UNIVERSITY OF CALIFORNIA
RIVERSIDE

Functional Partitioning for Low Power

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy
in
Computer Science
by
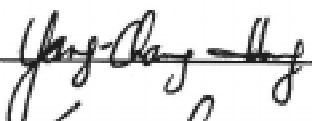Enoch Oi-Kee Hwang
June, 1999

Dissertation Committee:
        Dr. Frank Vahid, Chairperson
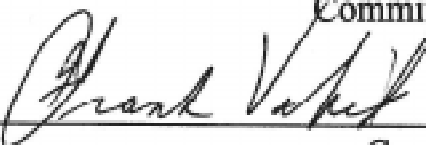   Dr. Yu-Chin Hsu, Co-Chairperson
        Dr. Yang-Chang Hong

The Dissertation of Enoch Oi-Kee Hwang is approved:

_____

_____
Committee Co-Chairperson

_____
Committee Chairperson

University of California, Riverside

Acknowledgments

I want to thank my professors Dr. Frank Vahid and Dr. Yu-Chin Hsu for all the support and help given to me.

Dedication

I want to dedicate this work to my loving wife Windy, my son Jonathan and my second expecting child.

Abstract of the Dissertation

Functional Partitioning for Low Power

by

Enoch Oi-Kee Hwang

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June, 1999
Dr. Frank Vahid, Chairperson
Dr. Yu-Chin Hsu, Co-Chairperson

Power reductions in VLSI systems have recently become a critical metric for design evaluation. Although power reduction techniques can be applied at every level of design abstraction, most automated power reduction techniques apply to the lower levels of design abstraction. Previous works have shown that sizable power reductions can be achieved simply by shutting down a system's 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 therefore investigated the power reduction attainable by the evolving automated technique of functional partitioning in which a process is automatically divided into multiple simpler, mutually exclusive, communicating processors, and then shut down the inactive processors. By shutting down the entire inactive processor, we have in effect shut down both the controller and datapath. Power reduction is accomplished because only one smaller processor is active at a time.

We have applied this functional partitioning technique to either the procedural or the finite-state machine with datapath (FSMD) behavioral level. From either level, the original process is partitioned into multiple parts. For the procedural level, a coarse-grained partitioning of procedures is done. Data transfers between the parts are simply the parameters in the procedural call. In contrast, FSMD partitioning has no concept of procedures, but rather states. A dataflow analysis is first performed to determine the data transfers between the parts. A power partitioning algorithm is then used to separate the states into multiple parts. The parts are then individually synthesized down to the gate level netlist. Finally, communication is added between the parts so that they are functionally equivalent to the original unpartitioned process.

Partitioning introduces extra power consumption for inter-processor communication. 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. Our results show that this functional partitioning technique can reduce power, on average, by 42% over unoptimized systems. In addition to power reduction, functional partitioning also provides solutions to a variety of synthesis problems.

Table of Contents

List of Tables

List of Figures

Chapter 1.   Introduction

Electronic circuit optimization for area and timing has been well studied. Recently, power consumption has become one of the more critical design parameters for very large scale integration (VLSI) systems. The reduction of area for an integrated circuit (IC), which was a big issue not too long ago, is not as big an issue today because with new IC production technologies, many millions of transistors can be put on a single IC. On the other hand, there is a trend towards portable battery operated devices. The shrinking sizes of integrated circuits calls for reduced power consumption in order to extend battery life for these portable devices. Furthermore, in the deep submicron technologies, there is a limitation of circuit density because of excessive heat generation from high power dissipation. Hence, power consumption is now one of the most important criteria in circuit designs.

While power reduction techniques can be applied at every level of design abstraction, most of the previous power optimization techniques apply to the lower levels of the design process; namely transistor and logic gate levels [1]. 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 in the circuit. We therefore investigate the power

reduction attainable by the evolving automated behavioral level technique of functional partitioning.

Power is consumed when capacitors in the circuit are either charged or discharged due to switching activities. Power reduction at the higher levels is mainly achieved through the reduction of these switching activities by shutting down portions of the system when they are not needed [2]. The idea is that for a large system, not all components are required to be active at all times and thus, large power savings are possible merely by cutting down on wasted switching activities. Large VLSI circuits such as processors contain different components such as the controller, memory and functional units. Recent high-level shutdown techniques focus on shutting down only portions of the controller or the functional units of a single custom hardware processor. Two such areas of shutdown techniques for power reduction have been addressed in recent literature.

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* using a few high order bits one cycle before they are needed. If precomputation (e.g., comparing the highest bits of a 32-bit comparison) indicates that the full computation is not necessary, then the entire original logic circuit can be turned off in the next clock cycle. Thus, switching activity is reduced and power is saved. 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 parts that are not needed are then shut off, thus

saving the power used in all the useless transitions in that part of the circuit.

For example, given the FSMD description in Figure 1.1, the corresponding

unoptimized RTL description is shown in Figure 1.2. 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 will be needed. These two functional units will be wasting power in the

unoptimized circuit of Figure 1.2 because they will have switching activities even though

```
loop
 case State_Var is
   when s0  =>
     p := 1 ;
     i := 1 ;
     if (1 < x) then -- x is a primary input
       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  =>
     if (i < x) then
       State_Var := s1 ;
     else
       State_Var := s3 ;
     end if ;
   when s3  =>
     p := p – 1 ;
     y <= p ;
     i := i – 1 ;
     State_Var := s2 ;
 end case;
end loop;
```

Figure 1.1. Sample unpartitioned FSMD code.

their results will not be needed. Furthermore, the controller can also be reduced to save power.

Figure 1.3 shows the result of applying the guarded evaluation technique to the unoptimized circuit of Figure 1.2. Latches are added in front of all the functional units. Since only the comparator and the subtract unit are needed in the execution of states s0
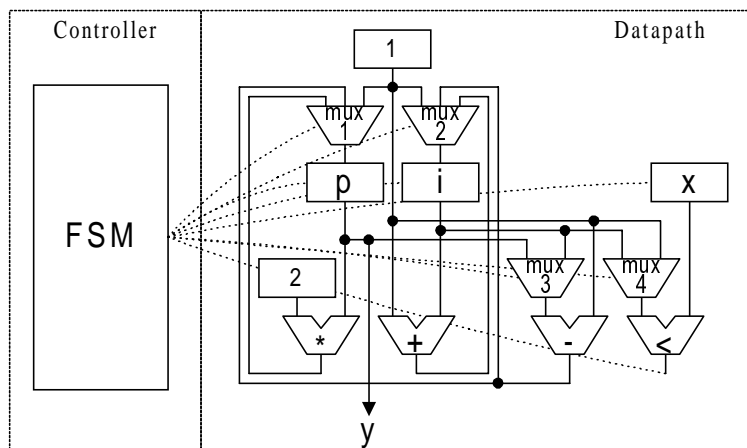
Figure 1.2. Unoptimized RTL design exmple.
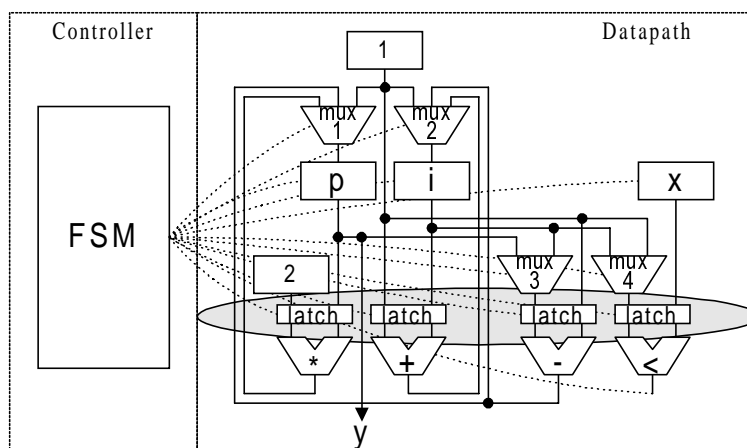
Figure 1.3. Guarded evaluation technique.

and s3, the inputs to the adder and multiplier can be latched, thus, preventing the inputs from changing. Power is saved because there will be no switching activities in these two functional units during the execution of states s0 and s3.

In *controller* shutdown techniques [5][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. Figure 1.4 shows an example of applying the selectively-clocked FSM technique. Here we have split the original FSM into two sub-FSMs. FSM1 includes state s1 which controls the portion of the datapath for the multiplier and adder while FSM2 includes states s0, s2, and s3, which controls the portion of the datapath for the comparator and subtract unit. Since we only need the use of the comparator and subtract unit to execute states s0 and s3, FSM1 can be made inactive by stopping the clock to it.
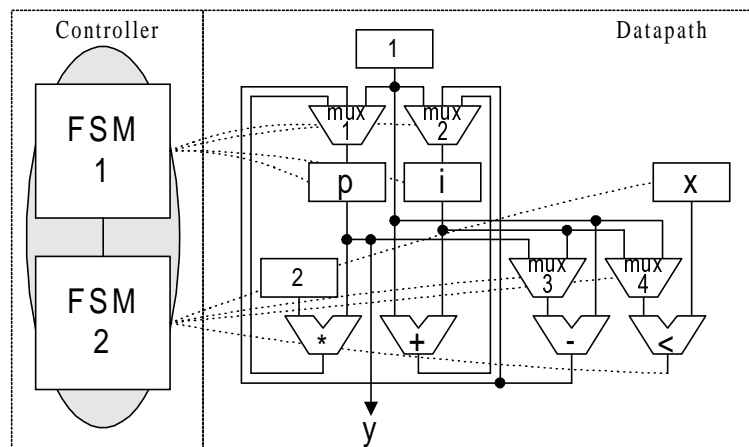
Figure 1.4. Selectively-clocked FSM technique.

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 unnecessary 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 [4] and [5] that the power savings would be even larger if both the controller and the datapath were considered together and if the techniques were applied on the complete circuit, rather than on individual blocks. Hence, we propose a new functional partitioning shutdown technique for reducing power where both the controller *and* the datapath are considered together.

Our functional partitioning technique for power reduction is based on the finite state machine with datapath (FSMD) model. Instead of trying to separately optimize individual components of a system and then trying to combine the different optimized circuits together, our technique optimizes the original monolithic system by partitioning it. 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, each 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 1.5 shows the result of applying the FSMD functional partitioning technique to the sample circuit of Figure 1.2. Here, we have two smaller mutually exclusive processors. The first processor contains the controller and datapath for executing state s1, and the second processor contains 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 between the processors and possible duplication of registers.
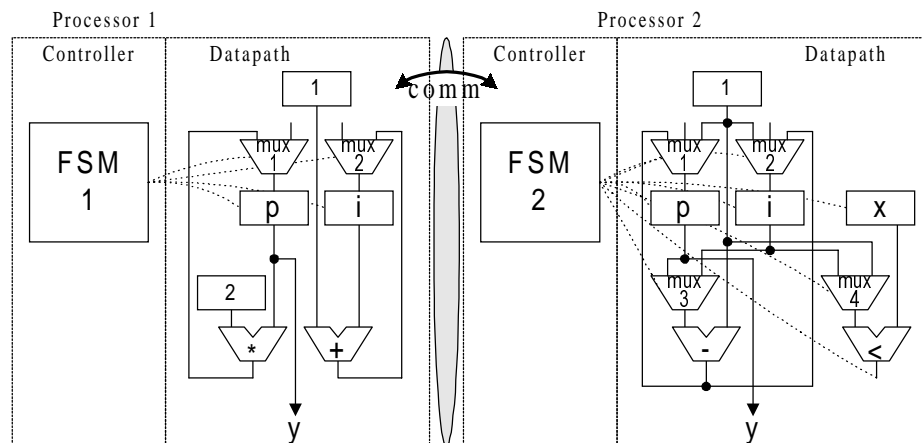
Figure 1.5. FSMD functional partitioning technique.

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, the technique does not require the modification of synthesis tools because it is applied before synthesis. The relevance of using the FSMD model is that many circuit designs are 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, and preserving the cycle-by-cycle behavior.

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] 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.

[6] J. Monteiro & A Oliveira, "Finite State Machine Decomposition For Low Power," *Proceedings of the Design Automation Conference*, pp. 758-763, 1998.

[7] Frank Vahid, Thuy DmLe, and Yu-Chin Hus, "Functional Partitioning improvements over structural partitioning for packaging constraints and synthesis: tool performance," *ACM Transactions on Design Automation of Electronic Systems* 3, 2, pp. 181-208, April 1998.

Chapter 2.   Previous Work

Power reduction techniques can be applied at every level of design abstraction. In this chapter, we will review previous significant contributions on power reduction at various design abstraction level.

2.1.   Transistor Level

Power reduction at the transistor level deals with the physical aspect of the transistor and how they are laid out inside a gate. There are basically two methods for reducing power dissipation at this level: transistor sizing and transistor reordering.

2.1.1.   Transistor Sizing

The size of a transistor can have significant impact on the gate delay and the power dissipated by the gate. The larger the transistor size, the shorter the delay, but more power is consumed. Thus, the goal is to find the smallest transistor or gate that will still satisfy the delay constraint. Work in this area includes [1], [2], [3], [4] and [5].

2.1.2.   Transistor Reordering

Gates (such as a NAND gate) have input pins that are functionally equivalent. In such a case, inputs can be permuted on these pins without affecting the correctness of the

result. However, ordering the gate inputs will affect both the power and the delay of the gate. Consider the transistor implementation of a 2-input NAND gate as shown in Figure 2.1. It was observed that the power consumption for the NAND gate is lower when the inputs to pins A and B are 1 and 0 respectively as opposed to 0 and 1. Thus, an ordering of the inputs using minimal power can be found. Work in this area include [2], [4], [6], [7] and [8]. It was reported in [6] that average power reductions of 12% are achievable using transistor reordering techniques.

2.2. Logic Gate Level

Power reduction at the logic gate level mainly deals with manipulating the logical expression to reduce the number of gates required to implement the expression and to reduce the switching activities. By reducing the number of gates, the overall capacitance in the circuit will also be reduced, thus, the power consumption. Using this technique to reduce power is basically a side effect from reducing the area. Optimization at this level occurs in two phases: technology-independent and technology-dependent optimization.
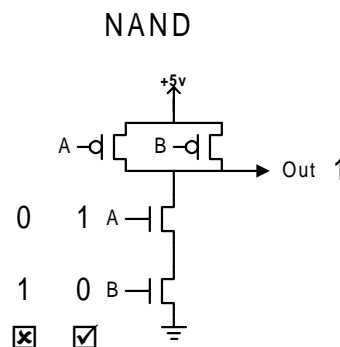
Figure 2.1. Transistor implementation of a 2-input NAND gate.

2.2.1. Technology-independent

In the technology-independent phase, no knowledge of the actual physical gates is assumed; only logic equations are manipulated to reduce the area, number of gates, delay and power consumption. Optimization techniques in this phase include exploiting the don't-care sets for reducing the switching activities as in [9] and [10]; path balancing by adding buffers to reduce glitches as in [11], [12] and [13]; and factorization of logical expressions presented in [14].

