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

L. Benini P. Vuillod G. De Micheli
Computer Systems Laboratory
Stanford University
Stanford CA 94305-9030

Claudionor Coelho
Department of Computer Science
University of Minas Gerais (UFMG)
Brazil

Abstract

In this paper we propose a technique for synthesizing low-power systems from a high-level specification. We analyze the control flow of the specification 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 at any given clock cycle, while all the others are idle and their clock is stopped. Our interacting FSM implementation achieves consistently lower power dissipation than the functionally equivalent monolithic implementation. In average 37% power savings are obtained with a 30% area overhead.

1 Introduction

Power dissipation is an important design constraint and numerous techniques for automatic synthesis and analysis of low-power circuits have been proposed in the academic and commercial environment [2, 3]. With the acceptance of behavioral synthesis [4, 10, 12, 8, 9] and the appearance of commercial products [6, 7] in this area, a significant effort has been dedicated to extending low-power synthesis techniques to the behavioral level of abstraction. Early work focused on the minimization of supply voltage under throughput constraints [16]. Unfortunately, designers cannot always choose arbitrarily choose the optimal supply voltage. These methods, albeit very effective, are limited in scope to full-custom designs and a restricted class of applications where power supply voltage is a design parameter.

More recently, another class of behavioral synthesis methodologies for low power has emerged. These techniques are more general in scope, because they do not require control on the power supply voltage. Power reduction is obtained by minimizing the *switching activity* of the circuit. Resource allocation techniques have been proposed [15, 16] to increase the temporal correlation (thereby reducing switching) on the inputs of registers and data-path units. While the approaches presented in [15, 16] optimize the resource allocation after scheduling, in [14, 18] both scheduling and resource allocation algorithms are modified to take into account transition activity. Again, the key idea is to increase the correlation between data stored in registers, consumed by operators or transmitted on busses during successive clock cycles.

Other techniques focus on the reduction of *useless* switching activity, by means of selective shutdown and clock control [15, 17]. In these works the main insight is that when a unit or a register is not used (i.e. the reservation table of the unit has an empty slot) during a control step, we can stop the clock or disable the loading of unneeded data. A different approach to selective shutdown is taken in [19]: the scheduling algorithm can be modified to exploit the *slack* in the schedule and pre-compute multiplexer control signals. If the control signals can be computed before the minimum start time of the operations whose results are multiplexed, we can use the control signals to completely stop the resource performing the unneeded operation.

In this paper we introduce a control-flow technique that achieves sizable power reductions by exploiting the *mutual exclusiveness* of sections of the computation. 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 detects idle conditions that are not apparent at the lower levels of abstraction. Moreover, the knowledge about mutual exclusion conditions in the control flow can be fruitfully exploited for scheduling and resource allocation in the data flow. In this work we focus on the syn-

0-8186-7563-2/96 $5.00 © 1996 IEEE

thesis of low-power controllers from high-level specification, but we are currently working on using the control-flow information for low-power data-path synthesis.

Our tool for low-power controller synthesis leverages the synthesis system presented in [5]: an environment for the automatic synthesis of control-dominated hardware from system-level specifications, but it can be adapted to other high-level synthesis methodologies. We have tested our approach on benchmark high-level specifications obtaining in average 37% reduction in power dissipation in the controller.

2 Behavioral synthesis

We consider a specification style based on procedural HDLs with imperative semantic (e.g. Verilog HDL, VHDL). The hardware model contains four kinds of statements: i) assignments (with complex operators), ii) conditional statements, iii) loops, iv) procedure calls. All conditionals, loops and procedure calls can be nested. For the sake of simplicity, we consider hardware models represented by a single process.

