FSMD Functional Partitioning for Low Power

Enoch Hwang Frank Vahid Yu-Chin Hsu
Department of Computer Science
University of California, Riverside, CA 92521
ehwang@cs.ucr.edu vahid@cs.ucr.edu hsu@cs.ucr.edu

Abstract

Previous works have shown that sizable power reductions can be achieved by shutting down sub-circuits when they are not needed. However, these shutdown techniques focus on shutting down only portions of the controller or the datapath of a single custom hardware processor. We propose a higher level shutdown technique that considers both the controller and datapath simultaneously; in particular, we partition a processor into multiple simpler mutually exclusive communicating processors, and then shut down the inactive processors (i.e., the inactive controller/datapath pairs). Power reduction is accomplished because only one processor is active at a time. In addition to power reduction, functional partitioning also provides solutions to a variety of synthesis problems and does not require the modification of the synthesis tool. We present results which show that this FSMD functional partitioning technique can save, on average, 42% over unoptimized systems in power reduction.

1. Introduction

Power reduction of VLSI systems is an important goal for system designs and much work has been done in this area as surveyed in [1]. While power reduction techniques can be applied at nearly every design abstraction level, many previous works have focused on the lower levels of abstraction. Recently, there is a focus on power reduction at the higher levels [2] where large power savings are possible merely by cutting down on wasted power. High-level power reduction mainly involves shutting down unnecessary portions of the circuit, thereby reducing the total amount of switching activities. Two areas of shutdown techniques for power reduction have appeared in recent literatures.

In *datapath* shutdown techniques, portions of the combinational logic in the datapath can be shut down for some cycles when those results are either precomputed or are not required. In [3], the output values are selectively *precomputed* before they are needed and are used to reduce the switching activity in the next clock cycle. The *guarded evaluation* technique in [4] tries to determine, on a per clock cycle basis, which part of a combinational circuit are computing results that will be used, and which are not. The sections that are not needed are then shut off, thus saving the power used in all the useless transitions in that part of the circuit. The clock gating method used in a popular synthesis tool considers only static power. Static power, however, accounts for less than 10% of the total power consumption in a circuit [5].

In controller shutdown techniques [6], the controller is partitioned into two or more mutually exclusive interacting FSMs and their clocks are selectively gated. 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.

For example, given the FSMD description in Figure 5, the corresponding RTL description is shown in Figure 1. Suppose that the input for *x* is 0 in the FSMD code, then only states s0 and s3 will be executed, and so neither the adder nor the multiplier are needed. These two functional units will be wasting power in the unoptimized circuit of Figure 1 because they will have switching activities even though their results are not needed. Furthermore, the portions of the controller for generating the control signals for the three registers *p*, *i*, and *t*2, and the two multiplexers *mux*1 and *mux*2 can be reduced to save power. Figure 2 shows the result of applying the guarded evaluation technique to the unoptimized circuit of Figure 1. Latches are added in front of all the functional units. Since only the comparator is needed in the execution of states s0 and s3, the inputs to the adder and multiplier can be latched, thus,

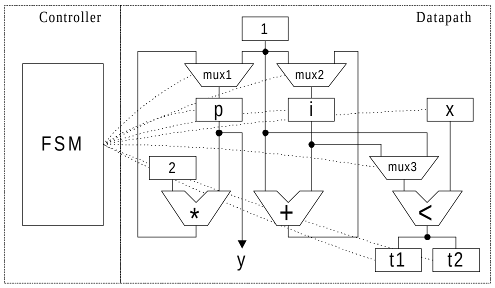

Figure 1. Unoptimized RTL design example.

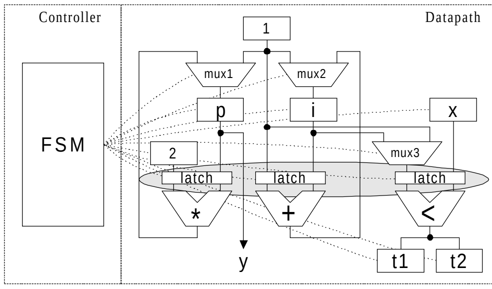

Figure 2. Guarded evaluation technique.

preventing the inputs from changing. Power is saved because there will be no switching activity in these two functional units. Figure 3 shows an example of applying the selectively-clocked FSM technique. Here we have split the original FSM into two sub-FSMs. FSM1 controls the portion of the datapath for the multiplier and adder while FSM2 controls the portion of the datapath for the comparator. Since we only need the use of the comparator, FSM1 can be made inactive by stopping the clock to it. The power savings from this technique come directly from the fact that there are multiple smaller FSMs instead of one large one. As a result, we have a shorter local clock line, fewer states, and simpler and smaller next state logic. While this method prevents unnecessarily power consumption in the control unit, there is no power reduction in the datapath.