Any gate in a combinational circuit has an associated don't-care set where the input combinations either never occur at the gate inputs or that they produce the same values at the circuit outputs. Since the power dissipation of a gate is dependent on the probability of the gate evaluating to a 1 or a 0, this probability can be changed by utilizing the don't-care sets.

It was observed in [13] that spurious transitions account for 10% to 40% of the switching activity in typical combinational logic circuits. In path balancing, the idea is to add delay buffers in a path in order to reduce the glitching activities of a circuit.

Factorization makes use of the fact that factoring an expression can reduce the number of literals, and therefore, reduce the number of transistors required to implement the expression. For example, the expression $a \bullet c + a \bullet d + b \bullet c + b \bullet d$ can be factored into $(a + b) \bullet (c + d)$, thus, reducing the transistor count considerably.

2.2.2. Technology-dependent

In the technology-dependent phase, logic equations are mapped to the target technology library gates, again optimizing for area, number, delay and power. A typical technology library will contain hundreds of gates with different transistor sizes. The problem is to find suitable gates requiring the least amount of power to satisfy the logic equations produced from the technology-independent phase. Much work has been done in this area in terms of area and delay. Work on extending the original approaches to power dissipation include [3], [15], [16] and [17].

2.3.   Register Transfer Level

Register transfer level (RTL) deals with the way data is transferred between registers from one clock cycle to the next. A circuit at the RT level can be broken down into two parts: sequential and combinational. The sequential portion contains the finite state machine (FSM) or controller for the circuit. The combinational or datapath portion contains the functional units, registers and multiplexers for performing the operations. The extent to which hardware is shared and the sequence of variables mapped to each register affect the total switched capacitance in the datapath. Functional units, registers and portions of the controller can be shut off during certain clock cycles to further reduce the power consumption.

2.3.1. Sequential Circuit

For a sequential circuit, the states of a finite state machine can be encoded in such a way that if a state s has a large number of transitions to state t, then the two states should

be given uni-distant codes, so as to minimize the switching activity at the register output. Works in this area include [14], [18], [19] and [20].

Recent work in reducing the power consumption in the sequential circuit uses a partitioning technique on the FSM [21] and [22]. The controller is partitioned into two or more interacting FSMs. Each FSM controls the execution of one section of computation and only one sub-FSM is active at any given clock cycle. Power is saved because the remaining FSMs are idle. This technique was discussed in the introduction.

2.3.2. Combinational Circuit

Switching activity is reduced through appropriate register allocation and binding techniques such as in [23] and [24]. Operand reordering and operand sharing between registers can also reduce switching as in [25] and [26]. The goal of operand reordering is to find an appropriate input operand order for commutative operations in such a way that switching activity is reduced. The operand sharing technique attempts to schedule and bind operations to functional units in such a way that the activity of the input operands is
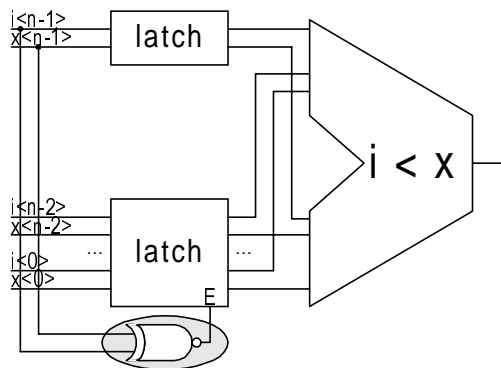
Figure 2.2. Precomputation example of a comparator.

reduced. Operations sharing the same operand are scheduled in control steps as near as possible. Thus, the potential for a functional unit to reuse the same operand value (and, therefore, to decrease its input activity) is higher. Precomputation logic can be added to the datapath to compute the output values for a subset of input conditions as in [27] and [28]. If the output values can be precomputed, then the switching activity in the original circuit can be reduced. Of course the power savings in the original circuit is offset by the power consumed in the extra logic.

An example taken from [27] is a n-bit comparator that computes the function $i < x$. The optimized circuit with precomputation logic is shown in Figure 2.2. In this example, the precomputation logic is the exclusive NOR gate. The comparison can be precomputed using only the most significant bit of the two inputs $i<n-1>$ and $x<n-1>$. Certainly if $i<n-1>$ is less than $x<n-1>$ then $i < x$. Thus, if the output can be determined from this precomputation, then the remaining bits need not be compared. By latching the remaining bits when the condition is satisfied, the amount of switching activities in the comparator is reduced and thus, power is saved.

Similar to disabling portions of the controller, unnecessary functional units and registers can also be shut off during certain cycles in the execution. During these times of unnecessary activity, the clock signal to registers can be stopped or the register can be disabled as in [30] and [31]; inputs to functional units can be latched as in [29].

Figure 2.3 shows an example taken from [29] of an ALU containing an adder and a shifter. A multiplexer is used to select the result from either one of the functional units. In

15

any clock cycle only one of the two functions needs to be computed. However, the multiplexer does the selection after both units have completed their evaluation. The proposed *guarded evaluation* technique to reduce the power from the unnecessary functional unit is to place a transparent latch with an enable at the input to each of the functional units. The input to the functional unit that is not required can be latched and thus unnecessary switching activities can be prevented.

2.4.   Behavioral Level

At the behavioral level, power reduction is obtained mainly by reducing the switching activities using circuit transformations or partitioning. Transformations of the circuit are typically aimed at reducing either the number of cycles in a computation or the number of resources used in the computation during high-level synthesis as in [32], [33], [34], [35] and [36]. The basic idea is to reduce the number of control steps so that slower control clock cycles can be used for a fixed throughput, allowing for a reduction in supply voltage. The reduction in control step requirements is most often possible due to the

(a)   (b)
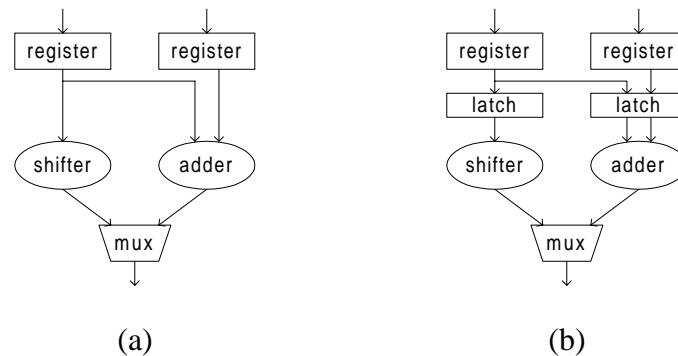
Figure 2.3. Guarded evaluation technique: (a) unoptimized RTL circuit, and (b) optimized with transparent latches.

exploitation of concurrency. Transformations that affect the amount of concurrency include retiming/pipelining, algebraic transformations, and loop transformations.

Partitioning provides the possibility of disabling mutually exclusive or inactive portions of the circuit when not needed in the execution during a certain time as in [21], [22], [29], [37] and [38]. Except for [38], these techniques are applied only to a small portion of the whole circuit. In [21] and [22], only portions of the finite state machine is disabled, and in [29] and [37] only portions of the datapath is disabled. In [38] both the controller and the datapath are disabled.

Coding techniques for reducing the switching activities on the I/O pins and address busses were presented in [39], [40] and [41]. In [41], different bus interfaces including bus width and coding schemes are compared for low power. More parallelism in a circuit can be introduced to speed it up and then reduces the voltage until it realizes its originally required speed as shown in [42].

2.5.   System Level

The main focus for power savings at the system level is from turning off portions of the system that are not being used and thus minimizing the use of power-intensive operations. This includes turning off the monitor and the disk drive [43]; turning off inactive hardware modules [44] and providing optimum supply voltage and/or mixed voltages to the modules [44] and [45]. Communications to and from memory modules can also be minimized as in [46]. Software may be compiled so as to minimize the power dissipation when it is executed on a given hardware platform as shown in [47] and [48].

2.6. Summary

In this chapter, we have reviewed previous significant contributions on power reduction at various design abstraction levels. Recent research in power reduction is focusing more at the higher design abstraction levels.

References

[1] Lin & Hwang, "Power reduction by gate sizing with path-oriented slack calculation," *Proceedings of the 1st Asia-Pacific DAC*, 1995, pp7-12.

[2] C.H. Tan & J. Allen, "Minimization of Power in VLSI Circuits Using Transistor Sizing, Input Ordering, and Statistical Power Estimation," *Proceedings of the Int'l. Workshop on Low Power Design*, April 1994, pp 75-80.

[3] Y. Tamiya, Y. Matsunaga, & M. Fujita, "LP based cell selection with constraints of timing, area and power consumption," *Proc. of the IEEE ICCAD*, 1994, pp378-381.

[4] M. Pedram, "Power estimation and optimization at the logic level," *Int. J. High Speed Electron. Syst.*, June 1994, pp179-202.

[5] Berkelaar & Jess, "Gate sizing in MOS digital circuits with linear programming," *Proc. of the ACM European DAC*, 1990, pp217-221.

[6] E. Musoll & J. Cortadella, "Optimizing CMOS Circuits for Low Power using Transistor Reordering," *European Design & Test Conference*, 1996

[7] Shen, Lin, & Wang, "Transistor reordering rules for power reduction in CMOS gates," *ICCAD*, 1995.

[8] Prasad & Roy, "Circuit optimization for minimization of power consumption under delay constraint," *Proc. of the 1994 Int. Workshop on Low Power Design*, pp15-20.

[9] A. Shen, S. Devadas, A. Ghosh, & K. Keutzer, "On Average Power Dissipation and Random Pattern Testability of Combinational Logic Circuits," *Proc. of the Int. Conf. on CAD*, Nov. 1992, pp402-407.

[10] S. Iman & M. Pedram, "Multi-Level Network Optimization for Low Power," *Proc. of the Int. Conf. on CAD*, Nov. 1994, pp371-377.

[11] C. Lemonds & S. Shetti, "A Low Power 16 by 16 Multiplier Using Transition Reduction Circuitry," *Proceedings of the Int'l Workshop on Low Power Design*, April 1994, pp 139-142.

[12] Murgai, Brayton, & Sangiovanni-Vincentelli, "Decomposition of Logic Functions for Minimum Transition Activity," *EDTC*, 1995, pp

[13] A. Ghosh, S. Devadas, K. Keutzer, and J. White, "Estimation of Average Switching Activity in Combinational and Sequential Circuits," *Proceedings of the Design Automation Conference*, pp. 253-256, June 1992.

[14] K. Roy & S. Prasad, "SYCLOP: Synthesis of CMOS Logic for Low Power Applications*," Proc. Of the Int. Conf. On Comp. Design: VLSI in Computers and Processors*, Oct. 1992, pp464-467.

[15] V. Tiwari, P. Ashar & S. Malik, "Technology Mapping for Low Power," *Proc. of the 30th DAC*, June 1993, pp74-79.

[16] C.Y. Tsui, M. Pedram & A. Despain, "Technology Decomposition and Mapping Targeting Low power Dissipation, *Proc. of the 30th DAC*, June 1993, pp68-73.

[17] B. Lin, "Technology Mapping for Low Power Dissipation," *Proc. of the Int. Conf. On Comp. Design: VLSI in Computers and Processors*, Oct. 1993.

[18] S.H. Chow, Y.C. Ho, T.T. Hwang, & C.L. Liu, "Low Power Realization of Finite State Machines – A Decomposition Approach," *ACM Trans. on Design Automation of Electronic Systems*, July 1996, pp315-340.

[19] C.Y. Tsui, M. Pedram, C.A. Chen, & A.M. Despain, "Low Power State Assignment Targeting Two- and Multi-level Logic Implementations," *Proc. of the Int. Conf. on CAD*, Nov. 1994, pp82-87.

[20] G.D. Hachtel, M. Hermida, A. Pardo, M. Poncino, & F. Somenzi, "Re-Encoding Sequential Circuits to Reduce Power Dissipation," *Proc. of the Int. Conf. on CAD*, Nov. 1994, pp70-73.

[21] L. Benini, P. Vuillod, G. DeMicheli, & Claudionor Coelho, "Synthesis of Low-Power Selectively-Clocked Systems from High-Level Specification," *ISSS*, 1996, p57-63.

[22] J. Monteiro & A. Oliveira, "Finite State Machine Decomposition for Low Power," *Proceedings of the Design Automation Conference*, pp. 758-763, 1998.

[23] Jui-Ming Chang, Massoud Pedram, "Register Allocation and Binding for Low Power", *Proceedings of the Design Automation Conference*, pp. 29-35, 1995.

[24] E. Musoll & J. Cortadella, "Scheduling and Resource Binding for Low Power," *Proceedings of the International Conference on Computer Aided Design,* pp104-109, 1995.

[25] E. Musoll & J. Cortadella, "High-level synthesis techniques for reducing the activity of functional units," *ISLPD*, 1995.

[26] Kim & Choi, "Power-conscious high level synthesis using loop folding," *DAC*, pp.441-445, 1997.

[27] M. Alidina, J. Monteiro, S. Devadas, A. Ghosh, & M. Papaefthymiou, "Precomputation-Based Sequential Logic Optimization for Low Power," *IEEE Transactions on VLSI Systems*, pp426-436, December 1994.

[28] J. Monteiro, J. Rinderknecht, S. Devadas, & A. Ghosh, "Optimization of Combinational and Sequential Logic Circuits for Low Power Using Precomputation," *Proc. of the 1995 Chapel Hill Conf. on Advanced Research on VLSI*, March 1995.

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

[30] G. Tellez, A. Farrahi, & M. Sarrafzadeh, "Acitvity-driven clock design for low power circuits," *Proc. of the IEEE Int. Conf. on CAD*, 1995, pp62-65.

[31] T. Lang, E. Musoll, & J. Cortadella. "Reducing Energy Consumption of Flip-Flops," *Universitat Politècnica de Catalunya DAC Technical Report 4,* 1996.

[32] A. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, & R. Brodersen, "Optimizing Power Using Transformations," *IEEE Trans. on CAD of IC and Systems*, Jan. 1995, pp12-31.

[33] A. Chandrakasan, M. Potkonjak, J. Rabaey, & R. Brodersen, "Hyper-LP: A System for Power Minimization Using Architectural Transformations," *Proceedings of the International Conference on Computer Aided Design*, pp. 300-303, 1992.

[34] A. Raghunathan, & N. Jha, "Behavioral synthesis for low power," *Proceedings of the International Conference on Computer Aided Design*, Oct. 1994, pp318-322.

[35] A. Raghunathan, & N. Jha, "ILP formulation for Low Power Based on Minimizing Switched Capacitance during Data Path Allocation," *Proc. of the Int. Sym. on Circuits & Systems*, 1995.

[36] R. Mehra & J. Rabaey, "Exploiting regularity for low-power design," *ICCAD*, 1996, pp166-172.

[37] Vaishnav & Pedram, "Delay Optimal Partitioning Targeting Low Power VLSI Circuits," *ICCAD*, 1995, pp638-643.

[38] Enoch Hwang, Frank Vahid, & Yu-Chin Hsu, "FSMD Functional Partitioning for Low Power," *Proceedings of the Design, Automation and Test in Europe*, pp. 22-28, March 1999.

