MUSTANG: State Assignment of Finite State Machines Targeting Multilevel Logic Implementations

SRINIVAS DEVADAS, HI-KEUNG MA, A. RICHARD NEWTON, FELLOW, IEEE, AND A. SANGIOVANNI-VINCENTELLI, FELLOW, IEEE

Abstract—**In this paper, we address the problem of the state assignment for synchronous finite state machines (FSM), targeted towards *multilevel combinational logic* and feedback register implementations. Optimal state assignment aims at a minimum area implementation. All previous work in automatic FSM state assignment has been directed at programmable logic array (PLA) i.e., two-level logic implementations. In practice, most large FSM's are not synthesized as a single PLA for speed and area reasons—multilevel logic implementations are generally used for smaller delay and area. In this paper, we present state assignment algorithms that heuristically *maximize the number of common cubes* in the encoded network so as to minimize the number of literals in the resulting combinational logic network *after* multilevel logic optimization. We present results over a wide range of benchmarks which prove the efficacy of our techniques. *Literal counts averaging* 20–40 percent less than other state assignment programs have been obtained.**

I. Introduction

IN THIS PAPER address the problem of encoding the states (*state assignment problem*) of synchronous finite state machines (FSM), targeted towards *multilevel* combinational logic and feedback register implementations. We assume that an optimal state assignment is a state assignment which yields minimum area in the final implementation.

All previous work in *automatic* FSM state assignments has been directed at the minimization of the number of product terms in a sum-of-products form of the combinational logic [1]–[4], [5]–[8] and hence, the results obtained are relevant for the cases where the combinational logic is implemented using programmable logic arrays (PLA's). In practice, most large FSM's cannot be synthesized as a single PLA for area and/or performance reasons—*multilevel* logic implementations are generally used for smaller delays or smaller areas (or both). Results using manual state assignment have shown that existing automatic state assignment techniques are inadequate for producing optimal multilevel logic implementations [9].

In this paper, we present a strategy for finding a state assignment of a FSM which heuristically minimizes an estimate of the area used by a multilevel implementation of the combinational logic.

1.1. Need for New Techniques of State Assignment

Existing automatic state assignment techniques targeting optimal PLA implementations of combinational logic are inadequate for producing optimal multilevel implementations. This is illustrated in Figs. 1–3. Using a state assignment program targeted toward PLA implementations the states of the FSM in Fig. 1 are given the codes in Fig. 2(a). This encoding produces a six product term PLA (Fig. 2(b)) after two-level minimization. After multilevel logic optimization, the resulting network contains 16 gates and is shown in Fig. 2(c). A different assignment of codes (Fig. 3(a)) produces a larger PLA with seven product terms (Fig. 3(b)), but a smaller multilevel logic network with 15 gates (Fig. 3(c)). This example illustrates the need for state assignment techniques targeted toward a different object, namely, optimal *multilevel* implementations of FSM combinational logic.

1.2. State Assignment for Multilevel Logic Implementations

In the sequel, we present a strategy for finding a state assignment of a FSM which minimizes an estimate of the area used by a multilevel implementation of the combinational logic. The estimate considered here is consistent with the estimate used by multi-level logic optimization algorithms [10]–[12]: the *number of literals in a factored form* for the logic. We have developed algorithms which produce a state assignment that heuristically minimizes the number of literals in the resulting combinational logic network *after* multilevel logic optimization.

Multilevel logic optimization programs like MIS [11] and SOCRATES [12] primarily use algebraic techniques for factorizing and decomposing the Boolean equations by identifying common sub-expressions. Our heuristics are based on *maximizing the number and size of common subexpressions* that exist in the Boolean equations that de-

Manuscript received December 22, 1987; revised July 18, 1988. This work was supported in part by the Semiconductor Research Corporation under Grant 442427-52055, by the Defense Advanced Research Advanced Projects Agency under Contract N00039-86-R-0365, and by a grant from AT&T Bell Laboratories. The review of this paper was arranged by Associate Editor M. R. Lightner.

This is an expanded version of the work originally presented at the ICCAD-87.

The authors are with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720.

IEEE Log Number 8824033.

```
-0 st0   st0 0
11 st0   st0 0
01 st0   st1 -
0- st1   st1 1
11 st1   st0 0
10 st1   st2 1
1- st2   st2 1
00 st2   st1 1
01 st2   st3 1
0- st3   st3 1
11 st3   st2 1
```

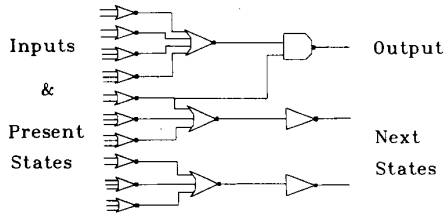
Fig. 1. Example FSM.

```
st0 -> 00    st1 -> 01
st2 -> 11    st3 -> 10
```

(a)

```
10 -1    10 0
11 1-    01 0
0- 10    10 1
01 0-    01 1
-0 -1    01 1
-1 1-    10 1
```

(b)



(c)

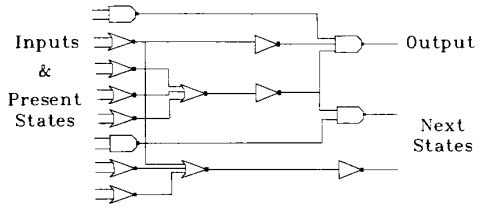
Fig. 2. (a) State assignment. (b) Minimized PLA implementation. (c) Optimized multilevel implementation.

```
st0 -> 00    st1 -> 10
st2 -> 01    st3 -> 11
```

(a)

```
10 10    01 1
0- 11    01 0
01 --    10 0
1- 01    01 1
0- 1-    10 1
-1 -1    01 1
0- -1    10 1
```

(b)



(c)

Fig. 3. (a) State assignment. (b) Minimized PLA implementation. (c) Optimized multilevel implementation.

scribe the combinational logic part of the FSM after the states have been encoded but *before* logic optimization. The state assignment algorithms find pairs or clusters of states which, if kept minimally distant in the Boolean space representing the encoding, result in a large number of common sub-expressions in the Boolean network.

We have obtained results over a wide range of benchmarks which illustrate the efficacy of our techniques. Literal counts averaging 20–40 percent less than the state assignment program KISS [5] and random assignment techniques have been obtained.

Preliminaries and definitions are given in Section II. In Section III, the basic approach followed to obtain a good state assignment is described. In Section IV, two algorithms are presented. The embedding algorithm used is presented in Section V. Results on the benchmark examples are presented in Section VI.