While the above mentioned techniques show significant power reductions, they focus only on either the datapath or the controller for a single custom processor. It was recognized in [3] and [4] that the power savings would be even larger if both the controller *and* the datapath are considered together and that the techniques are applied on the complete circuit, rather than on individual blocks. In this paper, we propose a new shutdown technique for reducing power where both the controller and the datapath are shut off together.

The rest of the paper is organized as follows. In Section 2, we will describe the problem. Section 3 gives a detail description of our technique. Section 4 shows our experimental results, and we conclude in Section 5.

2. Problem Description

When a circuit is synthesized from a behavioral description to a gate level netlist, only one control unit and one datapath is generated. What this means is that when there is a signal change at the primary input, the entire datapath and control unit may be affected. Partitioning the netlist (as in structural partitioning) does not reduce power because even though the partitioning creates more than one physical partition, there is still logically only one datapath and one control unit. When a primary input signal changes, it may still affect many of the gates in the datapath and/or controller regardless of which part they are in. Previous techniques only try to reduce the switching activities within a localized subset of gates either in the datapath or the controller.

Our FSMD functional partitioning technique is applied before synthesis. The original FSMD is first partitioned into several smaller mutually-exclusive FSMDs. Each of these smaller FSMDs is then synthesized to its own custom processor, having its own controller and datapath. The reason why FSMD functional partitioning can significantly reduce power is that each processor is smaller than the original one large processor implementing the entire process, and only one processor is executing a computation at any given time while the other processors will be idle. When a processor is idle, we have, in effect, shut down both the controller and the datapath for that processor. Thus, greater power saving is possible.

Figure 4 shows the result of applying the FSMD functional partitioning technique to the sample circuit of Figure 1. Here, we have two smaller mutually exclusive processors. The first processor containing the controller and datapath for executing state s1, and the second processor containing the controller and datapath for executing states s0, s2, and s3. Thus, when *x*=0, only processor 2 needs to be active. Processor 1 remains inactive in an idle state waiting for processor 2 to wake it up if necessary. The datapath of processor 1 is not consuming power because the inputs are not changing. The power consumed by both controllers is reduced because of their smaller size. Furthermore, the power consumed by processor 1's controller is reduced even more because it is in an idle state. The overhead in this technique is the communication and possible duplication of registers.

In addition to reducing power, FSMD functional partitioning also provides solutions to a variety of synthesis problems. These include I/O satisfaction by reducing total I/O by as much as 67% (which could impact physical design positively), reduced synthesis runtime by as much as 85%, and hardware / software tradeoffs [7]. Furthermore, our technique does not require the modification of the synthesis tool. The relevance of using the FSMD model is that many circuit designs are still specified at the register-transfer level using this model. However, partitioning introduces extra power consumption for inter-processor communication between the smaller FSMDs. Thus, the problem that must be solved is one of partitioning such that the reduction in power for computations far outweighs the power increase for communication, while also minimizing the increase in total circuit size and execution time.

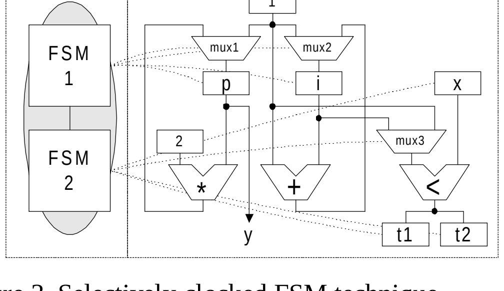

Figure 3. Selectively-clocked FSM technique.

