VLSI DESIGN
2001, Vol. 12, No. 2, pp. 167–186
Reprints available directly from the publisher
Photocopying permitted by license only

Automatic FSM Synthesis for Low-power Mixed Synchronous/Asynchronous Implementation

BENGT OELMANN[a,*], KALLE TAMMEMÄE[b], MARGUS KRUUS[b] and MATTIAS O'NILS[a]

[a]*Mid-Sweden University, Department of Information Technology, Sundsvall, Sweden;*
[b]*Tallinn Technical University, Department of Computer Engineering, Tallinn, Estonia*

(Received 20 June 2000; In final form 3 August 2000)

Power consumption in a synchronous FSM (Finite-State Machine) can be reduced by partitioning it into a number of coupled sub-FSMs where only the part that is involved in a state transition is clocked. Automatic synthesis of a partitioned FSM includes a partitioning algorithm and sub-FSM synthesis to an implementation architecture. In this paper, we first introduce an implementation architecture for partitioned FSMs that uses gated-clock technique for disabling idle parts of the circuits and asynchronous controllers for communication between the sub-FSMs. We then describe a new transformation procedure for the sub-FSM. The FSM synthesis flow has been automated in a prototype tool that accepts an FSM specification. The tool generates synthesizable RT-level VHDL code with identical cycle-to-cycle input/output behavior in accordance with the specification. An average power reduction of 45% has been obtained for a set standard FSM benchmarks.

Keywords: Low-power design; FSM decomposition; FSM partitioning; Asynchronous logic; Gated-clock techniques; RTL-synthesis

1. INTRODUCTION

Optimization techniques for low average power consumption in synchronous digital CMOS circuits often attempt to minimize the dynamic power consumption described as:

$$P = V_{DD}^2 \cdot f \cdot \sum_i \alpha_i \cdot C_i$$

where α_i is the probability of a signal transition within a clock period at node i, C_i is the *switched capacitance* in node i, V_{DD} is the power supply voltage and f is the clock frequency. Power optimization can be made on all abstraction levels, from IC technology to the system level. When optimizing on the gate level, or even higher abstraction levels, power optimization minimizes the product $\alpha \cdot C$, called *effective capacitance*. Here,

both power supply voltage and clock frequency are often regarded as fixed in the system specification and cannot be affected.

For architectural design, it is possible to reduce the effective capacitance by minimizing the communication over long wires that have high capacitance. Placing the required resources, such as processing units and memories, locally within the module reduces global communication [11]. It is also possible to shut down parts of the design that are idle, which makes the effective capacitance equal to zero during that period. Data path units, such as multipliers and ALUs, which are purely combinatorial logic, are shut down by disabling further changes of the values on the input signals. Here, additional logic is introduced to detect if the unit can be shut down or not. This technique has been described by Alidina *et al.*, in [1] and is called the *input- disabling precomputational-based* approach. For sequential circuits, gated-clock techniques are used to disable the clock signal to the parts of the design that are idle. For large circuits, such as complex microprocessors, this technique is often referred to as *dynamic power management*. Here, there are large functional units, such as cache memories and floating point units, with very specific tasks that are shut down when not used. This type of coarse-grained gated-clock technique is possible to apply manually by the designer thanks to the small number of places where clock-gating is introduced and to the fact that the different units are functionally well separated and therefore easy to identify. In order to use fine-grained clock-gating, a single functional unit is partitioned into several sub-units where each of them are conditionally clocked by a gated clock signal. An automated procedure is needed for synthesizing the original design to a gated-clock implementation optimized for power. The number of places where the clock is gated increases and it becomes less obvious as to how to partition the unit.

For FSMs, the most common approach to low-power design is to divide the FSM into two or more sub-FSMs where only one of these is active at a time. Both the precomputation-based technique and the clock-gating technique have been used. Dasgupta *et al.* [8] use the precomputation-based technique for PLA (*Programmable Logic Array*) implementations of FSMs and reduce the effective capacitance in the transition logic and output logic. Benini [5] *et al.* , detect self-loops, *i.e.*, when the next state is equal to the current state and the clock is gated under this condition. This approach has been extended in [5, 14] where states that are strongly connected, *i.e.*, that there is a high probability of having state transitions among them, are placed in the same cluster or super-state, and the state transitions within the super-state can be seen as a self-loop for that super-state. These approaches result in partitioning based on the description given in a STG (State Transition Graph). In the paper by Roy *et al.* [14], the partitioning is based on state assignment. For FSMs with few or no self-loops, *e.g.*, counters, it is possible to detect smaller FSMs that have self-loops. On higher levels of abstraction, the gated-clock approach has been applied for low-power optimization from high-level specifications of hardware. In [7], the control flow is examined and mutually exclusive sections of the computation are detected and determine the partitioning of the FSM that controls the execution. In another approach presented by Hwang *et al.* [9], the FSM is partitioned along with the data path, which leads to an implementation where both the data path and the controller are shut down when idle.

The amount of power that is saved by partitioning the FSM is mainly determined by how good the partitioning algorithm can cluster strongly connected states together in sub-FSMs and by how large the cost is, in terms of power, to make a state transition from one sub-FSM to another. In our work, we have focused on minimizing the cost of making state transitions from one sub-FSM to another. This has led us to a new implementation architecture that is based on a gated-clock technique for shutting down idle sub-FSMs, and asynchronous communication between the sub-FSMs. The two main benefits of having asynchronous communication are as follows: First of all,

the power overhead introduced by the circuits handling sub-FSM communication is up to five times lower than for the corresponding synchronous solution [12]. Secondly, a more power-efficient protocol can be employed that on its own, lowers the power consumption up to two times compared to existing ones [13].

The outline of the rest of this paper is as follows: The next chapter describes the principles behind gated-clock FSMs, points out the problems with fully synchronous implementation architectures and motivates the asynchronous approach we propose. Chapter 3 presents the decomposition model we use and Chapter 4 describes details on how partitioned FSMs are implemented. Chapter 5 presents experimental results from automatically synthesized FSM benchmark circuits.

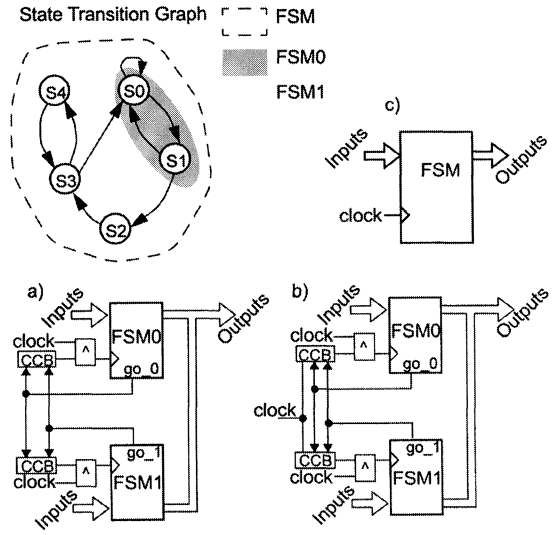


FIGURE 1 Implementation architectures for: (a) asynchronous partitioned FSM, (b) synchronous partitioned FSM, and (c) monolithic FSM.

2. BACKGROUND

The goal of an implementation architecture for partitioned FSMs is that it should provide an implementation with the same input/output behavior as the FSM specification describes. In our case, the specification is given in STG form for a monolithic FSM with synchronous behavior. The implementation architecture we propose in this paper is similar to the one that we have earlier presented in [12]. It also has much in common with the synchronous architecture that is used by Benini *et al.* [5]. The proposed architecture is depicted in Figure 1a and consists of: (1) a number of sub-FSMs, (2) an equally large number of CCBs (Clock Control Blocks), and (3) *AND* gates for gating the clock signal. Alternative implementation architecture is a fully synchronous partitioned FSM, shown in Figure 1b, and a monolithic implementation, shown in Figure 1c. The important difference between the two architectures for partitioned FSMs is that the communication between the sub-FSMs that is handled by the CCBs is asynchronous instead of synchronous.

The purpose of the sub-FSM interaction protocol is to control the activation and the deactivation of the sub-FSMs. When a state transition to a state in another sub-FSM takes place, the active sub-FSM generates an event on a communication control signal called a *go-signal*. This event has the following functional meaning: *Activate sub-FSM that contains the destination state of the transition and deactivate the currently active sub-FSM*. In general, a sub-FSM may submit many go-signals, one for each external state transition, and it can be activated by one of many incoming state transitions. The CCB that is associated with a sub-FSM enables or disables the clock signal based on go-signals both from its own and other sub-FSMs. In Figure 2, a timing diagram is shown for state

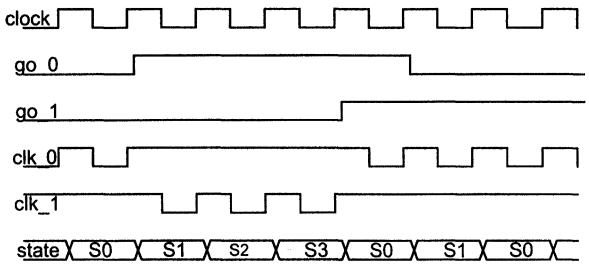


FIGURE 2 Timing diagram for transitions from FSM0 to FSM1 and then back to FSM0.

transitions between states residing in the two sub-FSMs (*FSM0* and *FSM1*), see Figure 1.