II. Preliminaries

2.1. Basic Definitions

A *variable* is a symbol representing a single coordinate of the Boolean space (e.g., a). A *literal* is a variable or its negation (e.g., a or $\bar{a}$). A *cube* is a set C of literals such that $x \in C$ implies $\bar{x} \notin C$ (e.g., $\{a, b, \bar{c}\}$ is a cube, and $\{a, \bar{a}\}$ is not a cube). A cube represents the conjunction of its literals. The trivial cubes, written 0 and 1, represent the Boolean functions 0 and 1, respectively. An *expression* is a set f of cubes. For example, $\{\{a\}, \{b, \bar{c}\}\}$ is an expression consisting of the two cubes $\{a\}$ and $\{b, \bar{c}\}$. An expression represents the disjunction of its cubes.

2.2. Representations of FSM's

An FSM is represented by two equivalent structures.

(1) It is *State Transition Graph* $G(V, E, W(E))$ where V is the set of vertices coresponding to the set of states S, were $\| S \| = N_S$ is the cardinality of the set of states of the FSM, an edge (v_i, v_j) joins v_i to v_j if there is a primary input that causes the FSM to evolve from state v_i to state v_j, and $W(E)$ is a set of labels attached to each edge, each label carrying the information of the value of the input that caused that transition and the values of the primary outputs corresponding to that transition.

(2) It is a *State Transition Table* $T(I, S, O)$ where I is the set of inputs, S is the set of states as above, and O is the set of outputs. We assume that the primary inputs and outputs of the FSM are in Boolean form. A row of the table corresponds to an edge in the state-transition graph. The table has as many rows as edges of state graph and as many columns as

$$N_i + N_o + 2$$

where N_i is the number of bits used to encode the inputs, N_o is the number of bits used to encode the outputs, and 2 refers to the present state and the next state. The matrix has Boolean entries for the inputs and outputs and "symbolic" entries for the columns corresponding to the present and next states, carrying the name of the present state and of the next state, respectively. The rows of the matrix are divided into two fields: the first field contains the input pattern and the names of the present state, the second field contains the output pattern and the names of

the next state. Note that the input pattern may contain don't care entries.

2.3. State Assignment for Multilevel Logic

The state assignment problem consists of assigning a string of bits (code) to each of the states, so that no two states have the same code. After a code has been assigned, the FSM can be implemented trivially once the storage elements (flip-flops) have been chosen, with a PLA. For example, assume that the storage elements are D flip-flops (one per bit). Then, each edge (v_i, v_j) of the state transition graph or row of the state transition table, corresponds to a product term, with the input part represented by the bits specified in the label $w((v_i, v_j))$ for the primary input and the bits forming the code for v_i (the present state), and the output part represented by the bits forming the code for v_j and the bits specified in $w((v_i, v_j))$ for the primary outputs. This representation of the FSM can be optimized using a two-level logic minimizer as ESPRESSO [13] to reduce the number of product terms needed to implement the logic function. Of course, different encoding of the states yield different logic functions. It is of great interest to assign codes to states so that the final optimized PLA has a few product terms as possible. Algorithms have been proposed that solve this problem by using a symbolic optimization step to determine a set of constraints on the encoding to guarantee that certain product terms could be eliminated in the final implementation [5], [7]. However, in some cases, the size of the PLA remains too large to satisfy timing or area constraints. In this case, a multilevel implementation of the logic is in order. The two-level logic description is then mapped into a multilevel implementation by factoring and decomposing the logic functions corresponding to the outputs. According to the particular target technology, e.g., CMOS standard cells, CMOS static gates laid out in the gate-matrix style or Weinberger arrays, a decomposition and factorization will be more effective than others.

Algorithms have been proposed that perform this step effectively (e.g., [10]–[12]). These algorithms represent the logic to implement as a *Boolean network*, i.e., a directed graph where each node corresponds to a logic function with *one output* and an arc is provided between two nodes if the output of one function is an input of the other. Because the output of each node is unique, a node and an output are in one-to-one correspondence.

In principle, in these algorithms, a cost function that is different according to the final implementation should be used. However, due to the many different technologies used, it is very difficult to identify a meaningful cost function that could be optimized effectively. Thus an estimate for the final area is used. An estimate that has been used successfully is the number of literals in a factored form of the logic function. Then, *the optimal state-assignment problem can be formulated as the problem of assigning codes to the states so that the total number of literals in the factored form of the logic function is minimized.*

It is certainly difficult to devise a measure of how many literals a particular state assignment will yield after multilevel logic optimization has been carried out, because of the great complexity of the algorithms used for this purpose [11].

III. The Basic Idea

3.1. Operations in Multilevel Logic Optimization

The key point in the proposed algorithms for the state assignment problem is the model used to predict the results obtained by the multilevel logic optimizer after the encoding has been performed. We focused on the operations of MIS [11], the Berkeley logic optimizer.

The algorithms in MIS [11] can be classified in two categories: algebraic and Boolean methods. It is very difficult to model the optimization achieved by MIS with the use of Boolean methods, while it is feasible to predict at least some of the operations that the algebraic division algorithms use to minimize the logic.

Among the several algebraic optimization algorithms used by MIS are factoring of logic equations, common sub-expression identification and common cube extraction. These three techniques are illustrated in Fig. 4. The latter two techniques are *algebraic division* techniques, expressions are divided by common cubes or sub-expressions in order to produce smaller expressions with new intermediate variables. Common cube extraction is actually a subset of common sub-expression identification—a sub-expression may be a single cube.

As illustrated in Fig. 4, extracting common cubes results in a network with fewer literals than the original network. The algorithm presented in this paper tries to *maximize the number of common cubes* that can be found by the logic optimization algorithms in the encoded two-level network. Maximizing the number of common cubes results in a large number of good factors that can be extracted during optimization to produce a reduced literal multi-level representation.

3.2. Influence of Encoding on the Number of Common Cubes

There are two basic processes behind the influence of state assignment on the number of common cubes in the encoded state transition table (a two-level representation) which is the starting point for multilevel logic optimization. We now analyze these two processes.

We focus on the second field (the present state field) in the STT of the machine shown in Fig. 1. If we assigned the states $st0$ and $st2$ with codes of distance N_d, then the lines of the next state $st1$ will have a common cube with $N_b - N_d$ literals (due to edges 3 and 8 in the STT). Similar relationships exist between other sets of states.

We shift focus to the third field (the next state field) of the STT. If we assign the states $st0$ and $st2$ with codes of distance N_d, then the present state $st1$ becomes a common cube for $N_b - N_d$ next state lines *whatever its code is* (due to edges 5 and 6 in the STT). The number of literals in the common cube is, of course, N_b. Again, similar relationships exist between other sets of states in the machine.

Factoring:
ace + bce + de -> ((a + b) c + d) e

Common sub-expression identification:
ace + bce + de -> *s*ce + de
ade + bde + af -> *s*de + af
s = a + b

Common cube identification:
ace + bcef + de -> *t*c + *u*c + de
ade + bdef + af -> *t*d + *u*d + af
t = bef *u* = ae

Fig. 4. Factoring, common sub-expression and common cube identification.