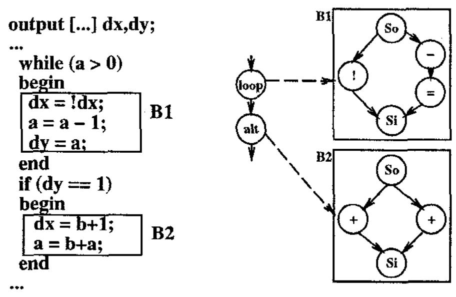
2.1 Computation model

Conditionals and loops express *control flow* information, while the assignments represent the *dataflow*. We use the *sequencing graph* [11] abstraction to represent the control/data flow information. The sequencing graph is a hierarchical graph where control-flow primitives are modeled through the hierarchy, whereas dataflow and serialization dependencies are modeled by graphs [4, 11].

The sequencing graph has two kinds of vertices: *operations* and *links*, the latter linking other sequencing graph entities in the hierarchy. Sequencing graph entities that are leaves of the hierarchy are called *basic blocks* and represent pure dataflow. Intuitively, referring to the HDL specification, a basic block is the straight-line code (sequence of assignments) within a loop or a conditional. Notice that a sequencing graph defines the computation as a set of operations (vertices) and a partial order among them (edges). The partial order does not depend on the order of the statements in the HDL code, but it represents data dependencies.

Vertices in the sequencing graph that are links to lower levels of the hierarchy correspond to control-flow statements. Branching is modeled by associating a sequencing graph entity with each branching body and a link vertex with the branching clause. Iteration is modeled by associating a sequencing graph entity with the body of the loop and a link vertex with the iteration clause.

Example 1 *Figure 1 shows a HDL specification and its sequencing graph. The vertices* `loop` *and* `alt` *are link vertices leading to lower levels of the hierarchy. They are associated with the control flow and implicitly represent the evaluation of the loop and branch clause, respectively. The dashed edges represent the hierarchy: two dataflow leaves of the hierarchy are represented. Notice that every sequencing graph has two dummy vertices called* `source` *and* `sink`.



```
output [...] dx,dy;
...
 while (a > 0)
 begin
  dx = !dx;
  a = a - 1;        B1
  dy = a;
 end
 if (dy == 1)
 begin
  dx = b+1;         B2
  a = b+a;
 end
...
```

Figure 1. Hierarchical sequencing graph of a HDL specification

2.2 Detecting mutual exclusion

The key contribution in our approach is to realize that the link vertices (i.e. the control flow constructs) allow us to identify *mutually exclusive* sections of the computation. We define two sequencing graph instances G_1 and G_2 to be mutually exclusive when the operations of G_1 *cannot be concurrently executed* with any operation in G_2.

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. Remember that a path between two vertices in a sequencing graph implies (transitive) functional dependency, therefore the two computations associated with the vertices cannot be concurrently executed. Moreover, in case of conditional statements, all alternative branching bodies are mutually exclusive.

The synthesis process requires one to schedule the sequencing graph, to determine resource allocation and to generate a controller FSM. The typical structure of a FSM generated by synthesis has a natural decomposition that closely matches mutually exclusive sections of the computation. Notice that scheduling introduces several additional mutual exclusion conditions: resource constraints translate directly into mutual exclusiveness of vertices in the sequencing graph. In this work we focus on the constraints created by the control flow. The rationale for this choice is that we look for a *coarse-granularity* decomposition, and control-flow induced decompositions have relatively few components.

Example 2 *The the state transition graph of the controller for the example in Figure 1 is shown in Figure 2. We assume that the schedule of the loop (including the test of the loop conditional) takes 5 clock cycles and the schedule of the*

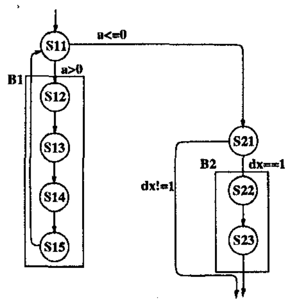


Figure 2. Controller for the HDL specification

body of the branch (plus the test of the conditional) takes 3 clock cycles. Since there is a direct path from the `loop` *and the* `alt` *vertices in Figure 1, the sequencing graphs associated with the two link vertices represent mutually exclusive computations. The FSM is divided in two components corresponding to the basic blocks $B1$ and $B2$ shown in Figure 1. In particular notice that the exits and entry points of a block correspond to the branch and loop clauses.*

2.3 Control-flow based state partitioning

After scheduling and resource allocation, the FSM for the controller is extracted from the sequencing graph (*controller generation*). We call the initial FSM *monolithic controller implementation.* In this sub-section we describe a three-step procedure that creates a partition of the controller's states. This is the key information required to generate the interacting FSM architecture which is our final target. The steps are: i) marking, ii) collapsing, iii) clustering. We outline each of them in sequence.

