Decomposition of Finite State Machines for Area, Delay Minimization

Rupesh S. Shelar †, Madhav P. Desai, H. Narayanan
Department of Electrical Engineering, Indian Institute of Technology, Bombay
Mumbai 400 076, India
Email: rupesh@sasi.com, {madhav, hn}@ee.iitb.ernet.in

Abstract

In this paper, we consider the state assignment problem as that of the decomposition of finite state machines and transform this decomposition problem into an orthogonal partitioning problem with a certain cost function. We attempt to justify this cost function in two ways, first by using an idealized model of multi-level logic implementation, and second by empirical studies of a particular benchmark circuit. We describe a greedy algorithm to minimize this cost function. We present results obtained by running the algorithm on a set of 16 MCNC benchmarks. We compare these results with other state assignment techniques such as JEDI [6] and NOVA [5]. For multilevel implementations of the benchmark state machines, we find that the implementations obtained after using JEDI [6] were, on average, 8.52% larger in area and 81.87% slower in delay than the implementations obtained using our approach. The implementations obtained after using NOVA were, on average, 4.40% larger in area and 104.96% slower in delay when compared with implementations obtained using our approach. Our scheme has the potential to serve as an alternative to conventional state assignment tools since we observe that it produces good results for larger finite state machines.

1 Introduction

Finite State Machines (FSMs) are important components of digital systems. Therefore, techniques for area efficient and fast implementation of FSMs are of great interest. The implementation of an FSM is strongly determined by the way codes are assigned to the states of an FSM. The state assignment problem can be stated as that of assigning codes to the states of a finite state machine while optimizing a given criterion. The state assignment problem has received considerable attention from researchers, because it is an important step in the process of sequential circuit synthesis. Some of the reported state assignment tools are NOVA [5] for area minimization of PLA implementations; JEDI [6] for multilevel implementations. Classical approaches for reduction in the complexity of the next state and output functions involve reducing input or output dependencies, state splitting etc. [7, 8].

The state assignment problem can also be viewed as that of the decomposition of a finite state machine. Recently, decomposition of finite state machine has attracted attention of researchers for area minimization and power reduction. Some of these approaches towards area minimization are Decomposition and Factorization [2], Modify and Restore [4] and Decomposition as Constrained Covering [3]. The decomposition and factorization approach tries to extract out repetitive parts of finite state machine, implement them only once and pass the control to them whenever a particular transition occurs. Since the repetitive part of finite state machine is implemented only once, it can achieve some area reduction. However, there may not be exact repetitive parts in an FSM. This difficulty is resolved by the Modify and Restore approach which involves modifying the next state and output functions to extract repetitive parts and restoring the original functions using restoring PLA and Ex-OR gates. In this approach, the restoring PLA might itself require a large area. The constrained covering approach has put forward strategies for handling various kinds of topologies such as cascade-cascade, cascade-parallel, parallel-parallel. These approaches are highly dependent on the structure of the machine.

There are essentially two approaches to machine decomposition. If the original state graph is partitioned into several pieces, with each piece being implemented by a separate machine with a wait state, then exactly one machine is active at any instant while the others are in the reset state [1]. In this case, decomposition is viewed as the join of two disjoint partitions on the set of states.

An alternative approach to decompose an FSM is based on factoring the original state graph, and is the focus of this paper. In this paper, we decompose a finite state machine into two interacting machines, for area effective as well as high performance implementation. Contrary to [1], our decomposition is a factoring of the original machine (with \(N\) states) into two much smaller machines (with \(\sqrt{N}\) states each) and this factoring can be viewed as the meet of the two orthogonal partitions of the set of states of the original machines. Note that the two smaller machines are both active simultaneously.

* Silicon Automation Systems Limited, Bangalore 560 008
We solve the decomposition of an FSM as an orthogonal partitioning problem. We demonstrate the effectiveness of our scheme from a comparison of the area and delay of the implementations obtained by our algorithm, with the area and delay of the implementations obtained using conventional state assignment tools. The areas of the implementations we obtain are competitive, while the delays are, on average, considerably smaller.