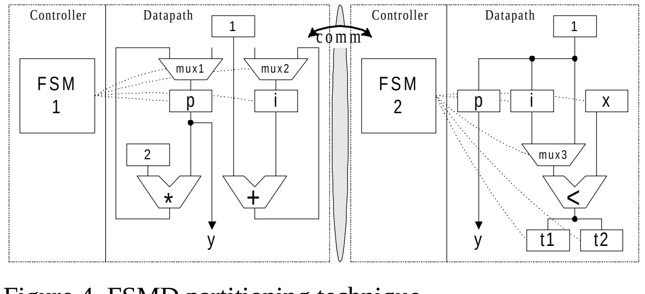

Figure 4. FSMD partitioning technique.

Since we based our partitioning technique on the FSMD model, we will now define it. An FSMD differs from a traditional FSM in that it may include variables with various data types, as well as complex data operations in its actions. Formally, a finite-state machine with dataflow [8] is a 6-tuple defined as follows:

$$P = <S, s_0, I \cup STAT, O \cup A, \delta, \lambda>$$

where:

- $S = \{s_0, \ldots, s_m\}$ is a finite set of *states*
- s_0 is the *reset state.*
- $I = \{i_j\}$ is a set of primary input values.
- $STAT = \{Rel(a, b) \mid a, b \in EXP\}$ is a set of status signals as logical relations between two expressions from the set EXP.
- $EXP = \{f(x,y,z,\ldots) \mid x,y,z,\ldots \in VAR\}$ is a set of expressions.
- VAR is a set of storage variables.
- $O = \{o_k\}$ is a set of primary output values.
- $A = \{x \Leftarrow e \mid x \in VAR, e \in EXP\}$ is a set of storage assignments.
- δ is a *state transition* function that maps a cross product of S and I into S.
- λ is the *output function* that maps a cross product of S and I into O for Mealy models or S into O for Moore models.

3. Implementation Details

In contrast to procedural functional partitioning [9] which performs a coarse-grained partitioning of procedures and functions, FSMD functional partitioning has no concept of functions or procedures, but rather states. What we need in FSMD functional partitioning is to be able to determine the variables that need to be passed from one state to the next. A power partitioning algorithm is then applied to separate the states into two or more parts. Finally, communication between the parts is added. The details are discussed in the following sections.

```
case State_Var is
   when s0   =>
      p := 1 ;
      i := 1 ;
      t1 := 1 < x ; -- x is a primary input
      if t1 then
         State_Var := s1 ;
      else
         y <= p ;
         State_Var := s3 ;
      end if ;
   when s1   =>
      p := p * 2 ;
      y <= p ;
      i := i + 1 ;
      State_Var := s2 ;
   when s2   =>
      t2 := i < x ;
      if t2 then
         State_Var := s1 ;
      else
         State_Var := s3 ;
      end if ;
   when s3   =>
      State_Var := s3 ;
end case;
```

Figure 5. Sample unpartitioned FSMD code.

3.1 Dataflow analysis

Given an unpartitioned FSMD, we first construct a control flow graph by assigning a state in the FSMD to a node in the graph. For example, given the unpartitioned FSMD code of Figure 5, we obtain the initial control flow graph of Figure 7 (a). A dataflow analysis, similar to those use for compiler optimization [10], is then performed on the graph to obtain the variables that need to be passed from one state to another.

When we perform the FSMD partitioning, we are only interested in the amount of data that cross between two parts and not between two states that are in the same part. Thus, for each part P_i, we need to calculate the set of variables needed to be passed from the caller part, $P_i.in$, and the set of variables needed to pass to the callee part, $P_i.out$. The algorithm to evaluate $P_i.in$ and $P_i.out$ is shown in Figure 6. Note that before applying this algorithm, we must already know how many parts we will have and the set of nodes in each part. Continuing with our example, if we put node s1 in part P1 and the rest of the nodes in part P0, then after applying the algorithm of Figure 6, we obtain the results shown in Figure 7 (b).

Notice that the variable x is used in state s2 and was defined (primary input) in state s0, thus, it must be passed from s0 via s1 to s2. However, x is not in either of the sets P1.in or P1.out. This is because s0 and s2 are in the same part. If we had put s2 in P1, then x would have to be passed across the parts, thus increasing the power consumed by the communication.