[39] Su, Tsui, & Despain, "Low power architecture design and compilation techniques for high-performance processors," *COMPCON 1994 Digest of Technical Papers*, 1994, pp489-498.

[40] Stan & Burleson, "Limited-weight codes for low power I/O," *Proc. of the 1994 International Workshop on Low-Power Design*, 1994, pp209-214.

[41] Tony Givargis, & Frank Vahid, "Interface Exploration for Reduced Power in Core-Based Systems, "*International Symposium on System Synthesis*, pp. 117-122, 1998.

[42] A. Chandrakasan, S. Sheng, & R. Brodersen, "Low-power CMOS design," *IEEE J. Solid-State Circuits*, 1992, pp472-484.

[43] Inki Hong & Miodrag Potkonjak, "Power Optimization in Disk-Based Real-Time Application Specific Systems," *ICCAD*, 1996, p634-637.

[44] Chandrakasan, Sheng & Brodersen, "Low-power techniques for portable real-time DSP applications," *Proc. of VLSI Design*, 1992.

[45] R. Martin & J. Knight, "Power-profiler: optimizing ASICs power consumption at the behavioral level," *Proc. of the 32nd DAC*, 1995, pp42-47.

[46] Wuytack, Catthoor, Franssen, Nachtergaele, & DeMan, "Global communication and memory optimizing transformations for low power systems," *Proc. of the 1994 Int. Workshop on LP Design*, 1994, pp203-208.

[47] V. Tiwari, S. Malik, & A. Wolfe, "Power analysis of embedded software: a first step towards software minimization," *IEEE Trans. VLSI Sys.*, Dec 1994, pp437-445.

[48] V. Tiwari, S. Malik & A. Wolfe, "Compilation Techniques for Low Energy: an overview," *Proc. of IEEE Symposium on Low Power Electronics*, Oct. 1994, pp38-39.

Chapter 3.   Power Consumption

In this chapter, we will look at the various factors affecting power consumption in a circuit, how power is calculated in a circuit, and finally show how partitioning a circuit can reduce power consumption.

3.1.   Power Dissipation

Power consumption of a CMOS circuit is composed of three components [1]: 1) dynamic power consumption due to capacitive charging and discharging when a signal toggles (P_d); 2) dynamic power consumption due to short circuit dissipation (P_{sc}); and 3) static power consumption due to leakage currents (P_s). Thus, the total power consumption P of a CMOS circuit is

$$P = P_d + P_{sc} + P_s \tag{3.1}$$

Every time when the output of a gate switches from a '0' to a '1' or vice versa, the loading capacitors (from the gates that are connected to this output) need to be charged or discharged. This gives the dynamic power term P_d. Also during this time when the gate switches, there is a moment when there is a path created between the power supply and ground, thus, causing the short circuit dissipation P_{sc}. During times of inactivity, current

is still being drawn because of parasitic diodes within a gate. This contributes to the static leakage current P_s. The following sections will look at these three terms in more detail.

3.1.1. Static Power Dissipation

Consider a complementary CMOS gate as shown in Figure 3.1. If the input V_{in} = '0,' the associated n-device is "OFF" and the p-device is "ON." The output voltage is V_{DD} or logic '1.' When the input V_{in} = '1,' the associated n-channel device is biased "ON" and the p-channel device is "OFF." In this case the output voltage is 0 volts (V_{SS}). Since one of the transistors is always "OFF" when the gate is in either of these logic states, there is no DC current path from V_{DD} to V_{SS}, and the resultant quiescent (steady-state) current, and hence the static power P_s, is zero.

However, there is some small static dissipation due to reverse bias leakage between diffusion regions and the substrate. A profile of an inverter shown in Figure 3.2 shows how the source-drain diffusions and the n-well diffusion form parasitic diodes with the p-substrate. Since parasitic diodes are reverse-biased, their leakage current contributes to the static power dissipation. The static power dissipation is the product of the device
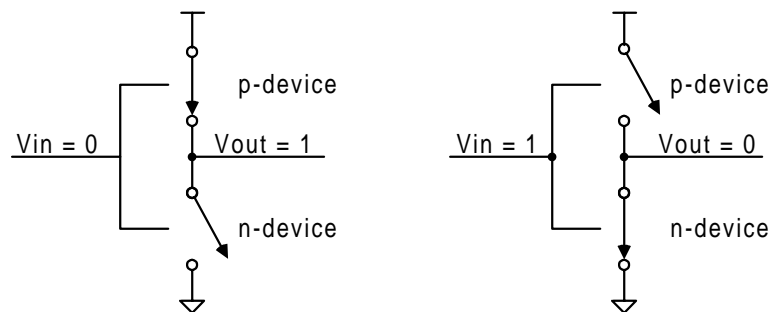
Figure 3.1. CMOS inverter model for static power dissipation evaluation.

leakage current and the supply voltage. Thus, the total static power dissipation, P_s, is obtained from

$$P_s = \sum_{1}^{n} leakage\ crrent \times V_{DD} \tag{3.2}$$

where

$$n = \text{number of devices.}$$

A useful estimate is to allow a leakage current of $0.1nA$ to $0.5nA$ per device at room temperature.

3.1.2. *Dynamic Power Dissipation due to Short-Circuit*

When a gate switches from a '0' to a '1' or vice versa, there is a moment when there is a path created between the power supply and ground, thus, causing the short circuit dissipation P_{sc}. The short-circuit power dissipation is given by
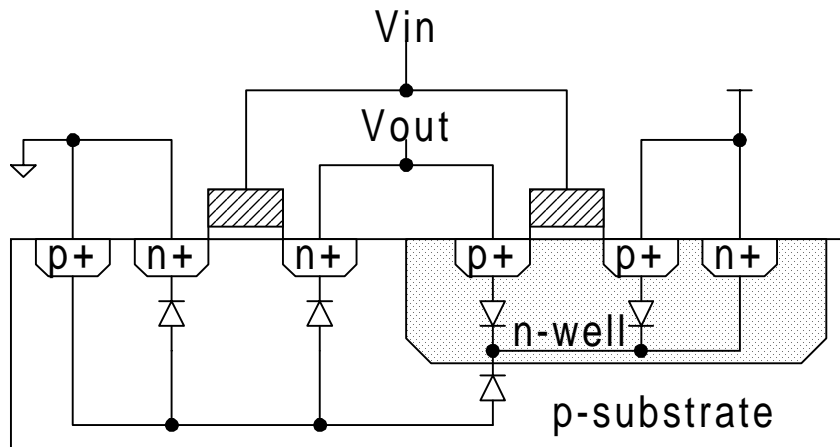
Figure 3.2. Parasitic diodes in a CMOS inverter.

25

$$P_{sc} = \frac{\beta}{12}(V_{DD} - 2V_t)^3 \frac{t_{rf}}{t_p}$$
(3.3)

where

β = gain factor of the transistor which is dependent on both the transistor manufacturing process parameters and the device geometry, and is given by

$$\beta = \frac{\mu\varepsilon}{t_{ox}}\left(\frac{W}{L}\right)$$

where μ is the dielectric constant, ε is the permittivity of the gate insulator, t_{ox} is the thickness of the gate insulator, W is the width of the channel, and L is the length of the channel.

V_t = threshold voltage ≈ 0.7v.

t_{rf} = rising and falling time of the input waveform (assuming that they are equal).

t_p = period of the input waveform.

As the load capacitance on the output of the gate is increased, the significance of the short-circuit dissipation is reduced by the capacitive dissipation P_d.

3.1.3. *Dynamic Power Dissipation due to Capacitive Charging and Discharging*

Each time during a signal transition from either '0' to '1' or, alternatively, from '1' to '0', current is required to charge or discharge the output capacitive load. This dynamic dissipation can be modeled by assuming that the rise and fall time of the step input is much less than the repetition clock period t_p. The average dynamic power P_d dissipated

during switching for a square-wave input V_{in}, and having a repetition frequency of $f_p = 1/t_p$, is given by

$$P_d = \frac{1}{t_p} \int_0^{t_p/2} i_n(t) V_{out} dt + \frac{1}{t_p} \int_{t_p/2}^{t_p} i_p(t)(V_{DD} - V_{out}) dt \qquad (3.4)$$

where

    i_n = n-device transient current.

    i_p = p-device transient current.

For a step input and with $i_n(t) = C_L \, dV_{out}/dt$, and $i_p(t) = C_L \, d(V_{DD} - V_{out})/dt$ where C_L = load capacitance, we get the equation

$$\begin{aligned}
P_d &= \frac{C_L}{t_p} \int_0^{Vdd} V_{out} \, dV_{out} + \frac{C_L}{t_p} \int_{Vdd}^{0} (V_{DD} - V_{out}) \, d(V_{DD} - V_{out}) \\
&= \frac{C_L V_{DD}^2}{t_p} \\
&= C_L V_{DD}^2 f_p
\end{aligned} \qquad (3.5)$$

In the behavioral context, this translates to

$$P_d = \frac{1}{2} C V_{DD}^2 f N \quad \text{watt} \qquad (3.6)$$

where C is the total loading capacitance of the gate output, V_{DD} is the supply voltage, f is the clock frequency, and N is the transition probability of the gate output. The transition probability or switching frequency is defined as the average number of gate output transitions per clock cycle and is given by

$$N_i = \frac{t_p n_i}{T_{exec}} \tag{3.7}$$

where $t_p = 1/f$ is the clock period, n_i is the total number of toggles at net i, and T_{exec} is the total execution time. These values are illustrated in Figure 3.3.

3.2.   Power Calculation

To simplify the total power calculation, many of the power calculation tools for complex circuits only consider P_d as an approximation to the total power because it has been shown that P_d accounts for over 90% of the total power [2].

Given a digital circuit, P_d is calculated for all the nets in the circuit and the sum of them is the average power consumption for the entire circuit
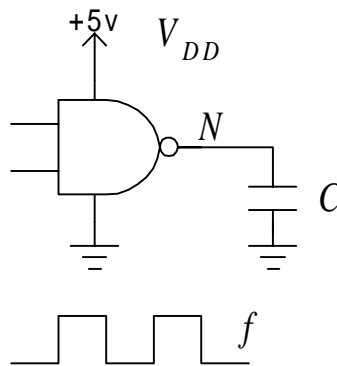
Figure 3.3. Variables affecting power consumption.

$$Power = \frac{1}{2t_p} V^2 \sum_{\forall i} C_i N_i \quad \text{watts}$$

$$= \frac{1}{2t_p} V^2 \sum_{\forall i} C_i \frac{t_p n_i}{T_{exec}} \tag{3.8}$$

$$= \frac{V^2}{2T_{exec}} \sum_{\forall i} C_i n_i$$

where i is an individual net in the circuit.

The total energy consumed for the entire circuit is

$$Energy = Power \times T_{exec} \quad \text{watt - second}$$

$$= \frac{V^2}{2T_{exec}} \sum_{\forall i} C_i n_i T_{exec} \tag{3.9}$$

$$= \frac{V^2}{2} \sum_{\forall i} C_i n_i \quad \text{joules}$$

Hence, to evaluate the energy usage of a circuit, we need to know the number of toggles and the loading capacitance for each net in the circuit, and the voltage used. The number of toggles can be obtained by simulation and counting the number of times the signal switches for each net. The capacitance for each gate can be obtained from the technology library used for the synthesis of the circuit. Knowing the capacitance for each gate, the loading capacitance for each net can be calculated from analyzing the netlist to see how the gates are connected.

For example, the netlist of Figure 3.4 has four gates and six nets. The nets are annotated with the toggle count and the gates are annotated with the gate capacitance. Assuming that the operating voltage is 5V, the energy consume by this netlist is:

$$Energy = \frac{5V^2}{2}\left[\begin{array}{l}(175\times3\times10^{-10}F)+(150\times3\times10^{-10}F)+(100\times2.8\times10^{-10}F)\\+(125\times(2\times10^{-10}F+2.8\times10^{-10}F)+(125\times3\times10^{-10}F)+(90\times3\times10^{-10}F)\end{array}\right]$$

$$= \frac{5V^2}{2}\left[\begin{array}{l}525\times10^{-10}F+450\times10^{-10}F+280\times10^{-10}F\\+600\times10^{-10}F+375\times10^{-10}F+270\times10^{-10}F\end{array}\right]$$

$$= \frac{5V^2}{2}\left[2500toggles\times10^{-10}F\right]$$

$$= 3.125\times10^{-6}J$$

$$= 3.125\mu J$$

3.3.   Power Reduction

Power reduction for a system can be achieved from all levels of the system design. At the lower levels of the design process, power reduction is obtained mainly through the reduction of the capacitance in the circuit. At the behavioral level, power reduction is obtained mainly from reducing the switching activities within the circuit.
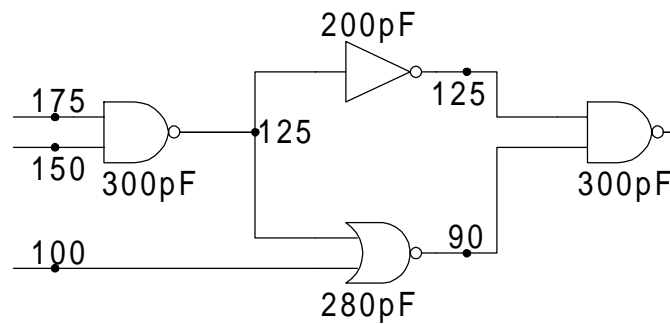
Figure 3.4. Sample netlist with four gates and six nets. The gates are annotated with the gate capacitance and the nets are annotated with the toggle count.

Switching activities in a circuit are reduced primarily by eliminating useless switching activities through proper power management. This can be accomplished either by disabling portions of the system that are not performing useful work, or by partitioning the system into several parts such that the switching activities are localized in only one part at a time. Partitioning is possible if the parts are mutually exclusive in their execution. Thus, only one part needs to be active at any one time while the remaining parts are inactive. This is analogous to deactivating certain devices in the system that is not needed, for example, powering down the disk drive when not in use.

An easy and effective method to disable a part is to prevent any changes to the inputs of the part. This in turn will prevent switching activities within the part. Thus, when the inputs to the circuit do not change, then there will be no switching activities in the entire circuit. For example, if the initial inputs to a NAND gate are all 1's, then the output is a 0. This output will remain at a 0 if the inputs do not change. Since this output is connected to the inputs of other logic gates, all the outputs to the other gates will also remain the same. Thus, there will be switching activities in the circuit only if the inputs
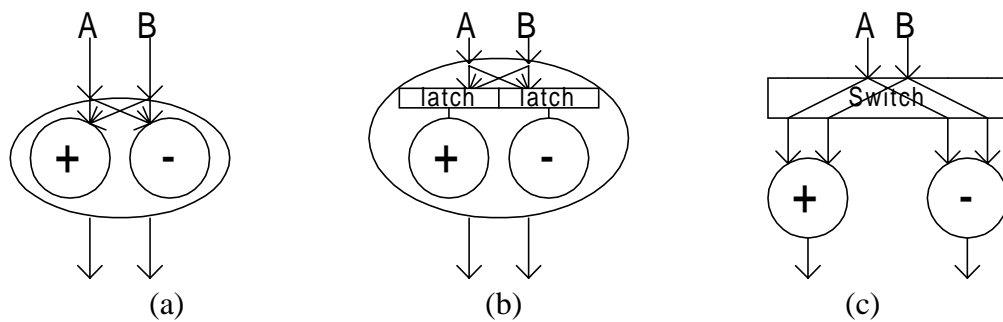
Figure 3.5. Combinational logic with: (a) common inputs, (b) latched inputs, and (c) partitioned inputs.

