Automatic Synthesis of Low-Power Gated-Clock Finite-State Machines

Luca Benini, *Student Member, IEEE*, and Giovanni De Micheli, *Fellow, IEEE*

Abstract— **The automatic synthesis of low power finite-state machines (FSM's) with gated clocks relies on efficient algorithms for synthesis and optimization of dedicated clock-stopping circuitry. We describe a new transformation for incompletely specified Mealy-type machines that makes them suitable for gated-clock implementation with a limited increase in complexity. The transformation is probabilistic-driven, and identifies highly-probable idle conditions that will be exploited for the optimal synthesis of the logic block that controls the local clock of the FSM. We formulate and solve a new logic optimization problem, namely, the synthesis of a subfunction of a Boolean function that is minimal in size under a constraint on its probability to be true. We describe the relevance of this problem for the optimal synthesis of gated clocks. A prototype tool has been implemented and its performance, although influenced by the initial structure of the FSM, shows that sizable power reductions can be obtained using our technique.**

I. Introduction

THE MAJORITY of the currently published papers in the area of automatic synthesis for low power focus on the reduction of the level of activity in some portion of the circuit [5]–[8], since in CMOS technology the largest fraction of the power is dissipated during switching events.

In synchronous circuits, it is possible to selectively stop the clock in portions of the circuit where active computation is not being performed. Local clocks that are conditionally enabled are called *gated clocks,* because a signal from the environment is used to qualify (gate) the global clock signal. Gated clocks are commonly used by designers of large power-constrained systems [11], [16] as the core of dynamic power management schemes. Notice, however, that it is usually the responsibility of the designer to find the conditions that disable the clock.

Three different approaches to the automatic synthesis of logic circuits that can be conditionally disabled by environmental signals have been reported so far. In [1], Alidina *et al.,* have described a *precomputation-based approach* that focuses mainly on data-path circuits, while the authors have described a method to generate gated clocks for systems described as Moore-type finite-state machines (FSM's) [3]. The methods in [1] have been extended to deal with general combinational circuits: in [2], Tiwari *et al.,* showed that it is possible to selectively disable parts of a combinational logic network without being restricted to stop the computation only at latch boundaries. This extended precomputation strategy has been called *guarded evaluation.*

Our paper is based on the observation that during the operation of a FSM, there are conditions such that the next state and the output do not change. Therefore, clocking the FSM only wastes power in the combinational logic and in the registers. If we are able to detect when the machine is idle, we can stop the clock until a useful transition must be performed and the clocking must resume. The presence of a gated clock has a two-fold advantage. First, when the clock is stopped, no power is consumed in the FSM combinational logic, because its inputs remain constant. Second, no power is consumed in the sequential elements (flip-flops) and the gated clock line (differently from the scheme proposed in [1] where enabling signals are used).

Obviously, detecting idle conditions requires some computation to be performed by additional circuitry. This computation dissipates power and requires time. Sometimes, it will be too expensive to detect all idle conditions. Therefore, it is very important to select a subset of all idle conditions that are taken with high probability during the operation of the FSM. We have shown in [3] that idle conditions correspond to self-loops of Moore FSM's and, therefore, it is relatively easy to detect them. Idle conditions in Mealy FSM's can also be detected, but with more effort.

In [3], two problems were left open. First, our method was applicable only to Moore-type FSM's. Second, the synthesis of the clock-stopping circuitry was based on a simple and fast heuristic. In this paper, we remove the limitation to Moore-type FSM's, extending the applicability of our approach to the more general class of incompletely specified Mealy-type FSM's.

We then address the synthesis of the clock-stopping logic. More in detail, we formulate and solve a new logic synthesis problem, namely the choice of a minimum-complexity subfunction F_a of a given Boolean function f_a, such that its probability of being true is larger than a predefined fraction of the total probability of f_a. This is the main theoretical result of our paper, and it is applicable to a variety of dynamic power management schemes, such as precomputation or guarded evaluation. From a practical point of view, our algorithm chooses a subset of all idle conditions such that the clock-stopping circuitry dissipates minimum power, but stops the clock with high efficiency.

A prototype tool has been implemented and applied to a number of benchmark circuits. In the current implementation, our tool assumes an explicit state-based description of a FSM

Manuscript received March 27, 1996. This work was supported by NSF under Contract MIP-9421129. This paper was recommended by Guest Editors M. Pedram and M. Fujita.

The authors are with the Center for Integrated Systems, Stanford University, Stanford, CA 94305 USA.

Publisher Item Identifier S 0278-0070(96)04854-3.

(state table or equivalent formalism) as a starting point. This may be a limiting factor because state tables are not suitable for the description of very large sequential systems. Nevertheless, controllers for large data path circuits *are* often specified as state tables (or equivalent representations) and efficient state table extraction procedures exist within synthesis tools [20]. Detecting idle conditions on the controller may lead to detect (and exploit) idle conditions for the controlled data path, obtaining much larger power savings.

In order to verify our results, we embedded our tool in a complete synthesis path from state-table specification to transistor-level implementation and we employed accurate switch-level simulation [27], because gate-level power estimation has limited accuracy. Since the glock-gating logic may add its delay to the critical path, particular care must be taken in detecting and eliminating timing violations that may arise when the cycle time closely matches the critical path of the original FSM. For some circuits more than 100% improvement [computed as $100(P_{orig}/P_{gated}-1)$] in average power dissipation has been obtained, but quality of the results is strongly dependent on the type of FSM we start with. In particular, our method is well suited for FSM's that behave as *reactive systems:* they wait for some input event to occur and they produce a response, but for a large fraction of the total time they are idle. Practical examples of such machines can be found for instance in microprocessors [10] and real-time systems.

II. Background

In this paper, we will assume a single clock scheme with edge-triggered flip-flops, shown in Fig. 1(a). This is not a limiting assumption. We have indeed applied our methods to different clocking schemes in [3] (where we used transparent latches and multiphase clocks). The FSM model of Fig. 1(a) is different from the FSM structure commonly used in CAD literature [22] where the inputs are connected directly to the combinational logic. Although the FSM model without input flip-flops is useful for discussing the properties of a FSM in isolation, it is seldom used in the design practice. In a large system, control logic and data-path are always decomposed in interacting subunits, for obvious reasons of complexity management. The interface between subunits (interacting FSM's in our case) is usually composed by sequential elements [12] (in our case, D flip-flops). If such boundary does not exist, there is a combinational path between adjacent subunits. This is seldom allowed in industrial design methodologies (it makes timing analysis harder and increases the risk of timing violations). Even if we consider a design that is simple enough to be described by a single FSM, flip-flops are usually inserted on the inputs to obtain better signal quality and synchronization.

From a more theoretical point of view, the FSM with latched inputs differs from its counterpart because the outputs in our model lag the outputs of the model without flip-flops by one clock cycle. The input-output behavior of the two models is therefore equivalent modulo a translation in time of the output stream (assuming that the flip-flop on the inputs are reset at the same values assumed in the first clock cycle by the primary inputs of the machine without input flip-flops).

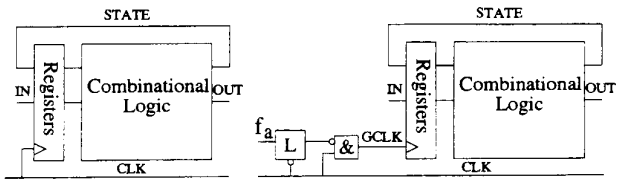


Fig. 1. (a) Single clock, flip-flop based FSM. (b) Gated-clock version.

A gated clock FSM is obtained modifying the structure in Fig. 1(a). We define a new signal called *activation function* (f_a) whose purpose is to selectively stop the local clock of the FSM, when the machine does not perform state or output transitions. When $f_a = 1$, the clock will be stopped. The modified structure is shown in Fig. 1(b). The block labeled "L" represents a latch, transparent when the global clock signal CLK is low. Notice that the presence of the latch is needed for a correct behavior, because f_a may have glitches that must not propagate to the AND gate when the global clock is high. Moreover, notice that the delay of the logic for the computation of f_a is on the critical path of the circuit, and its effect must be taken into account during timing verification.

The modified circuit operates as follows. We assume that the activation function f_a becomes valid before the raising edge of the global clock. At this time the clock signal is low and the latch L is transparent. If the f_a signal becomes high, the upcoming edge of the global clock will not filter through the AND gate and, therefore, the FSM will not be clocked and GCLK will remain low. Note that when the global clock is high, the latch is not transparent and the negated input of the AND gate cannot change at least up to the next falling edge of the global clock.