The organization of the rest of the paper is as follows. In Section 2, we present the decomposition model and give formal definitions in Section 3. In Section 4, the decomposition model is explained with an example. In Section 5, we propose a cost function which is a measure of the area of final implementation, and give theoretical as well as empirical justifications for this cost function. In Section 6, we present a greedy algorithm for the orthogonal partitioning problem (with the proposed cost function). In Section 7, we present the results obtained by running the algorithm on a set of 16 MCNC benchmarks. In Section 8, we conclude the paper by discussing future directions.

2 Decomposition Model

Any finite state machine $M$ can be described by a 6-tuple

$$M = (\Sigma, \Delta, Q, \delta, \lambda, q_{reset})$$

where, $\Sigma$ is the input alphabet, $\Delta$ is the output alphabet, $Q$ is the set of states, $\delta: \Sigma \times Q \rightarrow Q$ is the next state function and $\lambda: \Sigma \times Q \rightarrow \Delta$ is the next state function. The state $q_{reset}$ is the reset state.

In the classical literature [7, 8], various models for the decomposition of finite state machines have been proposed, such as parallel, serial, general. We will be considering the decomposition of an FSM into two interacting machines (the general model in [7, 8]). In Figure 1(a), we show the Huffman model of a finite state machine, and in Figure 1(b), we show the decomposed machine.

As shown in Figure 1(b), the decomposition model consists of three combinational logic blocks, two of which implement next state functions, and the remaining one implements an output function. In Figure 1(b), we show two interacting machines whose next state lines are their outputs, and a combinational network which generates the primary outputs.

3 Formal Definitions

We describe one way of decomposing an FSM $M(\Sigma, \Delta, Q, \delta, \lambda, q_{reset})$ into two interacting machines as shown in Figure 1(b). Suppose $Q = \{q_1, q_2, \ldots, q_n\}$. A partition $\Pi(Q)$ of a set $Q$ is a set of disjoint and non-empty subsets of $Q$ whose union is $Q$. The zero partition of $Q$ is denoted by $\Pi_0(Q)$, and is the partition whose elements are the singleton subsets of $Q$.

Let $\Pi_A(Q) = \{A_1, A_2, \ldots, A_m\}$ and $\Pi_B(Q) = \{B_1, B_2, \ldots, B_k\}$ be partitions of the set of states $Q$ (henceforth, when we write $\Pi$, we mean $\Pi(Q)$ because all partitions considered will be of $Q$). The partitions $\Pi_A$ and $\Pi_B$ are said to be orthogonal if the following conditions are met:

1. $m \times k \geq n$.
2. For $i \leq m$, $j \leq k$, either $A_i \cap B_j = \phi$ or $A_i \cap B_j = \{q_r\}$ for some $r \leq N$.

Conditions 1 and 2 are equivalent to saying that $\Pi_A \cdot \Pi_B = \Pi_0$ where “$\cdot$” represents the meet of the two partitions.

For partitions $\Pi_A$ and $\Pi_B$ respectively, two interacting machines $M_A$ and $M_B$ can be defined as follows: The states in $M_A$ (resp. $M_B$) are the elements of $\Pi_A$ (resp. $\Pi_B$). The inputs to $M_A$ (resp. $M_B$) are drawn from $\Sigma \times \Pi_B$ (resp. $\Sigma \times \Pi_A$). The outputs from $M_A$ and $M_B$ are the states themselves. A combinational circuit is used to generate the outputs of the original machine.

More precisely, we have

$$M_A = (\Sigma \times \Pi_B, \Pi_A, A_\delta, \lambda_A, A_{reset})$$
$$M_B = (\Sigma \times \Pi_A, \Pi_B, \delta_B, \lambda_B, B_{reset})$$

where $A_{reset}$ and $B_{reset}$ are the elements of $\Pi_A$ and $\Pi_B$ which contain $q_{reset}$.