The input and output spaces (the first and fourth fields) also have an influence on the number of common curves after encoding. If two different input combinations, i_1 and i_2, produce the same next state from different or same present states, then we have a common cube corresponding to $i_1 \cap i_2$ in the input space. Similarly, outputs asserted by different present states have common cubes corresponding to their intersections.

Given a machine, we have thus a large set of relationships between state encoding and the number/size of common cubes in the network prior to logic optimization. We can estimate the reduction in literal count or the "gains" that can be obtained by coding a given pair of states with close codes so single/multiple occurrences of common cubes can be extracted. Given these gains for each pair of states, we can attempt to find an encoding which maximizes the overall gain.

There arises a complication in gain estimation. Firstly, while the number of literals in the common cubes can be found exactly, the number of *occurrences* of these cubes depends on the encoding of the next states. In our example, assume that *st*1 was assigned *111* and *st*3 was assigned *110*. We have a common cube *11* (with 2 literals) for the next state lines but the number of occurrences of the this common cube depends on the number of 1's in the code of *st*2 (which we do not know). This problem is alleviated by treating the gains as *relative* merits rather than absolute and using an average-case analysis (Section 4.1.2).

It should be noted that these statically computed gains interact. Extracting some common cubes can *increase the level* (to the outputs) of other common cubes and can also decrease the gain in extracting them. For instance, a sequence of two cube extractions on a two-level network can produce a three- or a four-level network. Statically computing gains and maximizing the number of common cubes works because, given a particular encoding, the optimal *sequence* of cube extractions to produce a minimal literal multi-level network can be found by the logic optimizer. Our goal then is to find an encoding that maximizes the number of common cubes in the initial two-level network.

3.3 The Global Strategy

Our approach is to build a graph $G_M(V, E_M, W(E_M))$ where V is in one-to-one correspondence with the states of the finite state machine, E_M is a complete set of edges, i.e., every node is connected to every other node, and $W(E_M)$ represents the gains that can be achieved by coding the states joined by the corresponding arc as close as possible. These gains are statically and independently computed by enumerating the direct relationships between the input, state, and output spaces.

Then, the states are encoded, using this graph to provide the cost of an assignment of a state to a vertex of the Boolean hypercube.

A critical part of our approach is the generation of $W(E_M)$. We have experimented with two algorithms: one assigns the weights to the edges by taking into consideration the second and fourth fields of the state transition table, and is henceforth called *fanout-oriented*. The second algorithm assigns weights to the edges by taking into consideration the first and third fields and is henceforth called *fanin-oriented*.

The fanout-oriented algorithm attempts to *maximize the size* of the most frequently occurring common cubes in the encoded machine prior to optimization. The fanin-oriented algorithm attempts to *maximize the number of occurrences* of the largest common cubes in the encoded machine prior to optimization. These two algorithms are based on the two different processes behind the influence of state assignment on the number of common cubes in the network described earlier.

IV. Algorithms for Graph Construction

In this section, we present a fanout-oriented and a fanin-oriented algorithm which define a set of weights for the undirected graph $G_M(V, E, W(E_M))$. The weights represent a set of closeness criteria for the states in the machine which reflect on the number of common cubes in the encoded machine prior to optimization. Both these algorithms have a time- and space-complexity polynomial in the number of inputs, outputs and states in the machine to be encoded. In the sequel, the two algorithms are described and analyzed.

4.1. A Fanout-Oriented Algorithm

This algorithm works on the output and the fanout of each state. Present states which assert similar outputs and produce similar sets of next states are given high edge weights (and eventually close codes) so as to *maximize the size of common cubes* in the output and next state lines.

4.1.1. Algorithm Description:

The algorithm proceeds as follows:

(1) Construct a complete graph $G_M(V, E_M, W(E_M))$, with the edge weight set empty. For each output, all the labels, $W(E_M)$, in the state-transition graph G, are scanned to identify the nodes which assert that output. N_o sets of weighted nodes which assert each output are constructed. If a node asserts the same output more than once it has a correspondingly larger weight in the set.

(2) For each next state, sets of present states producing that next state are found (N_s sets are constructed).

The pseudocode below illustrates these steps of the procedure. *nw* stores the weight of the nodes in each of the different sets.

```
for( i = 1 ; i ≤ N_o ; i = i + 1 ) {
  foreach( edges e(v_k, v_l) ∈ G ) {
    if ( W(e).output[i] is 1) {
      OUTPUT_SET_i = OUTPUT_SET_i ∪ v_k
      nw(OUTPUT_SET_i, v_k) = nw(OUTPUT_SET_i,
      v_k) + 1
    }
  }
}
foreach( edges e(v_k, v_l) ∈ G ) {
  N_STATE_SET_l = N_STATE_SET_i ∪ v_k
  nw(N_STATE_SET_l, v_k) = nw(N_STATE_SET_l, v_k)
  + 1
}
```

(3) Using these N_o *OUTPUT_SET* AND N_s *N_STATE_SET* sets of nodes, $W(E_M)$ is constructed. The edge weight, we, is equal to the multiplication of the weights of the two nodes corresponding to it across all the sets. The weights corresponding to the next state sets have a multiplicative factor equal to the half the number of encoding bits, $N_b/2$. The reasoning behind the use of a multiplicative factor is given at the end of the section. The pseudocode for the calculation of we is shown below.

```
foreach( (v_k, v_l) ∈ G_M ) {
  for( i = 1 ; i ≤ N_s ; i = i + 1 )
    we (e_M(v_k,v_l)) = we (e_M(v_k, v_l)) +
      nw(N_STATE_SET_i, v_k) * nw(N_STATE_SET_i,
      v_l)
  we (e_M(v_k,v_l )) = we (e_M(v_k, v_l) * N_b/2
  for( i = 1 ; i ≤ N_o ; i = i + 1 )
    we (e_M(v_k,v_l)) = we (e_M(v_k,v_l)) +
      nw(OUTPUT_SET_i, v_k) * nw(OUTPUT_SET_i,
      v_l)
}
```

4.1.2. Analysis:

We now analyze the fanout-oriented algorithm. The first step of the algorithm entails enumerating the relationships between the present states and the output space. If two different present states assert an output, it is possible to extract a common cube corresponding to the intersection of the two state codes. By constructing the N_o different output sets and counting the number of times a pair of states occurs together in each output set, the algorithm effectively computes the number of occurrences of the common cube $X \cap Y$, for all states X and Y. We have to take into account the fact that a state may assert the same output for many input combinations—this corresponds to the weight *nw*( ). For two states that assert the same output a multiple number of times, each pair of edges will have the common cube. Accordingly, the weights *nw*( ) are multiplied.

In the second step, the next states produced by each pair of present states are compared. A state pair which produces the same next state has an associated common cube corresponding to the pairwise intersection. The number of occurrences of this common cube is dependent on the number of 1's in the code of the next state and therefore cannot be estimated exactly (unlike in the first step). We assume that the average number of 1's in a state's code is $N_b/2$. Since we are concerned with relative rather than absolute merits, the approximation that each common cube occurs in $N_b/2$ next state lines is a good one. Thus we have a multiplying factor of $N_b/2$ in the second step. Ideally, this factor should be a function of the encoding and not a constant for all state pairs.