31

change in value. Of course the reverse is not always true. For example, for a two input NAND gate, the output is a 1 for inputs {0,0}, {0,1}, and {1,0}.

Consider a combinational logic circuit containing two functional units with common inputs as shown in Figure 3.5(a). Assume that the computation requires the evaluation of an addition followed by a subtraction. While performing the addition, the input signals will also propagate to the gates in the subtraction unit. Thus, the subtraction unit is using power even though the result from the subtraction unit is not needed. Similarly when the subtraction is being performed, the addition unit is consuming power but the result from the addition unit is not needed.

There are basically two general methods to reduce the useless switching activities. In the first method, we can insert transparent latches at the inputs of the two functional units as shown in Figure 3.5(b). The latches can prevent the inputs to the functional unit that is not needed for a particular cycle from changing. For this method, the two functional units and the newly added latches are all within the same part. In the second method, we can partition the circuit such that the switching activities are localized in only one part at a time. Thus, we would put the addition unit and the subtraction unit in two different parts so that the input to one will not affect the input to the other as shown in Figure 3.5(c). The switch will control which part gets the input. When the inputs to a part remain constant there will be no switching activities within the entire part thus power is reduced.

As we will see in the following sections, the second method will result in more power reductions than the first method because the first method is only applied to the

combinational logic in the datapath, whereas, the second method is applied to the entire processor, namely the datapath *and* the controller. Thus, we want to apply this partitioning idea to the whole processor.

3.4.   Summary

In this chapter, we have shown how power is consumed and calculated in a CMOS circuit. The idea of partitioning a circuit was also introduced for power reduction.

References

[1] Neil H. E. Weste, & Kamran Eshraghian, *Principles of CMOS VLSI Design*, Addison-Wesley Publishing, California, 1993.

[2] 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.

Chapter 4.   Procedural Functional Partitioning

In this chapter, we will look at procedural functional partitioning and how power is reduced using this technique. In order to understand this, we need to have a general concept of the behavioral synthesis process. We will then show why the traditional method of structural partitioning does not reduce power and then show how functional partitioning is different and thus can reduce power.

4.1.   Behavioral Synthesis

Synthesis is the process of transforming and optimizing a digital circuit design from a high level of abstraction to a lower level of abstraction. In behavioral synthesis, the input is a behavioral description of the design specified in a Hardware Description Language (HDL) such as VHDL or Verilog. The synthesis process translates the behavioral description first into a structural description and finally to a physical gate level circuit netlist as shown in Figure 4.1. Figure 4.1(a) shows a behavioral description of a sample segment of code. Figure 4.1(b) shows the corresponding structural description with the separate FSM control unit and the datapath. Figure 4.1(c) shows the physical layout. The

```
if IR(3) = '0' then
   PC := PC + 1;
else
   BUF := MEM(PC);
   MEM(SP) := PC + 1;
   SP := SP - 1;
   PC := BUF;
end if;
```

(a)
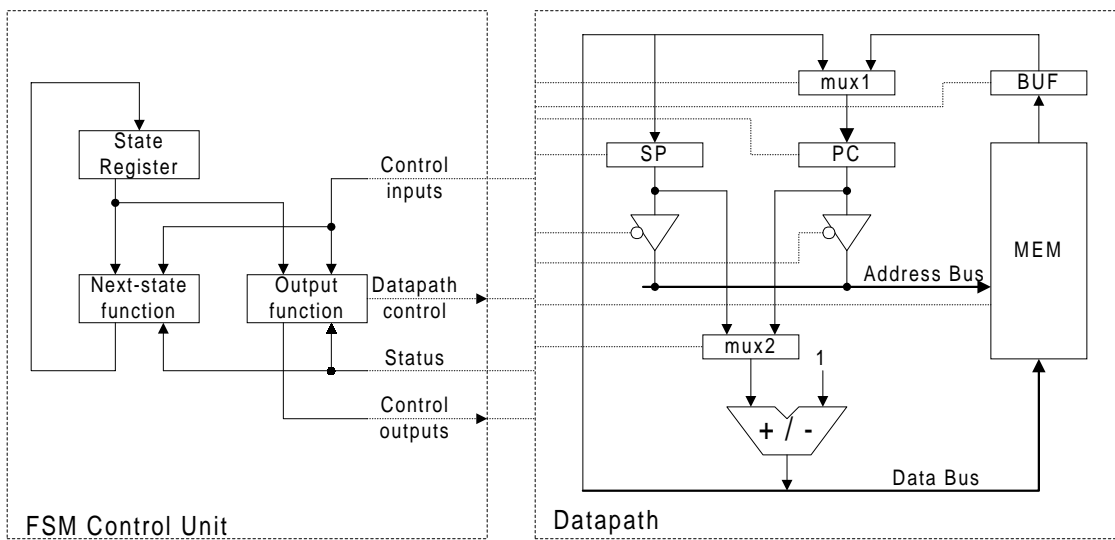
(b)
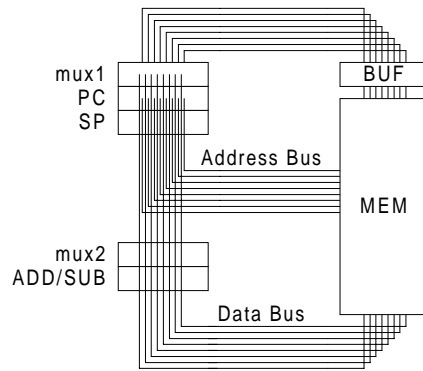
(c)

Figure 4.1. Example of the three levels of abstraction: (a) behavioral, (b) structural, and (c) physical.

major steps in the synthesis process include scheduling, allocation, and finally the generation of the gate level circuit netlist.

The internal data representation of a behavioral description is usually a control data flow graph (CDFG), which captures all the control and data-flow dependencies of the given behavioral description. Scheduling algorithms then partition this CDFG into subgraphs so that each subgraph is executed in one control step. Each control step corresponds to one state of the controlling finite state machine. Within a control step, a separate functional unit is required to execute each operation assigned to that step. Thus, the total number of functional units required in a control step directly corresponds to the number of operations scheduled in it. If more operations are scheduled into each control step, more functional units are necessary, which results in fewer control steps for the design implementation.

Allocation consists of two tasks: unit selection and unit binding. Unit selection determines the number and types of components to be used in the design. These components can be either functional units, storage elements, or interconnect wires. Unit binding maps the variables and operations in the scheduled CDFG to the selected components. For every operation in the CDFG, we need a functional unit that is capable of executing the operation. For every variable that is used across several control steps in the scheduled CDFG, we need a storage unit to hold the data values during the lifetime of the variable. Finally, for every data transfer in the CDFG, we need a set of interconnection units to effect the transfer.

When a circuit is synthesized from a behavioral description to a gate level netlist, only one FSM control unit and one datapath is generated as shown in Figure 4.1(b). The control unit consists of the state register, logic for generating the next state, and logic for generating control signals to control the operation of the datapath. The datapath consists of the data registers, functional units, multiplexers, and connecting wires for executing the operations of the specified behavioral instructions. At each cycle or time step, control signals from the control unit is sent to the datapath to perform the operations scheduled for that cycle. The state register is then updated by the next-state function in the control unit and the cycle repeats.

4.2. Structural Partitioning

Partitioning a design has been used as a solution to many circuit packaging problems.
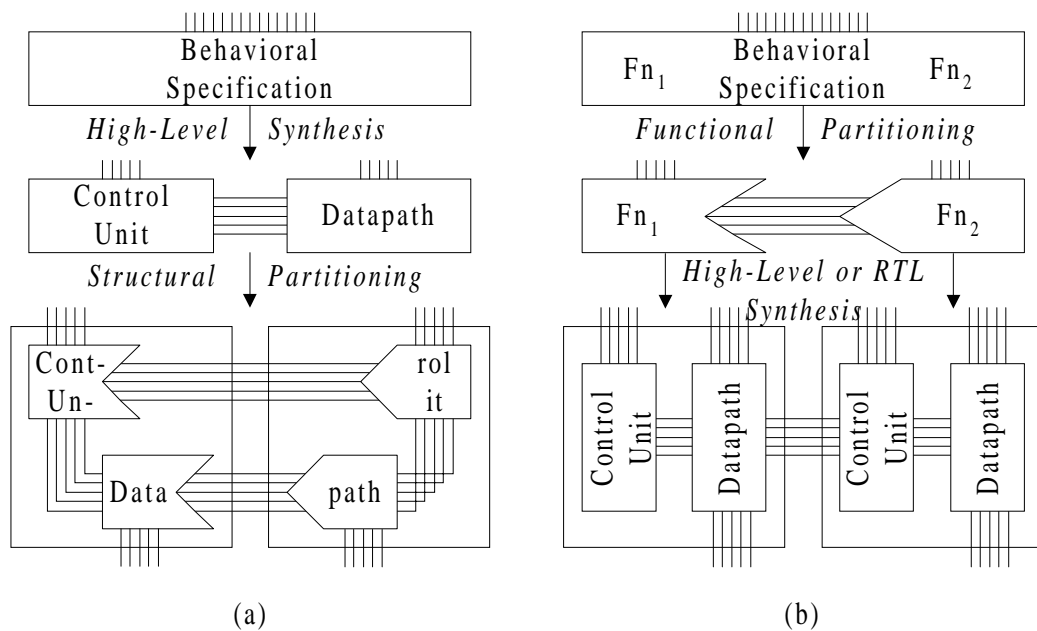
Figure 4.2. Partitioning: (a) structurally, (b) functionally.

Traditionally, partitioning a circuit is performed at the structural level. In structural partitioning (also known as circuit or netlist partitioning), as shown in Figure 4.2(a), a circuit is first synthesized from the behavioral level to the structural or gate level netlist. The partitioning is then performed at the gate level.

From a power reduction perspective, structural partitioning does not reduce switching activity. The reason is that behavioral synthesis, as mentioned in the previous section, generates only one control unit and one datapath from the design specification. Even though partitioning the netlist creates more than one physical partition, there is still logically only one processor consisting of one datapath and one control unit. When a primary input signal changes, the entire datapath may be affected regardless of which partition they are in. Moreover, all the gates in the control unit must also be active in order to provide the correct control signals to the datapath. Thus, even though, we have more than one part, the switching activities are not localized within a part. This results in much unnecessary switching activities and so power is not reduced.

Consider an unpartitioned processor containing two procedures, *Procedure1* and *Procedure2*, with common primary inputs as shown in Figure 4.3(a). Suppose we want to evaluate *Procedure1* followed by *Procedure2* sequentially. While performing *Procedure1*, the input signals will also propagate to the gates in *Procedure2* because they are all interconnected. A hypothetical switching activities for the gates of the control unit and datapath for both *Procedure1* and *Procedure2* are shown in Figure 4.3(b). Thus, power is being used by *Procedure2* although the result from it is not needed. In fact, the

39

amount of power used by *Procedure2* is the same regardless of whether the result is

needed or not. Similarly when *Procedure2* is being performed, *Procedure1* is consuming

power but the result from it is not needed.

If we take a hypothetical situation where both procedures require 1μW of power and

1μsec to execute, then for both procedures to execute sequentially, a total of 2μW of
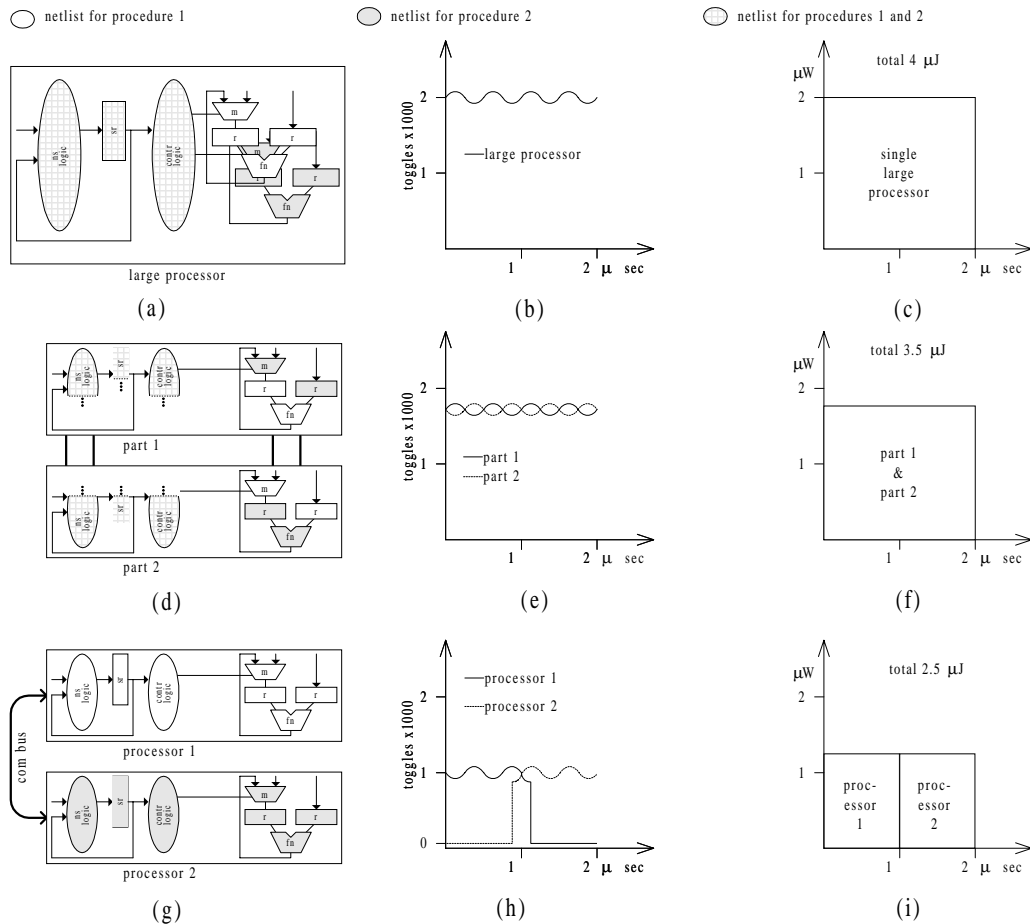
Figure 4.3. Netlist with control unit and datapath, hypothetical switching activity, and power usage of processor with two procedures: (a) single large processor with two procedures; (b) switching activity of large processor; (c) power usage of large processor; (d) structural partitioning of processor; (e) switching activity of (d); (f) power usage of (d); (g) functional partitioning of processor; (h) switching activity of (g); (i) power usage of (g).

power is required for 2μsec resulting in a total of 4μJ of energy being consumed by the single large processor as illustrated in Figure 4.3(c).

