We describe a new approach to technology mapping where matchings are recognized by means of Boolean operations. The computer-aided synthesis of a logic circuit involves two major steps: the optimization of a technology independent logic representation, using Boolean and/or algebraic techniques, and technology mapping. Technology mapping is the task of transforming an arbitrary multiple-level logic representation into an interconnection of logic elements from a given library of elements. Technology mapping is a very crucial step in the synthesis of semi-custom circuits for different technologies, such as sea-of-gates, gate-arrays or standard cells. The quality of the synthesized circuits, both in terms of area and performance, depends heavily on this step. For this reason, several approaches to technology mapping have been pursued and implemented in research and commercial mapping tools. Unfortunately, the mapping problem is a difficult one from a computational complexity stand-point. Therefore, rule-based technology mapping techniques [6, 12] and heuristic algorithms have been proposed [3, 8, 11, 13, 14, 17].

In this paper, we consider an algorithmic approach to technology mapping problem that extends the pioneering work of Keutzer on Dagon [11] and of Detjens [8] and Rudell [19] on MIS. To put our work in perspective, we briefly summarize their approach.

Technology mapping consists of three major tasks. First Boolean networks are partitioned into an interconnection of single-output sub-networks, with the property that each internal vertex has unit outdegree (i.e. fanout). Then each sub-network is decomposed into an interconnection of two-input functions (e.g. AND, OR, NAND or NOR). Each sub-network is modeled by a directed acyclic graph (DAG), called subject graph. Finally each subject graph is covered by an interconnection of library cells.

Finding a cover of a subject graph that optimizes area or timing is a difficult problem. Keutzer proposed to represent library functions by trees and to use a dynamic programming technique for optimal covering, based on fast tree matching algorithms. A similar approach was used by Rudell and Detjens [8, 19]. Noting that in a cell-based design (and timing) of a mapped network depends on the partitioning, decomposition and covering tasks. However, good results were achieved by this approach and extensions based on DAG matching presented by Detjens [8] did not show substantial improvements.

We consider in this paper an approach to the technology mapping that uses network partitioning and decomposition techniques similar to those used in [11], with an improved covering algorithm. The covering algorithm described in this paper does not use the tree-based representation. Instead, it uses Boolean matching techniques based on Shannon decomposition [4] for recognizing whether a logic function can be implemented by a library cell. The rationale for this choice is that the representation of single-output networks by trees makes cumbersome (and in some cases impossible) the efficient mapping of logic functions that have multiple occurrences of some variables into networks of gates that have also multiple occurrences of some variables (e.g. exclusive ORs or majority functions). Boolean techniques support uniformly the description and the matching of any single-output library cell, independently of its functionality. In addition, Boolean matching techniques can take advantage of don't care information.

The importance of the use of don't care conditions in multiple-level logic synthesis is well recognized [1]. In this paper we consider don't care conditions that are specified at the network boundary and that arise from the network interconnection itself [16]. Don't care conditions are usually exploited to minimize the number of literals (or terms) of each expression in a Boolean network. While such a minimization leads to a smaller (and faster) implementation in the case of multi-cell design style [5] (or PLA-based design), it may not improve the local area and timing performance in a cell-based design. For example, cell libraries exploiting pass-transistors might be faster and/or smaller than other gates having less literals. A pass-transistor based multiplexer is such a gate: assuming a function is defined by its on set \( F \) and its don't care set \( \overline{DC} \):

\[
F = (a+b)c
\]
\[
\overline{DC} = \overline{b}c
\]

then \((a+b)c\) is the representation that requires the least number of literals (3), and the corresponding logic gate is implemented by 6 transistors. On the other hand, \(\overline{a}b + \overline{b}c\) requires one more literal (4), but is implemented by only 4 pass-transistors, and is likely to be faster. Since Boolean minimization may lead to sub-optimal results, we propose to use directly the don't care conditions during Boolean matching in the search for the best implementation in terms of area (or timing).

This paper is organized as follows. We first show how to determine Boolean matchings and we show techniques for speeding-up the matching operation. We then examine how don't care conditions can be used in conjunction with Boolean matching. Eventually we present a covering algorithm based on Boolean matching and we conclude by showing the effectiveness of this technique on benchmark circuits.
2 Boolean Matching

Matching is the key operation of the technology mapping process. It identifies whether an element of the library can be used to implement a part of a given Boolean function. Matching can be formulated as checking the tautology between a given Boolean function, called the target function, and the set of functions representing a library element, for any permutation of its variables. We also consider the phase assignment problem in connection with the matching problem, because they are closely interrelated in affecting the cost of an implementation. Finally, we include the don't care set of the target function during the matching operation.