With asynchronous control for the CCB, we can remove the need for a clock signal. For low-power design, it is important to have as small effective capacitance as possible. The clock signal is the signal with the highest switching activity (twice as high compared to any data signal) and the capacitance added here will significantly contribute to increased power consumption.

The additional control circuitry introduced by the CCBs will naturally introduce additional power dissipation (*power overhead*). The number of CCBs are equal to the number of sub-FSMs, but only one of them enables the clock at a time. A CCB has three operational modes; they are:

- **Hand-over** When a transition from one sub-FSM to another takes place. In this mode, the asynchronous CCB is *active* and responds to the go-signal.
- **Enable** This is one of two passive modes. The CCB is passive and enables the local clock signal to the sub-FSM. In this mode, the CCB dissipates no power except from the AND gate enabling the clock.
- **Disable** The CCB is passive and disables the local clock signal. The power consumption comes from switching the input of the AND gate.

In Figure 3, the energy consumption for both the synchronous and asynchronous CCB is given.

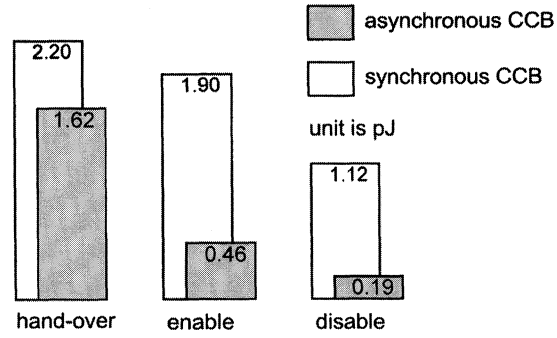


FIGURE 3 Energy consumption in synchronous and asynchronous CCB in different modes.

From this figure we can observe two things. The first is that energy consumption for the asynchronous CCB is lower in all modes. The second is that the difference in the energy dissipation of the asynchronous CCB in the different modes is larger than the difference of the synchronous CCB. This property is typical for asynchronous circuits where power is dissipated only when it is active. In a partitioned FSM when there is no *hand-over*, only one CCB is in *enable* mode while the rest are in *disable* mode. In the clock cycle when a hand-over occurs, one CCB is in *hand-over* mode. The total power consumption in the CCBs can be expressed as:

$$P_{\text{CCB}} = a \cdot P_{\text{hand-over}} + (1 - a) \cdot P_{\text{enable}} + (N - 1) \cdot P_{\text{disable}}$$

where $P_{hand\text{-}over}$, P_{enable}, and $P_{disable}$ are the power consumption for the CCBs in the different modes, a is the probability of a hand-over and N is the number of CCBs. With asynchronous CCBs, the power consumption can be reduced by five times for CCBs in disable mode, which constitutes the majority of the CCBs. This will have a significant impact on the total power overhead, especially when the number of CCBs (N) is large.

The second advantage of having asynchronous control is that the sub-FSM communication protocol can be made more power efficient. The existing synchronous solutions, *e.g.* [5], require that two sub-FSMs are clocked simultaneously at hand-over. The power consumption at hand-over will be largest here because two sub-FSMs will be active in this cycle. We have removed this requirement by implementing the CCB as an asynchronous controller. A synchronous controller updates its states only at clock edges. In contrast to this, an asynchronous controller can change state as a response to an input change after only some combinatorial delay. We used this property to design an asynchronous protocol that does not require simultaneous clocking of two sub-FSMs at hand-over. The total power consumption for the sub-FSMs in the synchronous ($P_{sub\text{-}fsm,synch}$) and the asynchronous ($P_{sub\text{-}fsm,asynch}$)

case is expressed as:

$$P_{\text{sub-fsm,synch}} = \sum_{i=0}^{N-1} T_i \cdot P_{\text{sub-fsm,i}} + \sum_{i=0}^{N-1} a_i \cdot P_{\text{sub-fsm,i}}$$

$$P_{\text{sub-fsm,asynch}} = \sum_{i=0}^{N-1} T_i \cdot P_{\text{sub-fsm,i}}$$

where T_i is the duty probability for the sub-FSM, $P_{sub\text{-}fsm,i}$ is the internal power consumption of the i^{th} sub-FSM, and a_i is the probability of activation of the ith sub-FSM.

The total power dissipation in a partitioned FSM is the sum of P_{CCB} and $P_{sub\text{-}fsm}$. Using the proposed asynchronous approach reduces both of these components.

3. FSM DECOMPOSITION

In this chapter, the decomposition model we use is presented first. We then make the necessary definitions that will be used for describing the FSM transformation procedures. Implementation of these procedures is discussed in Chapter 4.

In this chapter, we use abstract automata theory as has been described by Baranov in [3]. There is, however, a small difference in terminology between the work in [3] and other works we refer to in the area of implementation of decomposed FSMs, *e.g.* [5, 9]. In [3], the initial FSM, which corresponds to a monolithic FSM implementation, is referred to as a *source automaton* and a sub-FSM is referred to as a *component automaton*. In this chapter, we use the same abstraction and terminology as in [3]. Elsewhere in this paper, the terminology in most of the referred papers concerning implementation is used.

3.1. Decomposition Model

The source Mealy automaton is defined as a sextuple:

$$A = (S, X, Y, \delta, \lambda, s_0)$$

where S is the set of states, X is the set of binary inputs, Y is the set of binary outputs, δ is the transition function, λ is the output function and s_0 is the initial state. The automaton can be represented in the form of a transition table, where every row defines one transition from a source state to a destination state along with a certain output term according to a certain input term.

Let there be a partition on the set S:

$$\pi = \{S^1, \ldots, S^n\}$$

The automaton A can be decomposed into a set of component automata where every block $S^i \in \pi$ defines a component automaton:

$$A^m = (S^m, X^m, Y^m, \delta^m, \lambda^m, s_0^m)$$

We call states S^m internal states of component automaton. X^m is the set of input variables at all transitions from the states in S^m, and Y^m is the set of output variables at all transitions from the states in S^m. δ^m and λ^m are transition and output functions on the sets S^m and X^m. Such decomposition can be achieved by reordering the groups of transition table rows having the same source state, followed by segmentation according to π blocks.

3.2. Definitions

In this section, we will define different sets from the component automaton point of view. Let us define $V(s_k)$ to be the set of states from which there are transitions to the state s_k; s_k is not included in $V(s_k)$. With X_h we denote the existence of input valid in the expressions where it is used.

$$V(s_k) = \{s_j | \delta(s_j, X_h) = s_k,\ j \neq k\}$$

Similarly, we define a set of states, $T(s_k)$, not included in S^m to which there are transitions from the states of S^m.

$$T(s_k) = \{s_j | \delta(s_k, X_h) = s_j,\ j \neq k\}$$

We define two similar sets that are of more importance for the whole component automata.

$$V(S^m) = \{s_j | \delta(s_j, X_h) = s_k,\ s_j \notin S^m,\ s_k \in S^m\}$$

$$T(S^m) = \{s_j | \delta(s_k, X_h) = s_j,\ s_j \notin S^m,\ s_k \in S^m\}$$

Here, $V(S^m)$ is a set of states in S^m that have transitions to states outside S^m where s_k resides. $T(S^m)$ is a set of states not included in S^m, to which there are transitions from the states not included in S^m.

Let us define the set of states in S^m where there are transitions from other component automata as:

$$Q(S^m) = \{s_j | \delta(s_k, X_h) = s_j,\ s_j \in S^m,\ s_k \notin S^m\}$$

The position of the sets defined above is depicted in Figure 4.

The set $T(S^m)$ originates from another subset of S^m, which is denoted as a set $W(S^m)$.

$$W(S^m) = \{s_j | \delta(s_j, X_h) = s_k,\ s_j \in S^m,\ s_k \notin S^m\}$$

The position of the sets defined above is depicted in Figure 5.

We will use the shorter denotations V^m, W^m, Q^m, and T^m, in the rest of this chapter.

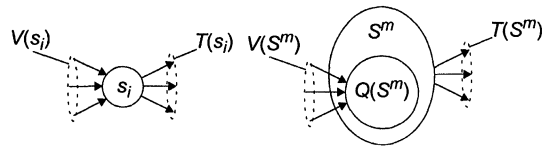


FIGURE 4 The transition sets of a state (left) and component automata (right).