The activation function is a combinational logic block with inputs the primary input IN and the state lines STATE of the FSM. No external information is used, the only input data for our algorithm is the behavioral description of the FSM and the probability distribution of the input signals. In the following subsections, we will describe some basic concepts from automata and probability theory that will be useful for the understanding of our algorithms. Refer to [13], [22] for a more detailed treatment.

A. Models of Finite State Systems

A Mealy-type FSM can be described by a six-tuple $(X, Y, S, s_0, \delta, \lambda)$ where X is the set of inputs, Y is the set of outputs, S is the set of states, and s_0 is the initial (reset) state. The next state function δ is given by $s_{t+1} = \delta(X, s_t)$. The output function λ is defined as: $y_t = \lambda(X, s_t)$.

The definition of Moore-type FSM is similar, with the only exception of the output function. For a Moore FSM the output *does not* depend on the input. Therefore we define λ_M as $y_t = \lambda_M(s_t)$. Conceptually, Mealy and Moore machines are equivalent, in the sense that it is always possible to specify a Moore machine whose input-output behavior is equal to a given Mealy machine behavior, and vice versa [17].

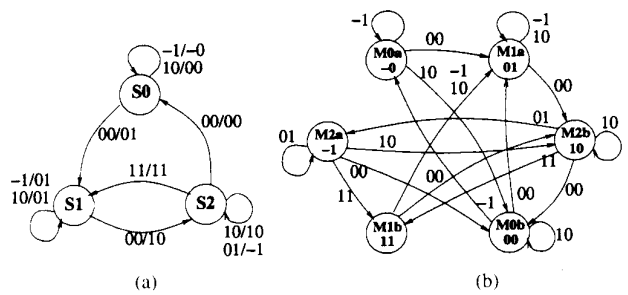


Fig. 2. (a) STG of a Mealy machine. (b) STG of the equivalent Moore machine.

However, there is an important difference. The Mealy model is usually more compact than the Moore model. Indeed the transformation from Mealy to Moore involves a state splitting procedure that may significantly increase the number of states and state transitions [17]. When *don't care* conditions (*DC*) are present, the FSM is called *incompletely specified,* λ and δ are *partial functions.*

Example 1: In Fig. 2(a), a Mealy machine is represented in form of state transition graph (STG). It is transformed into the equivalent Moore machine (using the procedure outlined in [17]) and the new STG is shown in Fig. 2(b). The STG of the Moore machine has the output associated with the states, while in the Mealy model, the outputs are associated with the edges. The higher complexity in terms of states and edges of the Moore representation is evident. Notice that both FSM's are incompletely specified, because of output *don't cares,* represented with "$-$" in the output fields.

B. Probabilistic Models of FSM's

We model the probabilistic behavior of a general FSM using a Markov chain [13] (as done in [8], [14], [24], [25]), whose structure is a weighted directed graph isomorphic to the STG of the machine. For a transition from state s_i to state s_j, the weight $p_{i,j}$ on the corresponding edge represents the *conditional probability* of the transition (i.e., the probability of a transition to state s_j *given* that the machine was in state s_i). Symbolically this can be expressed as

$$p_{i,j} = \text{Prob}\,(Next = s_j | Present = s_i). \tag{1}$$

The conditional probabilities $p_{i,j}$ are collected in a matrix $\mathbf{P}$ and depend on the probability distribution of the inputs, that is initially known.[1] However, using the conditional probability as an estimate of the total transition probability can lead to large errors, because the probability of a transition strongly depends on the probability for the machine to be in the state tail of the transition.

In order to find the probability of a transition without any condition, we need to know the *state probabilities* q_i that represent the probability for the machine to be in a given state s_i. Namely, the *total transition probabilities* we are looking for are

$$r_{i,j} = p_{i,j} q_i. \tag{2}$$

[1]The input probabilities of a FSM embedded in a digital system can be found by simulation.

Many methods have been proposed to calculate the state probabilities [13], [14]. In this paper, we have used the *Power Method.* Using this approach, the state probability vector $\mathbf{q} = [q_1, q_2, \cdots, q_{|S|}]^T$ can be computed using the iteration

$$\mathbf{q}_{n+1}^T = \mathbf{q}_n^T \mathbf{P} \tag{3}$$

with the normalization condition $\sum_{i=1}^{|S|} q_i = 1$ until convergence is reached. The convergence properties of this method are discussed in [15]. The power method has been chosen because of its simplicity and its applicability (if sparse matrix manipulation or symbolic formulation are used [14]) to FSM's with a very large number of states. In the following sections, we assume that the state probability vector and the total transition probabilities have already been computed using the power method and (2).

The knowledge of input and state probability distribution allows us to compute the probability of a Boolean function f whose support are the state and input variables of the machine in an exact fashion. Notice that this calculation is of vital importance in our algorithm, that performs a search based on the probability of the activation function.

III. Problem Formulation

Given the specification of the FSM and its probabilistic model, we first want to identify the idle conditions when the clock may be stopped. This is a simple task for Moore-type FSM's. For each state s_i, we identify all input conditions such that $\delta(x, s_i) = s_i$. We define for each state s_i, $i = 1, 2, \cdots, |S|$, a *self-loop function* $Self_{s_i} : X \rightarrow \{0, 1\}$ such that $Self_{s_i} = 1 \quad \forall x \in X$ where $\delta(x, s_i) = s_i$.

We then encode the machine. After the encoding step, every state s_i has a unique code e_i and $e_i = [e_{i,1}, e_{i,2}, \cdots, e_{i,|V|}]$, where V is the set of the state variables used in the encoding.

The *activation function* is defined as $f_a : X \times V \rightarrow \{0, 1\}$

$$f_a = \sum_{i=1,2,\cdots,|S|} Self_{s_i} \cdot e_i. \tag{4}$$

Example 2: For the Moore machine in Example 1, the self-loop function for state $M2a$ is $Self_{M2a} = in_0' in_1$ Similarly, all other self-loop state functions can be obtained. We encode the states using three state variables, v_1, v_2, v_3. The encodings are: $M0a \rightarrow v_1' v_2' v_3'$, $M1a \rightarrow v_1' v_2 v_3'$, $M2b \rightarrow v_1 v_2 v_3'$. $M0b \rightarrow v_1 v_2 v_3$, $M1b \rightarrow v_1' v_2 v_3$, $M2a \rightarrow v_1' v_2' v_3$. The activation function is therefore: $f_a = in_2 v_1' v_2' v_3' + in_2 v_1' v_2 v_3' + in_1 in_2' v_1' v_2 v_3' + in_1 in_2' v_1 v_2 v_3' + in_1 in_2' v_1 v_2 v_3 + in_1' in_2 v_1' v_2' v_3$.

If the machine is Mealy-type, the problem is substantially more complex. The knowledge of the state and the input is not sufficient to individuate the conditions when the clock can be stopped. If only the next state lines and the inputs are available for the computation of the activation function, we do not have a way to determine what was the output at the previous clock cycle. This is a direct consequence of the Mealy model: since the outputs are on the edges of the STG, we may have the same next state for many different outputs. The important consequence is that, even if we know that the state is not going to change, we cannot guarantee that the

output will remain constant as well, and therefore we cannot safely stop the clock.

Example 3: Consider the Mealy FSM in Example 1, and refer to Fig. 1 for the implementation. If we use only the lines IN and STATE as inputs for the calculation of the activation function, we may for example observe state $S2$ on the next state lines, and input 10. Observing the STG, we know that for this state and input configuration the state will not change. Unfortunately, we do not have any way to know what is the output value in the current clock cycle (it could be either 10 or -1). The Moore model does not have this problem, since we know the output when we know what the state is.

There are two ways to solve this problem. The simpler way is to use the outputs of the FSM as additional inputs to the activation function. The other approach is to transform the STG in such a way that the FSM will be functionally compatible with the original one, but only the input and state lines will be sufficient to compute the activation function.

We decided to investigate the second method for two main reasons. First, since for many FSM's, the number of output signals is large, it is likely that adding all output signals to the inputs of the activation function will produce poor results because of the high complexity of the activation function itself. Second, in the present implementation, our tool uses state transition tables as input, therefore we still have the freedom to modify the number of states and the STG structure (this is not the case if we start from a synchronous network that is an implementation of the STG).

The simplest transformation that enables us to use only input and state signals as inputs of the activation function f_a, is a Mealy to Moore transformation. The algorithm that performs this conversion is well known [17] and its implementation is simple, but it may sensibly increase the number of states and edges (correlated with the complexity of the FSM implementation).