3.2 Power partitioning algorithm

For our initial experiments, we have used a simple algorithm to partition the states such that the overall energy consumption of the entire circuit is minimized. The algorithm is based on the following energy estimation model. Details of the model can be found in [11]. Let Es_i be the energy consumed by state s_i. The total energy for a 2-part partitioned system (parts A and B) is

$$E_{partitioned} = \alpha_A E_A + \alpha_B E_B + E_{comm} \qquad (1)$$

```
for each part $P$ do
   begin
   $P.callee =$ set of all states $n_i : n_j \rightarrow n_i, i \neq j, n_i \in P,$ and $n_j \notin P.$
      // set of states in $P$ that transition from states that are not in $P$.
   $P.caller =$ set of all states $n_i : n_i \rightarrow n_j, i \neq j, n_i \in P,$ and $n_j \notin P.$
      // set of states in $P$ that transition to states that are not in $P$.
   $P.use = \bigcup n.use \quad \forall n \in P$ // set of variables used in $P$.
   $P.def = \bigcup n.def \quad \forall n \in P$ // set of variables defined in $P$.
   $P.in = (\bigcup P.callee.in) - (\overline{P.use})$
   $P.out = (\bigcup P.caller.out) - (\overline{P.def})$
   end
```

Figure 6. Partition dataflow analysis algorithm.

where $E_A = \sum_{\forall s_i \in A} Es_i$ (similarly for E_B) is the sum of the energy of the states in part A (part B), and α_A is the complexity for processor A due to its controller and datapath interconnect. α_A is approximated by the number of states in the part. The assumption made for α is that the complexities of two individual states are summed to get the complexity for both states combined, but in reality, the combined complexity will be less than the sum. E_{comm} is the communication energy. Currently, to simplify the partitioning heuristic, this term is ignored. Let E_U and α_U be defined similarly but for the unpartitioned system, then

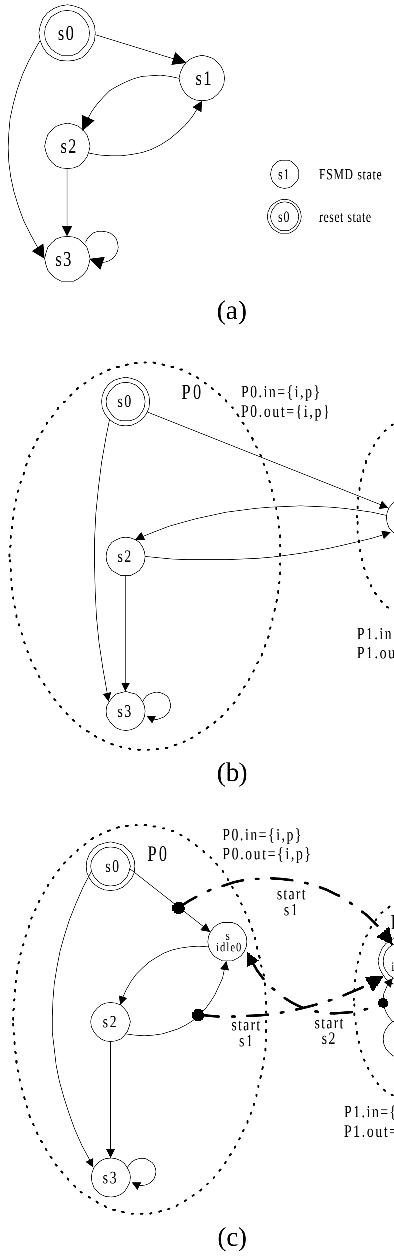

Figure 7. (a) Control flow graph of Figure 5. (b) Result after dataflow analysis and partitioning. (c) Result after refinement.

since $E_A + E_B = E_U$ and $\alpha_A + \alpha_B = \alpha_U$, we get

$$E_{partitioned} = \alpha_A E_A + (\alpha_U - \alpha_A)(E_U - E_A) \qquad (2)$$

To minimize $E_{partitioned}$, we need to minimize both terms. However, we cannot minimize the first term by minimizing both E_A and α_A because this will cause the second term to be maximized; likewise with the second term. Thus, to minimize a term, we can only minimize either α or E for the same term. From this observation, the partitioning algorithm below tries to balance the α and E between the two parts:

1. Calculate Es_i for all states.
2. Sort the states such that $Es_1 < Es_2 < \ldots < Es_n$.
3. Divide the states into two parts between s_i and s_{i+1} satisfying the following criteria:
   - Part $A = \{s_1, s_2, \ldots, s_i\}$
   - Part $B = \{s_{i+1}, s_{i+2}, \ldots, s_n\}$
   - $|E_A - E_B|$ is minimized.