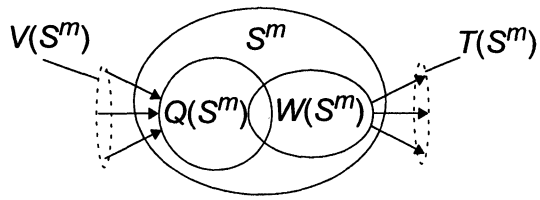


FIGURE 5 Input and output subsets of S^m.

3.3. Transformation of the Network

In this section, we will present the transformation sequence that results in a modified network description suitable for the implementation architecture we are targeting. In the following presentation, the definitions given in the previous section are used.

The transformation is carried out by the following steps:

1. Replace the transitions from the set W^m to T^m with transitions from W^m to additional states that we call *transition states*, G^m *i.e.*, $\delta(s_j, X_h) = s_k;\ s_j \in W^m,\ s_k \in T^m$ with $\delta'(s_j, X_h) = s_k;\ s_j \in W^m,\ s_k \in G^m$.

   There is a one-to-one mapping between the elements of T^m and G^m.

   Let us denote the set of states replacing the transitions originating from V^m with G^{m-}. These transitions cause the activation of component m.
2. Introduce new unconditional transitions from states in G^m to a single state d^m.

$$\delta(g_i, 1) = d^m; \quad g_i \in G^m$$

3. Introduce new transitions originating from the additional state d^m. The new transitions are based on all transitions from the set Q^m.

   There is a many-to-one mapping between the elements of G^{m-} and Q^m. We define additional inputs, one for every state in Q^m:

$$E^m = \left\{ e_j \middle| \bigcup_{s_i \in G^{m-}} \delta(s_i, X_h) = s_j; \quad s_j \in Q_m \right\}$$

   The new transition functions can now be evaluated: for every $\delta(s_j, X_h) = s_k;\ s_j \in Q^m$ transition functions $\delta'(d^m, (e_i, X_h)) = s_k;\ s_j \in Q^m$, $e_i \in E^m$ are added.
4. Introduce additional output functions: for every $\lambda(s_i, X_h) = s_k;\ s_i \in Q^m$ output functions $\lambda'(d^m, (e_i, X_h)) = s_k;\ e_i \in E^m$ are added. As there are as many entering transitions as there are exiting transitions in the network, we can say

that there is a one-to-one mapping between the source transitions, additional transitions and output functions.

5. The first transformation step (replacement of T^m with G^m) may result in states in the set Q^m, which do not have any incoming transitions. Such states are redundant and can be removed, except in the case where the state is an initial state of the network. An example of source decomposition and a transformed network is given in Figure 6.
6. The resulting network of the previous steps has the same behavior as the source automaton when the initial state is properly defined. Let s_0 be the initial state of the source automaton. The initial state of the network has to be defined as:
   1. The component containing the state s_0 is assigned s_0 as its initial state.
   2. Other components are assigned their initial states to the corresponding d-state of the component.

3.4. Functional Equivalence

The initial condition of the network, described in the previous section, guarantees that only one e-signal is active at a time. The equivalence of the source automaton and the transformed network can be proved. In the proof, the notation of transition tables is used. We will use a reordered and segmented source transition table.

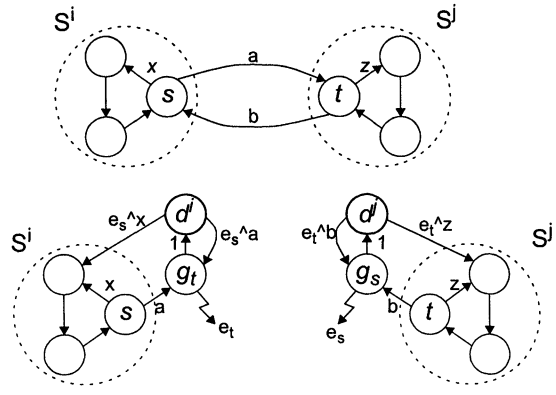


FIGURE 6 Example of source decomposition (top) and a transformed network (bottom).

Proof

1. **Transitions Inside of the Component** There are equivalent rows in the source and transformed transition tables.
2. **Exiting Transitions of the Component** For every exiting transition in the source table, there is a matching transition with the same output in the transformed table but with unique target states in the set G. Additionally, there are unconditional transitions from the states in G to a single d-state with a unique output signal in E.

   For every transition from the target state of an exiting transition, there is an additional matching transition in the transformed network from the d-state of the target component. This matching transition has the same target state, output vector and input term in conjunction with the appropriate signal in E.

According to the initial condition, only one component can be in the g-state at a time. Consequently, there can only be one e-signal active at a time. This will uniquely define the transitions to be taken, see Figure 6.

A condition that we call *static G^m-state* occurs when an automaton enters G^{m-} in the cycle followed by the entrance to G^m. This condition requires special considerations for the implementation. It will be described later in Section 4.4.

3.5. Example

Let us decompose the microprogram automaton A, given in Figure 7, into a two-component network using state-partition. According to the tables, there are three crossing transitions of the partition $\pi = \{\{s_1, s_2, s_3\}, \{s_4, s_5\}\}$ which define the number of g-states. From the tables it can be seen that there are as many e-signals in the network as there are crossing transitions. It can also be seen that inter-component communication is formed by

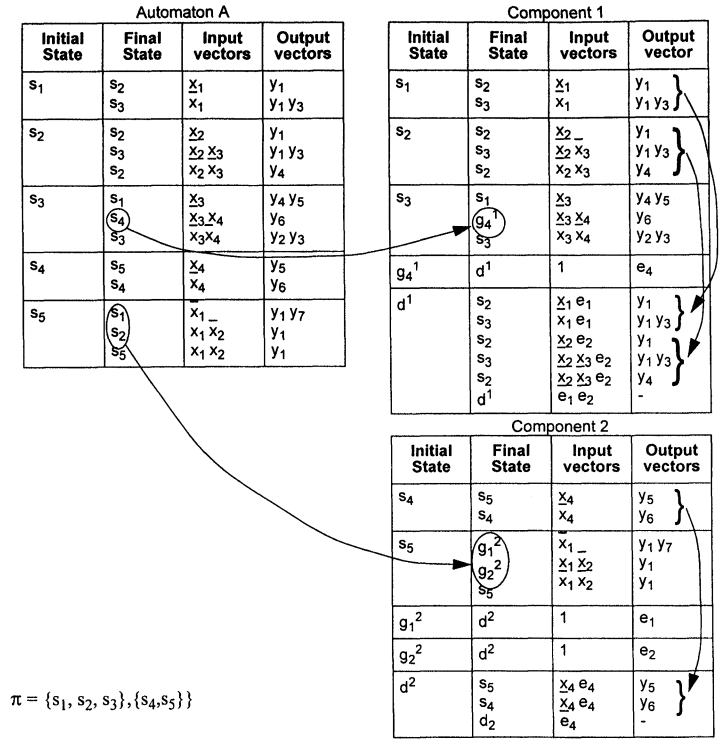


FIGURE 7 Decomposition example.

the signals $\{e_1, e_2, e_4\}$ where the index is bound to the target state in the source automaton.

4. IMPLEMENTATION

In this chapter, we will describe how decomposed FSMs are implemented. First we describe where in the design flow the decomposition takes place. Next, an overall picture of the tool that automatically carries out the decomposition is described. After that, the implementation of the FSM transformation steps presented in Chapter 3 are described. Hardware estimation is used to rank the different partition candidates that are generated by the tool. In Section 4.3, the estimation functions and their parameters are given. In order to have a complete synthesis design flow, additional cells must be added to a standard cell library. The implementation and the design details of the cells are given in Section 4.4.

4.1. Overview

4.1.1. Introduction

The position of the FSM power optimization procedures that have been implemented in the LIFS tool is depicted in Figure 8. FSM power optimization is one step of several synthesis steps in FSM synthesis, which are a part of RTL synthesis. Therefore, it is important that the computational complexity of the power optimization step is kept low in order to keep the total time spent in synthesis low.

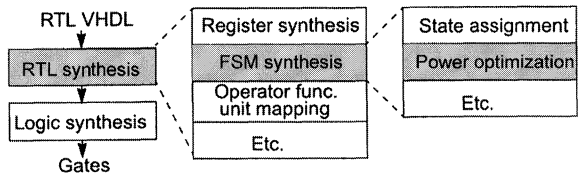


FIGURE 8 The LIFS tool positioned in a synthesis-based digital design flow.