A. Locally-Moore Machines

We now define and study a new kind of FSM transformation that enables us to use a Moore-like activation function without a large penalty in increased complexity of the FSM. We define a *Moore-state* as a state such that all incoming transitions have the same output. Formally, the subset of Moore-states of a Mealy machine is $\{s \in S | \forall x \in X, \forall r \in S, \delta(x, r) = s \Rightarrow \lambda(x, r) = const\}$. States that are not Moore-states will be called Mealy-states.

Proposition 1: A Mealy-state s with k different values of the output fields on the edges that have s as a destination can be transformed in k Moore-states. No other state splitting is required.

We could transform the FSM by simply applying the Mealy to Moore transformations locally to states that have self-loops. The local Moore transformation has the advantage that it allows us to concentrate only on states with self-loops, avoiding the useless state splitting on the states without self-loops. The potential disadvantage is an increase in the number of states related to the number of different outputs on incoming edges of Mealy states (Proposition 1). We devised a heuristic strategy to cope with this problem. We split Mealy states with self-loops into pairs of states, where one is Moore-type with a self-loop that has maximum probability.

Thus, for each state we define the *maximum probability self-loop function* $MPself_s : X \rightarrow \{0, 1\}$. Its ON-set represents the set of input conditions for a state that: i) are on self-loops; ii) produce compatible outputs (two outputs fields are compatible if they differ only in entries where at least one of the two is *don't care*); and iii) are taken with maximum probability.

The procedure that outputs $MPself_s$ is shown in Fig. 3. Its inputs are the state under consideration s, the self-loop function $Self_s$ (that includes all self-loops leaving state s), and the STG of the FSM. In the pseudocode, SLO is a partition of the self-loops. An element of SLO is composed of all self-loops from state s that have the *same* output (i.e., two self-loops with outputs differing only by *don't cares* will be in two different elements of SLO). The elements of SLO are mutually disjoint sets.

We then generate Q, a cover of the self-loops leaving state s. Initially Q is empty. In the first outermost iteration of the generation procedure, the first element of SLO becomes the first element of Q. Then, for each element of SLO, we check if it is compatible with any of the elements of Q. If this is the case, we incrementally modify the elements of Q. Otherwise, we create a new element.

The elements of Q are output-compatible, possibly overlapping sets of self-loops. Whenever we include an element of SLO in one of the elements in Q, we need to specify the *don't care* entries in the output field that are not always *don't cares* for all components of the set (this is done by procedure `merge_out_field` in the pseudocode). This step is needed to guarantee pairwise compatibility. Notice that an element of SLO can be included in more than one element of Q. Finally, the procedure `choose_max_prob_f` selects the output-compatible set of self-loops with maximum probability. The input conditions corresponding to this set form the ON-set of $MPself_s$. In general, function $MPself_s$ *does not* include all self-loop leaving state s. Consequently, $MPself_s \subseteq Self_s$, with equality holding when there is a single output-compatible class. Notice that the probability of the output-compatible classes can be compared using only the conditional input probability, because they are collection of self-loops leaving the same state.

Example 4: In the Mealy machine of Example 1, if we consider state $S2$, we have two self-loops: $in_1 in_2'$ with output 10 and $in_1' in_2$ with output -1. The two output fields are not compatible, therefore, we have two compatible classes (the same two functions). We will choose the class that is more probable. In this particular example, we assumed equiprobable and independent inputs and both functions have the same probability, therefore, one of the two is randomly chosen. Assume now that Prob $(in_1 = 1) = 0.7$ and prob Prob $(in_2 = 1) = 0.5$. The probability of the first class is $p_1 = 0.7 * 0.5 = 0.35$, while the probability of the second class is $p_2 = 0.3 * 0.5 = 0.15$. In this case, the first class will be chosen. Observe that the probability of state $S2$ does not come into play, because state $S2$ is the tail of both self-

```
find_Mpself(s, Self_s, STG) {
  SLO = self_loop_const_out(s, Self_s, STG) ;      /*Partition in classes with same output*/
  Q = ∅;
  foreach ( slo ∈ SLO ) {                          /*Iterate on all self-loops classes in SLO*/
    compatible = 0;
    foreach ( q ∈ Q ) {                            /*Iterate on all compatible classes of self-loops*/
      if ( is_compatible( slo, q ) ) {             /*increase compatible class*/
        q = q ∪ slo;
        merge_out_field( q );
        compatible = 1;
        break;
      }
    }
    if ( !compatible ) Q = Q ∪ {slo};              /*generate new compatible class*/
  }
  return( choose_max_prob_f( Q ) );                /*Choose the max. probability compatible class*/
}
```

Fig. 3. Algorithm for the computation of max probability self-loop function $MPself_s$.

loops, and the two total transition probabilities differ from the conditional probabilities only by the same scaling factor (i.e., the probability of $S2$).

Once the $MPself_s$ functions have been found for all states with self-loops, the second step of our transformation algorithms is performed. A Mealy-state s with at least one self-loop is split in two states s_a and s_b. State s_b has the same incoming and outgoing edges as the original one, with just one important difference: the edges corresponding to the self-loops represented by $MPself_s$ become transitions from s_a to s_b.

The second state s_b is reached only from s_a and has a self-loop corresponding to $MPself_s$. All the outgoing edges that leave s_a are replicated for s_b, keeping the same destination. State s_b is now Moore-type, because by construction, all edges that have s_b as head have the same output.

This procedure is advantageous for many reasons. First, the increase in the number of states is tightly controlled. In the worst case, if all states are Mealy-type and have self-loops, we can have a twofold increase in the number of states. Second, the self-loops with maximum probability are selected. Third, if we really want to limit the increase in the number of states, we may define a threshold: only the first k states in a list ordered for decreasing *total probability* of $MPself_s$ are duplicated.

We call the FSM obtained after the application of this procedure *locally-Moore* FSM, because in general only a subset of the states is Moore-type.

Example 5: The transformation of the Mealy machine of Example 1 produces the locally-Moore FSM shown in Fig. 4. The shaded areas enclose states that have been split. The Moore-states with self-loops are drawn with bold lines. The number of states and edges of the locally Moore machine is smaller than those that we obtained with the complete Mealy to Moore transformation (state $S0$ has not been split).

The inputs are in_1, and in_2. Assume that we use three state variables for the encoding: v_1, v_2, and v_3. The state codes are

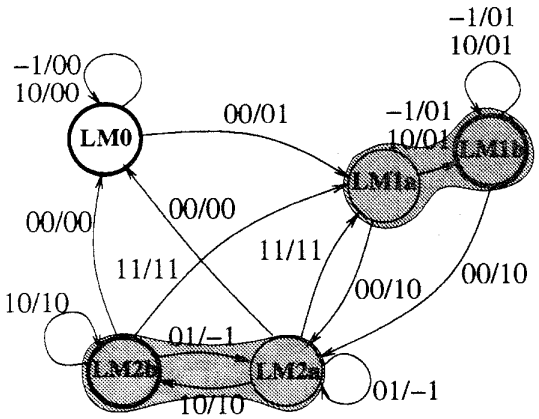


Fig. 4. STG of the locally-Moore FSM.

$LM0a \rightarrow v_1'v_2'v_3'$, $LM1a \rightarrow v_1'v_2'v_3$, $LM2a \rightarrow v_1'v_2v_3'$, $LM1b \rightarrow v_1'v_2v_3$, and $LM2b \rightarrow v_1v_2'v_3'$. The activation function includes all self-loops leaving Moore-states: $f_a = in_2v_1'v_2'v_3' + in_1in_2'v_1'v_2'v_3' + in_2v_1'v_2v_3 + in_1in_2'v_1'v_2v_3 + in_1in_2'v_1v_2'v_3'$.

Once the activation function has been found, we still need to solve the problem of synthesizing the clock-stopping logic in an optimal way. This problem will be addressed in the next section.

IV. Optimal Activation Function

The simplest approach is to try to use the complete f_a as activation function. This is seldom the best solution, because the size of the implementation of f_a can be too large, and the corresponding power dissipation may reduce or nullify the power reduction that we obtain by stopping the clock. Roughly speaking, it is necessary to be able to choose a function contained in f_a whose implementation dissipates minimum

power, but whose efficiency in stopping the clock is maximum. We call such function $F_a \subseteq f_a$ a *subfunction*[2] of f_a.

In [3], we proposed a simple greedy algorithm that will be shortly outlined. First, a minimum cover of f_a is obtained by a two-level minimizer. Then, the largest cubes in the cover are greedily selected until the number of literals in the partial cover exceeds a user-specified literal threshold. The rationale of this approach is that generally large cubes have high probability and the primes that compose a minimum cover are as large as possible.

There are several weak points in this approach.