We denote the target function \( F(x_1, \ldots, x_n) \) as a polynomial in \( x_1, \ldots, x_n \) having \( n \) inputs and one output. We denote the phase of variable \( x_i \) by \( \phi_i \in \{0,1\} \), where \( x_i^\phi = x_i \) for \( \phi = 1 \), \( x_i^\phi = \overline{x_i} \) for \( \phi = 0 \). We denote the don't care set of the target function by \( D(F(x_1, \ldots, x_n)) \). We denote the library by \( L : \{g_0, \ldots, g_m\} \), for \( m \leq 2^n \) elements \( g_i \) are multiple-input single-output functions. We define the matching problem as follows:

Given a target function \( F(x_1, \ldots, x_n) \), its don't care set \( D(F(x_1, \ldots, x_n)) \), and a library element \( g(y_1, \ldots, y_n) \), find an ordering \( \{i_1, \ldots, j_n\} \) and a phase assignment \( \{\phi_1, \ldots, \phi_n\} \) of the input variables of \( F \), such that either equation (1) or (2) is true:

\[
F(x_1^\phi_1, \ldots, x_n^\phi_n) = g(y_1^\phi_1, \ldots, y_n^\phi_n) \tag{1}
\]
\[
F(x_1^\phi_1, \ldots, x_n^\phi_n) = g(y_1^\phi_1, \ldots, y_n^\phi_n) \tag{2}
\]

for each value of \( \{y_1, \ldots, y_n\} \) and each care value of \( \{x_1^\phi_1, \ldots, x_n^\phi_n\} \notin D(F) \), i.e. equation (1) or (2) is a tautology for all minima not in the don't care set.

If no such ordering and phase assignment exist, then the element \( g \) does not match the target function \( F \). Furthermore, if no element in the library \( L : \{g_0, \ldots, g_m\} \) matches \( F \), then \( F \) cannot be covered by the library \( L \).

In other words, if we define the NPN-equivalent set of a function \( F \) as the set of all the functions obtained by input variable Negation, input variable Permutation and function Negation [18], we say that a function \( F \) matches a library element \( g \) when there exist a NPN-equivalent function which is tautological to \( g \) modulo the don't care set.

For example, any function \( F(y_1, y_2) \) in the set: \( \{y_1 + y_2 + \overline{y_1}, y_1 + \overline{y_2}, y_1 \overline{y_2}, y_2 \overline{y_1}\} \) can be covered by the library element: \( g(x_1, x_2) = x_1 + x_2 \). Note that in this example \( g(x_1, x_2) \) has \( n = 2 \) inputs, and can match \( 2^2 = 8 \) functions [10, 17].

2.1 A Simple Boolean Matching Algorithm

A Boolean match can be determined by verifying that there exists a matching of the input variables such that the target function \( F \) and the library function \( g \) are a tautology. Tautology can be checked by recursive Shannon decomposition [4]. The two Boolean expressions are recursively cofactorized generating two decomposition trees. The two expressions are a tautology if they have the same logic value for all the leaves of the recursion that are not in the don't care set. This process is repeated for all possible orderings of the variables of \( F \), or until a match is found.

The matching algorithm is described by the recursive procedure simple_boolean.match shown in figure 1, which returns TRUE when the arguments are a tautology for some ordering. At level \( n \) of the recursion, procedure simple_boolean.match is invoked repeatedly with arguments the cofactors of the \( i^{th} \) variable of \( F \) and the cofactors of all the variables of \( F \) until a match is found, in which case the procedure returns TRUE. If no match is found, the procedure returns FALSE. The recursion stops when the arguments are constants, i.e. when all variables have been cofactorized, in which case the procedure returns TRUE when the corresponding values match modulo the don't care condition. Note that when a match is found, the sequence of variables used to cofactor \( F \) in the recursion levels \( 1 \) to \( N \) represents the order in which they are to appear in the corresponding library element. The algorithm is shown in figure 1.