An overview of the information flow in LIFS is shown in Figure 9. The FSM description is given as an STG. Currently, we use the *KISS*2 format from Berkeley [15], but in principle, RT-level VHDL or graphical input could be used in that they all contain the same information. Power consumption in digital CMOS is dominated by the dynamic power consumption, which is highly data-dependent. In order to estimate power consumption, it is necessary to have a set of input data that is, under typical operating conditions, applied to the inputs of the FSM. In LIFS, it is possible to either give these input vectors in a testbench or, when no typical data is available, to specify probabilities for an input to be high (logic one) for each of the inputs. The power optimization is made for a user-given area constraint. The area constraint is given as the maximum acceptable increase in area relatively to a monolithic FSM. This will allow the designer to trade circuit area for reduced power consumption. The tool is designed to work in a standard-cell based design flow. In order to make early power and area estimates, data about power and area for three types of cells are needed: a clocked storage element (D flip-flop), a CCB and a gate (2-input *NAND* gate). The output of the tool is an RT-level VHDL description of the partitioned FSM. This description is normally passed on to a standard logic synthesis tool that produces the gate netlist. Along with the VHDL code, design specific scripts for logic synthesis are also generated.

The tool is divided into two main parts. The first part collects statistics about the FSM in order to find the state transition probabilities. This part may be omitted from the synthesis run if the transition probabilities are already known from the environment of the FSM or from a previous synthesis run where the *STG with probabilities* has been generated. The second part is where the actual partitioning takes place.

4.1.2. Statistics Collection

The purpose of the statistics collection methods is to determine the probabilities of state transitions in the STG. The statistics form the basis for the partitioner when *clustering* states, *i.e.*, group states that have the highest probability for their connections. One of the two implemented methods for collecting statistics is used by the tool.

The first method we call *profiling*. Profiling uses user-supplied input vectors for simulating the FSM. The tool first generates VHDL code that

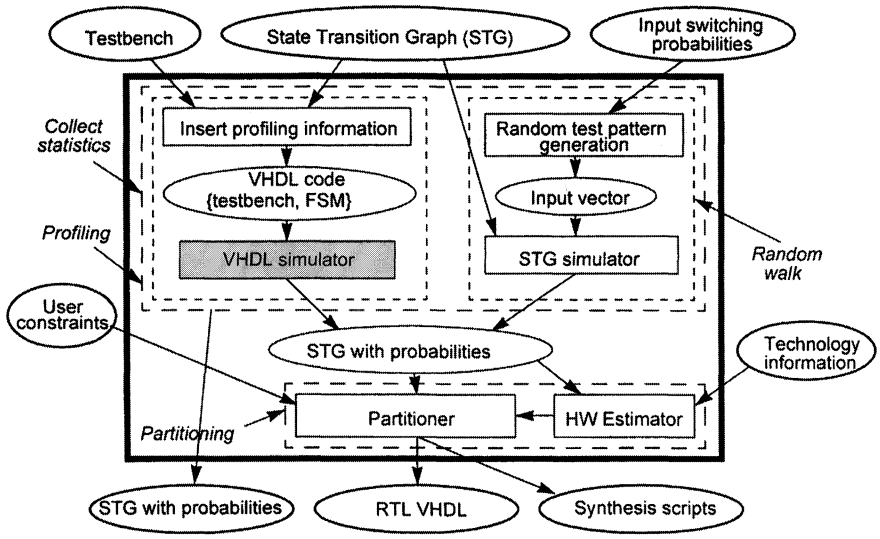


FIGURE 9 Overview of the LIFS tool.

corresponds to the initial STG specification. It also inserts profiling information during the generation of the VHDL code. The state transitions are traced during simulation and collected statistics are written back on file. The simulation is made in a standard commercial VHDL simulator.

The second method is based on *random walk*. Here, the state transition probabilities are based on the input probability vectors given by the user. The user specifies, for each of the inputs, the probability of the input being at high state. Random input vectors are generated from this given input probability vector using a uniform distribution function. The random input vectors are then used to simulate the STG. The simulation is carried out in a STG simulator that is embedded in LIFS. For the design examples that we present later in this paper, the length of the random walk simulation has been set to n^3, where n is the total number of arcs in the STG. The simulation time for determining the transition probabilities easily becomes very time-consuming for complex FSMs. It is desirable to run this part only once, even if the FSM must be re-synthesized. For that purpose, we have extended the KISS2 format by adding the transition probability for every transition in the FSM specification.

4.1.3. Partitioning

As previously mentioned, the power reduction strategy is to partition the FSM into a number of sub-FSMs. In a partitioned FSM, a state transition can take place inside the sub-FSM or between two different sub-FSMs, which we call a *crossing transition*. In Chapter 2, we showed that transitions within a sub-FSM dissipate less power than state transitions from one sub-FSM to another. It is also advantageous to have as few sub-FSMs as possible active to reduce the effective capacitance. But at the same time, dividing the FSM into smaller partitions tends to increase the probability of hand-overs occurring. The partitioning algorithm we use is divided into two phases. In the first phase, a cluster representation of all states in the FSM is built. The states are clustered according to a closeness measure that is based on the size of the mutual state transition probabilities between states. In the second phase, clusters from the first phase are grouped and the FSMs are synthesized. Implementation costs, which are estimates of power and area, are the basis for selecting the final partitioned FSM.

Phase 1: *Clustering* The input to the clustering algorithm is an STG with arcs, representing state transitions, labelled with state transition probabilities. We use a hierarchical clustering scheme to build a hierarchical system of clustering representation of the states in the FSM. Hierarchical clustering is a general technique of clustering similar objects together and it has found its application in many different fields [10]. The algorithm builds a binary tree as illustrated in Figure 10.

Phase 2: *Selection of best partition* From the binary tree built by clustering, it is possible to group the clusters into a large number of combinations that are all candidates for a partitioned FSM. Cutting the cluster tree at a certain level generates the clusters. For example, in Figure 10 the cutting level can be 1, 2, or 3. Cutting at level one, for example, gives a large number of small clusters ($\{s_0\}, \{s_1\}, \{s_2\}, \{s_3\}, \{s_4\}$) and cutting at level three gives two clusters ($\{s_0, s_1, s_2, s_3\}, \{s_4\}$). For each cut-level it is also possible to perform concatenation of clusters and new combinations of clusters can be generated.

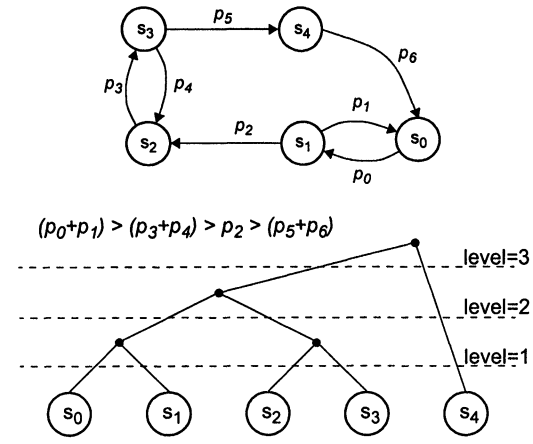


FIGURE 10 Example: hierarchical clustering.

```
partition select_partition(tree CT, real a_max) {
  real p_min ← ∞;
  partition BEST ← ∅;
  for h = 1 to H {
   clusters C ← cutlevel(CT, h)
   forall C {
    sort(C);
    partition TMP ← {c1},{c2,...,cN}
    for j = 2 to N-1 {
      partitioned_fsm PFSM ← synthesize(TMP);
      if p_min > power(PFSM) and area(PFSM) < a_max then
       BEST ← TMP;
      }
      TMP ← {c1},..,{cj}, {cj+1,...,cN};
    }
  }
  return BEST;
}
```

CT is cluster tree for the FSM.
p_{min} stores the minimum power.
a_{max} is a user constraint (max. area).
BEST stores the best partition.
TMP is the partition candidate.
PFSM is the partitioned FSM.
C is the number of clusters at a given cut-level
$c_x \in C$
N is the number of clusters in C
H is the height of the cluster tree
sort, sorts the clusters by activity, the cluster with the highest internal activity will receive the lowest index.
power and area are the HW estimation functions.
synthesize is the FSM synthesis function.
cutlevel returns the clusters in the cluster tree for a given cut-level

FIGURE 11 Procedure for selection of best partition.

Empirically, we found a procedure that for every cut-level generates a reduced number of clusters. The procedure, shown in Figure 11, takes a cluster tree and returns the partitioned FSM with the lowest power consumption for a given area constraint. In order to estimate power and area for the partitioned FSM, more details must be known about the implementation. The partitioned FSM, *i.e.*, all sub-FSMs and CCBs, are synthesized. After that, the estimation functions for circuit area and power consumption are applied. The functions for FSM synthesis and hardware estimation are described in more detailed below.

4.2. Sub-FSM Transformation

The FSM synthesis takes the clusters of states, given by the partitioning, and generates one sub-FSM for each of these clusters. The synthesis is made according to the transformation steps presented in Chapter 3. For the implementation, the transformation is divided into five steps.