Marking of the controller's states is the first step. During controller generation, we mark states associated with i) loop clause evaluations, ii) procedure calls, iii) return from procedures, iv) conditional evaluations and v) conditional joining points. The marking is a straightforward procedure because the sequencing graph is available, thus the high-level information on the control flow is still fully explicit in the hierarchy of the link nodes. Marked states are called *roots* of the decomposition. Notice that a single state may be a multiple root: for example, a state may correspond to a conditional join and to a loop clause evaluation as well (if they are scheduled in the same control step). Conversely, a single link node of the sequencing graph may induce the marking of multiple roots: consider for example the case of a partially unrolled loop.

Collapsing is a transformation performed on the marked FSM. It transforms the state transition graph into a weighted graph. The sequences of unmarked states between roots are merged in *simple nodes*. Each simple node is assigned a weight equal to the number of FSM states collapsed into it. The roots have weight equal to one. The subsequent steps of the algorithm are based on the observation that the roots have multiple outgoing incoming and/or multiple outgoing edges, while the simple nodes have a single incoming and outgoing edge.

Clustering is the third phase of the algorithm. Simple nodes with small weight are chosen first for clustering. The selected nodes are merged with their fan-in roots (i.e. roots with edges directed to the nodes) and the weight of the roots is augmented by the weight of the merged nodes. Clustering is carried on until the number of nodes left in the graph is equal to a user-specified number (the number of desired sub-machines in the decomposition). Notice that roots in the original unclustered graph may become simple nodes after one or more clustering steps.

The states clustered within each node in the final graph are the state sets of the sub-machines in the FSM decomposition. We call $\Pi(S) = \{S_1, S_2, ..., S_n\}$ the partition of the state set S of the original FSM. The rationale behind the clustering algorithm is to induce a balanced decomposition of the monolithic controller, where the component FSMs have similar number of states. The complete description of the clustering algorithm is involved, because several particular cases have to be considered and many heuristic techniques can be employed to direct the choice of which nodes are clustered first. We do not describe the algorithm in full details for space reasons.

The partition $\Pi(S)$ has two important properties. First, it is based on the natural hierarchy induced by the control flow in the sequencing graph. Second, and most important, all edges entering a node of the graph correspond to edges in the FSM which have a single destination state. We call this distinctive characteristic *single entry point property*. The *single entry point property* is fundamental for an effective low-power implementation based on selectively-clocked interacting FSMs, because it corresponds to a minimum number of additional signals to be generated in the final implementation. Unfortunately, there are correct HDL programs for which it is impossible to generate partitions with *single entry point*, namely programs with *goto*s and exceptions (*unstructured programs*). Thus, we need to guarantee correct (but sub-optimal) behavior even if the property does not hold. Intuitively, the root-based partition of controllers extracted by structured HDL programs have natural decompositions that lead to lower-power implementations, because structured code has better locality and mutually exclusive sections with coarser granularity.

