Low Power Realization of Finite State Machines—A Decomposition Approach

SUE-HONG CHOW, YI-CHENG HO, TINGTING HWANG
Tsing Hua University, Taiwan
and
C. L. LIU
University of Illinois at Urbana-Champaign

We present in this article a new approach to the synthesis problem for finite state machines with the reduction of power dissipation as a design objective. A finite state machine is decomposed into a number of *coupled* submachines. Most of the time, only one of the submachines will be activated which, consequently, could lead to substantial savings in power consumption. The key steps in our approach are: (1) decomposition of a finite state machine into submachines so that there is a high probability that state transitions will be confined to the smaller of the submachines most of the time, and (2) synthesis of the coupled submachines to optimize the logic circuits. Experimental results confirmed that our approach produced very good results (in particular, for finite state machines with a large number of states).

Categories and Subject Descriptors: B.6.1 [**Logic Design**]: Design Styles–*sequential circuits*; B.6.3 [**Logic Design**]: Design Aids—*automatic synthesis*, *optimization*; J.6 [**Computer Applications**]: Computer-Aided Engineering—*computer-aided design* (*CAD*)

General Terms: Design, Performance

Additional Key Words and Phrases: Decomposition of finite state machines, lower power design, state assignment

1. INTRODUCTION

In CMOS circuits, power is dissipated in a gate when the gate output changes from 0 to 1 or from 1 to 0. Minimization of power dissipation can be

This work was partially supported by R.O.C. NSC under grant NSC 85-2221-E-007-033 and by U.S. NSF under grant MIP 92-22408.
Authors' addresses: S.-H. Chow, Y.-C. Ho, and T. Hwang, Department of Computer Science, Tsing Hua University, Hsin Chu, Taiwan 30043; email: ⟨tingting@cs.nthu.edu.tw⟩; C. L. Liu, University of Illinois at Urbana-Champaign, 1304 W. Springfield Avenue, Urbana, IL 61801.

considered at algorithmic, architectural, logic, and circuit levels [Chandrakasan et al. 1992]. Studies on low power design are abundant in the literature[1] in which various techniques are proposed to synthesize designs with low transitional activities.

In sequential circuit design, an effective approach to reduce power dissipation is to "turn off" portions of the circuit, and hence reduce the switching activities in the circuit. Two attempts have been made to exploit such an approach. Alidina et al. [1994] proposes a precomputation-based approach in which the output values of a sequential circuit are precomputed so that the original circuit can be turned off in the next clock cycle. Benini and De Micheli [1995b] describe a scheme to stop the clocking of a finite state machine (FSM) when the machine is in a self-loop and the outputs do not change.

In this article we propose a technique that is also based on selectively turning off portions of a circuit. Our approach is motivated by the observation that, for an FSM, active transitions occur only within a subset of states in a period of time. Therefore, if we synthesize an FSM in such a way that only the part of the circuit which computes the state transitions and outputs will be turned on while all other parts will be turned off, power consumption will be reduced.

Consider as an example the FSM, $dk27$ from the MCNC benchmark set, shown in Figure 1. In the example, assume that the input signals 0 and 1 are equiprobable. The steady state probabilities which are shown next to the states in Figure 1 can then be computed accordingly [Papoulis 1984]. Now suppose we partition the FSM into two submachines M_1 and M_2 along the dotted line. Then around 40% of the transitions occur in submachine M_1, 40% of the transitions occur in submachine M_2, and 20% of the transitions occur between submachines M_1 and M_2. Now suppose that the FSM is synthesized as two individual combinational circuits for submachines M_1 and M_2. Then we can turn off the combinational circuit for submachine M_2 when transitions occur within submachine M_1. Similarly, we can turn off the combinational circuit for submachine M_1 when transitions occur within submachine M_2.

Using Jedi [Lin and Newton 1989] for state encoding and SIS [Sentovich et al. 1992] for logic optimization for the original and the two submachines, we obtain logic circuit realizations with 23, 10, and 14 literals, respectively. Let the average power dissipation model be $P_{avg} = 1/2 \cdot C \cdot ({V_{dd}}^2/T_{cyc}) \cdot E$, where C is the load capacitance, V_{dd} is the supply voltage, T_{cyc} is the clock period, and E is the transition count. Using the number of literals to approximate the load capacitance, the ratio between the power consumption of the decomposed machine and that of the original machine is computed as $((10 + 14)/23) \times 0.2 + (10/23) \times 0.4 + (14/23) \times 0.4 = 0.62$. In other words, power consumption in the decomposed machine is about 62%

[1]Please see Roy and Prasad [1992], Prasad and Roy [1993], Dresig et al. [1993], Lin and de Man [1993], Tsui et al. [1993], Tiwari et al. [1993], Benini and De Micheli [1995a], and Hachtel et al. [1994].

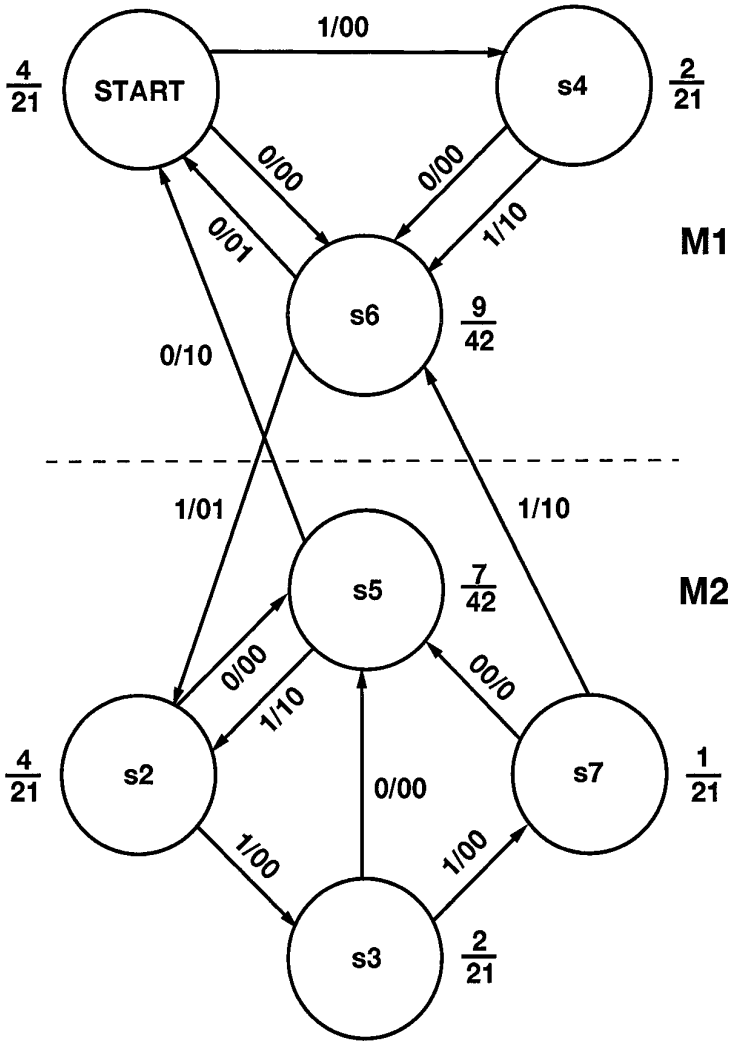

Fig. 1. Example of FSM ($dk27$).

of the original machine when the overhead for controlling the two submachines is ignored. (Such overhead is taken into consideration in our discussion later.)

In general, since the combinational circuit for each submachine is smaller than that for the original machine, power consumption in the decomposed machine will be smaller than that of the original machine. Besides reduction in power dissipation, realizing an FSM as a set of coupled submachines also provides the advantage of reducing the length of the critical path.

The remainder of this article is organized as follows. Section 2 discusses the computation model of a decomposed FSM. In Section 3, we describe a method for state partition. In Section 4, we discuss the state assignment

problem for the submachines. Section 5 describes an alternative approach to the state partition problem. In Section 6, we present experimental results on a set of circuits from the MCNC benchmark set. Finally, we provide some concluding remarks.

2. PARTITIONING A FINITE STATE MACHINE—THE PHYSICAL MODEL

In the general model of a synchronous FSM, a combinational circuit computes the output signals and the state signals for the next clock cycle, based on the input signals to the FSM and the state signals representing the current state of the machine. To reduce power consumption, we want to investigate the possibility of turning on only a portion of the gates in the combinational circuit in each clock cycle. To this end, we propose the partitioning of an FSM into submachines. At any moment, only one submachine is active (with its corresponding combinational circuit turned on) while all other submachines are inactive (with their corresponding combinational circuits turned off). The active submachine is in control. When the next input signal arrives, the active submachine might remain active. In that case, the other submachines will not be turned on and will remain inactive. On the other hand, when the next input signal arrives, the active submachine might turn on another submachine, set that submachine to the correct state, and turn itself off, becoming inactive.

