Synthesis of Low-Power Selectively-Clocked Systems from High-Level Specification

L. BENINI
Università di Bologna
and
G. DE MICHELI
Stanford University

We propose a technique for synthesizing low-power systems from behavioral specifications. We analyze the control flow of the specification model to detect mutually exclusive sections of the computation. A *selectively-clocked* interconnection of interacting FSMs is automatically generated and optimized, where each FSM controls the execution of one section of computation. Only one of the interacting FSMs is active for a high fraction of the operation time, while the others are idle and their clocks are stopped. Periodically, the active machine releases the control of the system to another FSM and stops. Our interacting FSM implementation achieves consistently lower power dissipation than the functionally equivalent monolithic implementation. On average, 37% power savings and 12% speedup are obtained, despite a 30% area overhead.

Categories and Subject Descriptors: B.1.2 [**Control Structures and Microprogramming**]: Control Structure Performance Analysis and Design Aids—*Automatic synthesis*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Gated clock, high-level synthesis, low power

1. INTRODUCTION

With the proliferation of portable devices, power dissipation has become an important design constraint. Numerous techniques for automatic synthesis and analysis of low-power circuits have been proposed in the academic and commercial environments [Rabaey and Pedram 1996]. In this paper we

This work is partially supported by NSF contract MIP-9421129.
Authors' addresses: L. Benini, DEIS, Università di Bologna, Viale Risorgimento 2, Bologna, 40136, Italy; G. De Micheli, CSL, Stanford University, Stanford, CA.
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2000 ACM 1084-4309/00/0700–0311 $5.00

introduce a behavioral-level technique that achieves sizable power reductions by exploiting the *mutual exclusiveness* of sections of the computations that have been defined in the literature as *basic blocks* [De Micheli 1994]. We use information at the behavioral level about *basic blocks* in the representation of a computation. We detect basic blocks that can never execute simultaneously and we generate a controller structure based on interacting finite state machines (FSMs). The interacting FSMs are then *selectively clocked*: only the part of the controller needed for the execution of the active basic block receives the clock signal. One important strength of our behavioral power reduction approach is that it exploits a careful analysis of the control flow to detect opportunities for power savings that are not apparent at lower levels of abstraction.

We consider single-process specifications based on procedural HDLs with imperative semantics (e.g. Verilog HDL, VHDL). We assume that the specification language contains the following kinds of statements: (i) assignments, (ii) conditional statements, (iii) loops, and (iv) procedure calls. Our tool for low-power controller synthesis exploits the features of the environment for the automatic synthesis of control-dominated hardware from system-level specifications presented in Ku and De Micheli [1992]. However, it can be adapted to other behavioral synthesis methodologies. We have tested our approach on benchmark behavioral specifications, obtaining in average 37% reduction in power dissipation in the controller.

1.1 Computation Model

Conditionals, loops, procedure calls, and exception-handling express *control-flow* information. We use a model similar to the *sequencing graph* [De Micheli 1994] abstraction to represent control/data-flow information. The sequencing graph is a hierarchical graph where data-flow and serialization dependencies are modeled by graphs and control-flow primitives are modeled through the hierarchy [De Micheli 1994; Ku and De Micheli 1992]. Therefore, the sequencing graph has two kinds of vertices: *operations* and *links*, the latter linking other sequencing graph entities in the hierarchy.

Vertices in the sequencing graph that are links to lower levels of the hierarchy correspond to control-flow statements. Sequencing graph entities that are leaves of the hierarchy can be basic blocks, exception-handling operations or assignments that were moved to the control flow. The *basic blocks* are groups of operations and represent pure data-flow information. Intuitively, referring to the HDL specification, a basic block is the straight-line code (sequence of assignments) within loops and conditionals. An example of HDL specification and its sequencing graph are shown in Figure 1(a) and (b).