Example 3 *Consider the controller of Figure 2. Among the states shown, S11 and S21 are roots and are identified exploiting the information in the sequencing graph (they correspond to link nodes). States S12 to S15 are collapsed into on simple node SN1. States S22 and S23 are collapsed as well in the simple node SN2. After marking and collapsing the initial FSM is transformed in the weighted graph shown in Figure 3 (a). Shaded nodes are roots (R1 and R2). The weight of the nodes is shown between parentheses.*

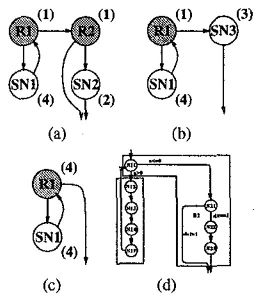


Figure 3. Marking, collapsing and clustering steps

Clustering is performed on the graph. Two successive steps are shown in Figure 3 (b) and (c). Observe that root R2, after the first collapsing with SN2 becomes a simple node SN3. Finally, the FSM decomposition induced by clustering is shown in Figure 3 (d).

3 Control structure based on communicating FSMs

The algorithm described in the previous section produces a partition $\Pi(S)$ of the FSM states. We implement the controller as a composition of communicating FSMs, one for each set of states in $\Pi(S)$. The specialized clocking scheme we use guarantees that only one communicating FSM is active at a given time (with an exception to be clarified later).

The FSM partition can be formally described as follows. Let F = $(X, Y, S, s_0, \delta, \lambda)$ be the original monolithic control FSM, where X is the input vector, Y the output vector, S the set of states, $s_0 \in S$ the reset state. The next state function is $s_{t+1} = \delta(x_t, s_t)$ and the output function is $y_t = \lambda(x_t, s_t)$. S has been partitioned in n sets. Our approach is to generate n interacting sub-FSMs with combined input/output behavior functionally equivalent to the monolithic controller F. For a set of states S_i in $\Pi(S)$, the corresponding sub-FSM F_i is created as follow.

- All states in S_i are included in F_i, with all transitions among them.
- A new reset state $s_{0,i}$ is added to F_i.
- For each transition from a state p of F_i to a state q of F_j $(i \neq j)$ a new signal $go_{p,q}$ is generated: $go_{p,q}$ is an input signal for machine F_j and an output signal for machine F_i.
- Every transition from p to q becomes a transition from p to $s_{0,i}$ in F_i and a transition from $s_{0,j}$ to q in F_j.
- The transition $p \longrightarrow s_{0,i}$ asserts the output signal $go_{p,q}$ (of F_i). The transition $s_{0,j} \longrightarrow q$ is performed only when the input (of F_j) $go_{p,q}$ asserted.

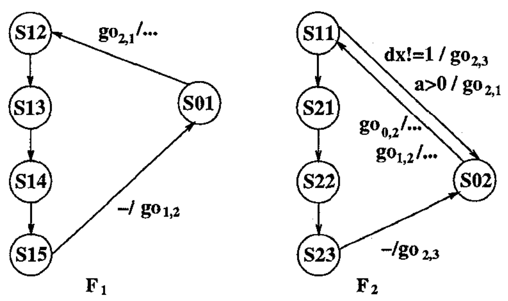


Figure 4. Interacting FSMs implementation

The intuition behind this transformation is that for each transition exiting from a block in the original FSM, the sub-FSM associated with the block performs a transition to its reset state. On the other hand, a transition entering a new block in the original FSM corresponds to a transition exiting the reset state in the interacting FSM of the block. The transformation can be clarified through an example.

Example 4 *Consider the monolithic controller of Figure 2. Its decomposition in interacting FSMs induced by the clustered graph of Figure 3 (c) is shown in Figure 4. 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 $go_{2,1}$ is asserted. The signal $go_{2,1}$ is an input for the F_1 that exits from $S01$ upon its assertion. The figure shows other activation signals for interfacing with sub-FSMs 0 and 3 (not shown).*

A sub-machine 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 sub-machine is performing state transitions and all other sub-machines are in reset state, ii) one sub-machine is transitioning toward its reset state, while another one is leaving it.

Notice that the *go* signals are *additional* inputs and outputs. All inputs and outputs of the original FSM are unchanged in the sub-machines. If an edge $s \longrightarrow t$ of the original machine has head and tail state included in sub-machine F_i, the edge is replicated in F_i, with the same input and output fields. Edges in the global FSM connecting states which belong to different partitions are associated with edges representing transitions to and from the reset states of the corresponding sub-FSMs. 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 sub-machine blocked in reset state are zero.