Figure 1 shows an example in which an FSM is partitioned into two submachines M_1 and M_2, with their sets of states being $S_1 = \{START, s_4, s_6\}$ and $S_2 = \{s_2, s_3, s_5, s_7\}$. We say that these two submachines are *coupled* in the sense that there are transitions from one submachine to another. Such transitions are referred to as *crossing transitions*. The overall machine containing the coupled submachines is referred as a *decomposed machine*. In this example, there is one crossing transition from submachines M_1 to M_2, and two crossing transitions from submachines M_2 to M_1. Clearly, when a crossing transition takes place, control is transferred from one submachine to another.

Some important questions arise. The first question is: for a given set of submachines obtained from the partitioning of an FSM, how do we determine the submachine to be turned on in each clock cycle? The second question is: when an inactive submachine becomes active, how do we set it to the correct state for the next clock cycle? The third question is: how does an active submachine relinquish control and pass it to the submachine that will become active in the next clock cycle? And the last question is: physically, how do we turn on and off a piece of combinational logic?

The first question can be answered as follows: on the basis of its current state and the input signals, the active submachine is capable of determining the submachine that is to become active in the next clock cycle. However, the complexity of the control logic that determines the submachine to be turned on is a function of the state codes assigned to the states of the FSM. To simplify the control logic, state codes are assigned according to the following rules: a certain number of bits in a state code are

designated *control bits* which distinguish the submachines from one another. Consequently, states in the same submachine will have identical control bits in their state codes, whereas states in different submachines will have different control bits. The remaining bits in a state code will be used to distinguish among states in the same submachine. Then, based on the values of the control bits of the state code of the present state, the control logic will turn on the corresponding submachine and turn off the others.

Let an FSM M be partitioned into n submachines, $M_1, M_2, \ldots, M_n$. Suppose these n submachines have $S_1, S_2, \ldots, S_n$ states, respectively. Then, in the state codes, log$n$ control bits are required to distinguish between the n submachines, and *maximum* (logS_1, logS_2, . . . , logS_n) bits are required to distinguish between the states in one submachine.

Example 2.1. We now use the FSM in Figure 1 as an example. The FSM is partitioned into two submachines M_1 and M_2, with their sets of states being $S_1 = \{START, s_4, s_6\}$ and $S_2 = \{s_2, s_3, s_5, s_7\}$. One control bit is needed to distinguish between the states in S_1 and S_2, and two more bits are needed to distinguish between the states within each submachine. The state assignment $START = 000$, $s_4 = 010$, $s_6 = 011$ for the states in S_1, and $s_2 = 101$, $s_3 = 111$, $s_5 = 110$, $s_7 = 100$ for the states in S_2, is a legal one. Submachine M_1 will be turned on and submachine M_2 will be turned off if the first bit of the state code of the present state is 0. Similarly, submachine M_2 will be turned on and submachine M_1 will be turned off if the first bit of the state code of the present state is 1.

The second question is when an inactive submachine becomes active, how do we set it to the correct state for the next clock cycle. Note that upon receiving the input signals, the currently active submachine recognizes that it should relinquish control in the next clock cycle, and it is also responsible for determining the state code of the next state of the machine. (The state code determines unambiguously both the submachine and the state inside that submachine.) What we need to do is to include the crossing transitions in the state transition tables of the submachines. Clearly, when a crossing transition takes place, we know not only that control is transferred from one submachine to another, but also the state code of the next state of the machine.

Example 2.2. For the example in Figure 1, there are two submachines, M_1 and M_2. In submachine M_1, there are five transitions among states in S_1 and one crossing transition, $s_6 \rightarrow s_2$ when the input signal is 1. Similarly, in submachine M_2, there are six transitions among states in S_2 and two crossing transitions, $s_7 \rightarrow s_6$ when the input signal is 1, and $s_5 \rightarrow START$ when the input signal is 0.

The third question is how does an active submachine relinquish control to allow another submachine to become active. We note that the state assignment scheme previously described together with the inclusion of crossing transitions in the submachines will allow control to be transferred

from one submachine to another with no additional circuitry. Suppose submachine M_1 is active in the present clock cycle, and control is to be transferred from submachine M_1 to submachine M_2 for the next clock cycle. According to our synthesis procedure, submachine M_1 will compute correctly the values of the state variables for the next clock cycle with the control bit set to be such that submachine M_2 will be turned on and submachine M_1 will be turned off.

Example 2.3. To be specific, let us examine the example in Figure 1 again. Suppose submachine M_1 is active, and is in state s_6. Since the first bit (which is the control bit) of the state code for s_6 is 0, submachine M_1 is turned on and submachine M_2 is turned off. Now, if the input signal is 0, submachine M_1 will transit to state *START*. Since the first bit of the state code for *START* is also 0, submachine M_1 will remain active in the next clock cycle. On the other hand, if the input signal is 1, submachine M_1 will transit to state s_2. Since the first bit of the state code for s_2 is 1, submachine M_2 will be turned on and submachine M_1 will be turned off in the next clock cycle.

The last question is, in realizing the circuit, how do we actually turn on and off a piece of combinational logic? We propose to use multiplexers, decoder, and control gates (e.g., AND, or NAND gates) in our control logic. Figure 2 shows a general architecture in which edge-triggered flip-flops are used. Through the multiplexers, the control bits of the state code of the present state will determine the portion of the combinational logic from which the next state registers will be loaded as well as the correct output signals. The decoder generates the enable signals which activate the portion of the combinational logic corresponding to the submachine that is to become active. The AND gates in front of the original combinational circuit will block the state and primary input signals from propagating through the combinational circuit if the corresponding submachine should remain inactive.

Suppose the original machine has S_o states, P inputs and Q outputs, and was partitioned into $M_1, M_2, \ldots, M_n$ submachines where submachine M_i has S_i states for $i = 1 \ldots n$. Then the number of flip-flops in the decomposed machine is $S_d = \log n + maximum(logS_1, \log S_2, \ldots, \log S_n)$. The control logic overhead will include $(\log S_o - S_d)$ flip-flops, one $\log n$ to n decoder, $(n \times (S_d + P))$ two-input AND gates, and $(S_d + Q)$ n to 1 multiplexers.

Example 2.4. To be specific, we examine the example in Figure 1 again. The state codes have been chosen as in Example 2.1 and the two submachines M_1 and M_2 have been synthesized as described in Example 2.2. Figure 3 shows the block diagram for the overall realization. Note that there are one control signal, $control_1$, which is the output of the first flip-flop; one 1×2 decoder which will generate the enable signals e_1 and e_2 for submachines M_1 and M_2; four AND gates A, B, C, D in front of M_1 and four AND gates E, F, G, H in front of M_2 which will block the state and