Basically, this algorithm finds the median of the states with respect to the energy consumption E. Alternatively, we can find the median with respect to α. This algorithm is very fast; steps 1 and 3 are $O(n)$ and step 2 is $O(n\log n)$ where n is the number of states. We are currently working on a more refined partitioning heuristic using a branch and bound scheme and comparing the power reduction results between them.

3.3 FSMD refinement method

The algorithm described in the previous section produces a partitioning of the FSMD states. We now generate a new communicating FSMD that is functionally equivalent to the original unpartitioned FSMD. The resulting communicating FSMDs from Figure 7 (a) is shown in Figure 7 (c).

For example, to transition from state s_0 to s_1 in the unpartitioned FSMD shown in Figure 7 (a), the equivalent transition in the partitioned FSMDs as shown in Figure 7 (c) is as follows. Initially, P0 is in s_0 and P1 is in its idle state s_{idle1}. To transition to s_1, P0 exits s_0, asserts $start_{s1}$, and enters its idle state s_{idle0}. Seeing that $start_{s1}$ is asserted, P1 exits s_{idle1} and enters s_1.

The FSMD partitioning can be formally described as follows. Let $P = \langle S, s_0, I \cup STAT, O \cup A, \delta, \lambda\rangle$ be the original unpartitioned FSMD. Our method is to partition P into n parts, $P_0, \ldots, P_{n-1}$ such that the combined behavior of the partitioned P_i's is functionally equivalent to the unpartitioned P. Each partitioned FSMD, P_i, is defined as follows:

$$P_i = \langle S_i, s_{0,i}, s_{idle,i}, I_i \cup STAT_i \cup IP_i, O_i \cup A_i \cup OP_i, \delta_i, \lambda_i \rangle$$

where the symbols are defined similarly to the unpartitioned FSMD except that they are for each part P_i. A new *idle* state $s_{idle,i} \in S_i$ is added to each P_i. Furthermore,

$$\bigcap_{i=0}^{n-1} S_i = \varnothing$$

and

$$\bigcup_{i=0}^{n-1} S_i = S + \bigcup_{i=0}^{n-1} s_{idle,i} \ .$$

P_0 is the main active part, and the other P_i's are the passive parts. For the main part P_0, the idle state is not the reset state, i.e. $s_{0,0} \neq s_{idle,0}$. Whereas, for the other parts $P_{i=1 \text{ to } n-1}$, the idle state is the reset state, i.e. $s_{0,i} = s_{idle,i}$. Besides the primary inputs and outputs I_i and O_i, each part also has data that is passed between the parts. These are IP_i and OP_i for data that is passed from and to another part respectively. $IP_i = <ip_1, \dots ip_a>$ where a = number of input parameters for P_i, and $OP_i = <op_1, \dots op_b>$ where b = number of output parameters for P_i. These parameters and values are determined from the dataflow analysis.

For each transition from a state u of P_i to a state v of P_j ($i \neq j$), a new signal $start_v$ is generated. $start_v$ is a uni-directional signal that goes from P_i to P_j. Every transition from state u to v in P becomes a transition from u to $s_{idle,i}$ in P_i and from $s_{idle,j}$ to v in P_j. The transition from u to $s_{idle,i}$ in P_i asserts the output signal $start_v$ of P_i. The transition from $s_{idle,j}$ to v in P_j is performed only when the input signal $start_v$ of P_j is asserted.

Partitioning the FSMD and introducing the extra idle state in each part according to the technique described above *do not change the cycle-by-cycle behavior* of the original unpartitioned FSMD. When there is a transition that crosses between two parts, the caller processor will transition to its idle state while at the same time, the callee processor transitions from its idle state to the next state. The transitions to and from its respective idle states for the two parts happen simultaneously, thus, no extra clock cycle is needed as shown in Figure 8. However, the critical path can be either lengthened (because of the added communication circuitry) or shortened (because of the smaller circuitry in the smaller part). Thus, the overall execution time can be longer or shorter than the unpartitioned system.