Figure 4.3(d) shows the result of structural partitioning the large processor of Figure 4.3(a). Here, the single datapath and control unit netlist is divided into two parts resulting in gates from both *Procedure1* and *Procedure2* to be spread across the two parts. As a result, even if we need the result from only one procedure, there will still be switching activities from both parts. Thus, the switching activities and energy consumption are decreased only slightly from the single large processor as shown in Figure 4.3(e) and Figure 4.3(f).

4.3.   Procedural Functional Partitioning

In procedural functional partitioning, the focus is on partitioning coarse-grained functions and procedures [1]. The behavioral process is first partitioned into several smaller mutually exclusive parts. Each of these smaller parts is then synthesized to its own custom processor, having its own controller and datapath as shown in Figure 4.2(b). From a power reduction perspective, functional partitioning can significantly reduce switching activity. The main reason is that partitioning occurs *before* synthesis, hence each part is a processor containing its own control unit and datapath. Each processor is now smaller than the original large processor implementing the entire process, and only one processor is executing a computation at any given time. Thus, at a given time, switching activity is limited to only one small processor; the other processors will be idle. A processor is made idle by preventing its primary inputs from changing as discussed in

Section 3.3. When a processor is idle, we have, in effect shut down both the controller and the datapath for that processor. Thus, power reduction is possible through functional partitioning because it reduces the overall switching activities of the entire system by localizing the activities within smaller processors, hence, power consumed per operation is less.

In our hypothetical example, we would first partition *Procedure1* and *Procedure2*. Synthesis is then applied to the two parts individually resulting in two separate processors having there own control units and datapaths as shown in Figure 4.3(g). Being two separate processors, the inputs to one will now have no effect to the inputs of the other. With two smaller mutually exclusive processors, the total amount of switching activities at any one time is reduced by about half as shown in Figure 4.3(h). However, partitioning introduces new switching activities for inter-processor communication. It is only during inter-processor communication that both processors will have switching activities at the same time. Thus, in our hypothetical example, each processor might now consume 1.25μW of power, resulting in a total energy of 2.5μJ as shown in Figure 4.3(i).

4.4. Procedural Functional Partitioning Example

We performed an experiment to compare the switching activities between an unpartitioned and a partitioned system. The example shown is for a factorization problem. Given the pseudo-code shown in Figure 4.4, we can partition it into three parts as shown in Figure 4.5. The program calls three separate procedures, *mod*, *divide*, and

```
input x;
count := 2;
while ((count * count) <= x) loop
 mod(x,count,mod_result);
 divide(x,count,divide_result);
 is_prime(count,prime_result1);
 is_prime(divide_result,prime_result2);
 if (mod_result = 0) and (prime_result1 = 1) and
   (prime_result2 = 1) then
   answer1 <= divide_result;
   answer2 <= count;
   exit;
 else
   count := count + 1;
 end if;
end loop;
```

Figure 4.4. Sample unpartitioned pseudo-code.

```
part 1
 …
 while ((count * count) <= x) loop
   call and wait for result from part 2;
   call and wait for result from part 3;
   if …
 end loop;

part 2
 wait for part 1 to call;
 get parameters x and count;
 mod(x,count,mod_result);
 divide(x,count,divide_result);
 return mod_result and divide_result;

part 3
 wait for part 1 to call;
 get parameters count and divide_result;
 is_prime(count,prime_result1);
 is_prime(divide_result,prime_result2);
 return prime_result1 and prime_result2;
```

Figure 4.5. Sample partitioned pseudo-code.

is_prime. We can put *mod* and *divide* in part two, *is_prime* in part three and the remaining code in part one. Part one controls the entire program flow. It performs the primary I/O, and calls the other parts when required. The part call includes the activation of the called part and passing the parameters to the called part. The part return will pass the results back to part one.

Figure 4.6 and Figure 4.7 show a plot of the switching activities for the unpartitioned and partitioned *Fac* example as shown in Figure 4.4 and Figure 4.5 respectively. Here we actually see the decrease in switching activities as a result of the partitioning. In the unpartitioned case, Figure 4.6, the number of toggles can go as high as 1601 in a clock cycle with an average of 141. The total power consumption is 19 mWatt. In the partitioned case, Figure 4.7, the maximum number of toggles in a clock cycle is only
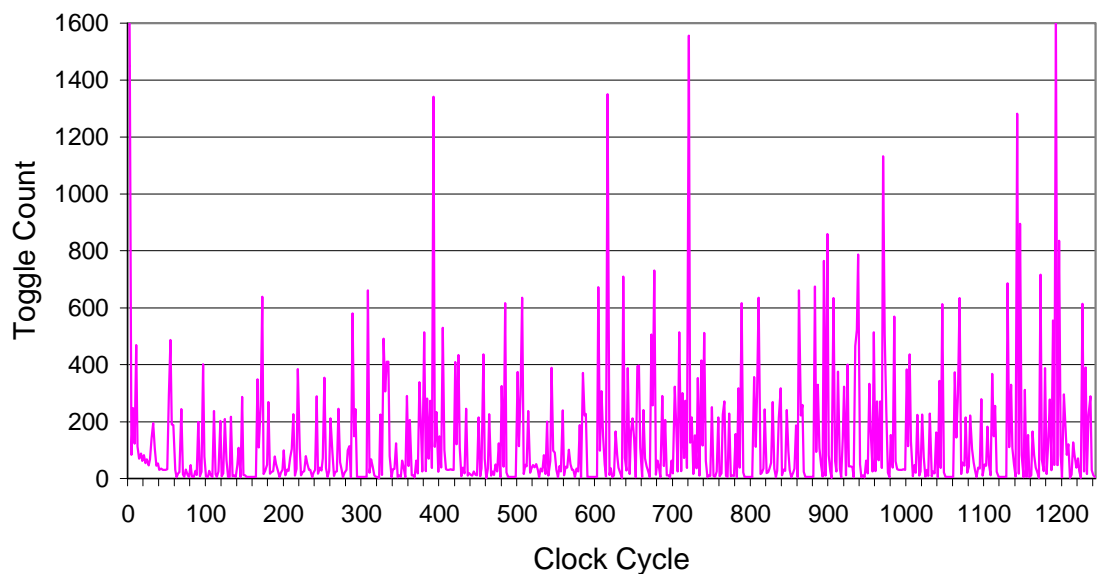
Figure 4.6. Plot of switching activities for the *Fac* unpartitioned example. Total power is 19 mWatt.

1353 with an average of 66. The total power consumption for this case is only 11 mWatt, a 41% reduction in power consumption.

In the partitioned plot, Figure 4.7, we can see that part two is called four times (at clock cycle 60, 380, 680, and 900). Part three is called nine times (at clock cycle 290, 550, 600, 790, 840, 1010, 1070, 1130, and 1190). Part one is the main controlling module, which is active only during the communication with the other two parts. We see that while part two is active, parts one and three do not have any switching activities whatsoever. Similarly, when part three is active, parts one and two are completely inactive. The only time when all three parts have switching activities is when handshaking signals are occurring over the communication bus. Notice also that the number of clock cycles is the same for both the unpartitioned and the partitioned system.
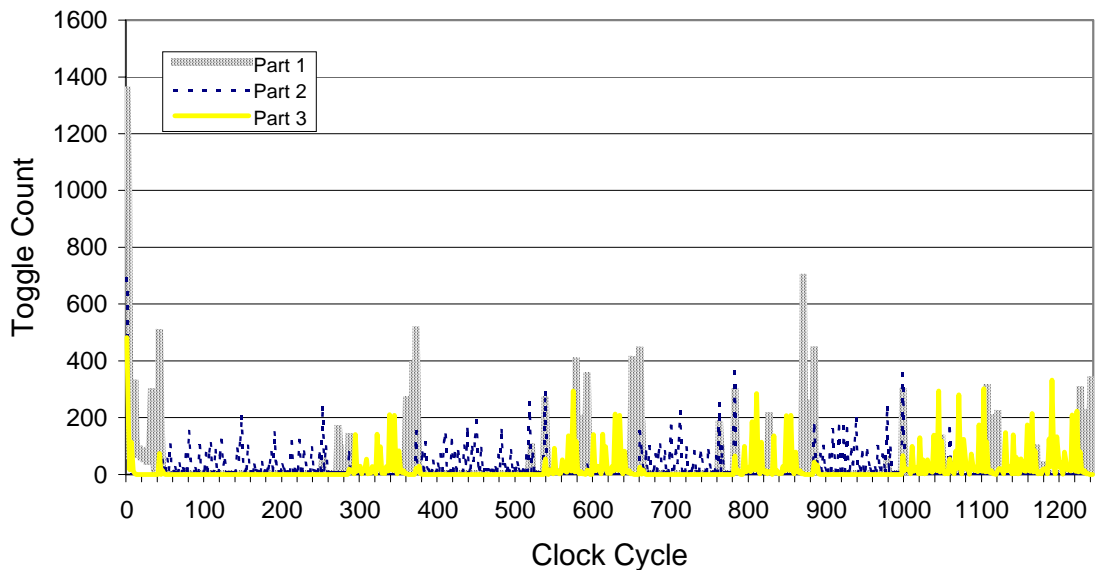
Figure 4.7. Plot of switching activities for the *Fac* partitioned example. Total power is 11 mWatt.

4.5. Summary

In this chapter, we have presented a procedural functional partitioning technique for reducing power consumption. We showed that the traditional structural partitioning technique cannot reduce power, whereas, the procedural functional partitioning technique can reduce power by as much as 41%.

References

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

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

Chapter 5.   FSMD Functional Partitioning

Many circuit designs are still specified at the register-transfer level using the finite-state machine with datapath (FSMD) model in which the behavior has been scheduled into states. The reason is that designers feel that they need more control over the cycle-by-cycle execution of the circuit, and this is something that a behavioral description and synthesis does not offer. In this chapter, we will describe a FSMD functional partitioning technique where we will apply the procedural functional partitioning technique described in the previous chapter to the FSMD model. We will first give a formal definition of a FSMD and then our FSMD functional partitioning technique will be described.

5.1.   FSMD Definition

A finite state machine with datapath differs from a traditional FSM in that it may include variables with various data types, as well as complex data operations in its actions. We can think of an FSMD as a behavioral process that has been scheduled, able to represent both control and data. Synthesis of the FSMD will split it into an FSM (representing control only) and a datapath (representing data only). A sample behavioral description of an FSMD is shown in Figure 5.1.

Formally, a finite state machine with datapath is a 6-tuple [1]

$$P = <S, s_0, I \cup STAT, O \cup A, \delta, \lambda> \qquad (5.10)$$

where:

$S = \{s_0, \ldots, s_m\}$ is a finite set of states.

$s_0 \in S$ is the reset state.

$I = \{i_j\}$ is a set of primary input values.

$STAT = \{\mathrm{Rel}(a, b) : a, b \in EXP\}$ is a set of status signals as logical relations between

```
loop
 case State_Var is
   when s0  =>
     p := 1 ;
     i := 1 ;
     if (1 < x) then -- x is a primary input
        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  =>
     if (i < x) then
        State_Var := s1 ;
     else
        State_Var := s3 ;
     end if ;
   when s3  =>
     p := p – 1;
     y <= p ;
     i := i – 1;
     State_Var := s2 ;
 end case;
end loop;
```

Figure 5.1. Sample unpartitioned FSMD code.

two expressions from the set *EXP*.

EXP = {*f*(*x*, *y*, *z*, …) : *x*, *y*, *z*, … ∈ *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* ⇐ *e* : *x* ∈ *VAR*, *e* ∈ *EXP*} is a set of storage assignments.

δ is a state transition function that maps a cross product of *S* and *I* ∪ *STAT* into *S*.

λ is the output function that maps a cross product of *S* and *I* ∪ *STAT* into *O* ∪ *A* for

Mealy models or *S* into *O* ∪ *A* for Moore models.

5.2.   FSMD Functional Partitioning Technique

The algorithm for our FSMD functional partitioning technique is listed in Figure 5.2. The input to the algorithm is a behavioral description of an FSMD such as the one shown in Figure 5.1. The output is multiple FSMDs with interconnections between them. In the first step of the algorithm the internal energy for each state is calculated using the internal energy power estimation technique described in Section 6.1. A dataflow analysis is then performed on the FSMD to determine the data transfer and communication bus width between them. This information is used in the evaluation of the communication energies

```
FSMD_Functional_Partitioning(FSMD){
 Calculate_State_Energy;
 Dataflow_Analysis;
 FSMD_Partitioning;
 Synthesis;
 FSMD_Refinement;
 }
```

Figure 5.2. FSMD functional partitioning algorithm.

between states. Having the internal state energy and the state communication energy, the next step is to perform the actual partitioning of the FSMD. This is described in Chapter 7. After partitioning, each of these smaller FSMDs is then synthesized to its own custom processor, having its own controller and datapath. Finally, in the refinement step, a communication bus is added to connect the processors together so that they are functionally equivalent to the original unpartitioned FSMD.

The architectural model after partitioning and refinement is shown in Figure 5.3.

5.3.   Dataflow Analysis

In contrast to procedural functional partitioning [1] as described in the previous chapter, 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. After partitioning, this information will be used to determine the data that need to be passed between the parts. This in turn will determine the maximum bus width required to connect the parts together.
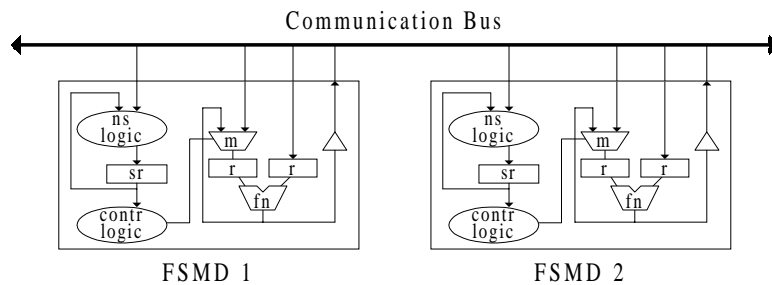
Figure 5.3. Architectural model.

5.3.1. *Basic 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. The edges in the graph correspond to the transitions between the states. For example, given the unpartitioned FSMD code of Figure 5.1, we obtain the initial control flow graph of Figure 5.5(a).

A dataflow analysis, similar to that use for compiler optimization [3], is then performed on the control flow graph to obtain the variables that need to be passed from one state to another. The algorithm for the basic dataflow analysis is shown in Figure 5.4. For each node *n*, four sets of variables are used: *n.def*, *n.use*, *n.in*, and *n.out*. The set *n.def* contains all variables defined in node *n*. A variable is defined when it is written to, for example, when it occurs on the left side of an assignment statement. The set *n.use* contains all variables first used in node *n*. A variable is used when it is read from, for example, when it occurs on the right side of an assignment statement or in an IF statement. A variable is first used in a block if it is used before it is defined, if any, in that block. For example, given the following two statements in a block:

$$j := i * 2;$$
$$k := j + 1;$$

the set *n.def* will contain the two variables *j* and *k* because both of them are defined. Both variables *i* and *j* are used but the set *n.use* will contain only *i* because even though *j* is used in statement two, it is not first used; *j* is defined in statement one before it is used in statement two. However, if the two statements are switched around:

$$k := j + 1;$$
$$j := i * 2;$$

then both *i* and *j* will be in *n.use*.