4.2.1. Sub-FSM Communication

The crossing transitions in the STG (see Fig. 13b) are implemented by the CCBs, clock-gating, and structural composition of the sub-FSMs and CCBs in the partitioned FSM. Let us consider sub-FSM A^m and its associated CCB CCB^m. The function of the CCB is to control the gated clock. The activation of sub-FSM A^m is made by incoming crossing transitions. These transitions are detected by decoding the incoming transition states, denoted G^{m-}. Deactivation of A^m occurs at the same time as another sub-FSM is activated (only one sub-FSM is active at a time). The outgoing crossing transitions from A^m are detected by decoding the transition states G^m. Signals decoded from G^{m-} we call *g*-signals and signals decoded from G^m we call *d*-signals, see Figure 12. The detection of an activating crossing transition can only be made based on the transitions of the *g*-signals. The CCB behavior and implementation are shown in Figure 14.

The CCB is an AFSM that holds the one-bit state variable *e*, reflecting the state of one crossing transition. The collection of all these state

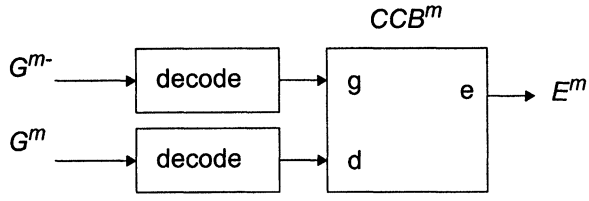
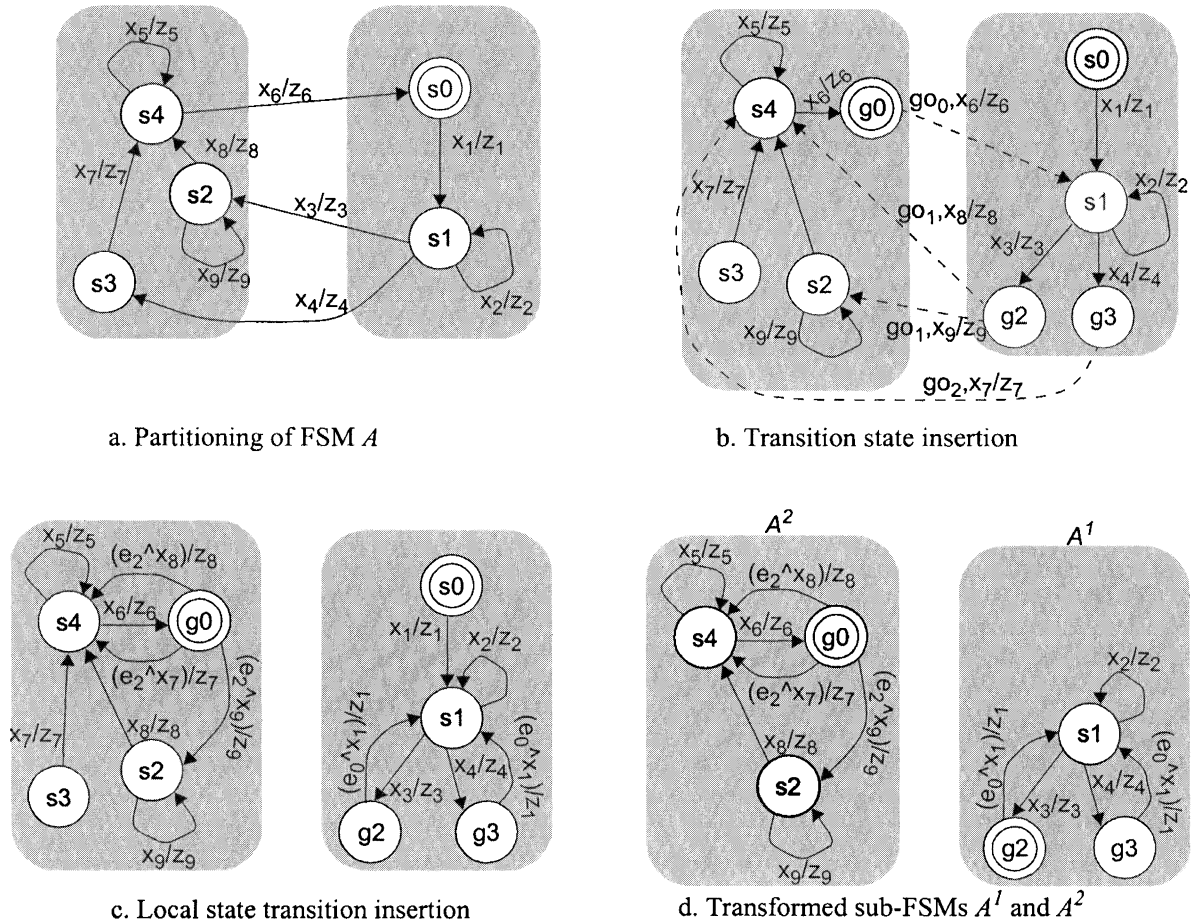


FIGURE 12 Signal interface of the CCB.



a. Partitioning of FSM A

b. Transition state insertion

c. Local state transition insertion

d. Transformed sub-FSMs A^1 and A^2

FIGURE 13 FSM transformation example.

variables gives a global state vector E. This state vector is one-hot encoded where only one bit is set high at a time. A high value indicates the last active crossing transition. E is decoded and used as input signals to the sub-FSMs.

4.2.2. *Transition-state Insertion*

The implementation architecture we use with asynchronous CCBs, see Figure 1, requires hazard-free g-signals. The g-signals must therefore be decoded from the state variable only. For example, the crossing transitions in Figure 13 are conditioned by the inputs of X. At these locations, where the crossing transition is conditioned by an input signal, the *Mealy* state transition is transformed to a *Moore* state transition. In the example given in Figure 13a, the initial machine consists of the set of states $S = \{s_0, s_1, s_2, s_3, s_4\}$ and the partitioned machine is $\pi = \{S^1, S^2\}$, where $S^1 = \{s_0, s_1\}$ and $S^2 = \{s_2, s_3, s_4\}$. For every crossing transition we insert the transition states $G(S^1) = \{g_2, g_3\}$ and $G(S^2) = \{g_0\}$. At this stage, see Figure 13b, the two sub-FSMs are still coupled by the crossing transitions, indicated by the go-signals in the STG. In the actual implementation, these transitions are handled by the hand-over mechanism that involves the CCBs and clock-gating.

A g-state is not added if it has no other outgoing transition beside the crossing transition (unconditional transition).

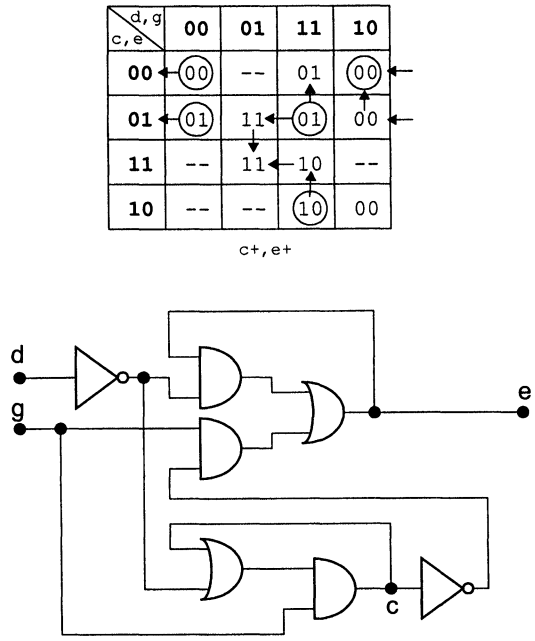


FIGURE 14 1-bit CCB, transition map and gate-level implementation.

4.2.3. Local Transition Insertion

Here, the coupled STGs will be separated. The purpose of having a separate STG for each sub-FSM is that standard synthesis procedures can be used on the STG to get to gate- level implementations. At the occurrence of a crossing transition, there is a state transition to a transition state in the active machine. As a consequence, the sub-FSM containing the destination state is activated. We know that this machine is in one of its transition states. Therefore, all transition states in combination with a global state E act as one of many possible entry states. In the example in Figure 13, the global state vector is $E=\{E^1, E^2\}$, where $E^1=\{e_0\}$ and $E^2=\{e_2, e_3\}$.

4.2.4. Removal of Unreachable States

From Figure 13c, it can be seen that some states do not have incoming transitions. These redundant states are $R^1=\{s_0\}$ and $R^2=\{s_3\}$, and their function is now, after the two previous steps, located in the transition states.

4.2.5. Setting of Initial States

Each of the sub-FSM must have an initial state. The initial state, given by the specification of the original machine, will be the reset-state of the sub-FSM in which it is located. For all the other sub-FSMs, an arbitrary transition state can be selected as the initial state, see Figure 13d.