Note that in the worst-case all permutations and phase assignments of the input variables are considered. Therefore, up to \( n! \cdot 2^n \) different Shannon decompositions may be required for each match. The worst-case computational complexity of the algorithm makes it practical only for small values of \( n \).

simple_boolean.match(\( F(x_1, \ldots, x_n), \varphi \), \( g(y_1, \ldots, y_n) \))

if \( (\varphi = 1) \) return (TRUE)

if \( (F \text{ and } g \text{ are constant}) \) or \( (|F| = 0) \) return \( (\varphi = 0) \)

which \( \varphi = 1 \)

\( gw = \text{pick a variable}(\{x_1, \ldots, x_n\}, \varphi) \)

remaining \( gw, g = \text{get remaining}(\{y_1, \ldots, y_n\}, g, \varphi) \)

while ( \( \text{which} \varphi \neq \text{true}(\varphi) \) )

\( g = \text{shannon decomposition}(\{y_1, \ldots, y_n\}, \varphi) \)

\( g' = \text{shannon decomposition}(\{y_1, \ldots, y_n\}, \varphi) \)

\( g' = \text{shannon decomposition}(\{y_1, \ldots, y_n\}, \varphi) \)

return \( (\varphi = 0) \)

\( \text{return} \) (TRUE)

if \( (\text{simple_boolean.match}(\{g \text{ library} \}) \text{ and } \text{remaining \( g \text{ library} \}) \neq \text{remaining \( g \text{ library} \}) \) \)

\( \text{return} \) (TRUE)

\( \text{return} \) (FALSE)

Figure 1: Simple Algorithm for Boolean Matching

2.2 Speeding Up Boolean Matching

We present in this section some techniques to speed-up the matching of completely specified functions. Uncompletely specified functions are dealt with in the following sections.

To increase the efficiency of the Boolean matching process, it is important to remark that the phase information of the unate variables is irrelevant to determine the matching. Therefore we define a transformation \( T \) that complements the input variables that are negative unate. Note that the phase information cannot be taken away from binate variables, where both the positive and negative phases are required to express \( F \). By using this transformation, we reduce the information required for the matching and therefore reduce also its computational cost. For example, any function \( F(y_1, y_2) \) in the set: \( \{y_1 + y_2 + \overline{y_1}, y_1 + \overline{y_2}, y_1 \overline{y_2}, y_2 \overline{y_1}\} \) can be represented by the set: \( \{y_1 + y_2, y_1 y_2\} \).

As a result, we redefine the matching problem as follows:

Given a target function \( F(x_1, \ldots, x_n) \) and a library element \( \{y_1, \ldots, y_n\} \), find an ordering \( \{i_1, \ldots, i_n\} \) and a phase assignment \( \{\phi_1, \ldots, \phi_n\} \) of the binate variables \( \{k_1, \ldots, k_n\} \) of \( F \), such that either (3) or (4) is true:

\[
T(F(x_1^\phi_1, \ldots, x_n^\phi_n)) = T(G(y_1^\phi_1, \ldots, y_n^\phi_n)) \tag{3}
\]
\[
T(F(x_1^\phi_1, \ldots, x_n^\phi_n)) = T(G(y_1^\phi_1, \ldots, y_n^\phi_n)) \tag{4}
\]

The following considerations are also important in reducing the computational complexity:

- Any input permutation must associate each unate (binate) variable in the target function to a unate (binate) variable in the function of the library element.
- Variables or groups of variables that are interchangeable in the target function must be interchangeable in the function of the library element.

The first point implies that if the target function has \( m \) binate variables, then only \( m! \cdot (n - m)! \) permutations of the input variables are needed.

The second point implies that symmetry classes can be used to simplify the search. A symmetry class is a set of variables that are interchangeable without affecting the logic functionality [15]. Techniques based on symmetry considerations to speed-up algebraic matching were also presented by Morrison in [17].

For a given function \( F(x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n) \), \( x_i \) and \( x_j \) belong to the same symmetry class if

\[
F(x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n) \equiv F(x_1, \ldots, x_j, \ldots, x_i, \ldots, x_n)
\]
The symmetry property is an equivalence relation (it is reflexive, symmetric and transitive), hence if \( [x_1, x_2] \) and \( [x_3, x_4] \) are two symmetry sets, then \( [x_1, x_2] \) is also a symmetry set.

Symmetry classes are used in two different ways to reduce the search space. First, they are used as a filter to quickly find good candidates for matching. A necessary condition for matching a target function \( \mathcal{F} \) by library element \( \mathcal{G} \) is that both have exactly the same symmetry classes. Hence only a small fraction of the library elements need be checked by the computationally intensive Boolean comparison to see if they match the logic equation.

The symmetry classes for each library element are calculated once before invoking the mapping algorithm.