- There is no guarantee that choosing the largest cubes in a cover will maximize the probability of the cover, because, in general, the probability of a cube depends on the input and state probability distribution. Even if we assume uniform input probability distribution, the state probability distribution is in general not uniform.
- The function sought may not be found by looking only at a subset of cubes of the minimum cover of the original activation function. The number of possible subfunctions of f_a is much larger than the functions that we can generate using subsets of the cubes of the minimum cover.
- Even if we restrict our attention to the list of cubes in the minimum cover of f_a assuming uniform distribution for input and states, the cubes of the cover are in general overlapping (the minimum cover is not guaranteed to be disjoint). Finding the minimum-literal, maximum-probability subset of cubes becomes a set covering problem that certainly is not solved exactly by a greedy algorithm.
- The relation between the number of literals in a two-level cover of the activation function and the power dissipation of a multilevel implementation is not guaranteed to be monotonic.

In the next section, we will propose a new algorithm that overcomes the first three limitations listed above. As for the last issue, we will assume that there is correlation between the number of literals of a two-level cover and the power dissipated in the final implementation, as suggested by experimental results presented in [4]. We now formulate the problem that we want to solve in a more rigorous way.

Problem 1: Given the activation function f_a, find $F_a \subseteq f_a$ such that its probability $P(F_a)$ is $P(F_a) \geq \text{MinProb} = \alpha P(f_a)$, (with $0 \leq \alpha \leq 1$) and the number of literals in a two level implementation of F_a is minimum.

We call this problem *constrained-probability minimum literal-count covering* (CPML). Notice that we could as well formulate the dual problem, constrained literal count cover with maximum probability. The two problems can be solved using the same strategy, and are equivalent for our purposes. With the assumption of a good correlation between number of literals and power dissipation, we propose an exact solution to CPML and, by consequence to the problem of finding the best reduced activation function given a complete f_a to start with.

[2]Here, we exploit the isomorphism between set theory and Boolean algebra: Boolean function are seen as sets of minterms. A *subfunction* of f_a is therefore a function whose ON-set is contained in the ON-set of f_a.

A. Finding a Minimum Power Implementation

Apparently, the first source of difficulty comes from the fact that we are not constrained to completely cover f_a, therefore, the "algorithmic machinery" developed in the area of two-level minimization seems not useful. We first show that this is not true. Consider the set of primes of f_a, called $Primes(f_a)$. Consider the set Sub_{f_a} of all possible subfunctions of f_a. The set of primes of a generic subfunction $F_a \in Sub_{f_a}$ is called $Primes(F_a)$. We state the following theorem.

Theorem 1: For every prime $p \in Primes(F_a)$, only two alternatives are possible.

- $p \in Primes(f_a)$.
- p is contained in at least one element q of $Primes(f_a)$ (consequently its literal count is larger than the literal count of q).

Proof: Assume that $p \in Primes(F_a)$, $(F_a \subseteq f_a)$. Two alternatives are possible: i) $p \in Primes(f_a)$ and ii) $p \notin Primes(f_a)$. We will prove by contradiction that, if ii) is true, there is always at least a prime $q \in Primes(f_a)$ such that $p \cdot q = p$ (in other words, p is contained in at least a prime of f_a). Assume that the assert is not true, therefore, p is not contained in any prime of f_a. Notice that p is an implicant of F_a, therefore, it is an implicant of f_a because $F_a \subseteq f_a$. By consequence, p is an implicant of f_a not contained in any prime of f_a. Therefore p is a prime of f_a by definition. This is not possible, because we assumed ii) to start with. □

The important consequence of this theorem is that we do not need to generate all possible subfunctions of f_a. We can restrict our search to subfunctions that are formed by subsets of $Primes(f_a)$ if we want to find a minimum literal subfunction. Functions that belong to this class have all primes in the first category of Theorem 1.

Now that we have defined our search space $[Primes(f_a)]$, we must find a search strategy that guarantees an optimum solution. The choice of a subset of $Primes(f_a)$ with minimum literal count satisfying the probability constraint cannot be done using a greedy strategy, because the primes are generally overlapping and a choice done in one step affects the following choices. An example will help to clarify this statement.

Example 6: Suppose that f_a is a function of four variables a, b, c, d. The set of primes is $Primes(f_a) = \{a'b', a'c', bc', ab\}$. Assume for simplicity that all minterms are equiprobable (Prob $= 1/16$) and our probability constraint MinProb (the minimum allowed probability of the subfunction) is MinProb $= 1/2$. All primes in this example are equiprobable (they have the same size). If we choose $a'b'$ first, the following choices are biased. Since $a'c'$ is partially covered by $a'b'$, it will not be the right next choice because we want to cover the largest number of minterms (remember that we are assuming equiprobable minterms). Consequently, either bc' or ab must be chosen.

CPML complexity is at least the same as two-level logic minimization, because CPML becomes two-level logic minimization for the particular case $\alpha = 1$. We describe here a branch-and-bound algorithm that has been shown to work efficiently on the benchmarks, even if its worst case behavior is exponential. Furthermore, the branch-and-bound can be

```
FindFa (f_a, α)
{
  PrimeList = Generate_primes(f_a);
  MinProb = αP(f_a);
  CurBest = FindFa_PH1(PrimeList, MinProb);                    /* Phase 1 */
  CurPartial = ∅;
  FindFa_PH2 (PrimeList, CurBest, CurPartial, MinProb);        /* Phase 2 */
  return(CurBest);
}
```

Fig. 5. Two-phases algorithm for the exact solution of CPML.

modified to provide heuristic minimal solutions when the exact minimum is not attainable in the allowable computation time.

B. Branch-and-Bound Solution

Our algorithm operates in two phases. In the first phase, a heuristic solution is found in polynomial time (in the number of primes N_P). The second phase finds the global minimum cost solution using a branch-and-bound approach. The pseudocode of the algorithm is shown in Fig. 5.

We exploit the similarity of this problem with *knapsack* [18]. We need to find the set of *items* (primes) with total *size* (probability) larger than or equal to the *knapsack capacity* (MinProb) minimizing the total *value* (number of literals). This formulation differs from knapsack in two important details. First, knapsack targets the maximization of value given a constraint of the maximum allowed size (we face the opposite situation). Second and most importantly, in knapsack the *size* of an item is a constant, while in our case, the probability of a prime varies when other primes are selected. To clarify this statement, observe that primes may overlap and they contribute to the total probability of the reduced activation function only with minterms that are not covered by other already selected primes (Example 6). As a consequence, CPML is not solved in *pseudopolynomial* time [18] by dynamic programming. Notice however that CPML reduces to knapsack if all primes are disjoint (by complementing values and sizes).

In the first phase of our algorithm, we employ a greedy procedure that is reminiscent of an approximation algorithm for the solution of knapsack [19]. The solution obtained is heuristic and it is employed as a starting point for the second phase of the algorithm, that provides an exact solution.

The pseudocode of first phase of the algorithm is shown in Fig. 6. Let us call P_{ME} the total probability of minterms in p which are not covered by already chosen primes. We define *value density* $\mathcal{D}$ for a prime p the ratio $P_{ME}(p)/N_{lits}(p)$. We greedily select the primes with largest $\mathcal{D}$ until the constraint on MinProb is satisfied. The selection is done by function `sel_prime_max` $P_{ME}N_{lits}$ `ratio` in the pseudocode of Fig. 6. We then check if there is a single prime whose probability satisfies the MinProb constraint and whose literal count is smaller than the total literal count of the greedily selected primes. If this is the case, the list is discarded and the single prime is selected.

The single prime solution is tested for two reasons. First, the same test is performed in the greedy algorithm for the heuristic solution of knapsack [19]. Second and most importantly, it corresponds to a particular case that can be encountered in practice. Some machines have a *halt* state and a *halt* input value. If the machine is in *halt* and the inputs are fixed at the *halt* value, no output and state transitions are allowed. The cube of the activation function corresponding to the *halt* state and inputs may have a small value of $\mathcal{D}$, because even if its probability is high, so is the number of specified literals in the cube. When synthesizing the activation function, we want to early detect the *halt* condition, that is indeed the most natural candidate for clock-gating. The single cube test helps in detecting such condition as soon as possible, before the time consuming exact search is started.

When the first phase of the algorithm terminates, it returns a feasible solution that is used as starting point for the branch-and-bound algorithm employed in the second phase. At the beginning of the second phase, we order the primes for decreasing ratio $P(p)/N_{lits}(p)$. The probability $P(p)$ of a prime is computed multiplying the conditional probability of the input part by the probability of the state part (remember that the probability of a transition is computed by multiplying the conditional input probability by the state probability), and it is computed once for all (contrasts with phase 1 of the algorithm, where P_{ME} that is recomputed whenever a new prime is chosen).