4.3. Hardware Estimation

The objective of hardware estimation is to enable ranking of the different partition candidates so that the best partition can be selected. The ranking is based on the implementation costs in terms of power consumption and circuit area. A small number of estimation functions are used, see Table I. The parameters in these functions are based on the technology that is used, data from statistics collection, or they are empirically determined. The parameters and their values are listed in Table II.

The empirically determined parameters are related to details that are not known on the current level of abstraction. For example, the size of the output logic is not known before a gate-level implementation, and the probability for a transition in the state-register is not known before state assignment.

4.4. Library Elements

The goal has been to use a standard cell-based design methodology. The output from the tool is a structural description of the partitioned FSM consisting of sub-FSMs and CCBs. The sub-FSM description is an RT-level description that can be fed to any commercial RTL synthesis tool. The CCBs are asynchronous FSMs and standard tools do not in general support synthesis of these circuits. In our approach, we design a 1-bit CCB as a library element on the gate level. We base this

TABLE I Energy estimate functions

	Function
Total sub-FSM energy	$E_{A^n} = E_{DFF} \times (1 + k_{P,\delta}) \times \sum_{m=1}^{n} T^m \times \lceil \log_2 \lvert S^{m^*} \rvert \rceil$
Total CCB energy	$E_{CCB} = a \times E_{\text{CCB,hand-over}} + (1-a) \times E_{\text{CCB,enable}} \times (n-1) \times E_{\text{disable}}$
Output logic energy	$E_{\lambda^*} = \left(p_{\text{SC}} \times \sum_{m=1}^{n} T^m \times \lceil \log_2 \lvert S^{m^*} \rvert \rceil + a + \sum_{i=1}^{\lvert X \rvert} p_{x_i} \right) \times \lvert Y \rvert \times E_{\text{ND}_2} \times k_\lambda$
Clock net energy	$E_{\text{cknet}} = E_{\text{ck,DFF}} \times k_{\text{ck}} \times \sum_{m=1}^{n} T^m \times \lceil \log_2 \lvert S^{m^*} \rvert \rceil$
Partitioned FSM energy	$E_{A^*} = E_{A^n} + E_{CCB} + E_\lambda + E_{\text{cknet}}$
Total sub-FSM area	$A_{A^n} = A_{DFF} \times (1 + k_{A,\delta}) \times \sum_{m=1}^{n} \lceil \log_2 \lvert S^{m^*} \rvert \rceil$
Total CCB area	$A_{CCB} = A_{\text{CCB,1 bit}} \times \lvert G \rvert$
Output logic area	$A_{\lambda^*} = \left(\sum_{m=1}^{n} \lceil \log_2 \lvert S^{m^*} \rvert \rceil + \lvert G \rvert + \lvert X \rvert \right) \times \lvert Y \rvert \times A_{\text{ND2}} \times k_\lambda$
Partitioned FSM area	$A_{A^*} = A_{A^n} + A_{CCB} + A_\lambda$

CCB on gates from the standard cell library, but for improved performance the CCB can be designed on the transistor level and be included as a cell in the cell library. Various multiple input CCBs are built based on the 1-bit version. The 1-bit CCB is a controller with two bits in the state variable and can be synthesized under the *fundamental mode* assumption [16] and with *single input change* (SIC) assumption. The transition map and the gate-level solution are given in Figure 14.

In general, a sub-FSM can be activated by one of many sub-FSMs and deactivated by one of many crossing transitions leaving the sub-FSM. For this, multiple-input CCBs are needed. These are generated from the 1-bit CCB. In this way, we can avoid synthesis of complex asynchronous controllers. The extension to multiple input CCBs is shown in Figure 15. For the 1-bit CCB, the value of e can be used directly for gating the clock. For the multiple-input CCB, an additional output *e_ck* must be generated and used for gating the clock.

TABLE II Parameters in estimation functions. Units are pJ and gate equivalents for energy dissipation and area respectively

Technology			Empirical			Statistical		
	Value	Comment		Value	Comment		Value	Comment
E_{DFF}	5.20	energy in D flip-flop cell	$k_{P,\delta}$	0.2	energy in transition function in units of E_{DFF}	T^m	–	duty probability of sub-FSM m
E_{ND2}	0.96	energy in 2-input NAND cell	k_λ	0.5	degree of shared logic in output logic	a	–	probability for a hand-over
$E_{ck,DFF}$	0.15	energy in clock net per DFF	p_{sc}	0.5	probability for a state change in state register	p_x	–	probability for an input transition
$E_{CCB,1bit}$	1.62/ 0.46/0.19	CCB energy in different modes	k_{ck}	1.3	energy overhead in clock buffers			
A_{DFF}	4.1	area of D flip-flop cell	$k_{A,\delta}$	0.2	area of transition function in units of A_{DFF}			
$A_{CCB,1bit}$	2.8	area of a 1-bit CCB						
A_{ND2}	1	area of 2-input NAND cell						

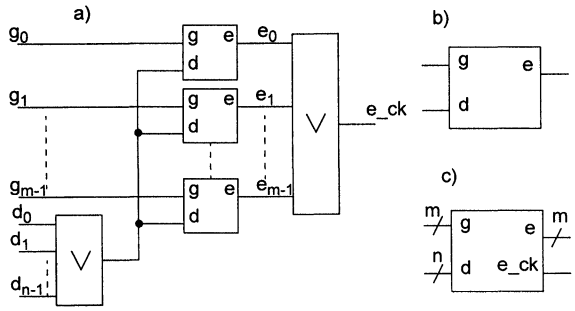


FIGURE 15 Multiple-input CCB, (a) structural composition, symbols for (b) 1-input CCB, and (c) m,n-input CCB.

As been described in Chapter 3, there are situations where we may have *static G^m-states* from a sub-FSM. This condition will prevent the transition on the *g*-signal, which is needed for triggering the CCB. In this work, we use a transistor-level solution for handling this situation. A special D flip-flop, called *GDFF*, has been designed and included in the cell library to be used in the sub-FSM state register. The *g*-input of the CCB is positive edge-triggered and to avoid the situation of static G^m-states, it must be guaranteed that all *g*-signals return to zero before assertion. With the GDFF, we can guarantee that the g-signals, which are decoded from the state register only, are zero during the first half of the clock period. The GDFF has an additional output (GQ) that is the state of the flip-flop gated with the clock signal. The normal Q-output is used for the transition function and the GQ-output is used only for decoding the g-signals. Due to uncertainties in loading conditions for the different nets between the cells after layout, a gate-level implementation of the function for GDFF may give hazardous results that cannot be accepted for signals to an edge-triggered input (*g*-input of the CCB). In Figure 16 we propose a transistor level solution of the GDFF. In order to attain a glitch-free output, it is important that the flip-flop structure has a small clock to output delay. Suitable flip-flops are based on RAM cell structures or CVSL style [17]. These flip-flops can be implemented directly and no special delay matching is required.

5. EXPERIMENTAL RESULTS

Our tool LIFS, which consists of a partitioning algorithm and a set of transformation rules, has been implemented as a software prototype tool in the Java language. LIFS, together with any

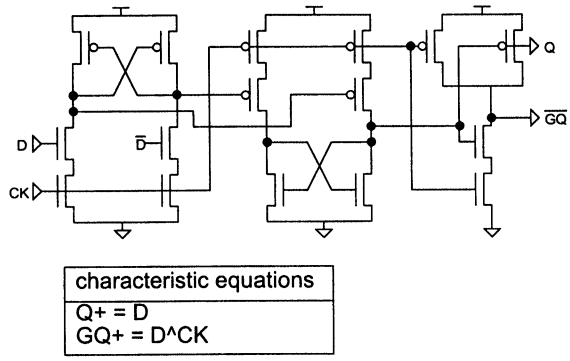


FIGURE 16 Transistor-level implementation of GDFF in static CVSL style.

standard RTL synthesis tool, forms a complete synthesis path from STG description of an FSM to its gate-level implementation, where the implementation architecture is our proposed mixed synchronous/asynchronous architecture. In order to demonstrate the effectiveness of the proposed architecture, eight of the MCNC standard benchmarks [18] were tested. The number of states in the benchmarks range from 10 to 121 states. When estimating energy consumption, the input data pattern to the circuit that is to be characterized is important. For FSMs, the sequence of the input vectors will determine the state transition probabilities and consequently determine how the FSM is partitioned. Unfortunately, typical input data is not specified by the MCNC benchmarks, which makes it difficult to compare the results with other reported works. In this work, we have set the input probability vector, used by the STG simulator in LIFS, to 0.5 for all inputs in all FSMs.

This chapter reports the experimental results from LIFS. First we illustrate the partitioning considerations by an example. We then describe the results of the structural decomposition. After that, the energy consumption and circuit area are reported separately for the sub-FSMs, output logic, and CCBs. Also, the total energy and area for the partitioned FSM are compared to its corresponding monolithic FSM implementation. Finally, we compare the timing by looking at the critical paths in the different implementations.