Second, symmetry classes are used during the Boolean comparison itself. Once a library element \( \mathcal{G} \) that satisfies the previous requirement is found, the symmetry sets of \( \mathcal{F} \) are compared to those of \( \mathcal{G} \). Then only variables belonging to symmetry sets of the same size can possibly produce a match.

Since all variables from a given symmetry set are equivalent, the ordering of the variables within the set is irrelevant. This implies that the permutations need only be done over symmetry sets of the same size. Thus the number of permutations required to detect a match is: \( \prod_{i} \big| S_i \big| \), where \( S_i \) is the number of sets of cardinality \( i \), and \( q \) is the size of the largest symmetry set. Although in the worst case logic equations might have no symmetry at all, our experience with commercial standard cells and programmable logic devices libraries (such as CMOS3, LSI Logic or Actel) is that the elements are highly symmetrical, the average \( S_i \) being less than 2, as shown in Table 1. For example, the gate MUX \( (x'\bar{y}+y'\bar{z}+z'y) \) in the Actel library has 4 sets of single unate variables (\( \{a\}, \{b\}, \{c\}, \{d\} \)), and 2 sets of single binate variables (\( \{x', y\} \)).

### Table 1: Average number of symmetry sets for different libraries

<table>
<thead>
<tr>
<th>Library</th>
<th>Average ( S_i )</th>
<th>Maximum ( S_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actel</td>
<td>1.29</td>
<td>4</td>
</tr>
<tr>
<td>CMOS3</td>
<td>1.27</td>
<td>4</td>
</tr>
<tr>
<td>LSI Logic</td>
<td>1.32</td>
<td>4</td>
</tr>
</tbody>
</table>

Unateness information and symmetry classes are used together to further reduce the search space. Unate and binate symmetry sets are distinguished, since both unateness and symmetry properties have to be the same for two variables to be interchangeable. Thus \( S_i = S_i' + S_i'' \), where \( S_i' \) is the number of sets of cardinality \( i \) made of unate variables, \( S_i'' \) is the number of sets of cardinality \( i \) made of binate variables. This further reduces the number of permutations to \( \prod_{i} \big| S_i \big| = \prod_{i} \big| S_i' \big| \times \prod_{i} \big| S_i'' \big| < \prod_{i} \big| S_i \big| \).

Hence, when considering the phase assignment of the binate variables, at most \( \prod_{i} \big| S_i' \big| \times (S_i - S_i'') \) trials have to be made in order to find a match. In the Actel library, the worst case occurs for the library element MUX \( (x'\bar{y}+y'\bar{z}+z'y) \), where \( S_1 = 7 \), and \( S_2 = 4 \). In that case, \( 4! \times 3! \times 2^2 = 1152 \times 7! \times 2^2 = 645,120 \), where \( 7! \times 2^2 \) represents the number of trials needed if no symmetry information is used.

Procedure boolean.match, a variation on procedure simple.boolean.match is shown in figure 2. It incorporates the symmetry information to reduce the search space: permutations are done only over symmetry sets of the same size.

### 2.3 Use of the Don’t Care Sets

When don’t care conditions are considered, the target function \( \mathcal{F} \) cannot be uniquely characterized by a symmetry set. Therefore the techniques based on symmetry sets presented in the previous section no longer apply.

A straightforward approach is to consider all the functions \( \mathcal{H} \) that can be derived from \( \mathcal{F} \) and its don’t care set \( \mathcal{P} \). Unfortunately, there are \( 2^N \) possible combinations, where \( N \) is the number of minterms in \( \mathcal{P} \). Therefore this approach can be used only for small don’t care sets. For large don’t care sets, a pruning mechanism has to be used to limit the search space.

We introduce now a representation of \( n \)-variable functions that exploits the notion of symmetry sets and NPN-equivalence and that can be used to determine matchings while exploiting the notion of don’t care conditions. For a given number of input variables \( n \), let \( G(V, E) \) be a graph whose vertex:

- \( \text{boolean.match}(g, \text{symmetry}_a, \text{symmetry}_b) \) if ( \( g \) and \( b \) are consistent or \( g = 0 \)) return ( \( g \) = \( g \) )
- ( \text{symmetry}_a = \text{get.next}(\text{symmetry}_a) \) \text{symmetry}_a = \text{min.diff}(\text{symmetry}_a)
- while ( \text{symmetry}_a set of \( g \) with \( \text{symmetry}_a \) size have still to be tried)
  - \( \text{symmetry}_a = \text{get.next}(\text{symmetry}_a) \)
  - \text{for} \( \text{var} \) in \( \text{variables}(\text{symmetry}_a) \)
  - \text{var} = \text{pick.a.variable}(\text{symmetry}_a)
  - \text{var} = \text{pick.a.variable}(\text{symmetry}_a)
- \( \text{return}(\text{TRUE}) \)