For space reasons, we have described the most straightforward implementation of the interacting FSMs. Several optimizations are possible to reduce the number of *go* signals. It is important to observe that if the *single entry point* *property* holds, the number of incoming signals for each sub-FSM can be reduced to the number of sub-FSMs 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 optimization is that if there is an edge between sub-machines 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 has been issued (because the starting state for the operation of F_j is always the same).

Example 5 *An example of optimization of the go signals is shown in Figure 4. The two sub-FSMs F_1 and F_2 have a single entry point. We also assume that sub-FSMs F_3 (not shown) has the same property. Thanks to the single entry point property, only one go signal is used for each couple of interacting FSMs, namely $go_{0,1}$, $go_{1,2}$, $go_{2,1}$ and $go_{2,3}$. We do not need to create separate signals, because for each sub-machine all transitions leaving the reset states are directed toward a unique entry state. Notice also that for this particular example the signals $go_{1,0}$ and $go_{1,3}$ are not needed, because there is no communication between the corresponding sub-machines.*

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 clock while they stay in reset state, we would 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. 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 when the 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 sub-FSM is in reset state. This happens when the sub-FSM machine 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 $F_{ai} = 1$.

$$F_{ai} = is\_in\_reset_i \wedge \overline{(\bigvee_{p\in F_j\neq F_i, q\in F_i} go_{p,q})} \qquad (1)$$

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 $go_{p,q}$ is asserted and the sub-FSM must exit the reset state. This activation function allows the newly activated sub-FSM to have its first active cycle during the last cycle of the previously active FSM. The two sub-FSMs 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.

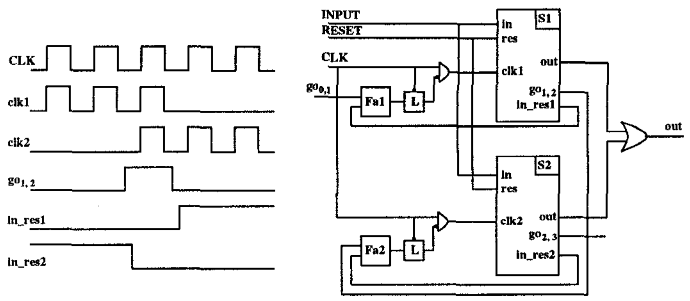


Figure 5. Gated-clock implementation of the interacting FSMs

Each local clock of the FSMs F_i is controlled by a gating block. The block is realized with the activation function F_{ai}. We could use F_{ai} as enable signal on the flip-flops, but this scheme would let the clock lines active and consume power. We gate the clock directly on its line. We use a low level sensitive latch for F_{ai}, in order to avoid spurious activity on the clock line [1]. Figure 5 shows two interacting FSMs and their clock control circuitry. The local clock of F_i is clocking (having the same waveform as CLK) when F_{ai} is 0. is blocked to 0 when $F_{ai} = 1$. One clock-gating block is instantiated for each sub-FSM.

Example 6 *The gated-clock implementation of the interacting FSMs of Figure 4 is shown in Figure 5. Notice how the external output is obtained by OR-ing the outputs of the sub-FSMs. This can be done because we specified that when a sub-FSM 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 sub-FSM.*

Figure 5 also shows the clock waveforms, the in_reset signals and the go signals. Notice that there is a clock cycle for which both local clocks are enabled. The waveforms show how sub-FSM 1 is deactivated and sub-FSM 2 activates thanks to the assertion of the $go_{1,2}$ signal.

3.2 Power savings

The interacting FSM implementation is power-efficient for several reasons. First, we save some power in the clock line. For a large fraction of the operation time, only one local clock is active and consumes power. A sub-FSM F_i has a fewer states than F, it has fewer state variables, and therefore the local clock of F_i is driving fewer flip-flops. Notice however that when there is a transition from F_i to F_j, both local clocks are active. In this cycle, the power consumed in the clock line is much higher than in the other cycles.