Given the number of occurrences of different common cubes in the machine, this algorithm assigns weights so as to maximize the size of the most frequently occurring cubes.

4.1.3. An Example:

The graph generated by the fanout-oriented algorithm for the example FSM of Fig. 1 is shown in Fig. 5. The output set corresponding to the single output is ($st0^2$, $st1^3$, $st3^2$). The next state sets are $st0 \rightarrow (st0^2, st1^1)$, $st1 \rightarrow (st0^1, st1^1, st2^1)$, $st2 \rightarrow (st1^1, st2^1, st3^1)$ and $st3 \rightarrow (st2^1, st3^1)$. The superscripts denote the weights *nw*( ) for each state in each set. The weight of the edge between the states $st2$ and $st3$ with $N_b = 2$ is $(1 \times 1 + 1 \times 1) \times N_b/2 + 3 \times 2 = 8$. Similarly, the other edge weights can be calculated.

4.2. A Fanin-Oriented Algorithm

The algorithm described above ignored the input space. The algorithm works well for FSM's with a large number of outputs and small number of inputs. However, the number of input and output variables could both be quite large. In this section, we describe a fanin-oriented algorithm which operates on the input and fanin for each state. Next states which are produced by similar inputs and similar sets of present states are given high edge weights (and eventually close codes) so as to *maximize the number of common cubes* in the next state lines.

4.2.1. Algorithm Description:

The algorithm proceeds as follows:

(1) The graph G_M is constructed. N_s sets of weighted next states which fan out from each present state in G are constructed as shown below. *nw* stores the weight of each node in all the sets.

```
foreach( edge e(v_k, v_l) ∈ G ) {
  P_STATE_SET_k = P_STATE_SET_k ∪ v_l
  nw(P_STATE_SET_k, v_l) = nw(P_STATE_SET_k, v_l)
  + 1
}
```

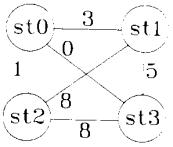


Fig. 5. Graph generated by fanout-oriented algorithm.

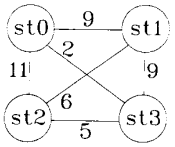


Fig. 6. Graph generated by fanin-oriented algorithm.

(2) For each input, sets of next states are identified which are produced when the input is 1 and when the input is 0. $2 \times N_i$ such sets are constructed as shown below.

```
for( i = 1 ; i ≤ N_i ; i = i + 1 ) {
  foreach( edge e(v_k, v_l) ∈ G ) {
    if( W(e)·input[i] is 1 ) {
      INPUT_SET^ON_i = INPUT_SET^ON_i ∪ v_l
      nw(INPUT_SET^ON_i, v_l) = nw(INPUT_SET^ON_i,
      v_l) + 1}
    if ( W(e)·input[i] is 0 ) {
      INPUT_SET^OFF_i = INPUT_SET^OFF_i ∪ v_l
      nw(INPUT_SET^OFF_i, v_l) = nw(INPUT_
      SET^OFF_i, v_l) + 1
    }
  }
}
```

(3) The weights on the edges in the graph, we, are found using the N_i *INPUT_SET*^{ON}, N_i *INPUT_SET*^{OFF} and N_s *P_STATE_SET* sets of nodes as illustrated in the pseudo-code below. Between each pair of nodes in G_M, an edge with weight equal to the multiplication of the weights of the two nodes across all the present state sets (scaled by N_b) and all the input sets is added

```
foreach( (v_k, v_l) ∈ G_M ) {
  for( i = 1 ; i ≤ N_s ; i = i + 1 )
    we (e_M(v_k,v_l)) = we (e_M(v_k,v_l)) + nw(P_
      STATE_SET_i, v_k) * nw(P_STATE_SET_i, v_l)
  we (e_M(v_k,v_l)) = we (e_M(v_k,v_l)) * N_b
  for( i = 1 ; i ≤ N_i ; i = i + 1 ) {
    we (e_M(v_k,v_l)) = we (e_M(v_k,v_l)) + nw
      (INPUT_SET^ON_i, v_k) * nw(INPUT_SET^ON_i, v_l)
    we (e_M(v_k,v_l)) = we (e_M(v_k, v_l)) + nw
      (INPUT_SET^OFF_i, v_k) * nw(INPUT_SET^OFF_i,
      v_l)
  }
}
```

4.2.2. Analysis:

We now analyze the fanout algorithm. The first step of the algorithm entails enumerating the relationships between the input and next state space. A next state produced by two different input combinations i_1 and i_2 has a common cube $i_1 \cap i_2$. The size of this cube can be found. By constructing the $2 \times N_i$ different input sets and counting the number of times a pair of states occurs together in each input set, the algorithm computes similarity relationships between all next state pairs in terms of the inputs. Giving next state pairs that are produced by similar inputs high edge weights will result in maximizing the number of occurrences of the largest common input cubes in the next state lines.

In the second step, the present states producing each pair of next states are compared. If two different next states are produced by the same present state, the state is common to some next state lines. The number of occurrences of this common cube is dependent on the intersection of the two next state codes. To maximize the number of occurrences of these cubes, next state pairs which have many common present states are given correspondingly high edge weights. Since each of these cubes have N_b literals (as opposed to a single literal for a single input), we have a multiplying factor of N_b while combining the weights computed in the two steps.

Given the sizes of the different common cubes in the machine, this algorithm assigns weights so as to maximize the number of occurrences of these cubes.

4.2.3. An Example

The graph generated by the fanin-oriented algorithm for the example FSM of Fig. 1 is shown in Fig. 6. As can be seen, the weights of the edges in the graph are different from those generated by the fanout-oriented algorithm (Fig. 5). Here we have the input sets $i_1(0) \rightarrow (st1^3, st3^2)$, $i_1(1) \rightarrow (st0^2, st2^3)$, $i_2(0) \rightarrow (st0^1, st1^1, st2^1)$ and $i_2(1) \rightarrow (st0^2, st1^1, st2^1, st3^1)$. The present state sets are $st0 \rightarrow (st0^2, st1^1)$, $st1 \rightarrow (st0^1, st1^1, st2^1)$, $st2 \rightarrow (st1^1, st2^1, st3^1)$ and $st3 \rightarrow (st2^1)$. The weight of the edge between $st0$ and $st1$ for $N_b = 2$ is $(1 \times 1 + 2 \times 1) + N_b \times (2 \times 1 + 1 \times 1) = 9$. The other edge weights are calculated in a singular fashion.