Figure 2: Algorithm for Fast Boolean Matching

Each vertex \( v \) in the graph is annotated with the library elements that matches the corresponding function, when a match exists. Then, for a given target function \( \mathcal{F} \) the corresponding vertex \( v \) is determined. There exists a matching to the cell \( \mathcal{G} \) if there is a path in the graph \( G(V, E) \) such that the function represented by \( v \) and \( v \) differ in one minterm. Such a graph \( G(V, E) \) for \( n = 3 \) is shown in figure 3.

![Figure 3: Matching compatibility graph for 3-variable Boolean space](image)

### 3 A Covering Algorithm

The logic circuit to be mapped is partitioned into subject graphs \( \{ f_1, \ldots, f_n \} \), that are decomposed into an interconnection of two-input
A cluster is a connected sub-graph of the subject graph \( G' \), having only one vertex with zero out-degree \( v_j \). It is characterized by its depth (longest directed path from \( v_j \)) and its number of inputs. The associated cluster function is the Boolean function obtained by collapsing [3] the Boolean expressions associated to the vertices into a single Boolean function. We denote all possible clusters containing the vertex \( v_j \) of \( G' \) by \( \{ v_{j_1}, \ldots, v_{j_n} \} \).

As an example, consider the Boolean network (after an AND/OR decomposition):

\[
\begin{align*}
\text{sink vertex } v_1 = f: \\
\text{node } 1 &= b \\
\text{node } 2 &= c + x_1 \\
\text{node } 3 &= x_3 + d \\
\text{node } 4 &= a + c
\end{align*}
\]

There are six possible cluster functions for the subject graph \( G' \), containing the sink vertex \( v_1: \)

\[
\begin{align*}
n_{1,2} &= b \\
n_{1,3} &= c + x_1 \\
n_{1,4} &= (c + x_1)(a + c) \\
n_{2,3} &= (c + x_1)(a + c) \\
n_{2,4} &= (c + x_1)(a + c) \\
n_{3,4} &= (c + x_1)(a + c)
\end{align*}
\]

The algorithm attempts to match each cluster function \( n_{ij} \) to a library element. The area cost of a cover is computed by adding to the cost of the matching of the cluster \( n_{ij} \) under consideration the cost of the clusters corresponding to the variables in the Boolean function for \( n_{ij} \). For any cluster, there is always a match, because the network was decomposed into AND/ORs in the initial setup phase. When matchings exist for multiple clusters, then the choice of the matching of minimal area cost guarantees minimality of the total area cost of the matched sub-graph [11, 8], for the particular AND/OR decomposition under consideration. The cost of the required inverters is also taken into account at this stage.

The timing cost of a cover can be computed in a similar way. The propagation delay through a cluster is added to the maximum of the arrival times at its inputs, to compute the total time at the vertex \( v_j \). [19] When matchings exist for multiple clusters, then the choice of the matching of minimal local time guarantees minimality of the total timing cost of the matched sub-graph, again for the particular AND/OR decomposition under consideration.

The covering algorithm is implemented by procedure \textit{get.bigger.function} shown in figure 5.

As the subject graph is being mapped, the \textit{don't care} set changes accordingly. For example, let us assume that the chosen matchings for \( k \), \( x_1 \) and \( x_2 \) are \( k = \text{OR3}(f, d, c) \), \( x_1 = \text{OR2}(a, c) \) and \( x_2 = \text{OR2}(f, d) \). Then the satisfiability \textit{don't care} sets associated with these variables are:

\[
\begin{align*}
P_{n,1} &= b \overline{c} + \overline{d} \overline{c} + c + d + e \\\nP_{n,2} &= x_3 \overline{e} + \overline{f} (a + c) \\\nP_{n,3} &= x_3 \overline{e} + \overline{f} (a + c) \\\nP_{n,4} &= x_3 \overline{e} + \overline{f} (a + c) \end{align*}
\]

These \textit{don't care} sets can be used for Boolean matching in the rest of the circuit. For example, let's assume the cluster \( v_{ij} \) of \( f \) is being processed. Then, its associated cluster function is \( f(a + c) \). However, \( f \) is part of the \textit{don't care} set \( P_{n,1} \), and can be used during the matching. In this particular case, the functions \( f(a + c) \) and \( a + b + c \) are both valid matches. For some technologies, the second option may be preferred, (i.e. a multiplexer may be better than an AND-OR gate).