The energy figures were obtained from gate-level power estimations by *Power Compiler* (Synopsys) and the area estimates are based on the cell area. The timing information was obtained from static timing analysis in *Design Compiler* (Synopsys). The target technology is a 0.5 μm CMOS standard cell technology. A wire-load model, supplied by the silicon vendor for this specific library [2] has been used.

Table III shows the main characteristics of the benchmark FSMs. It describes from the top row and down, the number of inputs, number of outputs and number of states.

The first phase of the partitioning (clustering) concentrates solely on state transition probabilities when grouping the strongly connected states. From the small FSM example given in Figure 17a, we can see that states $s0$ and $s1$ have self-loops with high probabilities and they are also, in relation to other states, strongly connected. The actual solution given by LIFS for this FSM was $s0$ and $s1$ located in one sub-FSM and the rest of the states located in a second sub-FSM. But only looking at state transition probabilities says very little about the implementation costs. The number of g- and entry-states of a sub-FSM plays an important role in implementation costs. In our example, there are two entry-states, $\{s0, s1\}$ and two g-states, $\{g2, g4\}$ in A^1. Sub-FSM A^2 also has two entry-states and two g-states. An increase in the number of g-states, $|G|$, will increase the size of the transition function and may also require larger state memory in the sub-FSM. For each entry-state, an internal enable signal, defined by E, is required. The number of enable signals, $|E|$, will influence the fan-in of the logic for both the transition function and the output function, see Figure 17b. In summary, a good partition has

TABLE III Characteristics of benchmark FSMs

	bbara	dk512	ex1	keyb	styr	donfile	tma	scf
$\|X\|$	4	1	9	7	9	2	7	27
$\|Y\|$	2	3	19	2	10	1	6	56
$\|S\|$	10	14	20	19	30	24	20	121

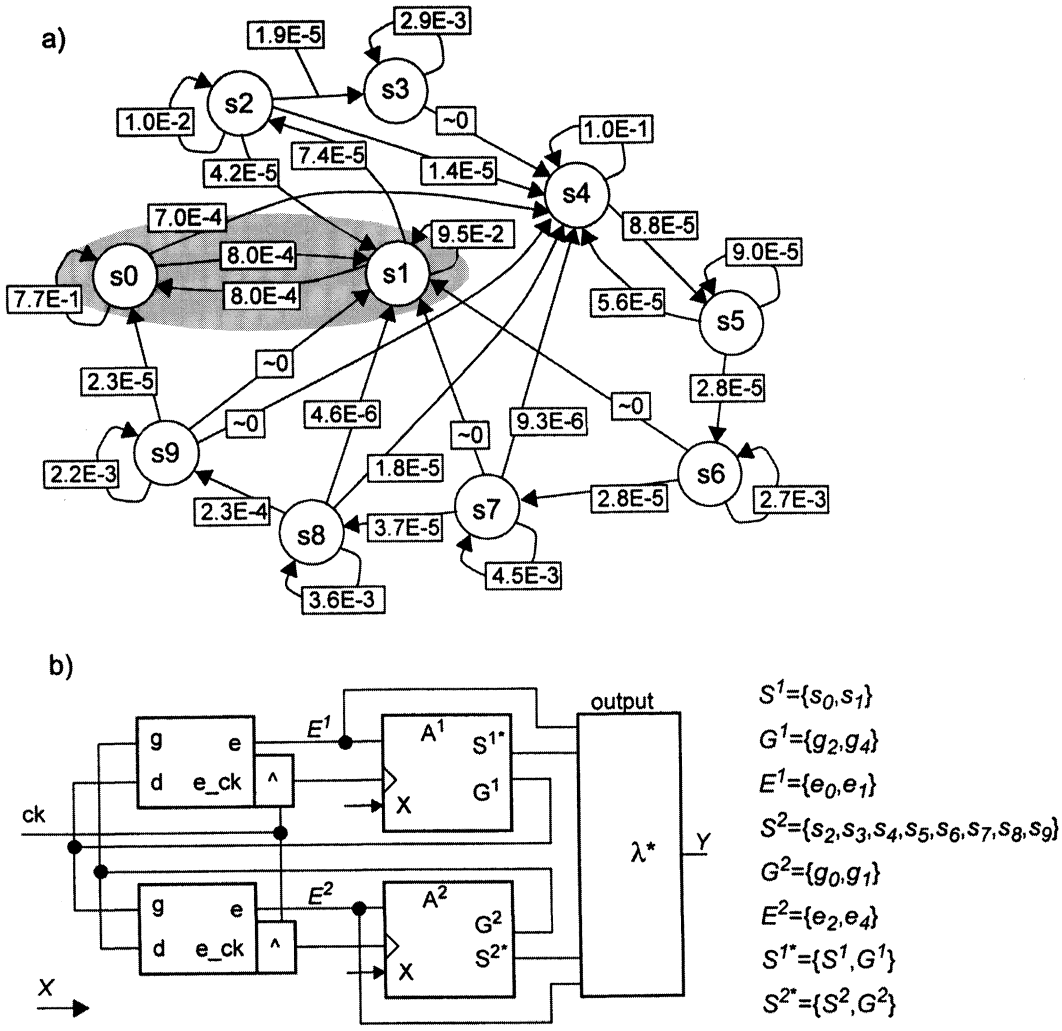


FIGURE 17 Example of partitioned FSM (bbara): (a) STG with transition probabilities, and (b) structure of the implementation.

sub-FSMs with high probabilities of state transitions within the sub-FSM and a small number of entry- and *g*-states.

Table IV presents structural information that influences the implementation costs of the partitioned FSMs. Here, T^i denotes the duty probability of the sub-FSM and $|S|$ denotes the number of original states located in the sub-FSM.

In Table V, the energy consumption and the circuit area for the power-optimized FSMs are presented. The column labelled *sub-FSM* gives the sum of energy and area for all sub-FSMs. The column labelled *output* gives the energy and area for the output function, and the column labelled *CCB* gives the sum of energy and area for all CCBs in the partitioned FSM. The column labelled *total* A^* gives the sum of energy respectively area for the three previously mentioned columns. The next three columns labelled *FSM*, *output*, and *total A*, contain energy and area for a monolithic implementation. The last column labelled *change* shows the decrease or increase in energy and area for the partitioned FSM in comparison to the corresponding monolithic implementation.

TABLE IV Structural information from the decomposition

	A^1				A^2				A^3				A^4			
A^m	$\|E^1\|$	$\|G^1\|$	$\|S^1\|$	T^1	$\|E^2\|$	$\|G^2\|$	$\|S^2\|$	T^2	$\|E^3\|$	$\|G^3\|$	$\|S^3\|$	T^3	$\|E^4\|$	$\|G^4\|$	$\|S^4\|$	T^4
bbara	2	2	2	0.87	2	2	8	0.13								
dk512	3	2	2	0.22	2	2	2	0.19	1	2	11	0.59				
ex1	4	2	2	0.38	2	2	1	0.19	1	2	1	0.19	1	2	16	0.25
keyb	1	3	1	0.72	3	1	18	0.27								
styr	1	3	1	0.64	3	1	29	0.36								
donfile	1	3	1	0.35	1	3	1	0.35	6	2	22	0.31				
tma	2	2	5	0.96	2	2	15	0.04								
sef	1	1	3	0.88	1	1	118	0.12								

TABLE V Energy consumption [pJ] and circuit area [#gate eq] for partitioned FSMs and monolithic FSM

	Sub-FSM		output		CCB	total A^*		FSM		output		total A		change		
	E_A^n	E_A^n	E_{λ^*}	A_{λ^*}	E_{CCB}	A_{CCB}	E_{A^*}	A_{A^*}	E_{fsm}	A_{fsm}	E_λ	A_λ	E_A	A_A	E	A
bbara	3.24	146	0.61	11	2.14	33	5.99	190	7.70	78	1.17	9	8.86	87	−32%	+118%
dk512	3.98	155	1.27	22	1.31	50	6.57	227	13.1	79	1.29	11	14.4	90	−54%	+151%
ex1	4.42	277	15.8	398	1.94	68	22.2	734	16.5	205	12.6	152	29.1	358	−25%	+107%
keyb	7.33	360	11.2	149	0.72	32	19.2	541	18.5	220	10.6	96	29.1	316	−34%	+71%
styr	5.54	356	7.47	178	0.72	32	13.7	566	16.8	245	11.9	169	28.8	489	−52%	+16%
donfile	7.00	318	0	0	2.00	66	9.00	384	16.0	148	0	0	16.0	148	−44%	+159%
tma	4.86	241	3.42	89	0.86	33	9.14	363	9.74	166	6.82	78	16.6	244	−45%	+49%
scf	4.54	492	4.77	259	1.03	15	10.3	766	19.6	437	11.7	201	32.1	638	−68%	+20%

For all the benchmarks we have tested, significant power reductions have been obtained. However, there is a large difference in achieved improvement for the different machines. For example, the *bbara* FSM seems to have good potential for large power reduction. Since it is small, the power overhead in sub-FSM communication becomes relatively large. For small FSMs, the area increases when partitioned. When using minimum length encoding of the states, the partitioned FSM will always require more flip-flops:

$$\lceil \log_2 |S| \rceil < \sum_{m=1}^{n} \lceil \log_2 |S^m| \rceil$$

In the case of *ex*1, we have a large sub-FSM (A^4) that is active most of the time. Further decomposition would have increased the area dramatically but with only small power savings as a result. Cases where large power reductions have been obtained are for *tma* and *scf*. Here, small clusters with high duty probabilities have been identified. For *tma*, one sub-FSM with five states is active 96% of the time while the other sub-FSM containing 15 states is only active 4% of the time. Both *tma* and *scf* are large enough so that partitioning them does not introduce excessively large area overhead.