From the sets *n.def* and *n.use*, the remaining two sets *n.in* and *n.out* are evaluated. *n.in* is the set of variables needed to be passed from the preceding node, and *n.out* is the set of variables needed to pass to the succeeding node.

The algorithm starts by initializing the sets *n.in*, *n.def*, and *n.use* for every node *n*. For each iteration of the WHILE loop, the two sets *n.in* and *n.out* are evaluated for every node.

```
for each node n do
 begin
 // initialization
 n.in = ∅;
 n.def = set of all variables defined in n;
 n.use = set of all variables first used in n; (a variable is first used in a block if it is used before it is
                                 defined, if any, in that block.)
 end
while changes to any of the in's occur do
 begin
 for each node n from last to first do
    begin
    n.out = ⋃ m.in where m is a successor node of n.
    n.in = n.use ⋃ (n.out − n.def )
    end
 end
```

Figure 5.4. Basic dataflow analysis algorithm.

This looping continues as long as there are changes to any of the *in* sets. At the termination of the algorithm, each node *n* will have the two sets *n.in* and *n.out* defined. Figure 5.5(b) shows the variables in the four sets after applying the algorithm to the sample code in Figure 5.1.

5.3.2. *Partition Dataflow Analysis*

When we perform the FSMD partitioning, we are only interested in the amount of data that cross between the 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 5.6. Note that before applying this algorithm, we must already know how many parts we will have and the set of nodes in each part. This is discussed further in Chapter 6 and Chapter 7. The four sets: *def*, *use*, *in*, and *out* are defined similarly as in the basic dataflow analysis but for the whole part. In addition, we define for each part a *callee* and *caller* set. *P.callee* is the set of states in *P* that transition *from* states that are not in *P*. *P.caller* is the set of states in *P* that transition *to* states that are not in *P*.

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 5.6, we obtain the results shown in Figure 5.5(c). Notice that the variable *x* is used in state *s*2 and was defined (primary input) in state *s*0, thus, it must be passed from *s*0 via *s*1 to *s*2. However, *x* is not in

s0

s1

s2

s3

s1   FSMD state

s0   reset state

(a)

0.def={p,i,t1}
0.use={x}
0.in={x}
0.out={p,i,x}

s0

s1

1.def={p,y,i}
1.use={p,i}
1.in={p,i,x}
1.out={p,i,x}

s2

2.def={t2}
2.use={i,x}
2.in={p,i,x}
2.out={p,i,x}

s3

3.def={p,y,i}
3.use={p,i}
3.in={p,i,x}
3.out={p,i,x}

(b)

P0   P0.in={i,p}
P0.out={i,p}

s0

P1

s1

s2

s3

P1.in={i,p}
P1.out={i,p}

(c)

P0   P0.in={i,p}
P0.out={i,p}

s0

start s1

s idle0

P1

s idle1

s2

start s1

s3

start s2
s1

P1.in={i,p}
P1.out={i,p}

(d)
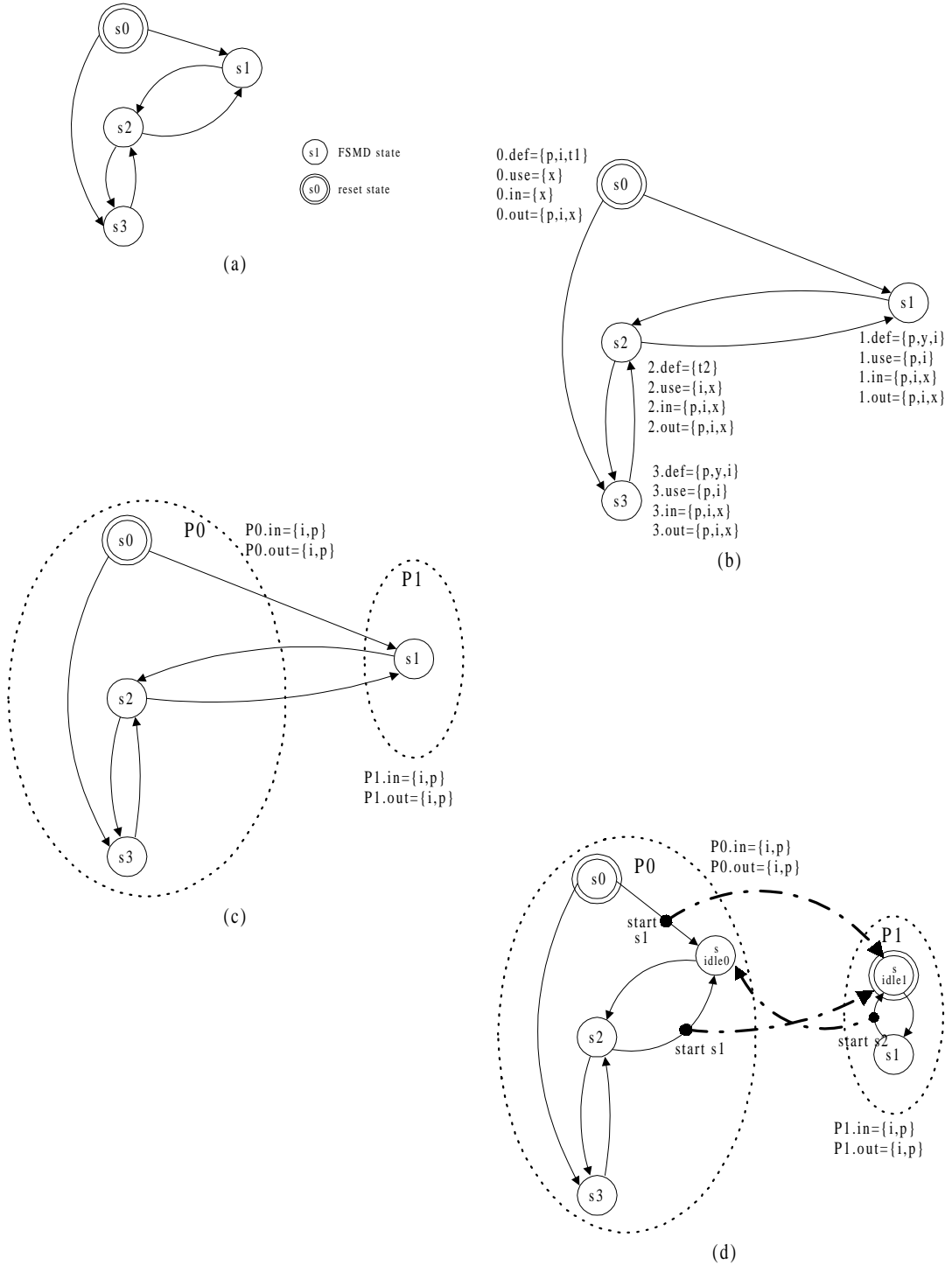
Figure 5.5. (a) Control flow graph for Figure 5.1, (b) after basic dataflow analysis, (c) after partition dataflow analysis, and (c) result after refinement.

55

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 bus width and therefore power consumed by the communication.

5.4. FSMD Refinement

The technique for actually partitioning the FSMD states is described in Chapter 7. After partitioning the FSMD states into mutually exclusive parts, each part is then individually synthesized down to the gate level. The next step in the process is to generate new communicating FSMDs such that they are functionally equivalent to the original unpartitioned FSMD. The resulting communicating FSMDs from Figure 5.5(a) are shown in Figure 5.5(d).

