_{\overline{x}_i} - d_{\overline{x}_i}| \leq |\overline{g}_{\overline{y}_j}|$)) begin
      $newM = C_{x_i} S_{y_j} (x_i \overline{\oplus} y_j) M$
      $(newf, newd, newg)$ = choose $(f_{x_i}, d_{x_i}, g_{y_j})$ or $(f_{\overline{x}_i}, d_{\overline{x}_i}, g_{\overline{y}_j})$
      if ($boolean\_match(newf, newd, newg, i + 1, newM)$ == match_found)
        return match_found
    end

    /* check the necessary conditions for $\overline{x}_i = y_j$ */
    if (($|f_{x_i} - d_{x_i}| \leq |g_{\overline{y}_j}|$) and ($|\overline{f}_{x_i} - d_{x_i}| \leq |\overline{g}_{\overline{y}_j}|$) and
      ($|f_{\overline{x}_i} - d_{\overline{x}_i}| \leq |g_{y_j}|$) and ($|\overline{f}_{\overline{x}_i} - d_{\overline{x}_i}| \leq |\overline{g}_{y_j}|$)) begin
      $newM = C_{x_i} S_{y_j} (x_i \oplus y_j) M$
      $(newf, newd, newg)$ = choose $(f_{x_i}, d_{x_i}, g_{\overline{y}_j})$ or $(f_{\overline{x}_i}, d_{\overline{x}_i}, g_{y_j})$
      if ($boolean\_match(newf, newd, newg, i + 1, newM)$ == match_found)
        return match_found
    end
  end
  return match_not_found
end
```

Figure 2: Boolean Matching

and sufficient condition for the matching to exist is that $newM = 1$ after all the variables are matched as in lemma 3.2.

The necessary condition given by lemma 3.6 to match x_i and y_j requires computing both f_{x_i} and $f_{\overline{x}_i}$ and comparing them with g_{y_j} and $g_{\overline{y}_j}$ respectively. When we match a second pair of variables $x_l = y_k$, we need to compute $f_{x_i x_l}, f_{x_i \overline{x}_l}, f_{\overline{x}_i x_l}$, and $f_{\overline{x}_i \overline{x}_l}$ and compare it with $g_{y_j y_k}, g_{y_j \overline{y}_k}, g_{\overline{y}_j y_k}$, and $g_{\overline{y}_j \overline{y}_k}$. This number grows exponentially as we match more variables.

When we set $x_i = y_j$, the pairs (f_{x_i}, g_{y_j}) and $(f_{\overline{x}_i}, g_{\overline{y}_j})$ must be matched respectively. We only choose one of the pairs (f_{x_i}, g_{y_j}) and $(f_{\overline{x}_i}, g_{\overline{y}_j})$ to be passed to the next step of the algorithm to be used for checking necessary conditions as given by lemma 3.6.

For example, let $f = x_1 x_2 x_3$ and $g = y_1 y_2 y_3$. First we try to find a match for variable x_1. $x_1 = y_1$ satisfies the necessary condition ($f_{x_1} = x_2 x_3, f_{\overline{x}_1} = 0, g_{y_1} = y_2 y_3$, and $g_{\overline{y}_1} = 0$). The pair $(f_{\overline{x}_1} = 0, g_{\overline{y}_1} = 0)$ cannot give us any further information because the necessary conditions are always satisfied for this pair irrespective of what variables are matched. On the other hand, the pair (f_{x_1}, g_{y_1}) contains all the information that we need. The following heuristic is used to choose one of the two pairs. If $(f_{x_i} - d_{x_i} = 0)$, or $(g_{y_j} = 1)$, the necessary conditions as given in lemma 3.6 are always satisfied. Therefore the other pair $(f_{\overline{x}_i}, g_{\overline{y}_j})$ is used to guide the matching. This same principle is used to check the other pair. If the above check is not enough, we choose either of f_{x_i} or $f_{\overline{x}_i}$ which has the larger difference between the number of onset points and offset points. The difference between the onset and offset points is computed as follows, $absolute\_value(|f_{x_i} - d_{x_i}| - |\overline{f}_{x_i} - d_{x_i}|)$ and $absolute\_value(|f_{\overline{x}_i} - d_{\overline{x}_i}| - |\overline{f}_{\overline{x}_i} - d_{\overline{x}_i}|)$.

This algorithm runs in linear time in the number of input variables for a library gate with one minterm in the onset or offset (AND, OR, NAND, NOR).

5 Don't Care Computation

The network is first decomposed into a set of trees. We compute compatible external plus observability don't cares at each of the nodes of the network as explained in [12]. These trees are sorted in topological order. Each tree is mapped after all its fanin trees. Image computation techniques are then used to find local don't cares at the leaves of the tree that is being mapped. The leaves of the tree correspond to primary inputs or roots of other trees that have been already mapped therefore their functions are fixed and the computed local don't cares are valid.