1.2 Detecting Mutual Exclusion

The key contribution in our approach is based on the following observations:

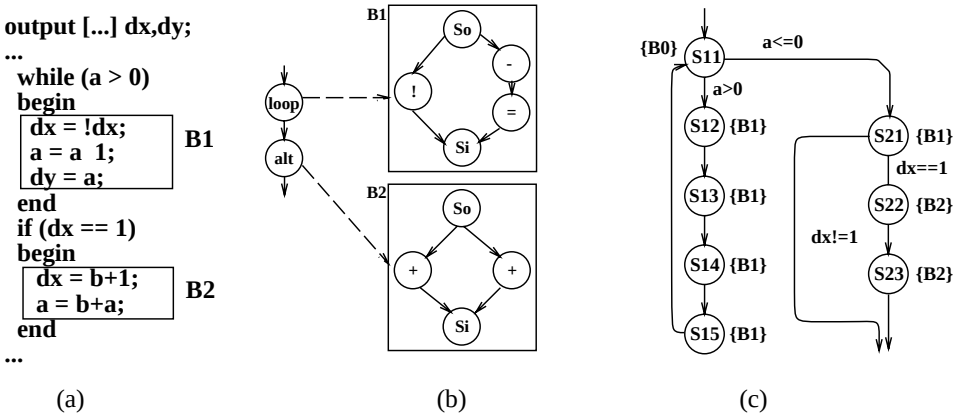

Fig. 1. Hierarchical sequencing graph of a HDL specification.

(1) A controller FSM of a behavioral specification consists of states and edges. Basic blocks are mapped into sets of states and control flow is mapped into the edges connecting the groups of states representing the basic blocks.

(2) The basic blocks of the model are disjoint, since concurrency is only allowed at the operation level, i.e., inside basic blocks.

At any given level of the hierarchy, if there is a path in the sequencing graph connecting two link vertices, the sequencing graph instances associated with the link vertices represent mutually exclusive computations. A path between two vertices in a sequencing graph implies a (transitive) functional dependency, therefore the two computations associated with the vertices cannot be executed concurrently. Moreover, in case of conditional statements, all alternative branching bodies are mutually exclusive. Using these rules we construct a mutual exclusiveness relation between basic blocks. It can be represented by a conflict graph whose vertices are basic blocks and the edges represent mutual exclusion constraints.

Our procedure generates a controller FSM and a marking of the states. A marking for a state is a set of basic block identifiers. It contains all basic blocks for which the state is a control step. Obviously, it is never the case that two mutually exclusive basic blocks appear in the marking of a state. The state transition graph of the controller for the sequencing graph of Figure 1(b) is shown in Figure 1(c). Notice the marking of the states.

2. CONTROL-FLOW-BASED STATE PARTITIONING

Our enhanced controller-generation procedure produces the state table of the controller FSM with state set S, the conflict graph C of the mutual exclusiveness relation, and a state marking M. The initial FSM is called *monolithic implementation*.

In this section we describe a two-step procedure that (i) creates a partition of the states of the monolithic implementation and (ii) clusters

blocks of the partition until a user-specified granularity is reached. The procedure is the key operation required to generate the low-power interacting FSM architecture that is our final target.

2.1 Building the Initial Partition

The partitioning algorithm takes as inputs the state set, the state marking, and the conflict graph as inputs and produces a state partition as output. Formally, a partition is defined as a collection of k sets (i.e., a set of sets) $\Omega(S) = \{P_1, P_2, \ldots, P_k\}$ such that $P_i \cap P_j = \emptyset$ for $i \neq j$, and $\bigcup_{i=1}^{k} P_i = S$.

Starting from any state `s`, the algorithm inserts in set `root` all states that are connected to `s` and whose marking is not mutually exclusive with any marking of states already in `root` (two markings are mutually exclusive when there is at least one basic block in the first that is mutually exclusive with one basic block in the second, and vice versa). When set `root` cannot be expanded due to mutual exclusion conflicts with all states reachable from states in `root`, it is inserted in the collection of sets $\omega(S)$. Then a new state is selected among those never included in any *root* and the process is repeated. The algorithm terminates when all states have been inserted in one of the `root` sets. The complexity of the algorithm is $O(n_s^2)$ where n_s is the number of states in the monolithic FSM.