V. The Embedding Algorithm

The algorithms presented above generate a graph and a set of weights, like the graphs of Figs. 5 and 6, to guide the state encoding process. The problem now is to assign the actual codes to states according to the analysis performed by the fanin and the fanout-oriented algorithms. This problem is a classical combinatorial optimization problem called *graph embedding*. Here G_M has to be embedded in the Boolean hypercube so that the adjacency relations identified by G_M are satisfied in an optimal way. Unfortunately, this problem is *NP*-complete and there is little hope to solve it exactly in an efficient way.

Several heuristic approaches have been taken to solve variations of this problem (e.g., [14], [8], [15]). In [14]

and [15] distance relations are *required* to be satisfied during graph embedding. Of course, it may not be possible to satisfy all of them and some constraints (which are heuristically picked) may be relaxed. Similarly in [8], where clusters of states are recognized as in the fanin/fanout-oriented algorithms, join and fork rules are specified which if satisfied result in the merging of product terms. Our problem is different in sense that the goal is to minimize a cost function rather than attempting to satisfy distance relations. We use a heuristic approach to this embedding problem that has given satisfactory results.

The heuristic algorithm is called *wedge clustering*. This algorithm is used to assign codes to the nodes in G_M to minimize

$$\sum_{i=1}^{N_s} \sum_{j=i+1}^{N_s} \text{we}\left(e_M(v_i, v_j)\right) * \text{dist}\left(\text{enc}\left(v_i\right), \text{enc}\left(v_j\right)\right)$$

where the v_k are the vertices in G_M, we $(e_M(v_k, v_l))$ is the weight of the edge, e, between vertices v_k and v_l, enc (v_k) is the encoding of vertex v_k. The function dist ( ) returns the distance between two bindary codes.

The graphs generated by the fanout- and fanin-oriented algorithms have a certain structure associated with them, especially for large machines. In these graphs, typically small groups of states exist that are strongly connected internally (edges between states in the same group have high weights) but weakly connected externally (edges between states not in the same cluster have low weights). The embedding heuristic has been tailored to meet the requirements of our particular problem. The heuristic exploits the nature of the graph by attempting to identify strongly connected clusters and assigning states within each cluster with uni-distant codes.

The embedding algorithm proceeds as follows. Clusters of nodes with the cardinality of the cluster no greater than $N_b + 1$ and consisting of edges of maximum total weight are identified in G_M. Given G_M, the identification of these clusters is as follows—a node, $v_1 \in G_M$, with the maximum sum of weights of any N_b connected edges is identified. The N_b nodes, $y_1, y_2, \cdots, y_{N_b}$ which correspond to the N_b edges from v_1 and v_1 are assigned minimally distant codes from the unassigned codes (v_1 may have been assigned already, so may the other y_i). A maximum of N_b nodes are chosen so the y_i can be (possibly) assigned unidistant codes from v_1. After the assignment, v_1 and all the edges connected to v_1 are deleted from G_M and the node section/code assignment process is repeated till all the nodes are assigned codes. The pseudocode below illustrates the procedure.

$GG = G_M$
while (GG is not empty) {
  Select $v_1 \in GG$, $y_i \in GG$ so $\sum_{i=1}^{N_b}$ we $(e_M(v_1, y_i))$ is maximum
  assign the y_i and v_1 minimally distant codes from unassigned codes
  $GG = GG = v_1$
}

We can prove the following optimality result about the embedding heuristic. For convenience in notation, we assume that we $(y_i, y_i) = 0 \; \forall \; i \; \exists \; y_i \in G_M$.

Theorem 1: At a given iteration, if the N_b states $y_1, y_2, \cdots, y_{N_b}$ are given unidistant codes from the selected state v_1 and if

$$\text{we}\left(e_M(v_1, y_i)\right) \geq \text{we}\left(e_M(y_i, y_j)\right) + \text{we}\left(e_M(y_i, y_k)\right), \quad 1 \leq i, j; k \leq N_b; i \neq j \neq k \tag{1}$$

then *the assignment is optimum* for this cluster of $N_b + 1$ states, i.e.,

$$\sum_{i=1}^{N_b} \text{we}\left(e_M(v_1, y_i)\right) * \text{dist}\left(\text{enc}\left(v_1\right), \text{enc}\left(y_i\right)\right) + \sum_{i=1}^{N_b} \sum_{j=i+1}^{N_b} \text{we}\left(e_M(y_i, y_j)\right) * \text{dist}\left(\text{enc}\left(y_i\right), \text{enc}\left(y_j\right)\right)$$

is minimum.

Proof: We have to prove that no assignment of codes to states can produce a cost which is less than the cost, $C(v_1)$, produced by assigning the N_b states with uni-distant codes from v_1. We have

$$C(v_1) = \sum_{i=1}^{N_b} \text{we}\left(e_M(v_1, y_i)\right) + 2 * \sum_{i=1}^{N_b} \sum_{j=i+1}^{N_b} \text{we}\left(e_M(y_i, y_j)\right)$$

since the $y_1, \cdots, y_{N_b}$ have uni-distant codes from v_1 and, therefore, are distance-2 from each other. To decrease the cost, the distances between the codes of the y_i have to be reduced from 2 to 1. This can only be done at the expense of an increase in the distance between some of the y_i and v_1 from 1 to 2. There are three possible ways of doing so. First, we can select any state y_S from $y_1, \cdots, y_{N_b}$ and code the rest of the y_i and v_1 with unidistant codes from y_S. Without loss of generality, assume we select y_1. We know, since we selected v_1 initially, that

$$\sum_{i=1}^{N_b} \text{we}\left(e_M(v_1, y_i)\right) \geq \sum_{j=1}^{N_b} \text{we}\left(e_M(y_k, y_j)\right) + \text{we}\left(e_M(y_k, v_1)\right) \; \forall \; k \tag{2}$$

Using (2) above, it can easily be shown that

$$C(y_1) = \sum_{i=2}^{N_b} \text{we}\left(e_M(y_S, y_i)\right) + \text{we}\left(y_1, v_1\right)) + 2 * \sum_{i=2}^{N_b} \sum_{j=i+1}^{N_b} \text{we}\left(e_M(y_i, y_j)\right) + 2 * \sum_{i=2}^{N_b} \text{we}\left(e_M(v_1, y_i)\right) \geq C(v_1)$$

and similarly, for $C(y_2), \cdots, C(y_{N_b})$.