First we build BDD's corresponding to global functions (functions in terms of primary inputs) at each of the leaves of a tree. The observability plus external don't care set at the root of the tree is already computed. We find the care set at the root node and cofactor (generalized cofactor [15]) the global functions at the leaves with respect to the care set. The recursive image computation method [15] is then used to find all the reachable points. The inverse of the reachable points gives the local don't cares at the leaves of the tree.

To compute the local don't cares for a cluster function within the tree we repeat the above procedure. The tree itself is considered like a network and its local don't cares are treated as external don't cares or input combinations that never occur. We build BDD's for each of the leaves of a cluster function in terms of the leaves of the tree and cofactor them with respect to the care set of the whole tree. We then find all the unreachable points for the cluster function in terms of its leaves. We have found the best match at the leaves of the cluster function already. Because of the observability and external don't cares, the positive and negative phase functions at the leaves of a cluster are not necessarily inverses of each other; therefore, the local don't cares computed are not necessarily valid for both phases. Let $f_1^p, \dots, f_r^p$ and $f_1^n, \dots, f_r^n$ be the positive and negative matches found for the leaves, $y_1, \dots, y_r$, of a cluster. Let E_t be the external don't care for the tree, and let $d_i = f_i^p \oplus f_i^n$. A valid local don't care set for both phases of the cluster is

$$D = \overline{S_x(\overline{E_t}(y_1 \overline{\oplus} f_1^p + d_1) \dots (y_r \overline{\oplus} f_r^p + d_r))}.$$

This local don't care set is correct whether f_i^p or f_i^n plus an inverter is used as the function for the ith leaf. If only external don't cares of the tree are used but not the observability don't cares within the tree, then $d_i \overline{E_t} = 0$; therefore, there is no need to compute d_i.

It must be also mentioned that, the choice of the functions at the leaves of a cluster affects the local don't cares of that cluster. Hence, dynamic programming might not give the best result for the mapping of a tree when observability don't cares are used within a tree. The choice of the best function at the leaves may shrink the local network for the cluster and thus worsen the final results, although this is not very likely in practice.

In a circuit with large trees, there are usually many clusters that one has to consider. Computing local don't cares for all such clusters is a costly operation.

6 Library Organization

Before technology mapping, a setup phase is used to process gates in the library and generate particular data structures called *NUTS*. The term *NUT* is the abbreviation for Negative Unate Transform introduced in [5]. All the gates in a NUT are equivalent to a NUT representative in the sense that the function of each gate can be obtained by inverting some of the inputs of the NUT representative. The NUT structure reduces the number of calls to the Boolean matching algorithm. Finding the best match between a cluster function and the set of gates in the library is therefore reduced to the use of the matching algorithm on the cluster function and all the NUT representatives with the same number of inputs. The matching with the remaining gates is derived directly from the assignment information computed during the setup phase.

In the groups we build, we also consider the inversion of the outputs of the gates. This reduction is possible because our matching algorithm considers the matching with both phases of the input nodes at the same time. In our implementation, the 2-input functions NOR, NAND, AND and OR are in the same NUT and any of them can be the representative. Matching both phases of a node with these gates requires either a single call to the Boolean matching algorithm if a matching is found with the first phase being tried or just another call to find if the other phase matches or no match is possible.

Instead of computing the negative unate transforms of the input variables as in [5], we use the Boolean matching algorithm to place each gate in its corresponding NUT structure. The setup phase parses the library, reading one gate at a time. A gate is added to a NUT if it or its complement matches the NUT representative. If the gate does not match any of the existing NUT's, then a new NUT is created with the gate as its representative. Symmetries and symmetry groups are also computed for each representative at this time.

7 Results

We run the new technology mapping algorithm on a set of benchmarks chosen from MCNC and ISCAS combinational circuits and compared the results with technology mapping for area in SIS. Table 1 shows the result for combinational circuits without any external don't cares. These circuits are well optimized using the *rugged script* [13] in SIS. The MCNC library *lib2* is used for the mapping. The column **start** shows the literal count in factored form for each of these circuits. The columns **SIS**, **bm_no_dc**, **bm_tree_dc**, and **bm_full_dc** show the area of mapped circuits. We divide numbers given by the mapper by 464 (half the area of the smallest inverter) to get round small numbers. As shown in the table, considerable improvements are obtained for some circuits by just using Boolean matching without any don't cares (**bm_no_dc**). For these circuits, we get 8 percent improvement in area compared to technology mapping in SIS while spending 3.9 times as much time. The best improvement is obtained for *C6288* which is about 25 percent. The column **bm_tree_dc** shows the obtained area when don't cares are computed only for the leaves of each of the trees. The CPU times and the circuit areas are almost the same as the case with no don't cares. The BDD's cannot be built for some of these trees as shown in the table. The column **bm_full_dc** shows the result obtained by computing don't cares for each single cluster. The times spent for mapping are an order of magnitude more than SIS while there is 12 percent improvement in the final area of the mapped circuits. Although the time spent is substantially more than the time spent by tree matching algorithms, it is comparable to the time spent for circuit optimization.