Example 1. Referring to Figure 1(c), assume that initially `root` = $S12$. Its neighbor states are $S11$ and $S13$. The marking of $S11$ ($\{B0\}$) is mutually exclusive with the marking of $S12$ ($\{B1\}$), therefore $S11$ is not included in `root`. On the other hand, $S13$ is compatible with $S11$, and it is included in `root`. The inner loop of the algorithm inserts $S14$ and $S15$ in `root` and terminates, since no more compatible states can be reached. Then, one of the remaining states ($S11$, $S21$, $S22$, and $S23$) is selected and the outermost loop continues. The partitioning algorithm produces a partition Ω with four sets. $\Omega(S) = \{\{S12, S13, S14, S15\}, \{S11\}, \{S21\}, \{S22, S23\}\}$.

2.2 Clustering

There are strong practical reasons for obtaining a coarse-grain partition: decomposition involves some tradeoff between size of the components and overhead due to the interaction between components. The larger the number of components, the larger the overhead for interfacing them. The procedure described in the previous section may generate an excessively fine-grained partition Ω. We present a procedure that *clusters* sets in the partition until a specified granularity is achieved. We attempt to produce a partition with the following two characteristics: (i) the partition blocks should be as balanced as possible and (ii) the probability of having a transition between partitions is minimized. The reason for these choices

will become clear in Section 3, when the FSM decomposition technique and its power saving will be analyzed in detail.

Given $\Omega(S) = \{P_1, P_2, \ldots, P_k\}$ and the desired number of partition blocks $n < k$, we order the blocks P_i for decreasing cardinality. The first n blocks are selected as *attractors*. The name stems from the fact that the attractors will attract smaller blocks and merge with them, until only attractors remain. After the attractors have been chosen, smaller blocks are considered, starting from the smallest one, say P_l. Block P_l is merged to one of its neighbor attractors (i.e., attractors containing at least one state connected to one of the states of P_l). The choice among the attractors is done using the following *affinity* function $\mathcal{A}$:

$$\mathcal{A} = KAvg_{attr}/(|P_q| + |P_j|) + Prob(P_l \rightarrow P_q \mid P_q \rightarrow P_l) \quad (1)$$

where $|P_q|$ is the cardinality of attractor q, $|P_j|$ is the cardinality of block j, Avg_{attr} is the average cardinality of the attractors, and $Prob(P_l \rightarrow P_q \mid P_q \rightarrow P_l)$ is the probability of a transition between a state in P_l and a state in P_q,, or vice versa. K is a constant for controlling the relative weight of the two contributions. The edge probabilities can be computed by simply simulating the FSM and measuring the frequency of state transitions. If neither simulation patterns nor input probabilities are available, all edges can be assumed to be equiprobable.

The affinity function expresses the tradeoff between balancing the size of the attractors and minimizing the probability of a transition between two partitions. The first term in Eq. (1) decreases with the size of the attractor (larger attractors are penalized). The second term increases with the probability of a transition between the attractor and P_l (i.e., blocks with high transition probability have higher affinity).

Given P_l and its attractors, P_l is merged with the attractor that the has highest affinity. Ties are broken by random choice. The process terminates when only attractors remain in the partition. The final partition, $\Pi(S)$, is guaranteed to have n blocks. We clarify the clustering procedure through an example.

Example 2. The clustering algorithm is applied to the FSM of Figure 1(c). First, $n = 2$ is specified. The initial partition is $\Omega(S) = \{\{S12, S13, S14, S15\}, \{S11\}, \{S21\}, \{S22, S23\}\}$. The two larger blocks become attractors. They are shown in Figure 2(a). Then one of the two smaller blocks is chosen (randomly, since they have the same size). Suppose that $\{S21\}$ is chosen first. It has only one neighbor attractor, $\{S22, S23\}$, and it is merged with it.