For example, to transition from state $s0$ to $s1$ in the unpartitioned FSMD shown in Figure 5.5(a), the equivalent transition in the partitioned FSMDs is shown in Figure 5.5(d). Initially, P0 is in $s0$ and P1 is in its idle state s_{idle1}. To transition to $s1$, P0 exits $s0$,

```
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 = $\bigcup n.use$    $\forall n \in P$  // set of variables used in P.
    P.def = $\bigcup n.def$    $\forall n \in P$  // 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 5.6. Partition dataflow analysis algorithm.

asserts $start_{s1}$, and enters its idle state s_{idle0}. Seeing that $start_{s1}$ is asserted, P1 exits s_{idle1} and enters $s1$.

The FSMD partitioning can be formally described as follows. Let

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

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:

$$P_i = <S_i, s_{0,i}, s_{idle,i}, I_i \cup STAT_i \cup IP_i, O_i \cup A_i \cup OP_i, \delta_i, \lambda_i > \qquad (5.11)$$

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. $IP_i = <ip_1, \ldots ip_a>$ where $a = $

number of input parameters for P_i, and $OP_i = <op_1, \ldots op_b>$ 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.

5.5.   Preserving the Cycle-By-Cycle Behavior

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 respective idle states for the two parts happen simultaneously, thus, no extra clock cycle is needed. A graphical representation of the execution of the partitioned FSMD is shown in Figure 5.7(a). The edges are annotated with the signals and data that are being sent across the communication bus. The corresponding transition timing diagram is shown in Figure 5.7(b).

5.6.   Critical Path Analysis

Communication involves driving buffers. Even though the delay to drive buffers is much shorter than the delay for all other operations, it is still possible that this added delay will lengthen the critical path. There are three cases where this may happen as shown in Figure 5.8. In the first case, Figure 5.8(a), the communication operation extends the critical path. If a state already contains the critical path and we need to add
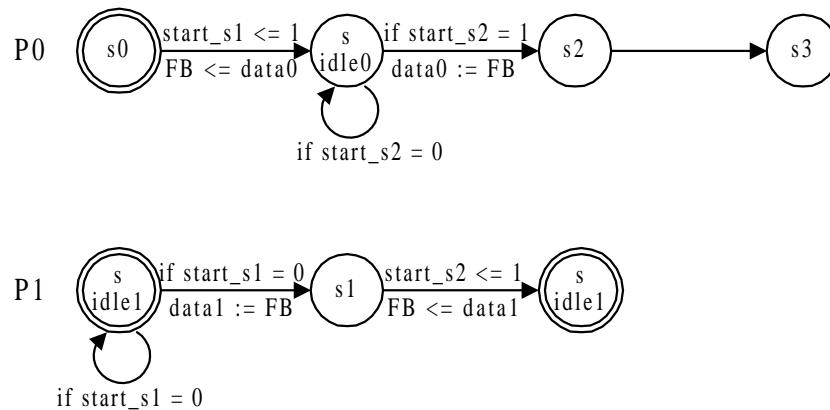
(a)
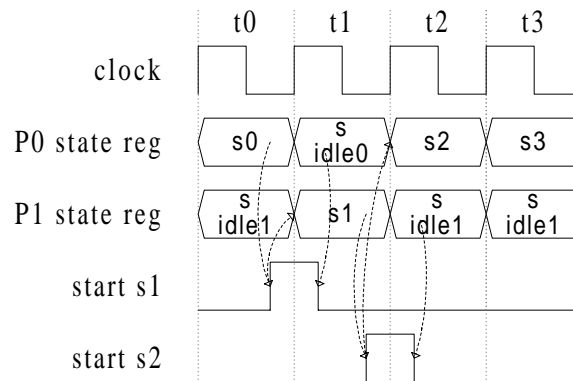
(b)

Figure 5.7. Cycle-by-cycle behavior preservation: (a) execution, and (b) transition timing diagram.

communication to this state, then of course the critical path will be lengthened. In the second case, Figure 5.8(b), the communication is added to a non-critical path. However, with the added communication delay, the non-critical path is now longer than the original critical path. Again, we have extended the critical path. In the third case, Figure 5.8(c), the communication is added to a non-critical path, but the total delay is still less than the original critical path. In this case, the critical path is not changed.

Even though the communication can extend the critical path in the partitioned system, it is still possible that this extended critical path is actually shorter than the critical path in the unpartitioned system. The reason is that because of the smaller circuitry in the smaller parts of the partitioned system, the critical path in the partitioned system is often actually shorter than the critical path in the unpartitioned system.

(a)

(b)

(c)

■ original critical path   ■ non-critical path   □ communication
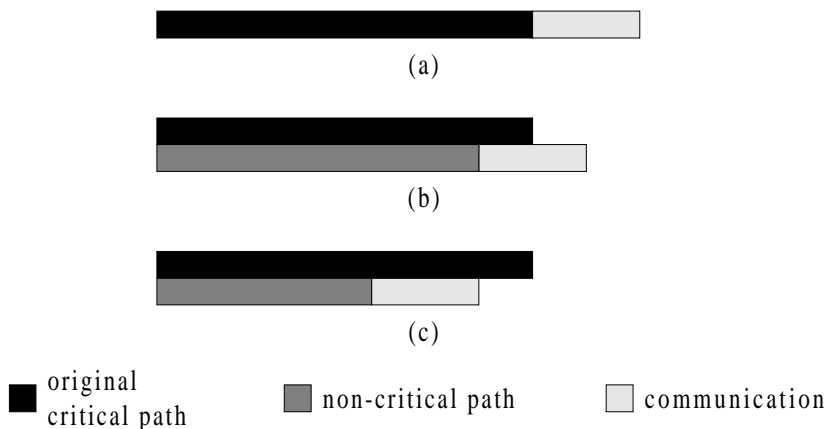
Figure 5.8. Three different situations for adding the communication operations: (a) extending the critical path from original critical path, (b) extending the critical path from non-critical path, and (c) not extending the critical path.

Thus the critical path can be either lengthened (because of the added communication circuitry) or shortened (because of the smaller circuitry in the smaller part). Hence, the overall execution time can be longer or shorter than the unpartitioned system.

5.7.   Preserving the Critical Path

An alternative to extending the critical path is to add the communication operation in a separate state. The original state will go to this new state to perform the communication

(a)

(b)
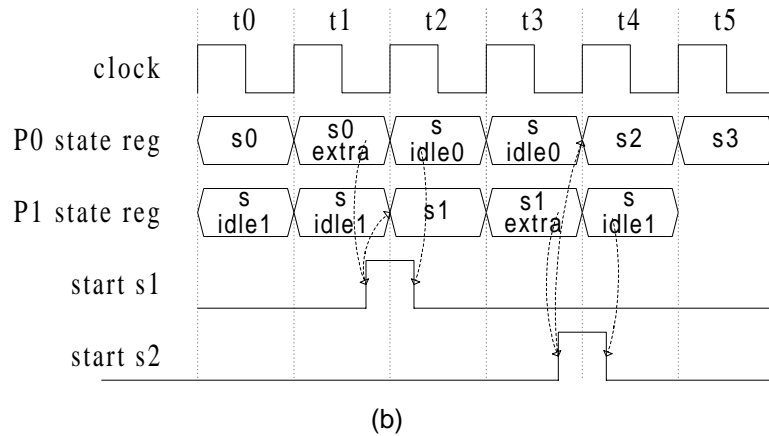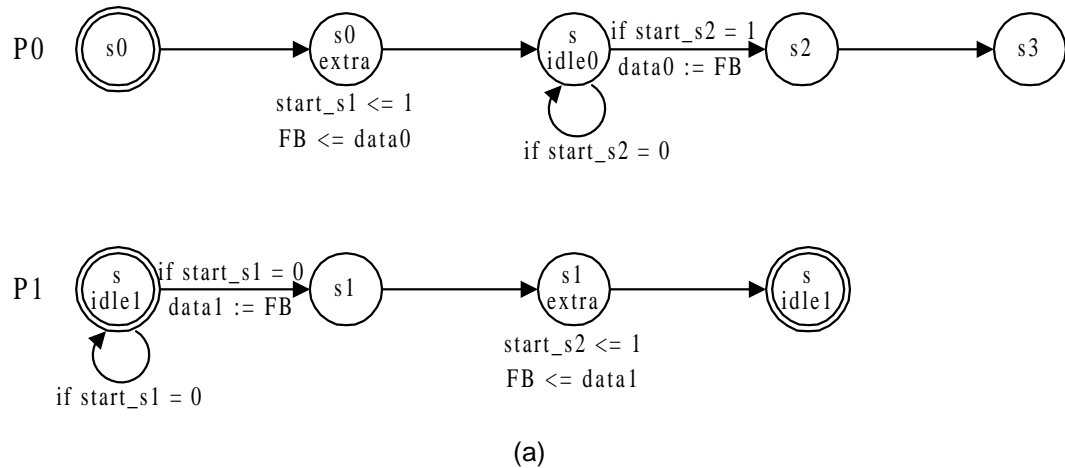
Figure 5.9. Critical path preservation: (a) execution, and (b) transition timing diagram.

before entering the idle state. The tradeoff here is that an extra state is needed and so the execution time will be lengthen by one clock cycle for each transition between the parts. This is shown in Figure 5.9(a) with the extra states *s0_extra* and *s1_extra* added in parts *P*0 and *P*1 respectively for sending the data over the communication bus. The corresponding transition timing diagram is shown in Figure 5.9(b).

This method is usually not used at the FSMD functional partitioning level because at this level, preserving the cycle-by-cycle behavior of the system is more important. However, this method can be used if such preservation is not necessary.

5.8.   Summary

In this chapter, we described a FSMD functional partitioning technique for power reduction. A processor is partitioned into two or more mutually exclusive parts before synthesis. After synthesis of the individual parts, the parts are reconnected via a communication bus so that they are functionally equivalent to the original unpartitioned FSMD. A dataflow analysis algorithm was used to find the data transfers between the parts and the communication bus width. This FSMD functional partitioning technique can be used to preserve either the cycle-by-cycle behavior of the circuit or to preserve the critical path.

References

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

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

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

Chapter 6.   Power Estimation Model

The power cost of a particular partitioning can be obtained either using a simulated approach or an estimated approach. While the simulated approach is much more accurate, it is also very time consuming because the switching activities of each node in the circuit is collected by simulating the entire design. Working at the FSMD level, the solution space is very large and so the simulated approach will drastically limit our exploration of the solution space. Much work has been done on performing very accurate but time consuming power estimation [1]. However, for our purposes, using these elaborate power estimation techniques is still impractical because it has to be evaluated many times during the partitioning optimization process. What we require is a very fast estimation technique that will give us a consistent relative evaluation of each partitioning. We therefore describe an efficient power estimation model and define theoretical energy bounds, which are used by the partitioning algorithm and heuristic.

Recall from Chapter 3 that the dynamic power consumption of a general design is given by

$$P = \frac{1}{2} C V^2 f N$$

where C is the average capacitance switched per access, V is the supply voltage, f is the clock frequency, and N is the switching frequency of the unit (or the activity factor). From this equation, we see that power estimation depends on several factors that are known only after hardware assignment, scheduling and/or placement. Furthermore, the activity factor is known only after executing the design. At the FSMD level, much of the information is not known. For example, in order to calculate the power consumption of a bus, the bus capacitance must be known. However, the bus capacitance is dependent on the length of the wire and proximity to other wires, and this information is not known until after placement and routing. Fortunately, for high-level optimization, relative evaluation of different designs is more important than absolute evaluation, and consistency is more important than accuracy.

Our power estimation model is divided into two parts: the internal and external energy models. The internal energy is the energy consumed by a single processor while the external energy is the energy consumed by the communication between the processors. The total energy consumed by a partitioned FSMD system is, therefore, the sum of the internal and the external energy. For the remaining discussion, we will restrict to partitioning the FSMD into two parts. The idea can be easily generalized to more than two parts.

6.1.   Internal Energy

We define the internal energy as the total amount of energy consumed by all the states in a part. This excludes the communication energy between states. The energy for

a state is the amount of power consumed by the state multiplied by the amount of time the state spends executing.

Let $U = \{s_1, s_2, \ldots, s_u\}$ be the set of u states in the unpartitioned FSMD. Let A and B be two partitions of the FSMD such that $A \cap B = \emptyset$, $A \cup B = U$, and a and b be the number of states in A and B respectively so that $a+b=u$. Let Es_i be the energy consumed by state s_i. From [2] we see that the power for a state can be approximated by the number of functional units and registers, and the amount of time spent executing in a state can be found by profiling. Furthermore, let

$$E_U = \sum_{\forall s_i \in U} Es_i \qquad (6.1)$$

be the sum of the energy of the states in the unpartitioned FSMD,

$$E_A = \sum_{\forall s_i \in A} Es_i \qquad (6.2)$$

be the sum of the energy of the states in the partitioned FSMD A, and

$$E_B = \sum_{\forall s_i \in B} Es_i \qquad (6.3)$$

be the sum of the energy of the states in the partitioned FSMD B. We claim that the total energy for an n-state FSMD is equal to

$$\alpha_n \sum_{\forall s_i \in FSMD} Es_i \qquad (6.4)$$

where α_n is determined by the complexity of the n-state FSMD. More detail on the complexity is given in Section 6.2. Therefore, the total energy usage for the unpartitioned

66

FSMD $E_{unpartitioned}$ is

$$E_{unpartitioned} = \alpha_u E_U \,, \tag{6.5}$$

and the total internal energy for the partitioned FSMD $E_{internal}$ is

$$E_{inernal} = \alpha_a E_A + \alpha_b E_B \,. \tag{6.6}$$

Figure 6.1 shows the results of an experiment where the energy usage for a FSMD with different numbers of states with identical actions is evaluated. The plot shows that adding the energy for an n-state FSMD with an m-state FSMD is less than the energy for an $n+m$ state FSMD. For example, using the 4 FU line, the energy for the 20-state and 28-state FSMDs (152+241=393) is less than the energy for the 48-state FSMD (467). In other words, when the complexities of two individual states are summed, the result will be less than the complexity of the two states combined. Thus, we have the inequality
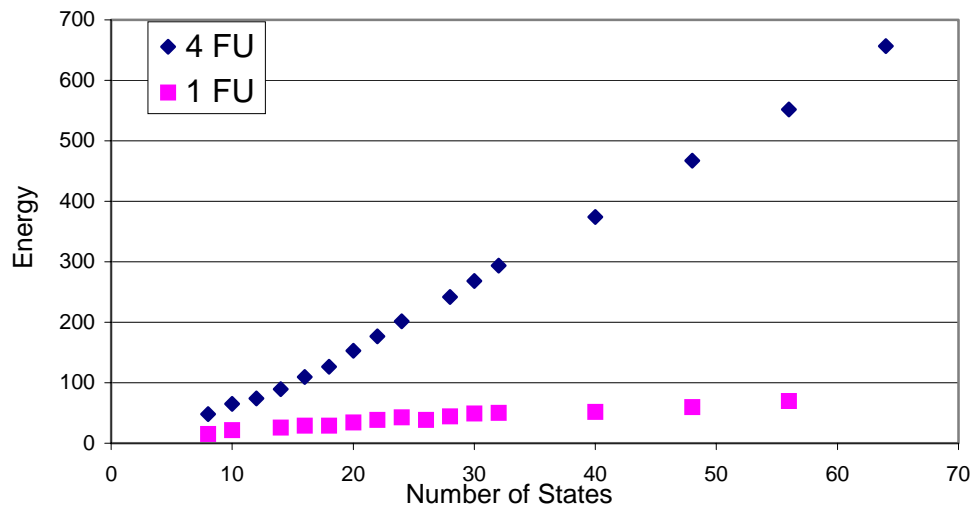
Figure 6.1. Energy versus the number of identical states for a FSMD with one and four functional units.

$$\alpha_a E_A + \alpha_b E_B < \alpha_u E_U . \tag{6.7}$$

6.2. FSMD Complexity

The FSMD complexity α addresses the issues of the internal interconnect and the size of the FSMD in terms of energy usage. The internal interconnect deals with the complexity of the datapath, whereas, the number of states deals with the complexity of the control unit. It was observed in [33] that smaller capacitance is achieved in smaller designs because there are fewer and/or shorter interconnects, and fewer functional units and registers, which are obstacles during floorplanning and routing, which indirectly influence interconnect capacitance, and therefore, power usage. Thus, the internal interconnect capacitance is dependent on, among other factors, the internal bus length which in turn is dependent on the number of functional units, multiplexers, registers, etc. that need to be connected together, and the final layout area.

Working at the FSMD level, this interconnect can be approximated by the number of states and functional units required in a state. Figure 6.1 shows that the energy usage of the FSMD is also related to its size and is approximated by the number of states in the FSMD. A similar relationship was also found in [2]. Figure 6.1 also shows how the number of states relates to the energy usage for different number of functional units. Thus, an approximation of the complexity, α_n, for an n-state FSMD is

$$\alpha_n = n \times (FU + mux + reg) \tag{6.8}$$

where *n* is the number of states in the FSMD, and *FU*, *mux* and *reg* are the average number of functional units, multiplexers, and registers respectively per state. α_1 is the complexity for one state. If the α_1s for two different states are not equal, then we use the smaller number.

6.3.   External Energy

The total energy of the partitioned system is not just $E_{internal}$. When there are two or more parts, communication must be added between the parts. We define the external energy as the energy consumed by the communication between parts. Thus, the total energy for the partitioned system is

$$E_{partitioned} = \alpha_A E_A + \alpha_B E_B + E_{comm}. \tag{6.9}$$

The communication energy E_{comm}, is simply the sum of the energy (weights) of all the edges crossing between the parts, E_{xedge}, multiply by their activity factor, β:

$$E_{comm} = \sum_{\forall cross-edge\, i} (\beta_i \times Exedge_i) \tag{6.10}$$

In our architectural model shown in Figure 5.3, only one external common bus is used to connect the two parts. All communication between parts occurs over this bus. Thus, the external bus is used every time when there is a transition from state s_i to s_j such that s_i and s_j are in different parts. A major factor that affects the bus energy is its length as reported in [33]. The bus length is approximated by the number of parts being connected together. The bus width is derived from the maximum data size crossing the parts. If the data is multiplexed over the bus, then the smaller data size is multiplied by the number of

69

times required to transmit all the data. The data size is obtained from the dataflow analysis.

Although in equation (6.7) we claim that

$$\alpha_a E_A + \alpha_b E_B < \alpha_u E_U.$$

However, with the added communication, the claim is not always true. In other words, it is possible that

$$\alpha_a E_A + \alpha_b E_B + E_{comm} > \alpha_u E_U.$$

Fortunately as we have found in most situations, there will be a partitioning such that

$$\alpha_a E_A + \alpha_b E_B + E_{comm} < \alpha_u E_U. \tag{6.11}$$

6.4. Finding the Energy Bounds

We will now evaluate lower (best) and upper (worst) bounds for the internal energy $E_{internal}$, the external communication energy E_{comm}, and finally the partitioned energy $E_{partitioned}$. These bounds will be used in our optimal partitioning algorithm.

6.4.1. Internal energy bounds

The following internal energy bounds progressively get tighter. We start with $[0, \alpha_u E_u]$ as the first bound. The lower bound is obvious. The upper bound is for an unpartitioned system. The second, tighter bound is $[\alpha_l E_U, (\alpha_u - \alpha_1)E_U]$. The reason for this second lower bound is that the minimum energy for a partition is when there is no added complexity when all the states are added into the part. Thus, $E_{internal} = \alpha_1 E_A + \alpha_1 E_B$

$= \alpha_1 E_U$. The upper bound comes from the fact that we need at least one state in one part in order to have a 2-way partition. Thus, we subtract the least energy for one state from the unpartitioned energy.

If some states are already assigned to either of the parts, we can get an even tighter third bound. Given the fact that some states are already assigned, we can calculate the internal energy for the current partitioning (i.e. currently known assigned states) using equation (6.6). From equation (6.7), we see that the worst that can happen is to put all the states in the same part. Thus, to get the upper bound, we put all the remaining unassigned states together in the same part. The resulting energy will be either $(\alpha_a + \alpha_{rs})(E_A + E_{rs})$, or $(\alpha_b + \alpha_{rs})(E_B + E_{rs})$, where α_{rs} is the complexity of the combined remaining states and E_{rs} is the total internal energy of the remaining states. Since we know that some states are already assigned to another part, therefore, $E_{internal}$ can be either $[(\alpha_a + \alpha_{rs})(E_A + E_{rs}) + \alpha_b E_B]$ or $[(\alpha_b + \alpha_{rs})(E_B + E_{rs}) + \alpha_a E_A]$. We select the one that is the largest. Thus, the upper bound, $ub_{Einternal}$, is

$$ub_{E\,internal} = max \begin{cases} (\alpha_a + \alpha_{rs})(E_A + E_{rs}) + (\alpha_b E_B), \\ (\alpha_b + \alpha_{rs})(E_B + E_{rs}) + (\alpha_a E_A) \end{cases} \qquad (6.12)$$

In fact, this is the exact maximum for $E_{internal}$ given that some states are already assigned to the parts because $(\alpha_a + \alpha_b + \alpha_{rs})(E_A + E_B + E_{rs}) = \alpha_u E_U$ is the absolute maximum.

71

For the lower bound, we add to the current partitioning energy the total energy for the remaining unassigned states, E_{rs}, using the least complexity (i.e. α_1). Thus, the lower bound, $lb_{Einternal}$, is

$$lb_{E\,internal} = \alpha_a E_A + \alpha_b E_B + \alpha_1 E_{rs}.$$ (6.13)

To get an even tighter lower bound, we note that all the remaining states must be assigned to either of the two parts, thus, the complexity for these remaining states must be at least $\min(\alpha_a, \alpha_b)$. Thus,

$$lb_{E\,internal} = \alpha_a E_A + \alpha_b E_B + (\min[\alpha_a, \alpha_b]) \cdot E_{rs}.$$ (6.14)

Furthermore, since at least one of the remaining states must be added to one part, the complexity of that part must at least be increased by α_1. Thus, an even tighter lower bound is

$$lb_{E\,internal} = \alpha_a E_A + \alpha_b E_B + (\min[\alpha_a, \alpha_b] + \alpha_1) \cdot E_{rs}$$ (6.15)

6.4.2. External energy bounds

We will now provide bounds for the external communication energy, E_{comm}. Recall from equation (6.10) that E_{comm} is the sum of the energy (weights) of all the edges crossing between the parts. Thus, the first lower and upper bounds are when no edges and all edges respectively cross between parts. However, given the fact that some states are already assigned to either of the parts, therefore, some edges are already determined as to whether they cross between parts or not. Thus, knowing the current communication

72

energy, E_{cc}, the upper bound for the communication energy, ub_{Ecomm}, is when all the remaining edges, E_{rc}, will cross between parts:

$$ub_{Ecomm} = E_{cc} + \sum E_{rc} \tag{6.16}$$

The lower bound for the communication energy, lb_{Ecomm}, is the sum of the currently known communication energy, E_{cc}, plus the minimum of all the remaining communication edges, E_{rc},

$$lb_{Ecomm} = E_{cc} + \min(E_{rc}) \tag{6.17}$$

6.4.3. Partitioned energy bounds

The bounds for the partitioned FSMD are simply the sum of the internal and communication energy bounds. Thus, the lower bound for the partitioned energy, $lb_{Epartitioned}$, is

$$
\begin{aligned}
lb_{Epartitioned} &= lb_{Einternal} + lb_{Ecomm} \\
&= \left[\alpha_a E_A + \alpha_b E_B + (\min[\alpha_a, \alpha_b] + \alpha_1) \cdot E_{rs} \right] + \left[E_{cc} + \min(E_{rc}) \right]
\end{aligned} \tag{6.18}
$$

and the upper bound for the partitioned energy, $ub_{Epartitioned}$, is

$$
\begin{aligned}
ub_{Epartitioned} &= ub_{Einternal} + ub_{Ecomm} \\
&= \left[max \begin{cases} (\alpha_a + \alpha_{rs})(E_A + E_{rs}) + (\alpha_b E_B), \\ (\alpha_b + \alpha_{rs})(E_B + E_{rs}) + (\alpha_a E_A) \end{cases} \right] + \left[E_{cc} + \sum E_{rc} \right]
\end{aligned} \tag{6.19}
$$

6.5. Model accuracy

We have compared the accuracy of the results obtained using our power estimation model with that of the simulated approach. For the simulated approach, we used an event-

driven simulator to simulate the execution of the design to collect the switching frequency of each node in the circuit. The loading capacitance for each node is obtained from the synthesized gate level netlist and a low-power technology library. The power equation, $P = \frac{1}{2}CV^2fN$, is then used to calculate the total energy consumed.

Comparing the simulated results shown in Table 8.4 and the estimated results shown in Table 8.7, we see that on average the difference between the estimated data and the simulated data is only 17%.

6.6.   Summary

We have presented an efficient power estimation model and defined theoretical energy bounds for the FSMD model. This power estimation model is used by our branch-and-bound partitioning algorithm and simulated annealing heuristic.

References

[1] F. Najm, "A Survey of Power Estimation Techniques in VLSI Circuits," *IEEE Transactions on VLSI Systems*, 2(4):446-455, December 1994.

[2] R. Mehra & J. Rabaey, "Behavioral Level Power Estimation and Exploration," *First International Workshop on Low Power Design*, pp. 197-202, April 1994.

[3] A. Chandrakasan, M. Potkonjak, J. Rabaey, & R. Brodersen, "Hyper-LP: A System for Power Minimization Using Architectural Transformations," *Proceedings of the International Conference on Computer Aided Design*, pp. 300-303, 1992.

Chapter 7.   Partitioning Algorithm and Heuristic

We will now describe a branch-and-bound optimal algorithm and a near optimal simulated annealing heuristic for functional partitioning the FSMD for low power. Our objective is to find a partitioning among the FSMD states such that the total energy for the partitioned system is minimized.

The cost function used by the algorithm and heuristic is based on the power estimation model described in the previous chapter. Using the power estimation model and the dataflow analysis, we can evaluate the internal energy usage for each state and the communication energy between any two states in the FSMD.

7.1.   Branch-and-Bound

Our optimal algorithm is based on a branch-and-bound technique. In this algorithm, a binary tree structure is used. Nodes that are promising are kept for further processing and those that are guaranteed to be worst are pruned. We start with the root having only one state. At each successive level, we add a new state and assign to the nodes in that level all possible combinations of the states for the two parts. Using equations (6.18) and (6.19)

from the previous chapter, an upper and lower bound is calculated for each node. Depending on the bounds, a node is either kept or pruned. The most promising node for a particular level is the one with the minimum energy for that level. Nodes that satisfy one of the following two conditions are guaranteed to be inferior and are therefore pruned:

Condition 1: if $LBn_i \geq UBmin$ then prune n_i.

Condition 2: if $LBn_i \geq Emin$ then prune n_i.

where LBn_i is the lower bound for the node n_i, $UBmin$ is the current minimum upper bound, and $Emin$ is the current minimum energy for a complete solution seen so far.

For example, assume that we have the state energy and the communication energy between any two states obtained from the basic dataflow analysis for four states as shown in Table 7.1. The resulting branch-and-bound binary tree is shown in Figure 7.1. Each node is annotated with the node number, the FSMD states in each of the two partitions, Ep (the current energy for that partitioning), lb (the lower bound for that partitioning), and ub (the upper bound for that partitioning). For example, for node 7, part A contains state s0 and part B contains states s1 and s2. Using equation (6.9), the current energy for this three state partitioning is:

Table 7.1. Sample energy data.

States	s0	s1	s2	s3
State Energy	110	80	60	80
State Comm Energy				
s0		30	0	30
s1			30	0
s2				30

$$E_{partitioned} = \alpha_A E_A + \alpha_B E_B + E_{comm}$$
$$= 1(110) + 2(80 + 60) + 30$$
$$= 420$$

Note that this is not the energy for the complete solution. $E_{partitioned}$ is the energy for the complete solution only for the nodes at the lowest level. Using equation (6.18), the lower bound is:

$A = s0$ ①

$Ep = 110$
$lb = 360$
$ub = 1440$

$A = \frac{s0}{s1}$ ②

$Ep = 380$
$lb = 550$
$ub = 1440$

$A = s0$ ③ $B = s1$

$Ep = 220$
$lb = 530$
$ub = 980$

$A = \begin{matrix} s0 \\ s1 \\ s2 \end{matrix}$ ④

$A = \frac{s0}{s1}$ ⑤ $B = s2$

$A = \frac{s0}{s2}$ ⑥ $B = s1$

$A = s0$ ⑦ $B = \frac{s1}{s2}$

$Ep = 750$
$lb = 860$
$ub = 1440$

$Ep = 470$
$lb = 660$
$ub = 1020$

$Ep = 480$
$lb = 670$
$ub = 950$

$Ep = 420$
$lb = 610$
$ub = 860$

$\begin{matrix} s0 \\ s1 \\ s2 \\ s3 \end{matrix}$ ⑧

$\begin{matrix} s0 \\ s1 \\ s2 \end{matrix}$ ⑨ $s3$

$\begin{matrix} s0 \\ s1 \\ s3 \end{matrix}$ ⑩ $s2$

$\begin{matrix} s0 \\ s1 \end{matrix}$ ⑪ $\begin{matrix} s2 \\ s3 \end{matrix}$

$\begin{matrix} s0 \\ s2 \\ s3 \end{matrix}$ ⑫ $s1$

$\begin{matrix} s0 \\ s2 \end{matrix}$ ⑬ $\begin{matrix} s1 \\ s3 \end{matrix}$

$\begin{matrix} s0 \\ s3 \end{matrix}$ ⑭ $\begin{matrix} s1 \\ s2 \end{matrix}$

$s0$ ⑮ $\begin{matrix} s1 \\ s2 \\ s3 \end{matrix}$

$Ep = 1320$
$lb = 1320$
$ub = 1320$

$Ep = 890$
$lb = 890$
$ub = 890$

$Ep = 930$
$lb = 930$
$ub = 930$

$Ep = 720$
$lb = 720$
$ub = 720$

$Ep = 890$
$lb = 890$
$ub = 890$

$Ep = 780$
$lb = 780$
$ub = 780$

$Ep = 720$
$lb = 720$
$ub = 720$

$Ep = 830$
$lb = 830$
$ub = 830$

Figure 7.1. Sample branch-and-bound binary tree.

$$lb_{Epartitioned} = lb_{E\,internal} + lb_{Ecomm}$$
$$= [\alpha_a E_A + \alpha_b E_B + (\min[\alpha_a, \alpha_b] + \alpha_1) \cdot E_{rs}] + [E_{cc} + \min(E_{rc})]$$
$$= [1(110) + 2(140) + (\min[1,2] + 1)(80)] + [30 + 30]$$
$$= 610$$

Finally, using equation (6.19), the upper bound for this node is:

$$ub_{Epartitioned} = ub_{E\,internal} + ub_{Ecomm}$$
$$= \left[max \begin{cases} (\alpha_a + \alpha_{rs})(E_A + E_{rs}) + (\alpha_b E_B), \\ (\alpha_b + \alpha_{rs})(E_B + E_{rs}) + (\alpha_a E_A) \end{cases} \right] + \left[E_{cc} + \sum E_{rc} \right]$$
$$= \left[max \begin{cases} (1+1)(110+80) + 2(140), \\ (2+1)(140+80) + 1(110) \end{cases} \right] + [30 + 60]]$$
$$= 860$$

After evaluating nodes 1 to 7, we have *UBmin* = 860. Since the lower bound for node 4, LBn_4 = 860, therefore, condition one is satisfied and we can prune the subtree rooted at node 4.