The second alternative in assigning codes is to select a y_S and assign it a code which is unidistant from two other y_i (Only two y_i can be chosen since the y_i are distance-2 from each other). This code will be distance-3 from the unselected y_i and will be distance-2 from v_1. Without loss of generality, assume that y_1 was selected and assigned a unidistant code from y_2 and y_3. We have

$$C(y_1) = \sum_{i=2}^{N_b} \text{we}\left(e_M(v_1, y_i)\right) + \sum_{i=2}^{3} \text{we}\left(e_M(y_1, y_i)\right) + 2 * \text{we}\left(e_M(v_1, y_1)\right) + 2 * \text{we}\left(e_M(y_2, y_3)\right) + 2 * \sum_{i=4}^{N_b} \sum_{j=i+1}^{N_b} \text{we}\left(e_M(y_i, y_j)\right) + 2 * \sum_{i=2}^{3} \sum_{j=i+1}^{N_b} \text{we}\left(e_M(y_i, y_j)\right) + 3 * \sum_{i=4}^{N_b} \text{we}\left(e_M(y_1, y_i)\right).$$

Expanding $C(v_1)$, we have

$$C(v_1) = \sum_{i=2}^{N_b} \text{we}\left(e_M(v_1, y_i)\right) + 2 * \sum_{i=2}^{3} \text{we}\left(e_M(y_1, y_i)\right) + \text{we}\left(e_M(v_1, y_1)\right) + 2 * \text{we}\left(e_M(y_2, y_3)\right) + 2 * \sum_{i=4}^{N_b} \sum_{j=i+1}^{N_b} \text{we}\left(e_M(y_i, y_j)\right) + 2 * \sum_{i=2}^{3} \sum_{j=i+1}^{N_b} \text{we}\left(e_M(y_i, y_j)\right) + 2 * \sum_{i=4}^{N_b} \text{we}\left(e_M(y_1, y_i)\right).$$

Canceling terms from $C(y_1)$ and using relation (1), shows that $C(y_1) \geq C(v_1)$.

The third alternative in assigning codes is to select a state y_S and $2 < p < N_b - 1$ states from the remaining y_i and make these p states uni-distant from y_S. y_S will be uni-distant from v_1, and p states will be distance-2 from v_1 and will be distance-2 from each other. If $p \leq 2$ then we are back to the second alternative (or worse) which is nonoptimal. Similarly, $p = N_b - 1$ brings us back to the first alternative which is nonoptimal. Assuming y_1 and y_2, $\cdots$, y_{p+1} are selected, we have

$$C(y_1) = \text{we}\left(e_M(v_1, y_1)\right) + 2 * \sum_{i=2}^{p+1} \text{we}\left(e_M(v_1, y_i)\right) + \sum_{i=p+2}^{N_b} \text{we}\left(e_M(v_1, y_i)\right) + \sum_{i=2}^{p+1} \text{we}\left(e_M(y_1, y_i)\right) + 2 * \sum_{i=2}^{N_b} \sum_{j=i+1}^{N_b} \text{we}\left(e_M(y_i, y_j)\right).$$

Expanding $C(v_1)$, we have

$$C(v_1) = \text{we}\left(e_M(v_1, y_1)\right) + \sum_{i=2}^{p+1} \text{we}\left(e_M(v_1, y_i)\right) + \sum_{i=p+2}^{N_b} \text{we}\left(e_M(v_1, y_i)\right) + 2 * \sum_{i=2}^{p+1} \text{we}\left(e_M(y_1, y_i)\right) + 2 * \sum_{i=2}^{N_b} \sum_{j=i+1}^{N_b} \text{we}\left(e_M(y_i, y_j)\right).$$

Canceling terms from $C(y_1)$ and using relation (1), shows that $C(y_1) \geq C(v_1)$. In the general case, more than one y_S, each with an associated set of states from the remaining y_i, may be selected and each set made unidistant from y_S. The proof for this case is more involved but follows in a similar way as for the previous case, expanding $C(v_1)$ and using relation (1). Q.E.D.

Thus we have a heuristic which is optimal for a graph satisfying relation (1) at each iteration of the embedding if sets of minimally distant codes can be found. It produces good (though perhaps sub-optimal) solutions for graphs satisfying

$$\text{we}\left(e_M(v_1, y_i)\right) \geq \text{RAT} * \text{we}\left(e_M(y_i, y_j)\right) + \text{we}\left(e_M(y_i, y_k)\right), \quad 1 \leq i, j; k \leq N_b; i \neq j \neq k$$

where RAT is close to 1. This, coupled with the fact that typical graphs produced by the fanout- and fanin-oriented algorithms have strongly connected clusters of states, makes the embedding algorithm eminently suitable for our purpose.

The algorithm is quite fast and has a worst case time complexity of $O(N_S^2(\log(N_S) + N_b))$. Initially, the $N_S - 1$ fanout edges from *each* of the N_S states are sorted in decreasing order of weights which takes $O(N_S^2 \log(N_S))$ time. The embedding itself may require a maximum of $N_s - N_b$ iterations. This is because in the first iteration, $N_b + 1$ states are encoded and in the worst case only one state is encoded in following iterations. To select a state with maximum weight of any N_b connected edges can be accomplished in $O(N_S N_b)$ time, giving an overall time complexity of $O(N_S^2(\log(N_S) + N_b))$.

The embedding algorithm is illustrated in Fig. 7 using a small example with 5 states, to be encoded using 3 bits. Initially, the node corresponding to state $st3$ is selected—it is the node with the maximum set of any 3 edge weights. The states corresponding to these three edges are $st0$, $st1$ and $st2$. The three states are given codes uni-distant from $st3$. $st3$ and its edges are deleted from the graph. The selection process continues, picking $st1$ from the modified graph and encoding $st4$. This completes the encoding.

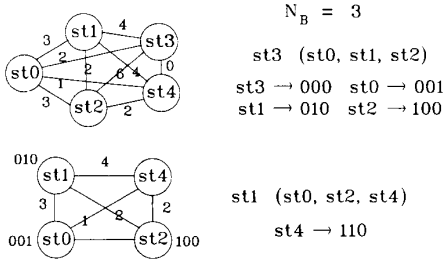


Fig. 7. An embedding example.

VI. Results

We have run 20 benchmark examples (which have been obtained from various university and industrial sources) representing a wide range of finite automata on different state assignment programs as well as on our two algorithms. The size statistics of the examples are given in Table I, with the minimum possible encoding for each FSM indicated under the column #enc.

The results obtained via random state assignment (RANDOM-A and RANDOM-B), using the state assignment program KISS (KISS), the fanout-oriented algorithm (MUST-P) and the fanin-oriented algorithm (MUST-N) for multi-level implementations are summarized in Tables II and III. The number of literals after running through two optimization scripts in the multi-level logic synthesis tool MIS [11] are given for each of the state assignment techniques. The literal counts of Table II were obtained using a short optimization script and those of Table III using a much longer optimization script (which produces better results).

The literal counts under RANDOM-A were obtained using a statistical average of 5 different random state assignments (using different starting seeds) on each example. RANDOM-B was the best result obtained in the different runs. RANDOM-B is significantly better than RANDOM-A especially for the smaller examples. MUSTANG is the best result produced by either the fanout or the fanin-oriented algorithm for each given example.

MUSTANG can be constrained to use any number of encoding bits greater than or equal to the minimum. For all examples MUSTANG was run using the minimum possible bit encoding. Minimum bit encoding has been found to be uniformly good for multi-level logic implementations. KISS typically uses a 1–3 bits more than the minimum encoding length. The time required by MUSTANG for encoding these benchmarks varied between 0.1 CPU seconds for the small examples to 100 CPU seconds for the largest example, **scf**, on a VAX 11/8650.