The next state function

$$\delta_A : \Sigma \times \Pi_B \times \Pi_A \rightarrow \Pi_A$$

is defined such that $\delta_A((\sigma, B_m), A_i) = A_j$ if and only if $\delta(\sigma, q_m) = q_r$ where $q_r \in A_i \cap B_m$ and $q_r \in A_j$ ($\delta_B$ is defined analogously). The functions $\lambda_A$ and $\lambda_B$ simply output the current state of $M_A$ and $M_B$ respectively.

The combinational network implements a function:

$$f : \Sigma \times \Pi_A \times \Pi_B \rightarrow \Delta$$

with $f(\sigma, A_i, B_j) = \lambda(\sigma, q_r)$, where $q_r$ is the unique state in $A_i \cap B_j$.

The following proposition states that this composite machine has the same behavior as the original FSM.

Proposition 3.1 The terminal behavior of the original FSM (started in the reset state) $M(\Sigma, \Delta, Q, \delta, \lambda, q_{reset})$ is identical to the terminal behavior of the network consisting of the two interacting machines $M_A$ $M_B$ (both started in their reset states) together with the combinational network described above.
Proof: The proposition can be proved by induction on length of the input sequence in a routine manner.

We demonstrate this decomposition of an FSM with a simple example in the following section.

4 Example

Consider an FSM whose state transition graph is shown in Figure 2. In this example, \( Q = \{1,2,3,4\} \) is a set of states. Two partitions satisfying the conditions in Section 3 are \( \Pi_A = \{\{1,2\},\{3,4\}\} \) and \( \Pi_B = \{\{1,3\},\{2,4\}\} \) because their meet is \( \Pi_0 \). Here, \( A_1 = \{1,2\}, A_2 = \{3,4\} \); and \( B_1 = \{1,3\}, B_2 = \{2,4\} \). The corresponding machine \( M_A \) has input alphabet

\[ \Sigma \times \Pi_B = \{(0,B_1),(0,B_2),(1,B_1),(1,B_2)\}. \]

Following the conditions in Section 3, \( \delta_A \) can be written as follows:

\[
\begin{align*}
\delta_A((0,B_1),A_1) &= A_1 & \delta_A((0,B_2),A_1) &= A_2 \\
\delta_A((1,B_1),A_1) &= A_2 & \delta_A((1,B_2),A_1) &= A_2 \\
\delta_A((1,B_1),A_2) &= A_2 & \delta_A((1,B_2),A_2) &= A_1
\end{align*}
\]

The first transition listed above is obtained by noting that, the state \( 1eA_1 \cap B_1 \) and \( \delta(0,1) = 2 \) the original FSM. Therefore, \( \delta_A((0,B_1),A_1) \) is the unique element of \( \pi_A \) which contains \( \delta(0,1) \), which is \( A_1 \). The other transitions can be obtained in a similar manner and the state transition graph for machines \( M_A \) as well as \( M_B \) can be drawn as shown in Figures 2(b) and 2(c) respectively (In Figure 2, we show the STG’s of machines, after simplifying the labels on some of the edges).

The function \( f \) computed by the Combinational network for generating the primary outputs can be written as follows:

\[
\begin{align*}
f(0,A_1,B_1) &= 0 & f(1,A_1,B_1) &= 1 \\
f(0,A_1,B_2) &= 1 & f(1,A_1,B_2) &= 0, \\
f(0,A_2,B_1) &= 1 & f(1,A_2,B_2) &= 0
\end{align*}
\]

Here, one can observe that state \( 1eA_1 \cap B_1 \) and \( \lambda(0,1) = 0 \), and thus, \( f(0,A_1,B_1) = 0 \). The outputs for other combinations are obtained in a similar manner.