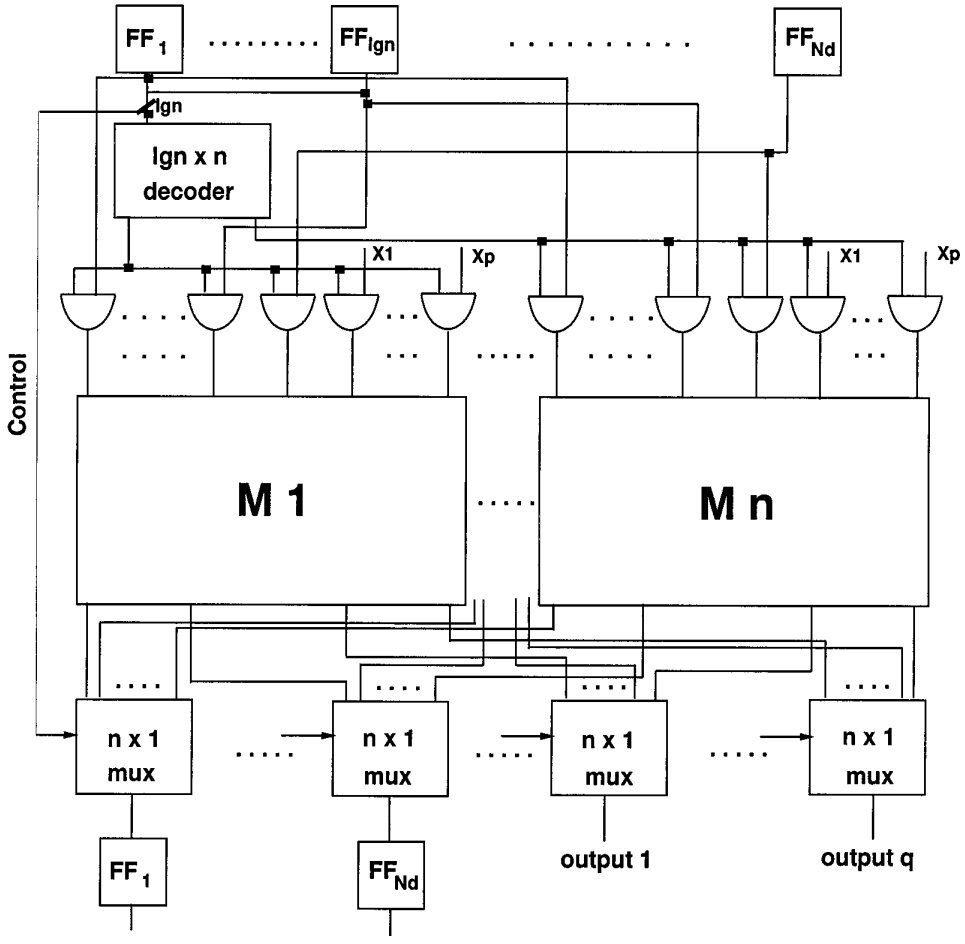

Fig. 2. General circuit structure of decomposed FSM.

primary input signals from propagating through M_1 and M_2, respectively; three multiplexers mux_1, mux_2, mux_3 which will determine whether the next state registers will be loaded from M_1 or M_2; and two multiplexers mux_4, mux_5 which will determine the correct output signals. Suppose submachine M_1 is active and is in state s_6. Since the first bit of the state code (the control signal $control_1$) of s_6, which is the input to the 1×2 decoder, is 0, the output signal from the decoder to Com_1, e_1, is 1 which will turn on all the AND gates A, B, C, D in front of submachine M_1. Thus, all signals fed to the circuit corresponding to submachine M_1 will propagate through. However, the output signal from the decoder to Com_2, e_2, is 0 which will turn off all the AND gates D, E, F, G in front of submachine M_2. Thus, all signals fed to the circuit corresponding to submachine M_2 will be blocked. Similarly, with the control signal $control_1$ being 0, signals for the next state flip-flops and output will come from Com_1 through the

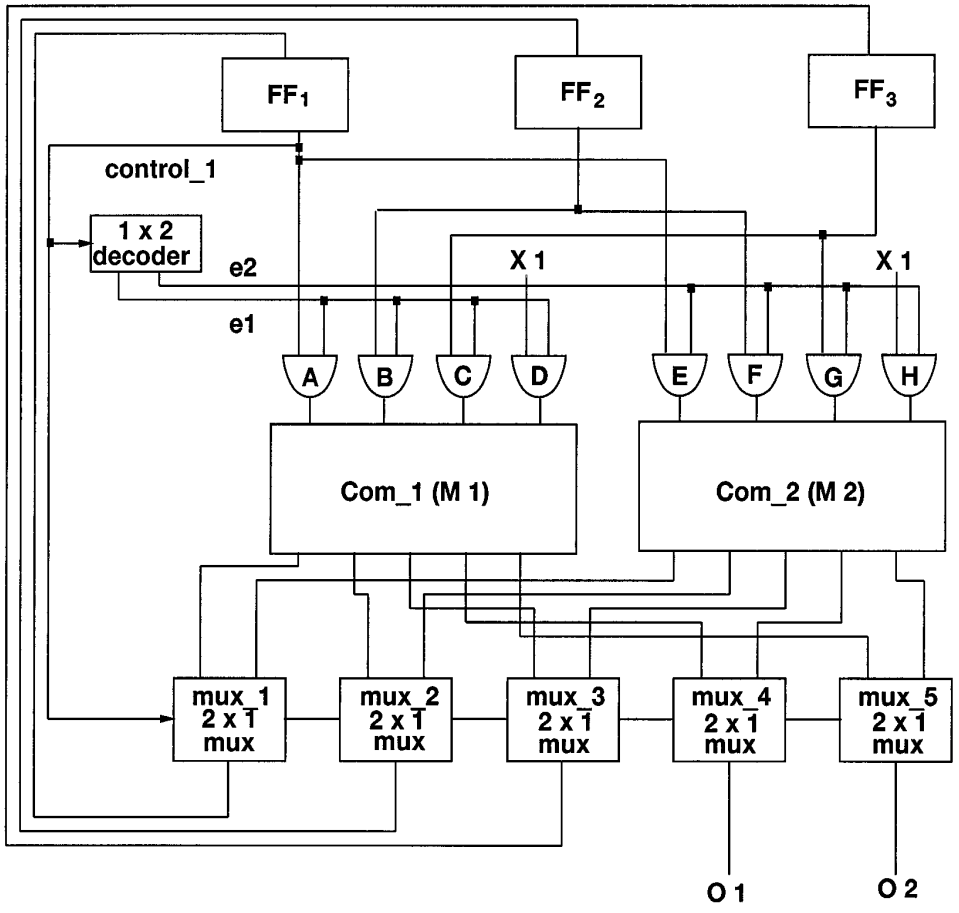

Fig. 3. Circuit of a decomposed FSM ($dk27$).

2×1 multiplexers. Now, if the input is 0, submachine M_1 will transit to state *START*. Since the first bit of the state code of *START* is also 0, submachine M_1 will stay active in the next clock cycle. On the other hand, if the input is 1, submachine M_1 will transit to state s_2. Since the first bit of the state code (the control signal $control_1$) of s_2 is 1, the output signal of the decoder e_2 will be 1 which will allow inputs to propagate through the AND gates E, F, G, H and Com_2, and thus turn on the submachine M_2. In the meantime, the value of e_1 is 0 which will inhibit the propagation of inputs through the AND gates A, B, C, D, and thus set all the inputs to submachine M_1 to 0s in the next clock cycle, rendering submachine M_1 inactive. If in subsequent clock cycles state transitions are confined to within submachine M_2, the value of e_1 will remain 0 and submachine M_1 will remain inactive.

It is clear from Example 2.4 that, at any moment, power consumption will be incurred in only one submachine M_i if transitions are confined

within that submachine. On the other hand, power consumption will be incurred in both submachines M_i and M_j when a transition from submachine M_i to M_j takes place. During the clock cycle in which M_i is turned off and M_j is turned on, both machines consume power.

3. PARTITIONING A FINITE STATE MACHINE INTO SUBMACHINES

As previously pointed out, we want to partition an FSM into a number of submachines. These submachines are said to be *coupled* in the sense that state transitions take place either within a submachine or between two submachines. To synthesize an FSM as a set of coupled submachines, we need to solve two problems: the first is to partition the FSM into submachines; that is, to partition the set of states of the FSM into subsets. The second is to synthesize the submachines; that is, to assign state codes to the states of the submachines. We present in this section a partitioning algorithm, and in the next section a synthesis procedure for the submachines.

According to our discussion in Section 1, we want to partition an FSM into submachines such that the probabilities of state transitions within a submachine are high and the probabilities of state transitions between two submachines are low.

We employ a two-phase partitioning algorithm. In the first phase, an initial clustering of the states of the FSM is performed in which states are clustered based on the notion of closeness of states. Two states are said to be close if there are many transitions between them. In the second phase, clusters obtained in the first phase are grouped to form a final partition based on an estimation of the total power consumption of all the submachines.

Our clustering algorithm is based on a method proposed by Hagen and Kahng [1992] for clustering the circuit elements in a circuit. The method is based on the notion of *sameness* between two circuit elements, which is a measure of the desirability of putting the two circuit elements in the same cluster. The sameness between two circuit elements is computed based on a *random walk* in the net-list graph of the circuit. For our algorithm, instead of a net-list graph with circuit elements as vertices, we have the state transition graph of an FSM with states as vertices. Furthermore, since we are given the probability distribution of the input signals, we construct a random walk according to such a probability distribution. For an FSM with n states, a random walk of length n^3 is constructed. A *cycle* in the random walk is defined to be a sequence of states $\{s_p, s_{p+1}, s_{p+2}, \ldots, s_q\}$ with $s_p = s_q$ and all s_i distinct, $i = p + 1, p + 2, \ldots, q - 1$. Intuitively, the set of states in a cycle corresponds to (part of) a natural cluster. The sameness of two states s_p and s_q, $sameness(s_p, s_q)$, is computed based on the cycles in the random walk that reflect the commonality of the set of states that are visited in cycles originating at s_p and s_q. Two states s_p and s_q will be placed in the same cluster if $sameness(s_p, s_q) > 0$.