The algorithms developed have achieved the goal of producing encodings which produce minimal area implementations after multilevel logic optimization as illustrated in Tables II and III. The literal counts obtained by MUSTANG are on the average 30 percent better than random state assignment and 20 percent better than KISS. In some cases, the fanout-oriented algorithm does better than the fanin-oriented algorithm, when ignoring the common

TABLE I
Statistics of Benchmark Examples

EXAMPLE	#inp	#out	#states	#enc
bbara	4	2	10	4
bbsse	7	7	16	4
bbtas	2	2	6	3
cse	7	7	16	4
dk15x	3	5	4	2
dk16x	2	3	27	5
keyb	7	2	19	5
lion	2	1	4	2
lion9	2	1	9	4
mark1	5	16	14	4
mc	3	5	4	2
modulo12	1	1	11	4
planet	7	19	48	6
s1	8	6	20	5
s1a	8	6	20	5
scf	27	56	128	7
shiftreg	1	1	8	3
tav	4	4	4	2
tbk.min	6	3	16	4
train11	2	1	10	4

TABLE II
Results Obtained Using Simple Chart

EXAMPLE	RANDOM-A		RANDOM-B		KISS		MUST-P	MUST-N	MUSTANG
	#lit	RAT	#lit	RAT	#lit	RAT	#lit	#lit	#lit
bbara	120	1.48	91	1.12	103	1.27	81	108	81
bbsse	214	1.48	190	1.31	145	1.01	144	177	144
bbtas	37	1.15	26	0.81	34	1.06	51	32	32
cse	405	1.33	339	1.11	264	0.87	304	319	304
dk15x	122	1.17	109	1.04	91	0.87	104	128	104
dk16x	553	1.59	516	1.49	411	1.18	383	346	346
keyb	810	2.45	663	2.00	474	1.43	495	330	330
lion	20	1.11	18	1.00	21	1.16	18	22	18
lion9	61	3.05	52	2.60	37	1.85	25	20	20
mark1	112	1.28	89	1.02	114	1.31	130	87	87
mc	40	1.11	37	1.02	43	1.19	37	36	36
modulo12	43	1.19	40	1.11	49	1.36	40	36	36
planet	1063	1.24	1012	1.18	869	1.01	1033	854	854
s1	852	4.26	805	4.02	690	3.45	639	200	200
s1a	649	4.0	583	3.59	382	2.35	514	162	162
scf	1674	1.31	1596	1.25	1441	1.13	1390	1274	1274
shiftreg	37	18.5	32	16.0	8	4.00	2	8	2
tav	25	1.04	24	1.00	24	1.00	25	24	24
tbk.min	540	1.12	532	1.10	563	1.16	515	482	482
train11	67	1.34	53	1.06	46	0.92	50	55	50
TOTAL	**7444**	**1.62**	**6807**	**1.48**	**5809**	**1.26**			**4586**

#lit : Number of literals after multi-level logic optimization using simple script
RAT: Ratio to number of literals produced by MUSTANG using simple script

TABLE III
Results Obtained Using Intensive Script

EXAMPLE	RANDOM-A		RANDOM-B		KISS		MUST-P	MUST-N	MUSTANG
	#lit	RAT	#lit	RAT	#lit	RAT	#lit	#lit	#lit
bbara	95	1.39	76	1.11	79	1.16	68	75	68
bbsse	153	1.29	131	1.11	118	1.00	118	144	118
bbtas	32	1.45	24	1.09	28	1.27	41	22	22
cse	273	1.24	240	1.09	203	0.92	224	220	220
dk15x	109	1.18	94	1.02	85	0.92	92	108	92
dk16x	406	1.40	394	1.35	315	1.08	326	290	290
keyb	369	1.75	311	1.48	213	1.01	320	210	210
lion	19	1.18	16	1.00	16	1.00	16	16	16
lion9	55	2.75	43	2.15	36	1.80	22	20	20
mark1	102	1.22	99	1.19	99	1.19	115	83	83
mc	39	1.08	37	1.02	42	1.16	37	36	36
modulo12	41	1.24	36	1.09	47	1.42	38	33	33
planet	697	1.23	654	1.16	547	0.97	597	563	563
s1	424	2.65	354	2.21	352	2.20	376	160	160
s1a	363	2.57	337	2.39	258	1.82	307	141	141
scf	939	1.10	922	1.08	861	1.01	881	852	852
shiftreg	36	18.0	24	12.0	8	4.0	2	8	2
tav	23	0.95	22	0.91	22	0.91	24	24	24
tbk.min	355	1.19	342	1.15	381	1.28	348	297	297
train11	64	1.30	54	1.10	44	0.90	49	55	49
TOTAL	**4594**	**1.39**	**4210**	**1.27**	**3754**	**1.14**			**3296**

#lit : Number of literals after multi-level logic optimization using intensive script
RAT: Ratio to number of literals produced by MUSTANG using intensive script

TABLE IV
RESULTS AFTER INTENSIVE SCRIPT AND TECHNOLOGY MAPPING

EXAMPLE	RANDOM-B		KISS		MUSTANG-N	
	#lit	#gate	#lit	#gate	#lit	#gate
cse	240	115	203	95	220	105
dk16x	394	175	315	143	290	124
keyb	311	158	213	112	210	112
planet	654	290	547	249	563	267
s1	354	174	352	173	160	93
s1a	337	169	258	131	141	83
scf	922	445	861	401	852	393
tbk.min	342	170	381	169	297	130

sub-expressions in the input space is a good approximation.

MUSTANG does comparatively better than random assignment or KISS in the shorter optimization script case than in the more complex optimization script. This is to be expected since MUSTANG models only the common cube extraction process in multilevel logic optimization. In the short optimization script, cube factors dominate in the reduction of the size of the network. More complicated factors, not modeled by MUSTANG, come into play in the complex optimization script.

For any given example, the literal counts obtained using MUSTANG and short or long optimization scripts are comparatively *closer* than using KISS or random assignment. For example, in **bbara**, using random assignment produces 120 literals after quick optimization versus 76 after a long optimization (on an average). The corresponding number for MUST-P are 81 and 68, respectively. MUSTANG eases the job of the multi-level logic optimizer, by providing a larger number of easily detectable factors in the network before optimization—a short script can produce good results. Also, the time taken by MIS to optimize MUSTANG encoded examples is significantly shorter (20–40 percent) than to optimize examples encoded using different techniques. Again, this is because a large number of easily detectable factors exist in the pre-optimized network.

Both MUSTANG and MIS optimize for literal counts rather than the number of gates in the network. MIS may produce arbitrarily complex gates in an optimized network. In many cases, these gates have to be *mapped* to a specific technology library. It is worthwhile to see if the gains in literal counts do produce networks with fewer gates. The number of gates in the benchmark examples after intensive logic optimization and technology mapping are given in Table IV for the different state assignment techniques. Only the largest examples are shown, the small examples had insignificant numbers of complex gates.