Finally, we present the timing results in Table VI. The column labelled *sub-FSM* gives the critical path in the sub-FSM and the following column labelled *output* gives the critical path in the output function for the partitioned FSM. The next two columns labelled *FSM* and *output* contain the critical paths for the monolithic implementation. The last two columns labelled *change* shows the decrease or increase in the critical path for the partitioned FSM in comparison to its corresponding monolithic implementation.

In general, one could expect a slight increase of the delay in the output logic for the partitioned FSM. Here, the fan-in will increase with $|E|$ signals. In the case of *ex*1, where we have four sub-FSMs and $|E| = 8$, the increase of the com-

TABLE VI Timing, critical paths [ns]

	sub-FSM	output	FSM	output	change FSM	change output
bbara	6.7	1.9	6.4	1.7	+5%	+12%
dk512	6.8	2.6	7.3	1.7	−7%	+53%
ex1	9.0	17.3	8.7	7.3	+3%	+136%
keyb	9.6	12.2	9.3	9.5	+3%	+28%
styr	12.4	7.7	12.7	7.2	−2%	+7%
donfile	9.3	0	8.1	0	+15%	0
tma	8.5	7.6	10.4	7.2	−18%	+6%
scf	14.5	9.0	12.6	8.8	+15%	+2%

plexity of the logic has significant influence on the delay in the output logic. For most of the benchmarks, we can observe only small changes in the delay of the critical path of the sub-FSMs.

6. CONCLUSIONS

Clock-gating is a common approach in reducing average power consumption in finite-state machines. In this paper, we have presented an automated synthesis flow for a new type of mixed synchronous/asynchronous implementation architecture for gated clock FSMs.

The advantages of having asynchronous communication between the sub-FSMs are:

- Asynchronous controllers dissipate less power than synchronous controllers when idle.
- A more power efficient hand-over protocol for communication between the sub-FSMs can be employed.

The effectiveness of the proposed implementation architecture accompanied with the automated synthesis procedure, implemented in a software tool, has been demonstrated by the MCNC FSM benchmarks. Power reductions of up to 68% have been achieved at the cost of an increase in area of 20%. We have found these results encouraging and we also see that further improvements are possible. More attention should be given to improving the partitioning algorithm. For some of the benchmarks, we can observe large increases in both area and power for the output logic. Alternative ways of organizing the logic for the output function will be further investigated.

Acknowledgments

Financial support from the Royal Swedish Academy of Sciences, the Estonian Science Foundation Grant "Multiparadigm System on Chip Design Environment" and the Foundation for Knowledge and Competence Development are gratefully acknowledged.

References

[1] Alidina, M., Monteiro, J., Devadas, S., Ghosh, A. and Papaefthymiou, M., "Precomputation-based sequential logic optimization for low power", *Proc. of the IEEE/ACM International Conf. on Computer-Aided Design*, pp. 74–81, November, 1994.

[2] Alcatel Microelectronics, *Technology and Design Kit Documentation C05M*, 1998.

[3] Baranov, S. (1994). *Logic Synthesis for Control Automata*, Kluwer Academic Publisher, ISBN 0-7923-9458-5.

[4] Benini, L., De Micheli, G. and Vermeulen, F., "Finite-state machine partitioning for low power", *Proc. of the IEEE International Symposium on Circuits and Systems*, **II**, 5–8, August, 1998.

[5] Benini, L., Siegel, P. and de Micheli, G. (1994). "Saving power by synthesizing gated clocks for sequential circuits", *IEEE Design and Test of Computers*, **11**, 32–41.

[6] Benini, L. and De Micheli, G. (1996). "Automatic synthesis of low-power gated clock finite-state machines", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **15**(6), 630–643.

[7] Benini, L., Vuillod, P., De Micheli, G. and Coelho, C., "Synthesis of low-power selectively-clocked systems from high-level specification", *Proc. of the International Symposium on System-level Synthesis*, pp. 57–63, November, 1996.

[8] Dasgupta, A. and Ganguly, S., "Divide and conquer: a strategy for synthesis of low power finite state machines", *Proc. of the International Conf. on Computer-Aided Design*, pp. 740–745, October, 1997.

[9] Hwang, E., Vahid, F. and Hsu, Y.-C., "FSMD functional partitioning for low power", *Proc. of Design Automation and Test in Europe*, pp. 22–28, March, 1999.

[10] Johnson, S. C. (1967). "Hierarchical clustering schemes", *Psykometrika*, No. 2, pp. 241–254.

[11] Mehra, R., Guerra, L. and Rabaey, J. (1996). "Low power architectural synthesis and the impact of exploiting locality", *Journal of VLSI Signal Processing*, **13**(2, 3), 239–258.

[12] Oelmann, B. and O'Nils, M., "Asynchronous control of low-power gated clock finite-state machines", *Proc. of the IEEE International Conf. on Electronics, Circuits and Systems*, pp. 915–918, September, 1999.

[13] Oelmann, B. and O'Nils, M., "A low power hand-over mechanism for gated-clock FSMs", *Proc. of the European Conf. on Circuit Theory and Design*, pp. 118–121, August, 1999.
[14] Roy, S., Banerjee, P. and Sarrafzadeh, M., "Partitioning sequential circuits for low power", *Proc. of the 11th International Conf. on VLSI Design*, pp. 212–217, January, 1997.
[15] Sentovich, E. M., Singh, K. J., Lavagno, L., Moon, C., Murgai, R., Saldanha, A., Savoj, H., Stephan, P. R., Brayton, R. K. and Sangiovanni-Vincentelli, A. (1992). *SIS: A System for Sequential Circuit Synthesis*, Electronics Research Laboratory, Memorandum No. UCB/ERL M92/41, Department of Electrical Engineering and Computer Science, University of California, Berkeley.
[16] Unger, S. H. (1969). *Asynchronous Sequential Switching Circuits*, Wiley & Sons, Inc.
[17] Weste, N. and Eshraghian, K. (1992). *Principles of CMOS VLSI design*, A Systems Perspective, 2nd edition, Addison-Wesley Publishing Company, ISBN: 0-201-53376-6.
[18] Yang, S. (1991). *Logic Synthesis and Optimization Benchmarks User Guide*, version 3.0, MCNC Technical Report.

Authors' Biographies

Bengt Oelmann has been with Ericsson Telecom, Stockholm, Sweden, Mid Sweden University, Sundsvall, Sweden, and Nordic VLSI, Trondheim, Norway. Currently, he is an Associate Professor at Mid Sweden University. His research interests include asynchronous logic design, VLSI implementation techniques, asynchronous logic design and its application to low-power design.

Kalle Tammemäe has served the faculty of Tallinn Technical University (TTU), Estonia, since 1994. He received diploma in Computer Engineering in 1981 and Ph.D. in Engineering in 1997 from TTU, fulfilling part of the Ph.D. studies at Royal Institute of Technology, Sweden. Currently, he is a part time Associate Professor in Department of Computer Engineering at TTU and full time rector of Estonian Information Technology College. His research interests include hardware/software co-design, hardware description languages, high-level synthesis, prototyping and control intensive system design. He is a member of the IEEE Computer Society and ACM.

Margus Kruus has served on the faculty of Information Processing of Tallinn Technical University since 1980. Currently, he is a Associate Professor and head of Department of Computer Engineering, Tallinn Technical University, Estonia. His research interests include decompositional design methods of digital systems, design-for-testability methods, computer arithmetics algorithms.

Mattias O'Nils has been with Ericsson Telecom, Stockholm, Mid Sweden University, Sundsvall, and Royal Institute of Technology, Stockholm, all in Sweden. Currently, he is an Associate Professor in Electrical Engineering at Mid Sweden University. His research interest include hardware/software codesign, interface synthesis, VLSI design methods, video signal processing, and low-power design. His research has resulted in three prototype CAD tools and over 20 research papers. He is a member of the IEEE.