In the second phase, clusters obtained in the first phase are grouped to form a final partition based on an estimation of the total power consumption of all the submachines. The total power consumption is estimated on the basis of the areas and the state transitional probabilities of the submachines. Suppose the original machine M has S_o states, P inputs and Q outputs, and was partitioned into $\pi = (M_1, M_2, \ldots, M_n)$ submachines where submachine M_i has S_i states. Let SP_i denote the probability that a transition will bring a state in M_i back to a state in M_i. SP_i is computed as follows. Let $s_{i1}, s_{i2}, \ldots, s_{it}$ denote the states of submachine M_i. Let $P(s_{i1}), P(s_{i2}), \ldots, P(s_{it})$ denote the probabilities that the machine is in states $s_{i1}, s_{i2}, \ldots, s_{it}$, respectively, at time equal to infinity. (In other words, $P(s_{i1}), P(s_{i2}), \ldots, P(s_{it})$ are the steady state probabilities.) Let the sum of the probabilities of occurrence of all input signals that will bring M_i from state s_{ij} to any state of M_i be PIM_{ij}. SP_i is computed as follows.

$$SP_i = \sum_{k=1}^{t} P(s_{ik}) \times PIM_{ik}. \tag{1}$$

Let CP_{ij} denote the probability that a transition will occur between a state in M_i and a state in M_j. Let the sum of the probabilities of occurrence of all input signals that will bring M_i from state s_{ij} to any state of M_j be POM_{ij}. CP_{ij} is computed as follows.

$$CP_{ij} = \sum_{k=1}^{t_i} P(s_{ik}) \times POM_{ik} + \sum_{k=1}^{t_j} P(s_{jk}) \times POM_{jk}. \tag{2}$$

Power consumption in a submachine is estimated to be proportional to the area of the submachine. Let $area_i$ denote the area of submachine M_i. $area_i$ is estimated as:

$$area_i = [P + \log(S_i)] \times \#\_rows \times \frac{Q}{2}, \tag{3}$$

where #_*rows* is the number of rows in the transition table specifying the state transitions in submachine M_i. The value of #_*rows* corresponds to the number of cubes in the submachine before any logic optimization is performed. The area of a submachine is estimated by the number of literals in a sum-of-product form realization of the submachine. To estimate the number of literals in a sum-of-product form realization, the number of cubes in the realization is estimated to be #_*rows*, and the average number of variables in a cube is estimated to be the sum of the number of input bits P and the state code length log S_i. Consequently, the total number of literals is estimated to be the total number of variables in all cubes multiplied by one half of the number of outputs, which is due to the assumption that the probability of each output being asserted is one half.

Now, the total power consumption is estimated as:

$$power\_cost(M, \pi) = \sum_{i=1}^{n} SP_i \times area_i + \sum_{i,j} CP_{ij} \times (area_i + area_j) + overhead(\pi), \quad (4)$$

where $overhead(\pi)$ is computed according to the definition in Section 2.

The second phase begins with the selection of the seeds for a partition. Suppose the clusters are to be merged to form a partition with n groups, where n is given. The steady state probability of a cluster is computed to be the sum of the steady state probabilities of the states in the cluster. The $n - 1$ clusters with the largest steady state probabilities are selected as the seeds of $n - 1$ groups, $G_1, G_2, \ldots, G_{n-1}$. The remaining clusters form the last group, G_n. Then, iteratively, a cluster in G_n is selected for possible movement into one of the other groups $G_1, G_2, \ldots, G_{n-1}$ according to the estimated power consumption computed based on Eq. (4). In each iteration, the cluster in G_n with the largest steady state probability C_k is selected. Power consumption is evaluated for all the possibilities of moving C_k into one of $G_1, G_2, \ldots, G_{n-1}$ as well as for the possibility of leaving C_k in G_n. The possibility with the least power consumption will be selected. If C_k is moved into one of $G_1, G_2, \ldots, G_{n-1}$, the iteration continues. If C_k remains in G_n, the iteration stops. The reason behind this heuristic is that when a cluster is moved into a group, it increases the steady state probability of the group, it also increases the area of the group. Clearly, groups with high steady state probabilities and small areas are desirable. Figure 4 shows the partitioning algorithm.

4. STATE ASSIGNMENT FOR SUBMACHINES

In this section we discuss how the submachines are to be realized. In the classical case of realizing an FSM, state codes are assigned to represent the states. Once such assignment is made, a corresponding logic circuit can be synthesized. In fact, there are existing software tools for solving the state assignment problem for FSMs. One of the most notable systems is Jedi [Lin and Newton 1989]. However, in our case, since the submachines to be realized are coupled (there are state transitions from one submachine to another), they cannot be synthesized individually in the traditional way. We propose here an algorithm which first "decouples" the submachines and then utilizes existing state assignment packages such as Jedi. In fact, the general state assignment problem for coupled submachines (assigning state codes to the states of all submachines simultaneously to optimize a certain objective function) is an interesting and challenging research problem in its own right.

Jedi assigns codes to states taking into account both fanin-oriented relations and fanout-oriented relations [Devadas et al. 1991]. Jedi com-

Algorithm *Partition*(M, n)
Input: M = The original FSM;
n = The number of groups to be partitioned ;
Output: return ($\pi = (M_1, ..., M_n)$);
Begin
Random walk for an initial clustering;
Sorting clusters in the initial clustering as $C_1, ...C_m$ in decreasing order according to state probability of a cluster;
for $i = 1$ to $n - 1$
$G_i = C_i$;
endfor
$G_n = C_n + C_{n+1} + ... + C_m$;
$\pi_{new} = (G_1, ..., G_n)$;
$Cost_{old} = \infty$;
$Cost_{new} = power\_cost(M, \pi_{new})$;
while ($Cost_{new} < Cost_{old}$)
C_k = the first cluster in G_n ;
$Cost_{best} = \infty$;
for $i = 1$ to $n - 1$
$\pi = (G_1, ..., G_n)$ where $G_i = G_i + C_k$ and $G_n = G_n - C_k$;
$Cost = power\_cost(M, \pi)$;
if ($Cost < Cost_{best}$)
$Cost_{best} = cost$;
$\pi_{best} = \pi$;
endif
endfor
$Cost_{old} = Cost_{new}$;
$Cost_{new} = Cost_{best}$;
$\pi_{new} = \pi_{best}$;
endwhile
return (π_{new})
end

Fig. 4. *Partition* algorithm.

putes a set of weights between pairs of states to reflect the relations imposed by the input and output parts of a transition. States with strong weighting relations will be assigned codes that are within short Hamming distance such that a large number of common cubes will result.

In a coupled submachine, there are two types of transitions; transitions among states within the submachine and transitions from states in the submachine to states in other submachines. The second type of transitions is referred to as *crossing transitions* of the FSM. In our state assignment algorithm, we first delete all crossing transitions in a submachine and then use Jedi to assign state codes to the states of the submachine. Note that if we do not delete the crossing transitions from the submachine, state codes will also be assigned by the state assignment tool (e.g., Jedi) to the fanout

states of the crossing transitions, which are states that belong to other submachines. This is undesirable because, in the first place, the state code length for the submachine might have to increase in order to accommodate these fanout states, and secondly, state codes are assigned to the fanout states without taking into consideration the structures of the submachines to which these states belong. However, if we simply delete the crossing transitions and let the state assignment tool assign state codes to states that belong only to the submachine without considering the structural constraints imposed by the crossing transitions, then the logic circuits that we eventually synthesize for the submachines are likely to be more complex. (The logic circuits are synthesized after the state codes for the states in all submachines have been determined.)

In our approach, all crossing transitions in a submachine are deleted. However, pseudo-outputs are introduced to force Jedi to assign state codes that are close together (in terms of Hamming distance) to some of the states for the purpose of minimizing the logic circuit that realizes the submachine. A pseudo-output bit is added corresponding to each relation imposed by the crossing transitions. Transitions (rows in the transition table) whose current states should be assigned close state codes will have a pseudo-output of value "1" and all other transitions (rows in the transition table) will have a pseudo-output of value "0". That is, we create an output relation (fanout-oriented relation) among those states that should be assigned codes that are close together (in terms of Hamming distance). Jedi will give larger weights to those states that have the value "1" in a pseudo-output bit due to the output relation (fanout-oriented relation). The effect of the pseudo-output bits is essentially to use output relations (fanout-oriented relations) to represent the relations imposed by the crossing transitions that have been deleted from the transition table.