Second, when a sub-FSM is not active, the output and state values do not change, and the switching activity in the combinational logic is substantially reduced. The next state logic of the F_i is significantly simpler and smaller than that of F, thus the power consumed by F_i is smaller. Since only one F_i is active at a given time, the whole system consumes less power than the original monolithic FSM.

The power saving would be bigger if we latch the primary inputs, thereby completely eliminating the switching activity in the inactive sub-FSM. On the other hand we would need to replicate the input latches for each sub-FSM, with sizable area penalty. In the current implementation, the inputs are not latched.

There is a loss of power to compute the activation function. This loss is small because the activation function is implemented by simple logic and does not consume much power. Some power is wasted also in the logic to compute the primary outputs. This logic is not significant compared to the whole system. It amounts in the worst case to one n-input OR gate for each primary output. However, the output logic is often minimized, because some of the primary outputs are not driven by all sub-FSMs.

In summary, notice that power savings depend on the choice of the state partitions of the control FSM. Good partitions are produced when the *single entry point property* holds because the number of additional signals needed for interaction and clock control is minimized. In many cases, clustering is of paramount importance, since mutually exclusive sections are initially numerous and small. This is an undesirable situation because a large number of small components implies frequent transitions across partition boundaries. In the current implementation, the user specifies the maximum number of desired partition blocks. The tool collapses mutually exclusive sections of small size until the required number of blocks is reached. For best power savings, the number of partitions has to be quite small.

4 Implementation and results

We have implemented a complete design flow from the system level to the gate-level. The front-end of our tool reads the behavioral description (in C or Verilog) and generates the controller's FSM. In this step, the decomposition in blocks required for the interacting FSMs implementation is created. The user specifies n, the maximum number of blocks in the decomposition.

In the second step we generate the interacting FSMs F_i, $i = 1, ...n$. We also generate the interface of the communicating system, we instantiate the FSMs and the clock block. The interface is written in hierarchical Verilog HDL. Gate-level synthesis of the FSMs is performed with *Synopsys design compiler.* We used the default options for the Finite State Machine synthesis. For technology mapping we specified maximum optimization effort, without timing constraints. The two implementations have been synthesized with the same optimization options and with the same gate library for synthesis and simulation.

Bench	NS	NP	Avg pow (μW)			Area ratio		
			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

Table 1. Results on HLS benchmarks.

The last step is the gate-level power simulation. We ran the simulations with PPP [20] (an accurate gate-level power simulator) and estimated the total average power dissipated in the circuit. We compare 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 neutral signal probability of 50% for all input signals.

We have tested our tool on control-dominated benchmarks described in [5]. All these designs where 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 showed the results for DRAM with 2 and 4 partitions. The remaining two benchmarks are with a FIFO queue controller and the speed control unit for an automobile (SPDCNT).

The results are shown in Table 1. We displayed 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 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).

The last three columns of the table reflects the area overhead. The figures are the ratio of the communicating FSMs area over the original FSM area, in terms of mapped cells. We distinguished 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. It can even be smaller thanks to the simplification the partition introduces. The area of SPDCNT has a decrease of 15%. This combinational overhead is approximatively 20% on average. The sequential overhead comes from the duplication of the state vectors. 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. Notice however that we used 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. Interestingly enough, for our largest examples the synthesis of all interactive FSMs was faster than the synthesis of the single FSM implementation, even considering the overhead of generating the partitions.

5 Conclusion and future work

We proposed an approach to power minimization at the behavioral level based on an interacting FSM implementation with selective shutdown. The selective shutdown is obtained by gating the clock of the state registers for inactive sub-FSMs. We achieved in average a 37% power reduction with a 30% increase in area. Additionally, our method allows the exploration of the area/power trade-off by reducing the number of partitions.

We obtain sizable improvements 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 on this process to get even better results.