Then, $\{S11\}$ is considered. It has 2 attractors, as shown in Figure 2(b). Assume that $Prob(S11 \rightarrow S21) = .01$, $Prob(S11 \rightarrow S12) = .1$ and $K = .1$. The affinity of $\{S11\}$ with the larger attractor is $\mathcal{A}_1 = .1(7/2)/(4 + 1) + .1 = .17$. The affinity with the smaller attractor is $\mathcal{A}_2 = .1(7/2)/(3 + 1) + .01 = .0975$. Since $\mathcal{A}_1 > \mathcal{A}_2$, $\{S11\}$ is merged with the largest attractor.

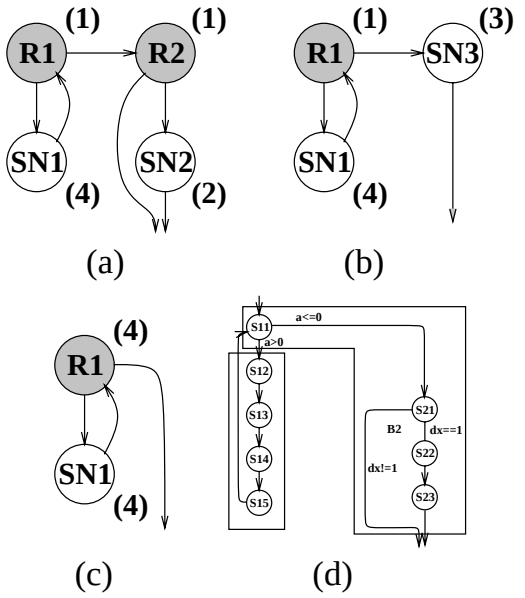

Fig. 2. Partitioning the FSM controller.

The final partition is shown in Figure 2(c): $\Pi(S) = \{\{S11, S12, S13, S14, S15\}, \{S21, S22, S23\}\}$.

For structured HDL specifications, the initial partition $\Omega(S)$ enjoys an important property. Given a partition block P_i, all transition in the FSM from states in blocks $P_j \neq P_i$ reach one single destination state $s_{entry} \in P_i$. We call this distinctive characteristic *single entry point property* (SEPP, for brevity) and the unique destination state is called *entry state*. SEPP is key for obtaining a low-power decomposition of the controller. The basic clustering algorithm does not ensure preservation of the SEPP, but it can be modified to do so. Due to space limitations we do not provide details on the extended algorithm.

3. CONTROL STRUCTURE BASED ON COMMUNICATING FSMS

We implement the controller as a network of communicating FSMs, one for each set of states in $\Pi(S)$. The decomposed FSM Φ is a set of n FSMs $\Phi = \{F_1, F_2, \ldots, F_n\}$. We call *submachines* the FSMs $F_i \in \Phi$. Submachines communicate through additional control signals, called *go* signals. For each submachine, a new *reset* state is defined.

A submachine can exit the reset state only upon assertion of a *go* signal by another submachine. At any given clock cycle, only two situations are possible: (i) one submachine is performing state transitions and all other submachines are in reset state and (ii) one submachine is transitioning toward its reset state, while another one is leaving it.

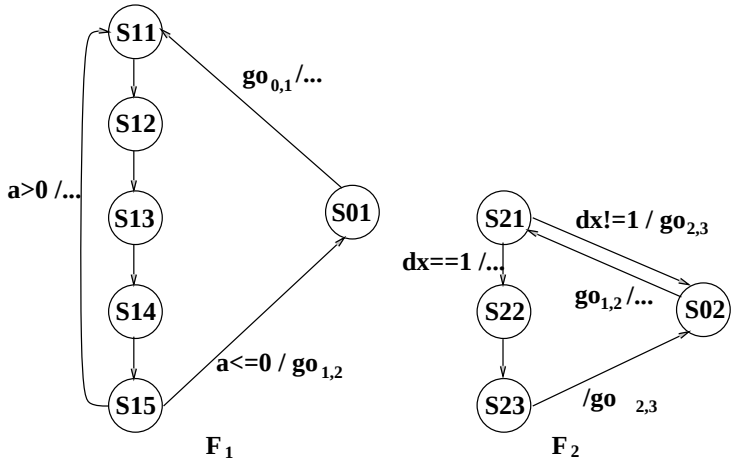

Fig. 3. Interacting FSMs implementation.