4. Experimental Results

We have implemented the FSMD functional partitioning technique described above. We start by describing a system using the FSMD model with VHDL. After applying our partitioning technique, the system is synthesized and simulated to obtain the switching activity data. Power results are calculated using the switching activity and the capacitance of the netlist obtained from the technology library. The unpartitioned and partitioned systems are compared in terms of their average and total power usage, area, and execution time.

Table 1 shows the statistics for the FSMD examples. *Fac* is a factorization program. *Chinese* evaluates the Chinese Remainder Theorem. *Diffeq* is an example from the HLSynth MCNC benchmark. *Volsyn* is a volume-measuring medical instrument controller. *NLoops* is an example with nested loops. *MP* is a small microprocessor. *DSP* is a digital signal processor. The second and third columns show the size in terms of gate count for the unpartitioned and partitioned systems respectively. For the partitioned size, the gate count for the individual parts are further broken down. The *states* column shows the number of states in the partitioned system and the last column shows the total bit width for the communication.

The results are summarized in Table 2. Columns 2 and 3 show the percent increase in area and execution time respectively. The absolute average and total power for the partitioned examples are shown in columns 4 and 5. The percent average and total power savings are shown in columns 6 and 7. In all cases except for *Diffeq*, both the average and total power is reduced. The savings in average power ranges from 2% to as much as 66% with an average of 42%. The savings in total power range from 33% to 64% with an average of 41%. For the *Diffeq* example, the average power is reduced by 2% but the total power is increased by 3%. A possible reason for this is that the *Diffeq* example is simply a repetition of a single algorithm several times, and thus, is not a good candidate for partitioning. The tradeoff for the area on the average is 24% and only 1% on average for the execution time. The reason why the execution time overhead is so small is because the critical path can be shortened as a result of a smaller processor. The 24% increase in gates is not as significant because chip capacities have continued to grow exponentially. The results do take into consideration the fact that the bus capacitance for communications between parts are larger than internal capacitance. In our power

Table 1: Example statistics.

Example	Unpart	Partitioned		
	Size	Size	States	Com
Fac	15251	17208=11166+2758+3284	20	230
Chinese	19766	33054=14137+2233+1669+15015	44	485
Diffeq	11487	12874=1654+11220	58	258
Volsyn	11193	13163=10798+2365	16	67
NLoops	2622	3484=1988+1496	12	66
MP	6210	6307=4623+1684	101	98
DSP	278	386=131+255	13	12

Table 2: Power reduction results.

Example	% Overhead		Absolute Partitioned		% Power Savings	
	Area %	Time %	Average Power (µW)	Total Power (µJ)	Average %	Total %
Fac	13	5	17.26	1347.40	66	64
Chinese	67	7	15.85	3491.43	37	33
Diffeq	12	5	54.74	8989.34	2	-3
Volsyn	3	9	7.54	1509.18	49	44
NLoops	33	-6	5.19	2511.00	42	45
MP	2	-4	13.29	425.27	51	51
DSP	39	-9	1.08	28.38	48	50
Average	24	1	16.42	2614.57	**42**	**41**

calculation, we have used a bus capacitance that is four times the internal capacitance.

Figure 9 shows a comparison of average power savings between our FSMD partitioning technique with the guarded evaluation [4] and selectively-clocked [6] techniques. We used two approaches to make the comparison and they both gave similar results. In the first approach, we analyzed our set of examples to estimate the power savings using the localized techniques. In the second approach, the power savings data for the localized techniques are taken directly from their respective papers and adjusted to our unoptimized examples. Since their savings are with respect to portions of the whole system, we have adjusted it accordingly to reflect the savings for the entire system. The data from [4] does not include examples with a power savings of less than 15%. Hence, to compare fairly, we have dropped all such examples in the comparison (in our case, the *Diffeq* data is dropped.) The percent power savings for the three techniques, guarded evaluation, selectively-clocked, and FSMD partitioning, over the unoptimized design are 31%, 7%, and 49% respectively. The power usage by the functional units and muxes is less for FSMD partitioning than for guarded evaluation because there is power savings from the muxes for the former but not the latter technique. Power usage by the registers is more than the unoptimized because some registers have to be duplicated, however, it is slightly less than that of guarded evaluation because fewer extra latches are needed. The controller power usage is about the same as that of the selectively-clocked technique.