At the starting point of the branch-and-bound we have a *search list* corresponding to all primes. Moreover, we have a *current best* solution generated by the first phase. The *current partial* solution is initially empty. Each time the recursive procedure is invoked, all primes in the prime list are considered one at a time. If the prime being considered (together with the current-partial solution) yields more literals than the best solution seen so far, the prime is discarded and another prime is considered. Otherwise, a solution feasibility check is done (i.e., if the probability is larger than, or equal to MinProb). If this is the case, the current best solution is updated. Otherwise, a new recursive search is started, where the prime just being considered is kept as part of the current partial solution, but the primes considered at the previous level of the recursion are discarded. The pseudocode of the algorithm is shown in Fig. 7.

Notice that the backtracking involved in the branch and bound is implicitly obtained in the pseudocode. For each

```
FindFa_PH1(PrimeList, MinProb, CurBest)
{
  Selected = ∅; Pr = 0; ;
  while (Pr < MinProb ) {                                  /* Greedy selection of primes */
    NewPrime = sel_prime_max_{P_ME N_lits}ratio(PrimeList);
    Pr = Pr + Prob(NewPrime);
    Selected = Selected ∪ NewPrime;
    Compute P_ME(NewPrime, PrimeList);          /* Recompute P_ME for unselected primes */
  }
  MaxPrPrime = maxProb_prime(PrimeList);
  Pr1 = Prob(MaxPrPrime);
  if ( Pr1 ≥ MinProb && N_lits(MaxPrPrime) < N_lits(Selected) ) {     /*Single-prime solution*/
    Pr = Pr1;
    Selected = MaxPrPrime;
    N_lits = N_lits(MaxPrPrime);
  }
  return(Selected);
}
```

Fig. 6. First phase of CPML solution.

```
FindFa_PH2 (PrimeList, CurBest, CurPartial, MinProb)
{
  if(Bound(PrimeList, CurBest, CurPartial, MinProb)) return;          /* Bounding step */
  DoneList = ∅;
  foreach (Prime ∈ PrimeList) {                                      /* Branching step */
    DoneList = DoneList ∪ Prime;
    NewPartial = CurPartial ∪ Prime;
    if ( N_lits(NewPartial) < N_lits(CurBest) ) {
      if ( Prob(NewPartial) ≥ MinProb )
        CurBest = NewPartial;                                        /* New Best solution */
      else
        FindFa_PH2 (PrimeList − DoneList, CurBest, NewPartial, MinProb);   /* Recursion */
    }
  }
}
```

Fig. 7. Second phase of CPML solution.

iteration of the inner loop, we generate a new partial solution adding to the original partial solution a single prime from the search list. In this way, each new iteration backtracks on the choice of the prime in the previous iteration. The algorithm terminates when all choices in the search list of the upper level of the recursion have been tried.

The bound is based on the approximation algorithm for the solution of knapsack mentioned above. The greedy procedure guarantees a solution to knapsack within a factor of two from the optimum [19]. The optimum knapsack solution itself is an upper bound to the solution of our problem (it becomes the exact solution if all primes are disjoint). Intuitively, the bound eliminates the partial solutions that could not improve the current best solution even if all primes in the search list were mutually disjoint and not overlapping with primes in the current partial solution.

The bounding procedure (Bound) is shown in Fig. 8. It works on the search list. If the search list (Primelist) is empty, obviously the return value is one. If the current partial solution (CurPartial) is empty, the return value is zero. In the general case, we select primes from the top of the search list until the sum of their literal count becomes larger than $N_{lits}(\texttt{CurBest}) - N_{lits}(\texttt{CurPartial})$. We compute the sum of the probabilities of all selected primes (excluding the last selected one) and call it P_{tot}. We choose the maximum $P_{\max}$ between P_{tot} and P_{one}, where P_{one} is the largest probability

value of a single prime in the search list whose literal count is less than $N_{lits}(\texttt{CurBest}) - N_{lits}(\texttt{CurPartial})$.

Once $P_{\max}$ has been obtained, we can prune the partial solution if the following inequality is verified

$$P_{\max} < \frac{\text{MinProb} - \texttt{Prob}\,(\texttt{CurPartial})}{2}. \tag{5}$$

The rationale behind this bound requires further explanation. In the Bound procedure, we are trying to discover if a selection of primes from PrimeList can increase the probability of the current partial solution by $\Delta P >$ MinProb − Prob (CurPartial) (an amount large enough to satisfy the bound on probability), without increasing the number of literals by more than DeltaLits (the difference between the number of literals in the best solution so far and the number of literals in the current partial solution).

If such a selection exists, its probability P_{exact} is $P_{exact} \leq P_{knap}$, where we obtain P_{knap} by assuming that all primes are disjoint (remember that primes can be overlapping, hence the inequality). An upper bound $P_{\max}$ for P_{knap} is obtained by the greedy algorithm described above, because finding P_{knap} requires the solution of a 0–1$knapsack$ problem and the greedy algorithm provides an approximate solution $P_{\max} \geq P_{knap}/2$ [19]. Thus, we have established the following chain of inequalities

$$P_{exact} \leq P_{knap} \leq 2P_{\max}. \tag{6}$$

If $2P_{\max} < \Delta P$, the same will hold for P_{exact}, thus proving the correctness of the bound.

Example 7: Assume that we have a current best solution that satisfies the constraint on the probability (MinProb = 0.4) with a cost of $Nlits = 40$. Assume that the current partial solution has cost $Nlits' = 36$ and probability Prob (CurPartial) = 0.35. Suppose that the first two element of the unselected prime list are $a'b'$ with probability 0.01 and $a'c'$ with probability 0.012. The maximum probability prime with at most four literals has probability 0.015. $P_{\max}$ is therefore $P_{\max} = \max\{0.015, 0.012 + 0.01\} = 0.022$. This branch of the search tree is pruned, because $P_{\max} <$ [MinProb − Prob (CurPartial)]/2 = 0.025. Notice that the two primes are partially overlapping, therefore the actual increase in probability for the current solution if we select the two primes would be smaller than the estimated one.

The bound can be made even tighter if after selecting a new prime in a partial solution, the probabilities of the remaining primes are reduced accordingly to the overlap with the chosen prime. Notice that the computation of this second bound requires the recalculation of all probabilities of the currently unselected primes (and the reordering of the search list). As a consequence, the second bound should be computed only after the first has been unsuccessful in pruning the search tree.

We want to point out that there are two possible sources of complexity explosion in our algorithm. First, the number of primes for a Boolean function is worst case exponential in the number of the function inputs. Second the branch-and-bound algorithm has a worst case exponential complexity in the number of primes that form the candidate list.

The double source of exponential behavior may seem worrisome. Nevertheless, the structure of our algorithm is flexible enough to generate fast heuristic solutions if the execution time exceeds some user-defined limit. The problem of the large number of primes can be avoided if we apply the algorithm to a reduced set of primes. The most natural candidate is obviously a prime and irredundant cover of the function, obtainable using two-level minimizers that can provide optimum or near-optimum covers for single-output functions with a large number of inputs [9].

If either the branch-and-bound is interrupted or a reduced set of primes is used, the exact minimality of the last solution found is not guaranteed, but we will have, in general, a good quality heuristic solution. Notice that the first phase of our algorithm finds a feasible solution in polynomial time, and we could even completely skip the branch-and-bound if we consider it too expensive.

Finally, an efficient implementation of the algorithm can be achieved using symbolic BDD-based techniques. Many parts of the current implementation already use symbolic techniques (for example, the prime generation is fully symbolic [23], and the cube probability calculation is also done in a symbolic fashion), but still the prime list is manipulated by the branch and bound algorithm in an explicit way.

C. The Overall Procedure

We can now briefly outline the full procedure used for the synthesis of our low-power gated clock FSM's. Our starting point is a FSM specified with a transition table or a compatible format. The synthesis flow is the following.

- The Mealy machine is transformed to an equivalent locally-Moore machine.
- The complete activation function f_a is extracted from the Moore-states of the locally-Moore machine.
- The probability of the complete f_a is computed.
- The prime set $Primes(f_a)$ is generated.
- The branch-and-bound algorithm finds the minimum literal count solution F_a whose probability is a prespecified fraction α of the probability of f_a.
- F_a is used as additional DC set for optimizing the combinational logic of the FSM.

The last step can sensibly improve the quality of the results, in particular if F_a is large [3]. Unfortunately, it is hard to foresee the effects of F_a used as DC set. Sometimes, it may be convenient to choose a F_a that is not minimal in the sense discussed above, if it allows a large simplification in the combinational part of the FSM. Our heuristic approach is to try different F_a that range from the complete f_a to a much smaller subfunction, in an attempt to explore the trade-off curve.