5 The Cost Function

The state transition graph of a decomposed machine may contain parallel edges. If all the parallel directed edges emanating from a state \( v_1 \) and terminating on the state \( v_2 \) are replaced by single directed edge from \( v_1 \) to \( v_2 \), then we get a digraph \( G(V,E) \), where \( V \) is a set of states. Note, that this digraph is allowed to contain self-loops. The number of edges in the digraph \( G(V,E) \) is our cost function. We expect that the larger the number of edges in this digraph, the larger will be the number of distinct state transitions, and the larger will be the number of terms in the next state functions. This in turn would imply that the number of edges in the digraph is correlated with the area of the implementation.

We provide two justifications for this simple cost function, first by assuming a particular model for the implementation of the machines, and second by using empirical results obtained after running the standard logic synthesis program SIS.

5.1 Justification of the cost function based on a simple implementation model

We present a four level implementation model of the decomposed interacting machines, and show that the area of this implementation is governed by number of edges in the digraph \( G(V,E) \).

We will first assume that \( M_A \) and \( M_B \) have about \( \sqrt{N} \) states each, and are both one-hot coded. In the implementation model, we further assume that the input symbols have already been implemented separately. These input symbols are combined with the state bits to implement next state functions using a four level network as shown in Figure 3. The next state function in machine \( M_A \) is dependent on primary input symbols, the present state of the machine \( M_A \) as well as present state of the machine \( M_B \). Since, both the machines are one-hot coded, there are only \( 2\sqrt{N} \) present state
variables. 

As shown in Figure 3, there are four levels of gates. Levels 1 and 2 represent one AND-OR and levels 3 and 4 represent second AND-OR. AND gates in first level are two input AND gates and they compute the logic function activating a particular transition, which is the AND of a primary input symbol, and the present state of another machine. The OR gates in second level, combines all the symbols on parallel edges generated by the first level AND gates, to create edge function. Third level gates are again two input AND gates which combine the present states of the machine with the edge functions. The fourth level OR gates combine all the ANDed edge functions which are fan-ins of a given state. Thus, output of fourth level is a next state function. Now, if the AND-OR logic is replaced by NAND-NAND logic, then the number of transistors in that implementation indicates the area of implementation of next state function.

If I is the number of input symbols and L is the number of edges in a digraph G(V, E) of given machine, then the number of transistors required in a CMOS implementation for first level, second level, third level, fourth level are 4I√N, 2I√N, 4L, 2L respectively. Therefore, the total number of transistors required for the implementation of machine equals 6I√N + 6L. In this expression, L, √N are fixed by the original machine.

Therefore, the number of transistors in the implementation depends on the number of edges L, which becomes a cost function that can be used to grade the orthogonal partitions. Thus, we conclude that the number of edges in the digraph induced by an orthogonal partition is an indication of the area of implementation of complete machine.

5.2 Empirical Justification of the Cost Function

Since we are targeting multi-level logic implementation of the FSM, the number of literals in factored form is a reasonable measure of the area of the implementation. We generated data for multi-level implementations of the MCNC benchmark scf (with 121 states) using 50 randomly generated orthogonal partition pairs. Each decomposition was implemented using the logic synthesis program SIS.

In Figure 4, we show a plot of literals in factored form versus the number of edges in digraphs of decomposed machines. From this plot, one can observe that the number of edges in the digraph are correlated with the number of literals in factored form. Similar behavior was observed for other benchmarks as well. We took this to be a justification for choosing the number of edges of the digraph as our cost function.

In the next section, we propose a greedy algorithm to reduce the cost function.

6 The Algorithm

We wish to find orthogonal partitions ΠA, ΠB such that the number of edges in digraphs corresponding to the machines is small. We propose a greedy algorithm which attempts to minimize the number of edges in the digraphs corresponding to each partition. It is obvious that, if two elements are in the same block of ΠA, then they must be in different blocks of ΠB; otherwise their meet will not be Π0. The greedy algorithm builds partition ΠA by forcing tightly connected states into the same block, so that the edges between them are replaced by a self-loop. While building the second partition ΠB, states are added one at a time to blocks by doing a local search (on assignments of the state to blocks) to determine which assignment creates the minimum number of additional edges in the directed graph of ΠB.