These rules should be followed when pseudo-output bits are added to represent the relations imposed by the deleted crossing transitions:

Rule 1: (fanout-oriented) If two or more states in one submachine have transitions going to the same state of another submachine under the same input, these states should be assigned codes that are close together (in terms of Hamming distance). Therefore, a pseudo-output bit is introduced. For each state that is a fanin state of a crossing transition, the pseudo-output bit of all transitions from this state will be "1". For all other transitions, the pseudo-output bit is "0". Figure 5(a) depicts this rule, where solid circles denote states that should be assigned codes that are close together.

Rule 2: (fanin-oriented) If one state has transitions going to two or more states in another submachine, all these fanout states in the other submachine should be assigned codes that are close together. In this case, a pseudo-output bit is introduced in the other submachine. For each state that is a fanout state of a crossing transition, the pseudo-output bit of all transitions from this state will be "1". For all other transitions, the

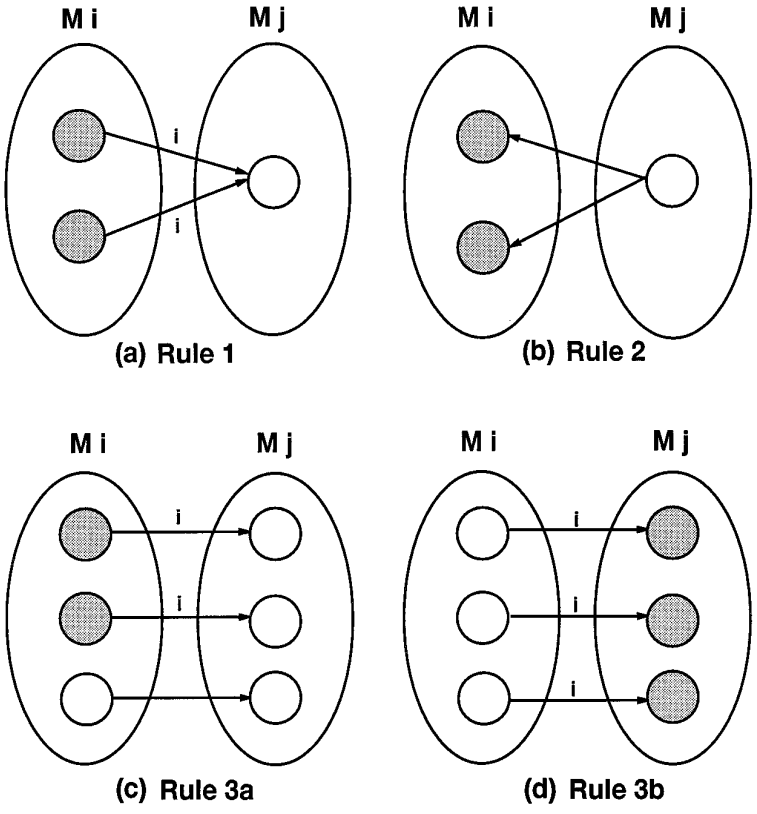

Fig. 5. Rules for adding pseudo-output bits.

pseudo-output bit is "0". Figure 5(b) depicts this rule, where solid circles denote states that should be assigned codes that are close together.

Rule 3a: (fanout-oriented) For states in one submachine that are fanin states of crossing transitions, if the fanout states of these crossing transitions are in the same submachine, and if these crossing transitions transit under the same input, then these states should be assigned codes that are close together. Therefore, one pseudo-output bit is introduced. For each state that is a fanin state of a crossing transition, the pseudo-output bit of all transitions from this state will be "1". For all other transitions, the pseudo-output bit is "0". Figure 5(c) depicts this rule, where solid circles denote states that should be assigned codes that are close together.

Rule 3b: (fanin-oriented) For states in the same submachine that are fanout states of crossing transitions, if the fanin states of these crossing transitions are in the same submachine, and if these crossing transitions transit under the same input, then these states should be assigned codes that are close together. Therefore, one pseudo-output bit is introduced. For each state that is a fanout state of a crossing transition, the pseudo-output bit of all transitions from this state will be "1". For all other transitions, the

Table I. Original Transition Tables

M_1			
input	present state	next state	output
0	s0	s1	1
1	s0	s3	0
0	s1	s2	1
1	s1	s3	0
0	s2	s1	1
1	s2	s5	1

M_2			
input	present state	next state	output
0	s3	s0	1
1	s3	s1	0
0	s4	s1	1
1	s4	s5	0
0	s5	s4	0
1	s5	s3	0

pseudo-output bit is "0". Figure 5(d) depicts this rule, where solid circles denote states that should be assigned codes that are close together.

After the introduction of pseudo-output bits according to the relations imposed by the crossing transitions in each submachine, Jedi is invoked to perform state assignment for each submachine individually. The following example illustrates how pseudo-output bits are added.

Example 4.1. Consider the transition tables of two submachines M_1, M_2 shown in Table I. The crossing transitions between these two submachines are shown in Figure 6. There are three crossing transitions from M_1 to M_2, namely, $s0 \rightarrow s3$, $s1 \rightarrow s3$, and $s2 \rightarrow s5$. There are also three crossing transitions from M_2 to M_1, namely, $s3 \rightarrow s0$, $s3 \rightarrow s1$, and $s4 \rightarrow s1$. After adding all pseudo-outputs and deleting all crossing transitions, we obtain the transition tables of the resultant submachines shown in Table II.

We now show how pseudo-output bits are added to the two submachines. For the two crossing transitions $s0 \rightarrow s3$ and $s1 \rightarrow s3$ in submachine M_1, since their fanout states are the same and their transitions are under the same input "1", according to Rule 1, one pseudo-output bit [the second output bit in the output column in Table II(a)] is added in which the first and second transitions have the pseudo-output value "1". Similarly, one pseudo-output bit (the third output bit in the output column) is added according to Rule 2, one pseudo-output bit (the fourth output bit in the output column) is added according to Rule 3a, and one pseudo-output bit (the fifth output bit in the output column) is added according to Rule 3b. In submachine M_2, two pseudo-output bits are added. One [the second output bit in the output column in Table II(b)] is added according to Rule 3a and another one (the third output bit in the output column) is added according to Rule 3b.

Note that if all the transitions from a state happen to be crossing transitions, all of them will be deleted from the submachine. So that this state will not disappear from the submachine, we introduce a transition from this state to a "don't care" state (denoted *).

5. AN ALTERNATIVE APPROACH

There is an alternative approach to the decomposition problem which is quite simple and straightforward. A given FSM is synthesized to obtain a

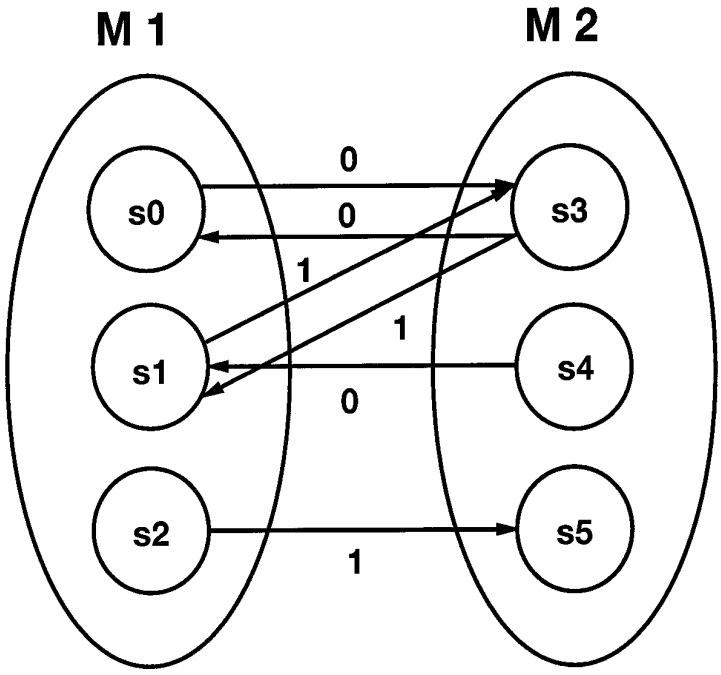

Fig. 6. Crossing transitions of submachines.