This iterative search strategy raises the problems of choosing appropriate values of the parameter $\alpha \leq 1$ used to scale down the probability of f_a when the reduced activation functions are generated. The approach that we adopted is to generate a set of reduced activation functions $Cand_F$ using different values of α, in such a way that the possible range of solutions is uniformly sampled. We have devised a heuristic procedure that generates suitable α values and we briefly

```
Bound(PrimeList, CurBest, CurPartial, MinProb);
{
  if ( PrimeList == ∅ ) return(1);
  if ( CurPartial == ∅ ) return(0);
  Selected = ∅; N_lits = 0; Pr = 0; exit = 0;
  DeltaLits = N_lits(CurBest) − N_lits(CurPartial);
  while ( !exit ) {                                   /* Greedy selection of primes */
    NewPrime = sel_max_PNratio(PrimeList);
    if ( N_lits + N_lits(NewPrime) > DeltaLits ) exit = 1;
    else {
      Pr = Pr + Prob(NewPrime);
      Selected = Selected ∪ NewPrime;
      N_lits = N_lits + N_lits(NewPrime);
    }
  }
  MaxPrPrime = maxProb_prime(PrimeList);
  Pr1 = Prob(MaxPrPrime);
  if ( Pr1 ≥ Pr && N_lits(MaxPrPrime) < DeltaLits ) {      /*Single-prime solution*/
    Pr = Pr1;
    Selected = MaxPrPrime;
    N_lits = N_lits(MaxPrPrime);
  }
  if ( Pr < .5 (MinProb − Prob(CurPartial)) ) return(1);          /*Bound test*/
  else return(0);
}
```

Fig. 8. Bounding function.

outline it

$$\alpha_i = \frac{i}{N_{cand}} \quad i = 1, 2, \cdots, N_{cand}. \tag{7}$$

It may be the case that for two or more consecutive values α_i, the algorithms generates the same solution. This happens, for example, when eliminating even a single prime from a solution generated for $i+1$ causes a decrease in the $P(F_a)$ in the solution generated for i larger than $P(f_a)/N_{cand}$. In this case, our algorithm adaptively select new values of α such that the new candidates will have a probability between those of solutions generated with two consecutive values of α that have maximum literal cost difference.

Obviously, if large N_{cand} are used, the computational time required to generated $Cand_F$ increases. Notice, however, that only the last two steps in the overall procedure describe before need to be iterated, and usually a small number of different values of α is sufficient to find a satisfying solution.

One more point is worth noting. Although our procedure for the synthesis of a constrained probability minimum literal cover of F_a is exact, the overall synthesis path is heuristic. As a consequence, finding an exact solution to CPML may not be essential, because the approximation introduced may cause large errors that we do not control. Nevertheless, the strength of our approach lies in its flexibility; our algorithm offers the possibility of exploring the search space with a fine granularity, and it can find heuristic solutions at a very low computational cost.

V. Implementation and Experimental Results

We implemented the algorithms as a part of our tool-set for low-power synthesis. The tool reads the state transition table of the FSM. The first step is the transformation of the Mealy machine to a locally-Moore machine and the extraction of the self-loops from the Moore-states.

An input probability distribution must be specified by the user (we assumed uniform input probability distribution, but this assumption is not restrictive). Moreover, we assumed that every input line has a maximum of one transition per clock cycle. This is an optimistic assumption, because multiple input transitions (high transition density) may increase the power dissipated in the activation function logic (but not in the FSM logic because the input are guarded by flip-flops). If input with high transition activity are present, smaller activation function should be allowed.

The power method is applied to compute the exact state probabilities given a conditional input probability distribution. Notice that this step can be modified to use the exact and approximate methods described in [15], [24], [25] that have been demonstrated to run on very large sequential circuits. Presently, our procedure employs sparse matrix techniques and

it has been able to process all MCNC benchmarks provided in state transition table format in a small time (less than 10 s on a DECstation 5000/240 for the largest example s298).

We then state assign both the original machine and the locally-Moore FSM using JEDI [26]. Once the state codes have been assigned, our probabilistic-driven procedure for the selection of the activation function can start. First, all primes of the activation function are generated using symbolic methods [23], then the probability of the minimized cover (obtained with ESPRESSO [20]) of the complete activation function f_a is computed. The number of literals of the complete minimized cover is used as initial literal cost limit in the branch-and-bound algorithm.

The user specifies the number of activation functions that the procedure should generate, and the branch-and-bound algorithms solves the CPML as many times as it is requested. Surprisingly, for all MCNC benchmarks this step has never been the bottleneck, the CPU time being in the order of 30 s maximum. This is certainly due to the fact that the majority of the FSM MCNC benchmarks do not have a large number of self-loops (in particular the larger ones). Nevertheless, even if difficult cases are found, our algorithm stops the search when a user specified CPU time limit has been reached. The solution becomes then suboptimal, but there are other sources of inexactness in the overall procedure. Therefore, the search for an exact optimum solution of CPML is not of primary practical importance.

The combinational logic of the locally-Moore FSM is then optimized in SIS [20] using the additional *DC* set given by the activation function. This step is repeated for all activation functions generated in the preceding step, and alternative solutions are generated. The *DC*-based minimization of the combinational logic using the activation functions is the main bottleneck of our procedure. In our tool, the user has the possibility to specify a CPU-time limit for each minimization attempt. This, of course, limits the possible improvements obtainable on large FSM's.

The activation functions are also optimized using SIS, then the alternative solutions are mapped with CERES [21], and the gated clocking circuitry is generated. Again the same optimization and library binding programs are used for both the original Mealy machine and the locally-Moore gated clock machines. We employed a simple target library which includes two, three, and four input gates. Our flip-flops have a master-slave structure, and their cost (in terms of area and input load capacitance) is approximatively equivalent to two three-input logic gates.

Finally, the alternative gated clock implementations and the implementation of the original Mealy FSM are simulated with a large number of test patterns using a switch level simulator (IRSIM [27]) modified for power estimation.

The quality of the results strongly depends on two factors. First, how much state splitting has been needed to transform the machine to a locally-Moore one. Second, for what percentage of the total operation time the FSM is in a self-loop condition (this depends on the FSM structure and on the input probability distribution). For machines with a very small number of self-loops or a very low-probability complete activation function, the chance of improvement is limited or null. This is the case for many MCNC benchmarks for which the final improvement is negligible. As for the first problem, it may be worth to investigate if, in case the state duplication is too high, using an activation function with the outputs of the FSM as additional inputs may lead to better results.

TABLE I
RESULTS OF OUR PROCEDURE APPLIED TO MCNC BENCHMARKS. SIZE IS NUMBER OF TRANSISTORS P (POWER) IS IN μW

	Original		Locally-M.		Gated				
Circuit	Size	P	Size	P	Size	P	%	F_a Size	α
bbara	330	67	422	72	408	34	97	74	1
bbsse	640	121	742	137	736	119	2	140	1
bbtas	142	56	138	57	164	44	27	34	.93
keyb	721	128	754	132	820	114	12	62	.91
lion9	188	60	226	60	248	52	15	8	.25
s298	7492	899	7496	900	7502	810	11	14	1
s420	544	132	544	132	602	108	22	44	.75
scf	3222	437	3222	437	3169	400	9	26	1
styr	1474	159	2468	230	2534	208	0	560	.75
test	348	73	442	76	374	32	128	64	.88

Example 8: The Mealy machine of Example 1 has been synthesized without any gated clock. The number of states is three, the mapped implementation has 124 transistors and a total nodal capacitance of 2.32 pF. The average power dissipation is 52 μW.

Using our algorithm, the minimum power implementation (obtained with the complete activation function in this case) of the equivalent locally-Moore gated clock machine has 178 transistors and a total nodal capacitance of 3.14 pF. The average power dissipation is 42 μW. Notice that the efficacy of the activation function in stopping the clock allows substantial power savings (24%) even if the total capacitance is larger (35%). This is due to the fact that the locally-Moore machine has five states, and its combinational logic is more complex. In contrast, with a complete Moore transformation the minimum power implementation has 196 transistors and total nodal capacitance of 3.39 pF. Its power dissipation is 48 μW.

Table I reports the performance of our tools on a subset of the MCNC benchmarks. The first six columns show the area (number of transistors) and the power dissipation of the normal Mealy FSM, the locally-Moore FSM without gated clock, and the locally-Moore machine with gated clock. The last three columns show the power improvement [computed as $100(P_{mealy}/P_{gated} - 1)$], the size (in transistors) of the activation function and the α factor used in the solution of CPML leading to the best result. Notice that, if there is no power improvement the improvement is set to zero.