If these circuits are not optimized first, the improvement over the technology mapping in SIS is very substantial. This is because by computing local don't cares and performing Boolean matching we do redundancy removal and a much stronger optimization on each circuit as opposed to tree matching. Even though the results on unoptimized circuits are better than the ones obtained from SIS, they are suboptimal to the ones obtained after running rugged script on each circuit.

circuit	start	SIS	CPU	bm_no_dc	CPU	bm_tree_dc	CPU	bm_full_dc	CPU
C432	218	437	5	398	31	397	157	381	574
C880	414	783	10	734	49	734	59	734	343
C1355	552	914	11	738	8	738	87	724	1184
C1908	535	933	11	810	27	810	118	793	1774
C2670	748	1339	20	1236	103	–	–	–	–
C3540	1283	2269	36	2176	213	–	–	–	–
C5315	1763	3055	45	3025	173	3025	229	3003	2285
C6288	3367	5453	68	4070	111	–	–	–	–
C7552	3022	4076	58	3690	190	–	–	–	–
z4ml	43	86	1	69	3	69	6	68	69
f51m	80	150	2	148	6	148	9	112	53
apex5	768	1473	19	1362	91	1361	127	1355	1180
apex6	732	1390	19	1345	97	1341	120	1336	882
alu4	102	200	2	196	10	196	11	180	103
rot	664	1283	16	1270	62	1267	74	1255	466
des	4214	5947	137	5698	596	5501	789	5498	11926

Table 1: Boolean Matching for Technology Mapping

start:	number of literals in factored form for the optimized circuits
SIS:	mapped using map -s in SIS
bm_no_dc:	mapped using boolean matching in SIS without don't cares
bm_tree_dc:	mapped using boolean matching in SIS with DC computed at the leaves of each tree.
bm_full_dc:	mapped using boolean matching in SIS with DC computed for each cluster
CPU:	in seconds on a IBM Risc System/6000 530

8 Conclusion

We have presented a new Boolean matching algorithm that can use don't cares and symmetries efficiently. We have applied this algorithm to technology mapping and have shown that the results of the mapper can be improved compared to tree matching techniques. The computation of local don't cares for each cluster function are discussed and techniques for such computations are presented. We have also organized the library of gates in an efficient way that reduces the number of times the Boolean matching algorithm is used. We developed ways to reduce the number of clusters generated in each tree and also more efficient don't care computation techniques to speed up the Boolean mapper. The same technique can be used for delay optimization and layout driven technology mapping.

9 Acknowledgements

We would like to thank Rajeev Murgai and Massoud Pedram for their helpful discussions and Professor Randy Katz for his support.

References

[1] R. E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[2] Ewald Detjens, Gary Gannot, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert Wang. Technology Mapping in MIS. In *International Conference on Computer Aided Design*, pages 116–119. IEEE, November 1987.

[3] Donald L. Dietmeyer and Peter Schneider. Identification of Symmetry, Redundancy and Equivalence of Boolean Functions. *IEEE Transactions on Electronic Computers*, EC-16(6):804–807, December 1967.

[4] K. Keutzer. Dagon: Technology Binding and Local Optimization by DAG Matching. In *24th ACM/IEEE Design Automation Conference*, pages 341–347, June 1987.

[5] F. Mailhot and G. D. Micheli. Technology Mapping Using Boolean Matching. In *European Design Automation Conference*, pages 180–185, March 1990.

[6] C. R. Morrison, R. M. Jacoby, and G. D. Hachtel. *Logic and Architecture Synthesis for Silicon Compilers*, chapter TECHMAP: Technology Mapping with Delay and Area Optimization, pages 53–64. Elsevier Science Publishers B.V.(North-Holland), 1989.

[7] S. Muroga. *Threshold Logic and its Applications*. John Wiley, 1971.

[8] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney. The Transduction Method - Design of Logic Networks Based on Permissible Functions. In *IEEE Transactions on Computers*, October 1989.

[9] M. Pedram and Bhat N. Layout Driven Technology Mapping. In *28th ACM/IEEE Design Automation Conference*, pages 99–105, San Francisco, June 1991.

[10] Rick Rudell. *Logic Synthesis for VLSI Design*. PhD thesis, U. C. Berkeley, April 1989. Memorandum UCB/ERL M89/49.

[11] H. Savoj and R. Brayton. The Use of Observability and External Don't Cares for the Simplification of Multi-Level Networks. In *27th ACM/IEEE Design Automation Conference*, Orlando, June 1990.

[12] H. Savoj, H. Touati, and R. K. Brayton. Extracting Local Don't Cares for Network Optimization. In *IEEE International Conference on Computer-Aided Design*, November 1991.

[13] H. Savoj, H-Y. Wang, and R. Brayton. Improved Scripts in MIS-II for Logic Minimization of Combinational Circuits. In *International Workshop on Logic Synthesis*, May 1991.

[14] H. Touati. private communication, 1990.

[15] H. Touati, H. Savoj, B. Lin, R. Brayton, and A. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *IEEE International Conference on Computer-Aided Design*, November 1990.