Note that the *go* signals are *additional* inputs and outputs. All inputs and outputs of the monolithic FSM are unchanged in the submachines. If an edge $s \rightarrow t$ of the monolithic machine has head and tail state included in submachine F_i, the edge is replicated in F_i, with the same input and output fields. Edges in the global FSM-connecting states that belong to different partitions are associated with edges representing transitions to and from the reset states of the corresponding subFSMs. These transitions are labeled as follows: (i) edges toward reset have the same input field as the original edge, an additional output *go* (set to 1) and all original outputs set to 0); ii) transitions leaving reset have only one specified input *go* and the same output field as in the original transition edge of the monolithic FSM. The outputs of a submachine blocked in reset state are zero.

Example 3. Consider the monolithic controller of Figure 1(c). Its decomposition in interacting FSMs induced by the clustered graph of Figure 2(c) is shown in Figure 3. For the sake of clarity, we do not show the complete input and output fields of the FSMs. The transition from $S11$ to $S12$ is taken only if $a > 0$. If the transition is taken, the additional output $go_{1,2}$ is asserted. The signal $go_{1,2}$ is an input for the F_2 that exits from $S02$ upon its assertion. The figure shows other activation signals for interfacing with subFSMs 0 and 3 (not shown).

Since we assumed that the single entry point property holds, the number of incoming *go* signals for each subFSM is reduced to the number of subFSMs connected to it. Thus the number of *go* signals is bounded from above by $n(n - 1)$, but is usually much smaller than that. The rationale behind this important property is that if there is an edge between submachines F_i and F_j, and F_j has a single entry point, it does not matter from which particular state of F_i the *go* signal was issued (because the starting state for the operation of F_j is always the same).

When the SEPP does not hold, the overhead for decomposing the monolithic controller may drastically increase. In the worst case, we may need as many go signals as there are edges between states belonging to different partition blocks. Hence, the quality of the decomposed implementation may be notceably reduced.

3.1 Clock-Gating

In the interacting FSM system, most of the machines F_i remain in state $s_{0,i}$ during a significant number of cycles. If we stop their clocks while they stay in reset state, we save power (in the clock line and in the FSM combinational logic) because only a part of the system is active and has significant switching activity. If clock-gating is not allowed in the design flow, less aggressive techniques such as those in Chow et al. [1996] can be used. To be able to stop the clock, we need to observe the following conditions:

—The condition under which F_i is idle. It is true that when F_i reaches the state $s_{0,i}$ we use the Boolean function $is\_in\_reset_i$ that is, 1 if F_i is in state $s_{0,i}$, 0 otherwise.

—The condition under which we need to resume clocking, even if the subFSM is in reset state. This happens when the subFSM receives a go signal and must perform a transition from s_{0i} to any other state.

We can derive the following activation function (in negative logic), F_{ai}. The clock is stopped when $F_{ai} = 1$.

$$F_{ai} = is\_in\_reset_i \wedge \overline{\left(\bigvee_{F_i, F_j, F_i \neq F_j} go_{j,i} \right)} \tag{2}$$

The first term $is\_in\_reset_i$ stops the clock when the machine reaches $s_{0,i}$. The second term ensures that clocking resumes when one of the $go_{j,i}$ is asserted and the subFSM must exit the reset state. The disjunction is extended over all subFSMs that have at least a transition to F_i. This activation function allows the newly activated subFSM to have its first active cycle during the last cycle of the previously active FSM. The two subFSMs make a transition in the same clock cycle: one is transitioning to its idle state and the other from its idle state. The local clocks of F_i and F_j are both active. We call *transitions of control* the cycles when a subFSMs shuts down and another activates. Each local clock of the FSMs F_i is controlled by a clock-gating block, taken from Benini and De Micheli [1996].