After the FSMD partitioning, we end up with several smaller processors, thus, we can further apply the localized techniques to the individual processors to get even better results. Our analysis [11] shows that an additional 18% power savings might be achievable resulting in a total savings of 58% as shown in the *FSMD partitioning and guarded evaluation* plot in Figure 9.

5. Conclusions

We have introduced an FSMD functional partitioning technique for reducing power consumption. Unlike previous power reduction shutdown techniques that focus only on either the datapath or the controller, our approach partitions the entire FSMD to shut down both the controller and the datapath. We achieved on average a 42% average power reduction with a 24% increase in gate count and only 1% increase in execution time. Furthermore, since our technique is applied at a higher level and in the early stages of the design process, further power reduction is still possible by applying localized power reduction techniques at the lower levels. In addition to power reduction, FSMD functional partitioning also provides solutions to a variety of synthesis problems and does not require the modification of the synthesis tool.

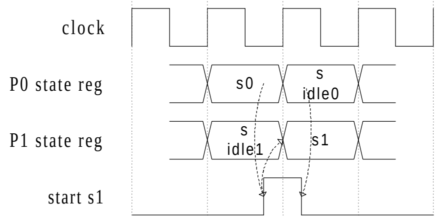

Figure 8. Partitioned FSMD transition timing diagram.

References

[1] Srinivas Devadas & Sharad Malik, "A Survey of Optimization Techniques Targeting Low power VLSI Circuits," *Proceedings of the Design Automation Conference*, pp. 242-247, 1995.

[2] Enrico Macii, Massoud Pedram, & Fabio Somenzi, "High-Level Power Modeling, Estimation, and Optimization," *Proceedings of the Design Automation Conference*, pp. 31-38, 1997.

[3] Mazhar Alidina, Jose Monteiro, Srinivas Devadas, & Abhijit Ghosh, "Precomputation-Based Sequential Logic Optimization for Low Power," *Proceedings of the International Conference on Computer Design*, pp. 74-81, October 1994.

[4] Vivek Tiwari, Sharad Malik, & Pranav Ashar, "Guarded Evaluation: Pushing Power Management to Logic Synthesis/Design," *International Symposium on Low Power Design*, 1995.

[5] A. Chandrakasan, T. Sheng, & R. Brodersen, "Low Power CMOS Digital Design," *Journal of Solid State Circuits*, Vol. 27, No. 4, pp. 473-484, April 1992.

[6] L. Benini, P. Vuillod, G. De Micheli & C. Coelho, "Synthesis of Low-Power Selectively-Clocked Systems from High-Level Specification," *International Symposium on System Synthesis*, pp. 57-63, Nov. 1996.

[7] F. Vahid, T. Le, & Y.C. Hsu, "A Comparison of Functional and Structural Partitioning," *International Symposium on System Synthesis*, pp. 121-126, November 1996.

[8] D. Gajski, N. Dutt, A. Wu, & S. Lin, *High-Level Synthesis Introduction to Chip and System Design*, Kluwer Academic Publisher, Boston, 1992.

[9] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and design of embedded systems*, New Jersey, Prentice Hall, 1994.

[10] A. V. Aho, R. Sethi, & J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, California, 1988.

[11] E. Hwang & F. Vahid, "Energy Estimation for FSMD Partitioning," UCR CS 98 06, University of California, Riverside.

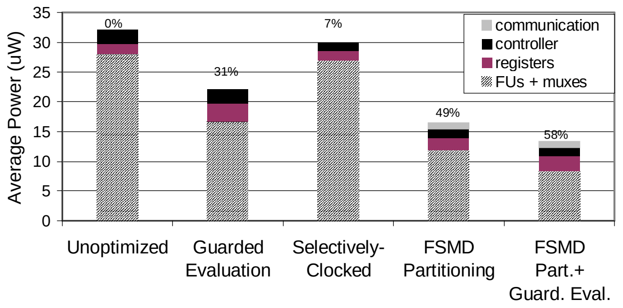

Figure 9. Average power savings compared. Percentages show power savings. Shorter bars are better.