**Figure 5: Algorithm for network covering**

<table>
<thead>
<tr>
<th>Function</th>
<th>Don't Care</th>
<th>Library element</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a(b + c) )</td>
<td>( \overline{b} )</td>
<td>AO32A,MX2</td>
</tr>
<tr>
<td>( a(b + c) )</td>
<td>( \overline{b} )</td>
<td>OA32,MA3</td>
</tr>
<tr>
<td>( a(b + d) )</td>
<td>( \overline{b} \overline{c} + \overline{d} )</td>
<td>AO32,A01C,A03</td>
</tr>
<tr>
<td>( a(c + d) )</td>
<td>( \overline{b} \overline{c} )</td>
<td>AND3,A01A</td>
</tr>
</tbody>
</table>

**Table 3: Compatible library elements (Actel)**

4 Implementation and Results

The technology mapping algorithms have been implemented in a program called Ceres. Ceres reads the logic description of the circuits and of the library in a description language called SLIF (Structure and Logic Interchange Format) [7]. SLIF allows for the description of sequential networks, which are also mapped by program Ceres. Search limiting heuristics, under the control of the user, are also available to speed up the execution times. Ceres has been tested on DECStation 3100 and 3200, as well as on SUN workstations. We used the benchmarks circuits provided for the 1989 MCNC Logic Synthesis Workshop. Starting from the original description of the benchmarks, we compared the results of our technology mapping algorithm to the ones of the technology mapper built in MIS-2I release 2.1. We used three different libraries: Actel, LSI Logic and CMOS3. Area was chosen as the metric for the final implementation, counting inverters as well as other logic gates in the total cost.

Procedures \textit{simple:boolean.match, boolean.match, traverse.graph and get.bigger.function} have been coded and tested in Ceres. However,
only procedure boolean.match is currently integrated within procedure
get.bigger.function. Hence, results of technology mapping reported in ta-
ble 2 do not reflect the use of don’t care. Examples of running procedure
traverse_graph are reported separately in Table 3, and show how functions
can match multiple library elements.

The run times on a DECstation 3100 ranged from a few seconds for
majority to less than 3 minutes for e4f. Table 2 summarizes our results.
It is worth noting that the run times from program Ceres were consistently
small, for the three libraries used. In particular, the run times using the Actel
library, which has a wealth of XORS, multiplexers and majority functions [9],
were comparable to the run times using the other two libraries, indicating that
the Boolean matching algorithm used in Ceres handles these types of
gates very efficiently.

5 Conclusions and Future Work

We have presented a different algorithmic approach to technology mapping.
Boolean operations are used during the matching step, making it possible to
recognize quickly such functions as exclusive ORs and majority functions.
Different filters and simplifications have been proposed to reduce the search
space that would otherwise be very large. Results have shown that this ap-
proach, in its present implementation, is competitive with other algorithmic
methods based on tree or DAG covering.

We have studied the use of don't cares in connection with technology map-
ning. Our experiments have shown that the present techniques are limited to
library cells of at most four variables. However, most library elements
fit into this class. The experiments have also shown that the use of don't
cares leads to a wider choice for matching library elements. Future work
will include the full integration of this technique into Ceres.

Acknowledgements

The authors wish to thank David C. Ku for numerous and fruitful discus-
sions. This work is supported in part by the National Science Foundation
under grant MIP-8719546, by DEC, ATT and NSF under a FYI award, by a
Stanford CIS seed grant, by the National Sciences and Engineering Council
of Canada, and by the Quebec Ponds P.C.A.R.

References

Sangiovanni-Vincentelli and A.S. Wong, Multilevel Logic Minimization Using Implicit


Level Logic Optimization System, IEEE Transactions on CAD, Vol. CAD-6, No. 6,


Sijllhorst, The Berkeley Silicon Compiler System, in Digital Silicon Compilation,


tecture of Electrically Configurable Gate Arrays, IEEE Journal of Solid-State circuits,


tically Synthesizing and Optimizing Combinational Logic, Proceedings of the 21st


[15] E.J. McCluskey, Detection of Group Invariance or Total Symmetry of a Boolean Func-

[16] P. McCreen and R.R. Brayton, Consistency and Observability Invariance in Multi-Level

Delay and Area Optimization, in Logic and Architecture Synthesis for Silicon Compilers,
G. Sacca and P.M. McLellan editors, North-Holland, pp.53-64, 1989.


216