Power savings depend on the choice of the state partition $\Pi(S)$. A good partition minimizes the probability of transitions of control (during which two machines are clocked and dissipate power) and has a reduced interface overhead (i.e, a small number of interface signals). Preserving SEPP is useful in reducing the number of interface signals.

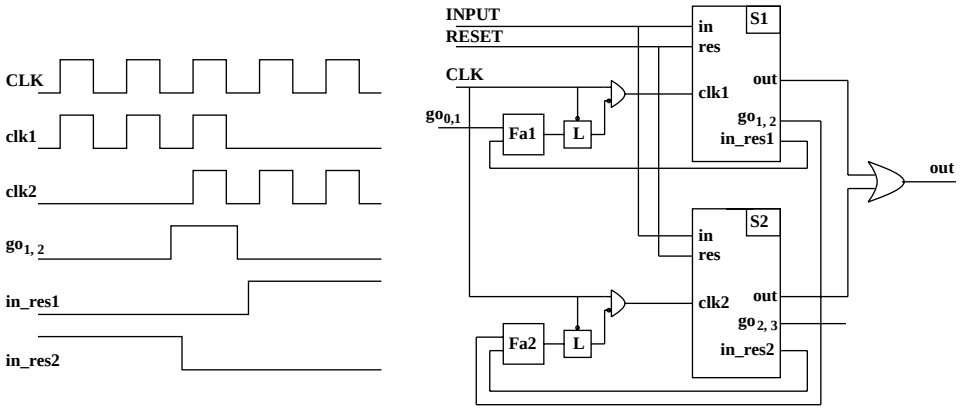

Fig. 4. Gated-clock implementation of the interacting FSMs.

Example 4. The gated-clock implementation of the interacting FSMs of Figure 3 is shown in Figure 4. Notice how the external output is obtained by OR-ing the outputs of the subFSMs. This can be done because we specified that when a subFSM is in reset state, all its output signals are zero, thus the only possible source of controlling 1 values for the outputs is the active subFSM.

Figure 4 also shows the clock waveforms, the *in_reset* signals, and the *go* signals. Note that there is a clock cycle for which both local clocks are enabled. The waveforms show how subFSM 1 is deactivated and subFSM 2 activates, thanks to the assertion of the $go_{1,2}$ signal.

4. IMPLEMENTATION AND RESULTS

We tested our decomposition procedure on a set of control-dominated benchmarks. All these designs were originally specified in C or Verilog. XFE, XF, and XB are the controllers of the transmission unit of an Ethernet coprocessor. These controllers are, respectively, the frame transmitter with exception handling (XFE), without exception handling (XF), and the bit transmitter (XB). DRAM is the controller for a PCI bus protocol conversion to a DRAM protocol. We show the results for DRAM with two and four partitions. The remaining two benchmarks consist of a FIFO queue controller and the speed control unit for an automobile (SPDCNT).

We performed power estimates with PPP [Bogliolo et al. 1996] (an accurate gate-level power simulator). We compared the circuit obtained from the original specification implemented as a single FSM and from the interacting FSMs with clock-gating. We applied a large number of randomly generated vectors for power simulation. Since we simulated the controller in isolation, we used the uniform signal probability of 50% for all input signals.

The results are shown in Table I. We display the original number of states, the number of partitions, the power consumption with the original system and the communicating system. The “ratio” column gives the ratio

Table I. Results on HLS Benchmarks

			Avg pow (μW)			Area ratio		
Bench	NS	NP	orig	inter	ratio	seq	comb	tot
XFE	177	5	2558	1700	0.66	3.5	1.49	1.64
SPDCNT	56	4	3075	1211	0.39	2.83	0.85	1.02
DRAM	34	4	1562	770	0.49	2.5	1.09	1.28
DRAM	34	2	1562	943	0.60	1.67	1.01	1.09
XF	27	2	811	516	0.63	1.6	1.16	1.26
fifo	24	2	1517	1337	0.88	1.60	1.44	1.46
XB	20	3	875	598	0.68	2	1.35	1.47

of power consumption with the original description and the gated description. We obtained 37% reduction of power consumption on average, much higher power reductions for systems with high locality (i.e., systems where one of the interacting FSMs is running most of the time).