In the future we plan to extend our method by considering datapath optimization. Power can be reduced in the datapath by selectively gating the clock of the input latches of units controlled by inactive sub-FSMs. Moreover, we are developing algorithms for partitioning based on estimates of the probability of execution.

6 Acknowledgments

This research is partially supported by NSF under contract MIP-9421129, by the scholarship 200212/90.7 from CNPq/Brazil, by a fellowship from Fujitsu Laboratories of America and by ARPA under contract DABT 63-95-C-0049.

References

[1] L. Benini and G. De Micheli, "Transformation and synthesis of FSMs for low power gated clock implementation," *International Symposium on Low Power Design*, pp. 21–26, April 1995.

[2] J. M. Rabaey and M. Pedram, Low power design methodologies. Kluwer Academic Publishers, 1996.

[3] A. Bellaouar and M. J. Elmasry, Low-power digital VLSI design : circuits and systems Kluwer Academic Publishers, 1995.

[4] G. De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill, 1994.

[5] C.Coelho, V. Mooney and G. De Micheli, "Hardware/Software partitioning and synthesis from mixed specifications," to appear in *European Design Automation Conference*, Sept. 1996.

[6] D. Knapp, T. Ly et al., "Behavioral synthesis methodology for HDL-based specification and validation," *Proceedings of the Design Automation Conference*, pp. 286–291, June 1995.

[7] R. Bergamaschi, R. O'Connor et al., "High-level synthesis in an industrial environment," *IBM Journal of Research and Development*, vol. 39, no. 1-2, pp. 131–148, 1995.

[8] S. Note, F. Chattoor et al., "Combined hardware selection and pipelining in high-level performance data-path design," *IEEE Transactions on CAD/ISCAS*, vol. CAD-11, pp. 413–423, April 1992.

[9] D. Thomas, E. Lagnese et al. *Algorithmic and Register-Transfer Level synthesis: the System Architect's Workbench*. Kluwer Academic Publishers, 1990.

[10] A. Wu D. Gajski et al. *High-Level VLSI synthesis - Introduction to chip and system design*. Kluwer Academic Publishers, 1992.

[11] D. Ku and G. De Micheli. *High-Level synthesis of ASICs under timing and synchronization constraints*. Kluwer Academic Publishers, 1992.

[12] W. Wolf, A. Takach et al., "The Princeton university behavioral synthesis system," *Proceedings of the Design Automation Conference*, pp. 182–187, June 1992.

[13] A. Chandrakasan, M. Potkonjak et al., "Minimizing power using transformations," *IEEE Transaction on CAD/ISCAS*, vol. 14, no. 1, pp. 12–30, January 1995.

[14] E. Mussol and J. Cortadella, "High-level synthesis techniques for reducing the activity of functional units," *International Symposium on Low Power Design*, pp. 99–104, April 1995.

[15] A. Raghunathan and N. K. Jha, "Behavioral synthesis for low power," *Proceedings of the International Conference on Computer Design*, pp. 318–322, October 1994.

[16] J. Chan and M. Pedram, "Register allocation and binding for low power," *Proceedings of the Design Automation Conference*, pp. 29–35, June 1995.

[17] A. Farrahi, G. Tellez and M. Sarrafzadeh, "Memory segmentation to exploit sleep mode operation," *Proceedings of the Design Automation Conference*, pp. 36–41, June 1995.

[18] A. Dasgupta and R. Karri, "Simultaneous scheduling and binding for power minimization during microarchitecture synthesis," *International Symposium on Low Power Design*, pp. 69–74, April 1995.

[19] J. Monteiro, S. Devadas et al., "Scheduling techniques to enable power management," *Proceedings of the Design Automation Conference*, pp. 349–352, June 1996.

[20] A. Bogliolo, L. Benini, and B. Riccò, "Power Estimation of Cell-Based CMOS Circuits," *Proceedings of the Design Automation Conference*, pp. 433–438, June 1996.