Table II. Resultant Transition Tables

(a) M_1			
input	present state	next state	output
0	s0	s1	11111
0	s1	s2	11111
0	s2	s1	10001

(b) M_2			
input	present state	next state	output
1	s3	*	011
1	s4	s5	010
0	s5	s4	001
1	s5	s3	001

gate-level description. The FSM can then be decomposed into submachines by computing the *cofactors* of the synthesized sequential circuit with respect to one or more of the state variables. For example, to decompose an FSM into two submachines, we can compute its cofactors with respect to $y_i = 0$ and $y_i = 1$ for any state variable y_i. The two subcircuits so obtained correspond to two coupled submachines with y_i being the control signal. We can compute the cofactors with respect to each of the state variables and select the decomposition that has the lowest power consumption. Clearly, we can also compute the cofactors with respect to two state variables to obtain a decomposition into four submachines, and so on. This type of decomposition is referred as *cofactor decomposition*.

As was expected intuitively and is also demonstrated experimentally in the next section, such an approach, although it indeed leads to a reduction in power consumption, is not as effective as the algorithm presented in

Sections 3 and 4 because transitional probabilities are not taken into consideration when decomposition of the FSM is carried out.

6. EXPERIMENTAL RESULTS

Our algorithm which consists of the partitioning algorithm in Section 3 and the state assignment algorithm in Section 4 has been implemented as a software tool called FSMD in the C language and executed on a SUN Sparc station. To demonstrate the effectiveness of our algorithm, 15 MCNC benchmark circuits were tested. Only circuits with more than 16 states and more than 100 literals in the original realization were selected because it is less meaningful to perform decomposition on small machines. For each machine, state assignment was performed by Jedi. The option we used for all examples was *jedi -e c*. After state assignment, SIS was called to perform logic optimization. For logic optimization, the standard script was first used. External "don't cares" were then extracted and *full simplify* was called. Power consumption was estimated for the circuit (including the added control logic) by using the command *power_estimate -t S*. Clock rate was set to the default value of 20 MHz in SIS.

The first set of experiments was performed to compare the literal count and the power consumption between the original machine and the decomposed machine. First, a two-way decomposition was performed on the original FSM. Then, for each submachine, we carried out the state assignment and logic optimization steps separately. Finally, the two submachines were connected with control logic added. Table III shows the experimental results. The columns labeled "#lit" show the number of literals in the realization of a machine. The columns labeled "#power" show the power consumption (in μW) in a machine. The columns labeled "#st" and "SP" give the number of states in a submachine, and the probability of transition within the submachine, respectively. The column "ratio" is the ratio between power consumption in the decomposed machine and in the original machine. For all examples except $s1$, the decomposed machines achieve a 15 to 59% reduction in power dissipation. Such reductions are achieved mainly because an FSM is decomposed in such a way that there is a high probability that transitions will take place within a submachine that has only a small number of states (i.e., a small area). For example, $s1488$ is decomposed into two submachines M_1 and M_2. M_1 has 4 states and a probability of 0.89 for its transitions to stay within the submachine. M_2 has 44 states but a probability of only 0.01 for its transitions to stay within the submachine. The crossing transitional probability is 0.1. Consequently, during the operation of the decomposed machine, only a small submachine will be active most of the time. This explains why the decomposed machine for $s1488$ attains a 59% power reduction. Only for the example $s1$, power consumption is increased by about 39% in the decomposed machine. We found that, in $s1$ there are adjacency relations among the states that, when properly utilized in the assignment of state codes, yield a relatively simple logic circuit realization of the machine. In other words, if $s1$ is synthesized

Table III. Comparisons of Original M and Decomposed M

circuit	original M		M1			M2			FSMD M		
	#lit	power	#st	SP	#lit	#st	SP	#lit	#lit	power	ratio
bbsse	103	373.2	4	0.94	39	12	0.01	57	152	318.1	0.85
cse	186	422.8	3	0.87	44	13	0.01	165	245	257.7	0.61
dk16	224	1,238.3	12	0.48	101	15	0.18	115	251	868.7	0.70
ex1	214	789.8	8	0.93	73	12	0.02	98	257	409.8	0.52
keyb	168	599.4	3	0.90	56	16	0.01	146	246	492.4	0.82
planet	472	2,375.5	19	0.63	232	29	0.30	251	536	1,337.9	0.56
pma	202	965.1	9	0.84	62	15	0.09	113	237	622.2	0.64
s1	113	502.4	7	0.50	76	13	0.24	147	278	700.2	1.39
s1488	503	1,448.6	4	0.89	41	44	0.01	489	593	592.9	0.41
s1494	514	1,276.2	4	0.89	52	44	0.01	503	625	938.3	0.74
s510	269	778.7	20	0.54	117	27	0.41	144	324	519.1	0.67
sand	527	1,525.5	12	0.30	321	20	0.62	120	488	634.7	0.42
sse	103	373.2	4	0.94	39	12	0.01	57	152	318.1	0.85
styr	407	1,073.0	4	0.87	63	26	0.03	405	501	695.8	0.65
tbk	330	978.2	4	0.64	78	28	0.02	180	293	744.8	0.76

* power : μW

as a single machine, many large cubes can be formed utilizing these adjacency relations. However, when the states are partitioned into two sets, some of the adjacency relations are destroyed. Consequently, many of the large cubes can no longer be formed in the submachines, which results in an increase in the number of literals. In fact, one submachine has a larger literal count than the original one. This is the major reason that the power consumption in the decomposed machine is worse than that of the original one.

In most cases, the decomposed machine synthesized by our algorithm attains a reduction in power consumption at a cost of increase in area. (For circuits *sand* and *tbk*, we attain reductions in both literal count and power consumption.) To measure the effectiveness of our approach, we define the effectiveness ratio as:

$$R_{eff} = \frac{\#lit_{decomposed} \times power_{decomposed}}{\#lit_{original} \times power_{original}} \tag{5}$$

If the effectiveness ratio is greater than 1, it means the reduction in power consumption we attain is offset by an increase in area. Table IV shows the effectiveness ratios for the machines we tested. Although almost all the examples tested attain a reduction in power, it is clear from the table that we have a high area overhead for small machines. For example, the effectiveness ratios for the three smallest machines (with literal counts close to 100) are all greater than 1. For larger machines, the effectiveness ratios are in fact quite small.

Table IV. Comparisons of Effectiveness Ratios

circuit	original M #lit×power	FSMD M #lit×power	R_{eff}
bbsse	38,439.6	48,351.2	1.26
cse	78,640.8	63,136.5	0.80
dk16	277,379.2	218,043.7	0.79
ex1	169,017.2	105,318.6	0.62
keyb	100,699.2	121,130.4	1.20
planet	1,121,236.0	717,114.4	0.64
pma	194,950.2	147,461.4	0.76
s1	56,771.2	194,655.6	3.43
s1488	728,645.8	351,589.7	0.48
s1494	655,966.8	586,437.5	0.89
s510	209,470.3	168,188.4	0.80
sand	803,938.5	309,733.6	0.39
sse	38,439.6	48,351.2	1.26
styr	436,711.0	348,595.8	0.80
tbk	322,806.0	218,226.4	0.68

We then conducted two sets of experiments to measure the performance of our partitioning algorithm (Section 3) and state assignment algorithm (Section 4). Our partitioning algorithm partitions the state set of an FSM into subsets, taking into account the probability distribution of the input signals. To confirm the effectiveness of our approach, we conducted a set of experiments on one example, $s1494$, assuming the probability distributions for the input signals "0" and "1" to be (0.9, 0.1), (0.8, 0.2), . . . , and (0.1, 0.9). For each of these probability distributions, we measured the power consumption for

(i) the original machine (the result is referred to as $s1494.orig$);
(ii) a decomposed machine obtained by our partitioning algorithm by *assuming* that the input signals 0 and 1 were equiprobable (the result is referred to as $s1494.equal\_partition$); and
(iii) a decomposed machine obtained by our partitioning algorithm using the given probability distribution for the input signals (the result is referred to as $s1494.prob\_partition$).

These results are plotted as shown in Figure 7. That $s1494.prob\_partition$ is superior to the other two sets of results confirms that our partitioning algorithm is indeed very effective.