The tool is able to process all benchmarks, but in the table, we list examples representative of various classes of possible results. The benchmarks `bbara` and `test` are reactive FSM's. The high number and probability of the self-loops allow an impressive reduction of the total power dissipation, even if the area penalty can be not negligible. For this class of FSM's, our tool gives its best results.

In contrast, for bbsse and styr there is no power reduction or even a power increase. The bbsse benchmark is representative of a class of machines where the number and probability of the self-loops is too small for our procedure to obtain substantial power savings. The styr benchmark has many self-loops, but they all have low probability. Moreover, the transformation to locally-Moore machine has a too large area overhead in this case, therefore, even if there are power savings with respect to the locally-Moore implementation without clock, the smaller Mealy implementation has the lowest power consumption.

For all other examples in the table the power savings vary between 10% and 30%. For some of these machines (s420 and scf), there is no area overhead for the locally-Moore transformation. This happens when all states with self-loops are already Moore states in the original FSM. We included some of the larger examples in the benchmark suite (s298 and scf) to show the applicability of our method to large FSM's.

From the analysis of the results, it is quite clear that several complex trade-offs are involved. First, the transformation to locally-Moore machine can sometimes be very expensive in terms of area overhead. Second, the choice of the best possible activation function is paramount for good results. In fact, for many examples, the complete activation function was too large, and reduced activation functions gave better results. Notice, however, that for some examples, the efficiency of the activation function in stopping the clock was such that the power was sensibly reduced even with large area overhead.

Having discussed *how much* power is saved, we address now the problem of *where* the power is saved. In our approach, the FSM clock is freezed only when the next state variables and the outputs are not going to change in the upcoming clock cycle. It may be possible to think that power is saved only in the flip-flops and the clock line. This intuitive observation is deceiving, because power is also saved in the combinational logic, as it is shown in Table II. We have compared in the table the power dissipation of the locally-Moore implementation with and without gated clock. We compare to the locally-Moore FSM because its STG is isomorphic to those of the gated-clock FSM. Hence, all modifications are due only to the insertion of the activation function. The comparison with the Mealy machine is less explicative because the locally-Moore transformation modifies the STG and consequently the next state and output function, making impossible to distinguish how the clock-gating circuitry alone affects the power dissipation.

In the first two columns of the table, for the two implementations, we compare the ratio of the power dissipated in the combinational logic (flip-flops outputs, all nodes in the FSM logic and outputs) and the power dissipated in the clock-related circuitry (activation function, clock lines, NAND gate, latch, inputs, and internal nodes of the flip-flops). In the last two columns, we show the power ratio for the combinational logic and the power ratio for the clock-related circuitry for the two implementations.

First, notice that the ratio P^{Comb}/P^{Clk} is almost always smaller for the gated-clock FSM's. This result is quite intuitive, because P^{Clk} in the gated-clock FSM includes the power dissipation of the activation function. The results of columns three and four are somewhat counterintuitive, because they show that *there is consistently higher power saving in the combinational logic than in the clock-related circuitry.* This result is due to three factors. First, the reduced switching activity on the outputs of the flip-flops, that are generally highly loaded. Second, the absence of propagation to internal nodes in the FSM logic of input transitions when the FSM is in a self-loops. Third and most importantly, the simplification in the FSM's logic that is obtained using the ON-set of the activation function as additional controllability *don't care* set.

TABLE II
PARTITION AND COMPARISON BETWEEN POWER DISSIPATION IN CLOCKING LOGIC AND FSM LOGIC FOR LOCALLY-MOORE AND GATED-CLOCK FSM'S

Circuit	$P^{Comb}_{Gated}/P^{Clk}_{Gated}$	$P^{Comb}_{Loc\_M}/P^{Clk}_{Loc\_M}$	$P^{Comb}_{Gated}/P^{Comb}_{Loc\_M}$	$P^{Clk}_{Gated}/P^{Clk}_{Loc\_M}$
bbara	1.2777	2.7006	0.3624	0.7660
bbsse	1.5317	2.5386	0.7601	1.2597
bbtas	1.2242	1.8715	0.6861	1.0488
keyb	2.4024	3.2242	0.8185	1.0985
lion9	2.0734	2.2749	0.8937	0.9806
s298	17.7560	17.9872	0.9887	1.0016
s420	1.3517	1.5251	0.8126	0.9169
scf	2.8988	2.7368	0.9274	0.8755
styr	4.2330	4.3990	0.8969	0.9320
test	1.1751	2.8957	0.3048	0.7512

We want to point out that our methodology attains consistent power savings not only when the clock line is heavily loaded and large flip-flops are used, but also for very small FSM's with optimized flip-flops. It is, however, important to remark that the full extent of the possible savings is obtained only if the combinational logic is reoptimized with the increased *don't care* set previously described. Moreover, for classes of FSM's such as synchronous counters, or more generally, FSM without self-loops, our methodology is ineffective in reducing power consumption.

In summary, the power savings depend on the fraction of the total operation time that the FSM spends in idle condition. Don't care optimization is very helpful when the FSM is small and the idle time is a relatively small fraction of the total, because it helps in reducing the overhead of the clock-gating circuitry. For large FSM's the impact of *don't cares* is generally less relevant. Even if the power savings are basically decided by the initial structure of the FSM, it is important to have an automated synthesis procedure such as ours, so as to avoid unnecessary effort from designers in trying to manually design clock-stopping logic.

VI. CONCLUSION AND FUTURE DIRECTIONS

We have described a technique for the automatic synthesis of gated clocks for Mealy and Moore FSM's. We want to emphasize that our method is part of a complete procedure, starting from FSM's state-table specification to fully mapped network, and it has been tested with accurate power estimation tools. The quality of our results depends on the initial structure

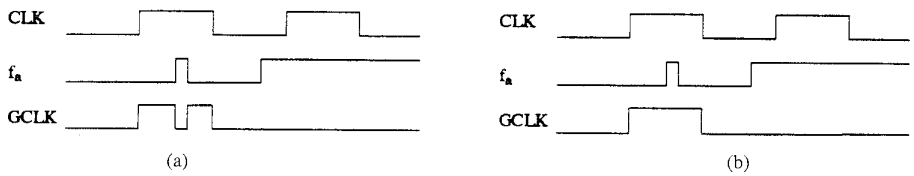


Fig. 9. Timing diagrams of the activation function f_a, the global clock CLK and the gated clock GCLK when (a) a simple AND gate is used and (b) a latch and the AND gate are used.

of the FSM, but we obtain important power reductions for a large class of FSM's, where the probability of being in a self-loop (idle) is high. Even if our tool cannot fully replace the knowledge of the designer in finding idle conditions at the architectural level, it may enable design exploration for cases where it is not clear if clock gating may produce sizable power savings.

While automatic synthesis of dynamic power management circuits is not a new idea, this paper provides two novel theoretical contributions. First, we presented a transformation for Mealy FSM's that makes them suitable for gated-clock implementation, allowing for greater flexibility in the choice of clock-stopping functions with small support and lower complexity. Second, we have proposed a logic optimization problem, called "constrained probability minimum literals" problem, and we have described its exact and heuristic solutions. Our solver has large applicability, and can improve the performance of any power management scheme that relies on optimized combinational logic that stops the clock with maximum efficiency.

For some FSM's, even the limited increases in the number of states and transitions produced by the locally-Moore transformation is unacceptable. To address this problem, a procedure that splits only on the states with high probability self-loops may become useful.

An area-oriented state assignment program has been used in the current implementation. Additional power savings could be obtained if the state assignment algorithm takes power dissipation into account. The relationship between state assignment and activation function synthesis requires further investigation.

Finally, future research will concentrate on the implementation of fully symbolic algorithm for the synthesis of the activation function and on the application of our techniques to large synchronous networks.

APPENDIX: TIMING ANALYSIS

The activation function uses as its inputs the state and input signals of the FSM; therefore, it is on the critical path of the circuit. In order to verify the correctness of the gated clock implementation, we need to make sure that the delay that the activation function adds to the delay of its inputs is less than the cycle time T of the circuit. We can test this condition performing static timing analysis on the network composed by the activation function and the logic that feeds its input (the combinational part of the FSM and the logic in the previous stages that computes the primary inputs). We call the critical path delay through this network T_{crit}.

If we collect the delays through the latch and the AND gate (Fig. 1) and the setup time of the input flip-flops in one worst-case parameter T_{wc} we obtain the following constraint inequality for the activation function

$$T_{crit} < T - T_{wc}. \tag{8}$$

Moreover, the presence of a gate on the clock path usually implies increased clock skew. In a completely automated synthesis environment, it should be possible for the designer to specify accurate skew control for the gated clock line, thus preventing possible races or timing violation involving the logic blocks in the fan-out of the FSM.