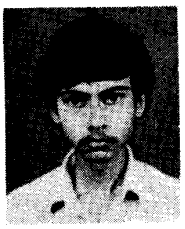
VII. CONCLUSIONS

All previous work in automatic state assignment has been targeted toward two-level logic implementations of finite state machines. Multilevel logic implementations can be substantially faster and/or smaller than corresponding two-level implementations. We have shown the need for new techniques for state assignment directly targeting multi-level logic implementations and developed algorithms for this purpose. As compared to existing techniques, significant reductions in literal counts, averaging 20–40 percent have been obtained on benchmark examples. Further work includes attempting to merge the fanin and fanout-oriented approaches and predicting more complicated multilevel logic optimizations like common kernel extraction and Boolean factoring.

REFERENCES

[1] D. B. Armstrong, "A programmed algorithm for assigning internal codes to sequential machines," *IRE Trans. Electron. Comput.*, vol. EC-11, pp. 466–472, Aug. 1962.
[2] T. A. Dolotta and E. J. McCluskey, "The coding of internal states of Sequential machines," *IEEE Trans. Electron. Comput.*, vol. EC-13, pp. 549–562, Oct. 1964.
[3] H. C. Torng, "An Algorithm for finding secondary assignments of synchronous sequential circuits," *IEEE Trans. Computers*, vol. C-17, pp. 416–469, May 1968.
[4] G. D. Micheli, A. Sangiovanni-Vincentelli, and T. Villa, "Computer-aided synthesis of PLA-based finite state machines," in *Proc. Int. Conf. on Computer-Aided Design*, Santa Clara, CA, November 1983, 154–156.
[5] G. D. Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Optimal state assignment of finite state machines," *IEEE Trans. Computer-Aided Design*, vol. CAD-4, pp. 269–285, July 1985.
[6] A. J. Coppola, "An implementation of a state assignment heuristic," in *Proc. 23rd Design Automation Conf.*, Las Vegas, NV, July 1986.
[7] G. D. Micheli, "Symbolic design of combinational and sequential logic circuits implemented by two-level macros," *IEEE Trans. Computer-Aided Design*, vol. CAD-5, pp. 597–616, Oct. 1986.
[8] G. Saucier, M. C. Depaulet, and P. Sicard, "ASYL: A rule-based system for controller synthesis," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 1088–1098, Nov. 1987.
[9] C. Tseng *et al.*, "A versatile finite state machine synthesizer," in *Proc. Int. Conf. on Computer-Aided Design*, Santa Clara, CA, pp. 206–209, Nov. 1986.
[10] R. K. Brayton and C. T. McMullen, "Synthesis and optimization of multistage logic," *Proc. ICCD*, Oct. 1984.
[11] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "MIS: A multiple level logic optimization system," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 000–0000, Nov. 1987.
[12] K. Bartlett, G. D. Hachtel, A. J. D. Geus, and W. Cohen, "Synthesis and optimization of multi-level logic under timing constraints," in *Proc. Int. Conf. on Computer-Aided Design*, Santa Clara, CA, pp. 290–292, Nov. 1985.
[13] R. K. Brayton, C. T. McMullen, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. New York: Kluwer Academic, 1984.
[14] G. D. Micheli and A. Sangiovanni-Vincentelli, "Multiple constrained folding of programmable logic arrays: Theory and applications," *IEEE Trans. Computer-Aided Design*, vol. CAD-2, July 1983.
[15] R. L. Graham and H. O. Pollak, "On embedding graphs in squashed cubes," in *Graph Theory and Applications 303*. New York: Springer Verlag, 1972.

*



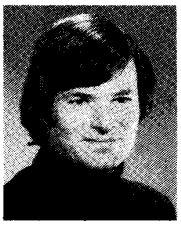
Srinivas Devadas received the B.S. in electrical engineering from the Indian Institute of Technology, Madras, India, in 1985, the M.S.E.E. and Ph.D. degree from the University of California, Berkeley in 1986 and 1988, respectively.

He is currently an Assistant Professor of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology, Cambridge. His research interests include the automatic layout of logic circuits, behavioral synthesis and verification, parallel processing, sequential testing and neural networks.

Hi-Keung Ma received the B.S. degree in electrical engineering from Queen Mary College, University of London, England, in July 1982. He is currently working towards the Ph.D. degree at the University of California, Berkeley, in the computer-aided design of integrated circuits.

His research interests include automatic layout, test generation and verification of sequential and combinational logic circuits, design for testability, and parallel processing.

*

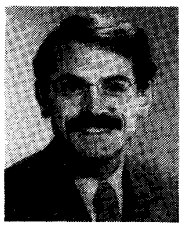


A. Richard Newton (S'73–M'78–SM'86–F'88) was born in Melbourne, Australia, on July 1, 1951. He received the B.Eng. (elect.) and M.Eng.Sci. degree from the University of Melbourne, Melbourne, Australia, in 1973 and 1975, respectively, and the Ph.D. degree from the University of California, Berkeley, in 1978.

He is currently a Professor and Vice Chairman of the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley. He is the Technical Program Chairman of the 1988 ACM/IEEE Design Automation Conference, and a consultant to a number of companies for computer-aided design of integrated circuits. His research interests include all aspects of the computer-aided design of integrated circuits, with emphasis on simulation, automated layout techniques, and design methods for VLSI integrated circuits.

Dr. Newton was selected in 1987 as the national recipient of the C. Holmes McDonald Outstanding Young Professor Award of Eta Kappa Nu. He is a member of Sigma Xi.

*



Alberto Sangiovanni-Vincentelli (M'74–SM'81–F'83) received the Dr.Eng. degree from the Politecnico di Milano, Italy, in 1971.

From 1971 to 1977, he was with the Politecnico di Milano, Italy. In 1976, he joined the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, where he is presently Professor. He is a consultant in the area of computer-adided design to several industries. His research interests are in various aspects of computer-aided design of integrated circuits, with particular emphasis on VLSI, simulation and optimization. He was Associate Editor of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS, and is Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN and a member of the Large-Scale Systems Committee of the IEEE Circuits and Systems Society and the Computer-Aided Network Design (CANDE) Committee. He was the Guest Editor of two special issues of the IEEE TRANSACTIONS ON CAD (*CAD for VLSI*, Sept./Nov. 1987). He was Executive Vice-President of the IEEE Circuits and Systems Society in 1983.

In 1981, Dr. Sangiovanni-Vincentelli received the Distinguished Teaching Award of the University of California. At both the 1982 and 1983 IEEE-ACM Design Automa. Conferences, he was given a Best Paper and a Best Presentation Award. In 1983, he received the Guillemin-Cauer Award for the best paper published in the IEEE TRANSACTIONS on CAS and CAD in 1981–1982. He is a member of ACM and Eta Kappa Nu.