We measured the critical path for the monolithic and the decomposed implementation. Interestingly, the critical path decreases on average by 12% in the decomposed implementation. Our tool does not optimize for speed, hence such speedup is somewhat unexpected. An intuitive explanation for this phenomenon is that our decomposition can be seen as a form of state assignment based on a wide encoding that requires a number of flip-flops larger than the minimum one. Wide encodings are often used to increase the speed of controllers, with a price in area.

The last three columns of the table reflect the area overhead. The figures are the ratio of the communicating FSMs area over the monolithic FSM area, in terms of mapped cells. We emphasize the combinational area and the sequential area. The area overhead in combinational logic is mainly due the additional logic on the outputs (OR gates) and the activation functions. The area of the combinational logic can even be reduced, thanks to the simplification introduced by the partition. The SPDCNT area decreases by 15%. This combinational overhead is approximatively 20% on average.

The sequential overhead comes from the duplication of the state information. This overhead is significant, but it depends on the number of partitions. For a 4-partition, the area of the sequential part is increased by more than a factor of 2. For a 2-partition, the sequential overhead is around 1.6. Note, however, that we use minimum-length state encoding for all FSMs. For different encoding styles (such as one-hot encoding) the sequential overhead is substantially reduced. The total area increase is on average around 30%.

The computation time is dominated by the FSM synthesis step. This result is quite expected, since our decomposition and clustering algorithms have polynomial complexity in the number of states, and in a few seconds complete all FSMs in our experiments, while FSM synthesis with commercial tools can take a few hours. Interestingly enough, for our largest examples, the synthesis of all interactive FSMs is faster than the synthesis of the single FSM implementation, even considering the overhead for generating the partitions.

5. CONCLUSIONS

We propose an approach to power minimization at the behavioral level on the basis of an interacting FSM implementation with selective shutdown. Selective shutdown is obtained by gating the clock of the state registers for inactive subFSMs. On average we achieved a 37% power reduction and 12% speedup, despite a 30% increase in area. The run-time of the decomposition algorithm is negligible with respect to the time spent on logic synthesis and optimization. We obtained sizable improvements at a low computational cost because power minimization is targeted in the early stages of the design process. Power optimizations techniques at the FSM level (such as state assignment) and at the gate level can be plugged in during this process to get even better results.

ACKNOWLEDGMENTS

The authors wish to thank Claudionor Coelho for is help on an earlier version of this paper.

REFERENCES

BENINI, L. AND DE MICHELI, G. 1996. Transformation and synthesis of FSMs for low power gated clock implementation. *IEEE Trans. Comput.-Aided Des. Integr. Circuits 15*, 6 (June 1996), 630–643.

BOGIOLO, A., BENINI, L., AND RICCÒ, B. 1996. Power estimation of cell-based CMOS circuits. In *Proceedings of the 33rd Annual Conference on Design Automation* (DAC '96, Las Vegas, NV, June 3–7), T. P. Pennino and E. J. Yoffa, Eds. ACM Press, New York, NY, 433–438.

CHOW, S.-H., HO, Y.-C., HWANG, T., AND LIU, C. L. 1996. Low power realization of finite state machines—a decomposition approach. *ACM Trans. Des. Autom. Electron. Syst. 1*, 3, 315–340.

DE MICHELI, G. 1994. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., New York, NY.

KU, D. AND DE MICHELI, G. 1992. *High-Level Synthesis of ASICs Under Timing and Synchronization Constraints*. Kluwer Academic Publishers, Hingham, MA.

RABAEY, J. AND PEDRAM, M. 1996. *Low Power Design Methodologies*. Kluwer Academic, Dordrecht, Netherlands.

Received: October 1997; accepted: February 1999