The pseudo code for the algorithm is shown in Figure 5. The algorithm simultaneously generates the orthogonal partitions ΠA and ΠB. We assume that we have been provided positive integers n1, n2, with n1n2 ≥ N. The algorithm generates partitions such that ΠA will consist of n1 blocks, each containing at most n2 states, and ΠB will consist of n2 blocks, each containing at most n1 states.

The routine put_in_most_suitable_block() chooses one block among all the available blocks of ΠB, such that when a state is put in that block, the additional number of edges created in the directed graph corresponding to ΠB is minimized. The routine find_most_adjacent_state() returns the state with the maximum number of fan-in edges from the states already in a given block of the first partition ΠA. The following proposition states the complexity of the greedy algorithm.

Proposition 6.1 The greedy algorithm shown in Figure 5 produces two orthogonal partitions in time no more than O(n1n2 + n1n2(d_{in}^{max} + d_{out}^{max})), where d_{in}^{max}, d_{out}^{max} is the maximum fan-in and fan-out degree respectively.

Proof: The procedure Put_in_most_suitable_block() takes O(n1(d_{in}^{max} + d_{out}^{max})) time, while the procedure Find_most_adjacent_state() takes O(||V_{in}||) time. Initially, ||V_{in}|| ≤ n1n2 and each of these procedures is called at most n1n2 times. Therefore, the algorithm takes O(n1n2 + n1n2(d_{in}^{max} + d_{out}^{max})).
JEDI, NOVA, One-hot

Table 2. Comparison of Decomposition with JEDI, NOVA, One-hot

<table>
<thead>
<tr>
<th>Examples</th>
<th>JEDI</th>
<th>NOVA</th>
<th>DECOMP</th>
<th>One-Hot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Area</td>
<td>Delay</td>
<td>Area</td>
<td>Delay</td>
</tr>
<tr>
<td>planet</td>
<td>2138</td>
<td>8.26</td>
<td>105.54</td>
<td>8.77</td>
</tr>
<tr>
<td>C128</td>
<td>2138</td>
<td>24.16</td>
<td>277.76</td>
<td>8.74</td>
</tr>
<tr>
<td>s1488</td>
<td>6.89</td>
<td>5.29</td>
<td>6.89</td>
<td>5.29</td>
</tr>
<tr>
<td>s1944</td>
<td>23.28</td>
<td>5.24</td>
<td>23.28</td>
<td>5.24</td>
</tr>
<tr>
<td>s25</td>
<td>4.69</td>
<td>2.52</td>
<td>4.69</td>
<td>2.52</td>
</tr>
<tr>
<td>s50</td>
<td>39.44</td>
<td>5.78</td>
<td>39.44</td>
<td>5.78</td>
</tr>
<tr>
<td>s120</td>
<td>6.15</td>
<td>5.67</td>
<td>6.15</td>
<td>5.67</td>
</tr>
<tr>
<td>NOVA</td>
<td>6.81</td>
<td>6.75</td>
<td>6.81</td>
<td>6.75</td>
</tr>
<tr>
<td>ONE-HOT</td>
<td>6.48</td>
<td>6.81</td>
<td>6.48</td>
<td>6.81</td>
</tr>
<tr>
<td>Total</td>
<td>105.54</td>
<td>8.77</td>
<td>105.54</td>
<td>8.77</td>
</tr>
</tbody>
</table>

The same procedure was followed to produce circuits using JEDI [6] and NOVA [5]. In Table 1, we show information about the benchmarks and in Table 2, we show the comparison of the DECOMP results with the JEDI [6] and NOVA [5] results. In Table 2, we show the area of the multi-level implementation and the maximum delay in the combinational logic. The area figures for DECOMP were obtained by adding the area of decomposed machines as well as output logic. The reported delay for the implementation using DECOMP is the one which is the highest among the next state functions of the decomposed machines and the combined output logic.