Another set of experiments was performed to demonstrate the effectiveness of the state assignment algorithm in Section 4 that utilizes the concept of pseudo-output bits. After an FSM was decomposed into subma-

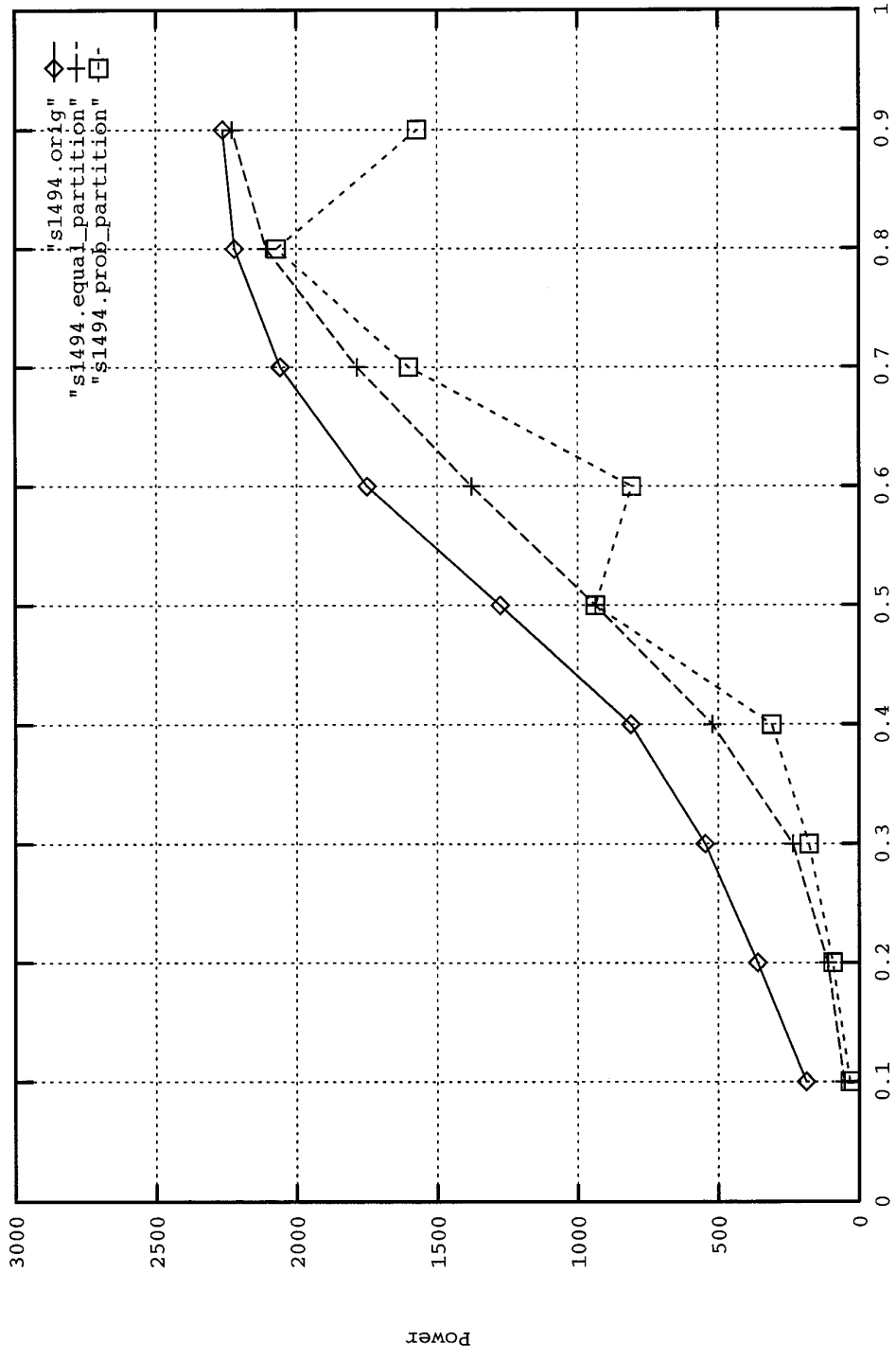

Fig. 7. Power consumption of *s*1494 with different input signal probabilities.

Table V. Comparisons of Literal Counts Without and With Pseudo Outputs

circuit	M1		M2		M1 + M2		
	without	with	without	with	without	with	ratio
bbsse	39	39	58	57	97	96	0.99
cse	42	44	177	165	219	209	0.95
dk16	103	101	138	115	241	216	0.90
ex1	82	73	108	98	190	171	0.90
keyb	54	56	240	146	294	202	0.69
planet	236	232	251	251	487	483	0.99
pma	64	62	142	113	206	175	0.85
s1	82	76	134	147	216	223	1.03
s1488	43	41	428	489	471	530	1.13
s1494	49	52	494	503	543	555	1.02
s510	117	117	144	144	261	261	1.00
sand	330	321	148	120	478	441	0.92
sse	39	39	58	57	97	96	0.99
styr	75	63	319	405	394	468	1.19
tbk	103	78	254	180	357	258	0.72

chines, we (i) used Jedi to assign state codes to the states of the submachines by simply deleting the crossing transitions, and (ii) assigned pseudo-output bits to the submachines and then used Jedi to do the state assignment. The results after logic minimization are shown in Table V. The literal counts of the two submachines are shown under the columns labeled "M1" and "M2," respectively. The column labeled "M1 + M2" gives the sum of the literal counts of the two submachines. The columns "without" and "with" give the literal counts when state assignment was carried out without and with the introduction of pseudo-output bits, respectively. The table shows that our state assignment algorithm which utilizes the pseudo-output bits outperforms the one without utilizing the pseudo-output bits in 10 of the 15 examples.

Realizing an FSM as a set of coupled submachines also provides the additional advantage of reducing the length of the critical path. The quantity *action* is defined to measure the energy consumption and critical path delay in a circuit:

$$action = En \times \tau_d = P \times \tau_d^2 = \frac{1}{2} \cdot C \cdot V_{dd}^2 \cdot E \cdot \tau_d, \tag{6}$$

where En is the energy consumption, τ_d is the critical path delay, C is the load capacitance, and E is the transition count. We computed the critical

Table VI. Comparisons of Values of Action of Original M and FSMD M

circuit	original M			FSMD M			ratio
	delay	energy	action	delay	energy	action	
bbsse	23.80	18.66	444.11	22.00	15.905	349.91	0.79
cse	36.80	21.14	777.95	33.20	12.885	427.78	0.55
dk16	54.80	61.915	3,392.94	32.80	43.435	1,424.67	0.42
ex1	34.00	39.49	1,342.66	27.80	20.49	569.622	0.42
keyb	28.40	29.97	851.15	41.20	24.62	1,014.34	1.19
planet	73.20	118.775	8,694.33	51.60	66.895	3,451.78	0.40
pma	33.80	48.225	1,630.97	31.20	31.11	970.63	0.60
s1	24.20	25.12	607.90	38.00	36.67	1,393.46	2.29
s1488	57.00	72.43	4,128.51	68.40	29.645	2,027.72	0.49
s1494	58.40	63.81	3,726.5	50.00	46.915	2,345.75	0.63
s510	33.60	38.935	1,308.22	30.00	25.955	778.65	0.60
sand	58.80	76.275	4,484.97	36.00	31.735	1,142.46	0.25
styr	60.00	53.65	3,219.00	55.80	34.79	1,941.28	0.60
tbk	59.00	48.91	2,885.69	38.60	37.24	1,437.46	0.50

* energy: $\mu W/MHz$; delay: ns

path delay using the command *map -s* in SIS to map a design to the cell library $lib1.2.sis2lib$. Table VI shows the values of *action* for the original and the decomposed machines (including the added control logic).

The column labeled "delay" is the critical path delay of the mapped circuits measured by *map* and the column labeled "energy" is the energy consumed (power-delay product) which is calculated by dividing the power consumed by the clock frequency (20 MHz). The column labeled "action" is the value of *action* which is calculated as the product of energy consumed and maximum delay of the mapped circuit. The column labeled "ratio" is the ratio of the values of *action* between the decomposed machine and the original machine. For all examples except *keyb*, $s1$, $s1488$, critical path delay is reduced in the decomposed machines. The increase in critical path delay for examples *keyb,* $s1488$ is due to the fact that states are partitioned unevenly. For *keyb*, one submachine has 16 states out of a total of 19 states, and for $s1488$, one submachine has 44 states out of a total of 48 states. After adding the control overhead, the reduction in the critical path delay due to a smaller number of states in a submachine is offset by the additional control logic. In terms of the value of *action*, the decomposed machine outperforms the original machine for all examples but two.