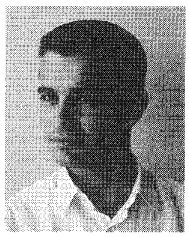
Finally, it should be noticed that the presence of the latch L is fundamental for the correct behavior of the proposed gated clock implementation. A simple combinational AND gate is not acceptable because the activation function is not guaranteed to change when the global clock signal is low. If the activation function changes when the clock is high and it is not latched, it may create spurious pulses (glitches) on the local clock line. An example of this problem is shown in Fig. 9. In Fig. 9(a), the behavior of the gated clock line simply ANDed with f_a' is shown. The glitch on f_a produces a glitch on the gated clock line that will very likely produce incorrect behavior in the FSM. In contrast [Fig. 9(b)], when f_a is latched, the glitch does not pass through the latch when the clock is high. Function f_a may also produce glitches when the clock is low, but in this case the AND gate itself will filter out the spurious transitions, because the global clock signal has the controlling value.

The presence of the latch could be avoided if we could guarantee that the activation function changes only after the falling edge of the global clock, or that the circuitry that implements the activation function is hazard free. These constraints may be acceptable in some particular examples, but the general solution that we have discussed has a small overhead (only one latch) and it does not require specialized techniques for the synthesis of f_a.

REFERENCES

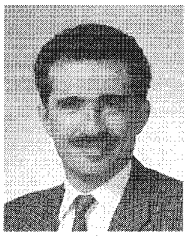
[1] M. Alidina and J. Monteiro *et al.*, "Precomputation-based sequential logic optimization for low power," *IEEE Trans. VLSI Syst.*, vol. 2, pp. 426–436, Jan. 1995.
[2] V. Tiwari, S. Malik, and P. Ashar, "Guarded evaluation: Pushing power management to logic synthesis/design," in *Proc. Int. Symp. Low Power Design*, Apr. 1995, pp. 221–226.
[3] L. Benini, P. Siegel, and G. De Micheli, "Automatic synthesis of gated clocks for power reduction in sequential circuits," *IEEE Design Test Comput.*, vol. 11, pp. 32–40, Dec. 1994.
[4] P. E. Landman and J. M. Rabaey, "Activity-sensitive architectural power analysis for the control path," in *Proc. Int. Symp. Low Power Design*, Apr. 1995, pp. 93–98.

[5] A. Shen, A. Ghosh, S. Devadas, and K. Keutzer, "On average power dissipation and random pattern testability of CMOS combinational logic networks," in *Proc. Int. Conf. Computer-Aided Design,* Nov. 1992, pp. 402–407.
[6] C. Tsui, M. Pedram, and A. Despain, "Technology decomposition and mapping targeting low power dissipation," in *Proc. Design Automation Conf.,* 1993, pp. 68–73.
[7] K. Roy and S. Prasad, "Circuit activity based logic synthesis for low power reliable operations," *IEEE Trans. VLSI Syst.,* vol. 1, pp. 503–513, Dec. 1993.
[8] L. Benini and G. De Micheli, "State assignment for low power dissipation," in *Proc. IEEE Custom Integrated Circuits Conf.,* May 1994, pp. 136–139.
[9] P. McGeer, J. Sanghavi *et al.,* "ESPRESSO-SIGNATURE: A new exact minimizer for logic functions," *IEEE Trans. VLSI Syst.,* vol. 1, pp. 432–440, Dec. 1993.
[10] N. Yeung *et al.,* "The design of a 55SPECin92 RISC processor under 2W," in *Proc. IEEE Int. Solid-State Circuits Conf.,* Feb. 1994, pp. 206–207.
[11] B. Suessmith and G. Paap, III, "Power PC 603 microprocessor power management," *Comm. ACM,* vol. 37, no. 6, pp. 43–46, June 1994.
[12] N. Weste and K. Eshragian, *Principles of CMOS VLSI Design,* 2nd ed. Reading, MA: Addison-Wesley, 1992.
[13] K. Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1982.
[14] G. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Symbolic algorithms to calculate steady-state probabilities of a finite state machine," in *Proc. IEEE Euro. Design Test Conf.,* Feb. 1994, pp. 214–218.
[15] ——, "Probalistic analysis of large finite state machines," in *Proc. Design Automation Conf.,* June 1994, pp. 270–275.
[16] J. Schutz, "A 3.3 V 0.6 μ BiCMOS superscalar microprocessor," in *Proc. IEEE Int. Solid-State Circuits Conf.,* Feb. 1994, pp. 202–203.
[17] J. Hartmanis and H. Stearns, *Algebraic Structure Theory of Dequential Machines.* Englewood Cliffs, NJ: Prentic-Hall, 1966.
[18] M. R. Garey and D. S. Johnson, *Computers and Intractability. A Guide to the Theory of NP-Completeness.* San Francisco, CA: Freeman, 1983.
[19] S. Martello and P. Toth, *Knapsack Problems. Algorithms and Computer Implementations.* New York: Wiley, 1990.
[20] E. Sentovich *et al.,* "Sequential circuit design using synthesis and optimization," in *Proc. Int. Conf. Computer Design,* Oct. 1992, pp. 328–333.
[21] F. Mailhot and G. De Micheli, "Algorithms for technology mapping based on binary decision dBagrams and on Boolean operations," *IEEE Trans. Computer-Aided Design,* vol. 12, pp. 599–620, May 1993.
[22] G. De Micheli, *Synthesis and Optimization of Digital Circuits.* New York: McGraw-Hill, 1994.
[23] O. Coudert and C. Madre, "Implicit and incremental computation of primes and essential primes of Boolean functions," in *Proc. Design Automation Conf.,* June 1992, pp. 36–39.
[24] R. Marcelescu, D. Marculescu, and M. Pedram, "Switching activity analysis considering spatiotemporal correlations," in *Proc. Int. Conf. Computer-Aided Design,* Nov. 1994, pp. 294–299.
[25] J. Monteiro, S. Devadas, and B. Lin, "A methodology for efficient estimationof switching activity in sequential logic circuits," in *Proc. Design Automation Conf.,* June 1994, pp. 315–321.
[26] B. Lin and A. R. Newton, "Synthesis of multiple-level logic from symbolic high-level description languages," in *Proc. IEEE Int. Conf. Computer Design,* Aug. 1989, pp. 187–196.
[27] A. Salz and M. Horowitz, "IRSIM: An incremental MOS switch-level simulator," in *Proc. Design Automation Conf.,* June 1989, pp. 173–178.



Luca Benini (S'95) received the Laurea degree from the University of Bologna, Italy, and the M.S. degree in electrical engineering from Stanford University, Stanford, CA, where he is also pursuing the Ph.D. degree in electrical engineering.

He was a Research Assistant with the Department of Electronics and Computer Science, University of Bologna. His research interests are in synthesis, behavioral synthesis, and design for testability.



Giovanni De Micheli (S'79–M'82–SM'89–F'89) received the Dr. Eng. degree in nuclear engineering from the Politecnico di Milano, Italy, 1979, the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California, Berkeley, in 1980 and 1983, respectively.

He is (by courtesy) Professor of Electrical Engineering and of Computer Science with Stanford University, Stanford, CA. Previously, he held positions with the IBM Thomas. J. Watson Research Center, Yorktown Heights, NY, the Department of Electronics, Politecnico di Milano, and Harris Semiconductor, Melbourne, FL. His research interests include several aspects of the computer-aided design of integrated circuits and systems, with particular emphasis on synthesis of hardware/software codesign. He is the author of *Synthesis and Optimization of Digital Circuits,* (McGraw-Hill, 1994) and coauthor/editor of three other books. He was also Codirector of the NATO Advanced Study Institute on Hardware /Software Codesign, held in Tremezzo, Italy, in 1995, and on Logic Synthesis and Silicon Compilation, held in L'Aquila, Italy, in 1986 and 1987.

Dr. De Micheli was granted a Presidential Young Investigator Award in 1988, the 1987 IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN Best Paper Award, and two Best Paper Awards at the Design Automation Conference in 1983 and 1993. He is a member of the Board of Governors of the IEEE Circuits and Systems Society and a member of the Editorial Board of the IEEE PROCEEDINGS. He is an Associate Editor with the IEEE TRANSACTIONS ON VLSI SYSTEMS, *Integration: the VLSI Journal,* and *Design Automation for Embedded Systems.* He is technical Cochair (design tools) of the 1996–1997 Design Automation Conference and was Technical and General Chairman of ICCD in 1988 and 1989, respectively.