From Table 2 we observe that, on an average, DECOMP produces circuits which have 8.52% less area and 81.87% less delay than circuits obtained using JEDI [6] and 4.40% less area and 104.96% less delay than the circuits obtained using NOVA [5]. As compared to ONE-HOT, on an average, DECOMP produces circuits which have 24.40% less area, while the delays are nearly the same. One can also observe from Table 1 and Table 2 that, for larger MCNC benchmarks, DECOMP produces circuits which have not only less area but also less delay.

In Table 2, consider the implementations of larger FSMs such as c128, planet, s1488, s1494, scf, s510. For c128, NOVA produces the best implementation, since it assigns a code which yields the lowest number of literals after the minimization. In this case, NOVA [5] is able to assign the vertices of the seven dimensional hypercube to the 128 states in an optimum manner. On the other hand, JEDI assigns a gray code which produces a higher number of literals in the next state function. DECOMP implements the 128 state counter as a 12 state and 11 state counter interacting with each other. The implementation of these two counters and the output combinational logic uses less area than JEDI and ONE-HOT implementations. We also note that in the case of the 12 state
counter modulo12, NOVA produces a poorer solution compared to DECOMP, which implements the 12 state counter using 4 state and 3 state counter. This implementation via DECOMP uses less area than NOVA and ONE-HOT. However, in this case, JEDI assigns a gray code which produces a smaller number of literals than DECOMP. It is clear that JEDI and NOVA produce good results for particular counters. Note that DECOMP always implements the \( N \) state counter using two \( \sqrt{N} \) state counters.

In case of planet, \( s1488 \), ONE-HOT produces the best results in terms of area. In case of \( s1488 \), planet, DECOMP produces better results than JEDI and NOVA considering the area of the implementation while the delays are comparable to ONE-HOT and less than JEDI and NOVA. In case of \( s1494 \), JEDI produces the best results area wise, but DECOMP produces an implementation slightly more expensive area wise, but having the lowest delay. In case of \( scf \) and \( s510 \), DECOMP produces implementations which not only have smaller areas, but also have smaller delays compared to JEDI, NOVA, ONE-HOT.

In case of small examples such as \( bhara \), \( bbtas \), \( train11 \), \( s27 \) DECOMP produces implementations which are expensive area wise and the delays obtained are also not significantly less. In the case of small examples, the area saved by using \( 2\sqrt{N} \) instead of \( N \) flip-flops is not as significant as it is in case of larger FSMs.

From this discussion, it appears that decomposition can be used as an alternative to conventional state assignment tools for area effective, high performance implementation of larger FSMs.

## 8 Conclusions

In this paper, we have described a technique for the decomposition of an \( N \) state machine into two interacting \( \sqrt{N} \) state machines. We observe that an effective implementation of the original machine can be obtained by one-hot implementations of the two smaller interacting machines, especially when the size of original state machine is large. Based on an empirical study, we observe that decomposition followed by one-hot implementations of the constituent machines can be superior to conventional state assignment tools when area effective and high performance implementations of large state machines are desired. Especially in the case of delays, our results are almost always significantly better than those obtained using conventional state assignment tools.

There are several future directions in the study of machine decompositions. First, note that the process of decomposition can be continued to form interactions of a larger number of machines. The implementation of each constituent machine also need not be in the one-hot style. One of the future directions can be decomposition for power reduction. In the case of decomposition, when states are put in a single block of the partition, it can create a number of self-loops in the corresponding machine. In fact, the number of self-loops in both machines can be a good cost function which can be maximized to reduce the power dissipation. Moreover, these self-loops can be implemented separately to disable the clock, and save more power. Also, these self-loop conditions can be used as don’t care conditions for implementation of the combinational logic of the machines. There is of course a price to paid in implementing the clock disabling logic, which consumes power, and might lead to additional delays in the clock path and result in an increased clock period. These issues need to be studied further.

## References


8. Frederick Hennie, Finite state models for logical machines, John Willey and Sons, 1968, pp. 140-175.