Another set of experiments was performed to compare the results of two-way decomposition and four-way decomposition. In this set of experiments, we chose only those machines the literal counts of which are larger than 400 in the original machine realization because the gain obtained by decomposing a small machine will be offset by the overhead of a four-way

Table VII. Results of Four-Way Decomposition

circuit	M1			M2			M3			M4		
	#st	SP	#lit	#st	SP	#lit	#st	SP	#lit	#st	SP	#lit
planet	11	0.13	132	11	0.22	129	8	0.18	65	18	0.19	196
s1488	1	0.71	16	3	0.01	26	5	0.01	98	39	0.00	419
s1494	1	0.71	16	3	0.01	26	5	0.01	99	39	0.00	420
sand	9	0.20	123	4	0.13	145	7	0.07	135	12	0.45	47
styr	1	0.42	32	3	0.06	39	4	0.04	57	22	0.01	483

decomposed machine. The results of four-way decomposition are shown in Table VII. The comparison between two- and four-way decompositions is shown in Table VIII. We can see from the table that significant improvement in power dissipation was attained for $s1494$ and $s1488$ because they were decomposed in such a way that there is a high probability that transitions will take place within a submachine that has only a small number of states.

Another set of experiments was performed to compare the literal count and the power consumption of the decomposed machine obtained by our algorithm and that of the machine obtained by computing the cofactors as described in Section 5. We first performed state assignment and logic optimization on the original machine. For each state variable, the cofactors with respect to that state variable were computed. The subcircuits of the cofactors were optimized again and then connected with control logic. Cofactor decompositions were performed for all state variables, and the best result was selected. Table IX shows the results. For all examples except $s1494$, cofactor decomposition is consistently inferior to our decomposition method. As was previously pointed out, this is because cofactor decomposition does not take into consideration the state transitional probabilities.

Finally, a set of experiments was performed to compare our results with those produced by LPSA [Wu 1996]. LPSA is a state assignment tool for low power realization of sequential machines developed at the VLSI CAD Laboratory at Tsing Hua University. LPSA assigns state codes that are close together (in terms of Hamming distance) to states that have high transitional probabilities. It produces results that are comparable to those of Hachtel et al. [1994]. Table X shows the results. The column labeled "ratio" is the power ratio of FSMD to LPSA. For this set of examples, FSMD outperformed LPSA in all cases except pma. On the average, machines synthesized by FSMD consumed 34% less power than those by LPSA.

7. CONCLUDING REMARKS

In this article we present a new approach to the synthesis problem for FSMs with the reduction of power dissipation as a design objective. An FSM is decomposed into a number of submachines. Most of the time, only one of the submachines will be activated which, consequently, could lead to

Table VIII. Comparisons of Two- and Four-Way Decompositions

circuit	2-way		4-way		ratio
	#lit	power	#lit	power	
planet	536	1,337.9	651	1,170.6	0.87
s1488	593	592.9	670	373.9	0.63
s1494	625	938.3	678	503.1	0.54
sand	488	634.7	565	818.0	1.29
styr	501	695.8	483	673.0	0.97

* power : μW

Table IX. Comparisons of Cofactor- and FSMD-Decompositions

circuit	cofactor-decomp M		FSMD M	
	#lit	power	#lit	power
bbsse	168	373.6	152	318.1
cse	224	284.7	245	257.7
dk16	293	1,063.5	251	868.7
ex1	402	842.4	257	409.8
keyb	240	513.2	246	492.4
planet	548	2,025.8	536	1,337.9
pma	243	828.8	237	622.2
s1488	683	1,205.6	593	592.9
s1494	678	887.5	625	938.3
s510	372	773.4	324	519.1
styr	504	808.0	501	695.8
tbk	453	920.6	293	744.8

substantial savings in power consumption. The key steps in our approach are: (1) decomposition of an FSM into submachines so that there is a high probability that state transitions will be confined to the smaller of the submachines most of the time, and (2) synthesis of the coupled submachines to optimize the logic circuits. Experimental results confirmed that our approach produced very good results (in particular, for FSMs with a large number of states).

We also have confirmed individually the effectiveness of our decomposition procedure and our synthesis procedure. Our partitioning algorithm presented in Section 3 produced results that are superior to those produced by the cofactor decomposition approach which does not take into account state transitional probabilities. Our synthesis procedure employs the concept of pseudo-outputs to decouple the submachines and has also been shown to be quite effective. We believe that the problem of synthesizing

Table X. Comparisons of LPSA and FSMD

circuit	LPSA		FSMD		ratio
	#lit	power	#lit	power	
bbsse	102	334.0	152	318.1	0.95
cse	165	341.7	245	257.7	0.75
dk16	258	1,206.7	251	868.7	0.72
ex1	227	586.3	257	409.8	0.70
keyb	190	527.7	246	492.4	0.93
planet	519	2,570.3	536	1,337.9	0.52
pma	182	589.6	237	622.2	1.05
s1	211	712.0	278	700.2	0.98
s1494	525	1,601.5	625	938.3	0.59
s510	296	1,022.9	324	519.1	0.50
sand	541	1,449.4	488	634.7	0.44
styr	386	995.6	501	695.8	0.70
tbk	281	951.8	293	744.8	0.78
total	3,883	12,889.5	4,433	8,539.6	0.66

coupled (sub)machines is an interesting research problem in its own right, and certainly deserves further investigation.

Besides reduction in power dissipation, realizing an FSM as a set of coupled submachines also provides the advantage of reducing the length of the critical path. In terms of the product of energy consumption and the critical path delay, our decomposition approach also produces good results.

REFERENCES

ALIDINA, M., MONTEIRO, J., DEVADAS, S., GHOSH, A., AND PAPAEFTHYMIOU, M. 1994. Precomputation-based sequential logic optimization for low power. In *Proceedings of ICCAD-94*, 74–81.

BENINI, L. AND DE MICHELI, G. 1995a. State assignment for low power dissipation. *IEEE J. Solid State Circuits 30*, 3 (March), 258–268.

BENINI, L. AND DE MICHELI, G. 1995b. Transformation and synthesis of FSMs for low-power gated-clock implementation. In *Proceedings of the International Symposium on Low Power Design*, 21–26.

CHANDRAKASAN, A. P., SHENG, S., AND BRODERSEN, R. W. 1992. Low-power CMOS digital design. *IEEE J. Solid-State Circuits 27*, 4 (April), 473–484.

DEVADAS, S., MA, H., AND NEWTON, R. 1991. MUSTANG: State assignment of finite state machines targeting multilevel logic implementations. *IEEE Trans. CAD* (Dec.), 1290–1300.

DRESIG, F., LANCHES, P., RETTIG, O., AND BAITINGER, U. G. 1993. Simulation and reduction of CMOS power dissipation at logic level. In *Proceedings of the EDAC'93 EURO-ASIC* (Feb.), 341–346.

HACHTEL, G., HERMIDA, M., PARDO, A., PONCINO, M., AND SOMENZI, F. 1994. Re-encoding sequential circuits to reduce power dissipation. In *Proceedings of ICCAD'94*, 70–73.

HAGEN, L. AND KAHNG, A. B. 1992. A new approach to effective circuit clustering. In *Proceedings of ICCAD* (Nov.), 422–427.

LIN, B. AND DE MAN, H. 1993. Low-power driven technology mapping under timing constraints. In *Proceedings of ICCD'93* (Oct.), 421–427.

LIN, B. AND NEWTON, A. R. 1989. Synthesis of multiple level logic from symbolic high-level description language. In *Proceedings of the IFIP International Conference on VLSI*, 187–196.

PAPOULIS, A. 1984. *Probability, Random Variables and Stochastic Processes*. McGraw Hill, New York.

PRASAD, S. C. AND ROY, K. 1993. Circuit activity driven multilevel logic optimization for low power reliable operation. In *Proceedings of the EDAC'93 EURO-ASIC* (Feb.), 368–372.

ROY, K. AND PRASAD, S. C. 1992. SYCLOP: Synthesis of CMOS logic for low power applications. In *Proceedings of the ICCD*, 464–467.

SENTOVICH, E., SINGH, K., MOON, C., SAVOJ, H., BRAYTON, R., AND SANGIOVANNI-VINCENTELLI, A. 1992. Sequential circuit design using synthesis and optimization. In *Proceedings of ICCD'92* (Oct.), 328–333.

TIWARI, V., ASHAR, P., AND MALIK, S. 1993. Technology mapping for low power. In *Proceedings of the 30th Design Automation Conference* (June), 74–79.

TSUI, C. Y., PEDRAM, M., AND DESPAIN, A. M. 1993. Technology decomposition and mapping targeting low power dissipation. In *Proceedings of the 30th Design Automation Conference* (June), 68–73.

WU, S.-S. 1996. State assignment for low power and high speed. M.S. thesis, Dept. of Computer Science, Tsing Hua University, Taiwan.

Received November 1995; revised June 1996; accepted July 1996