```
T = initial_temperature;
c_old = InitialCost(initial_partition);
while stopping_criterion is not satisfied do
  while inner_loop_criterion is not satisfied do
    i = RANDOM(1,number_of_states);        // next state to move
    MOVE{Sᵢ};                               // move Sᵢ from current part to other part
    c_new = IncrementalCost(Sᵢ);
    Δc = c_new – c_old;
    x = F(Δc, T);
    r = RANDOM(0,1);
    if r < x then
      c_old = c_new;
    end if
  end while
  T = Update(T);
end while
```

Figure 7.2. Simulated annealing heuristic.

In this example, we have two optimal solutions. The first solution is at node 11 where the partitioning is $A = \{s0, s1\}$ and $B = \{s2, s3\}$, and $E_{partitioned} = 720$. The second solution is at node 14 with the partitioning $A = \{s0, s3\}$ and $B = \{s1, s2\}$.

7.2.   Simulated Annealing

To tradeoff accuracy with speed, we implemented a simulated annealing heuristic. Near optimal solution is possible with this simple and fairly fast heuristic as first introduced in [1] and further discussed in [2].

The simulated annealing heuristic, as shown in Figure 7.2, starts with a random initial partition. For each move, the heuristic randomly selects a state to be moved from one partition to another. The acceptance of the new partition depends on the function F and a random number. The function F is defined as

$$F(\Delta c, T) = min(1, e^{-(\frac{\Delta c}{T})}).$$

where Δc is the change in cost from the old to the new partition, and T is the annealing temperature. If the cost for the new partitioning is better than the old (i.e., Δc is negative) then F returns a 1 and so the new partitioning is definitely accepted; otherwise, it is accepted with a probability determined by the annealing temperature and a random number. The annealing temperature, T, is high at the beginning and is decreased during each iteration by the function *Update* which is defined as

$$Update(T) = \alpha T$$

80

where $0 < \alpha < 1$. The *stopping_criterion* is satisfied when T is approximately zero. The *inner_loop_criterion* is satisfied when the solution does not improve for a certain number of iterations. The accuracy of the heuristic is determined by the annealing temperature T, *stopping_criterion*, *inner_loop_criterion*, and α for updating T.

The heuristic starts by calling the *InitialCost* function shown in Figure 7.3 for calculating the initial energy cost of the starting random partitioning. After each move in the heuristic, the total energy is re-calculated using the *IncrementalCost* function shown in Figure 7.4. The global variables (*EA*, *EB*, *sizeA*, *sizeB*, and *Ecomm*) used in the *IncrementalCost* function are also initialized in the *InitialCost* function. Both of these cost functions are based on the energy estimation model described in the last chapter. The

```
InitialCost(partitioning){
 EA = EB = 0;
 sizeA = sizeB = 0;
 Ecomm = 0;
 for all states Sᵢ do {              // calculate EA,
   if Sᵢ.part = A then {             //    EB,
     EA = EA + ESᵢ                    //      sizeA,
     sizeA = sizeA + 1;              //    and sizeB
   } else {
     EB = EB + ESᵢ
     sizeB = sizeB + 1;
   }

 for all edges edgeᵢ,ⱼ do {          // calculate Ecomm
   if Sᵢ.part ≠ Sⱼ.part then
     Ecomm = Ecomm + edgeᵢ,ⱼ.weight
   }

 Epartitioned = sizeA*EA + sizeB*EB + Ecomm;
 return Epartitioned;
 }
```

Figure 7.3. Initial cost function.

complexity for the *IncrementalCost* function is $O(k)$ where k is the maximum number of adjacent edges for a node. The worst is when k equals the number of states.

The annealing parameters used in the experiments are as follows: annealing temperature = 30; random seed = fixed for all examples; outer loop stopping criterion = 1×10^{-6}; inner loop stopping criterion = 10000; alpha for updating the annealing temperature = 1×10^{-6}.

7.3. Summary

In this chapter, we have presented an optimal branch-and-bound algorithm and a near optimal simulated annealing heuristic for partitioning the FSMD states such that the total energy for the partitioned system is minimized.

```
IncrementalCost(S_move){
  if S_move.part = A then {              // moved to part A
    sizeA++;
    sizeB--;
    EA = EA + ES_move;
    EB = EB – ES_move;
    }
  else {        // moved to part B
    sizeA--;
    sizeB++;
    EA = EA - ES_move;
    EB = EB + ES_move;
    }

  for all states S_i adjacent to S_move { // update Ecomm
    if S_i.part ≠ S_move.part then {
      Ecomm = Ecomm + edge_{i,move}.weight;
      }
    }

  return sizeA*EA + sizeB*EB + Ecomm;
  }
```

Figure 7.4. Incremental cost function.

References

[1]   S. Kirkpatrick, C.D. Gelatt and M.P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671-680, 1983.

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

Chapter 8.   Experiments and Results

We implemented the techniques discussed in the preceding chapters. We will now present the results from the experiments that we have performed.

8.1.   Power Reduction for Procedural Functional Partitioning

For our procedural functional partitioning experiments, we described five examples at the behavioral level using VHDL. We applied the procedural functional partitioning technique described in Chapter 4. After partitioning, the system is synthesized and simulated to obtain the switching activity data. NSYN [1], a behavioral synthesizer, was used to synthesize the partitioned and unpartitioned systems from the behavioral level to the gate level. Purespeed [2], an event-driven simulator, was used to collect the switching frequency data for the power calculation. Loading capacitance was obtained from the synthesized gate level netlist and a low-power technology library. Power results are calculated using the switching activity and the loading capacitance for each node. The unpartitioned and partitioned systems are compared in terms of their average and total power usage, area, and execution time.

Table 8.1 shows the statistics for these examples. *Fac* is a factorization program. *Chinese* is to evaluate 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 while loops. For the *Fac* example, two different ways of partitioning the system was done. The first way is shown in Figure 4.5. For the *Chinese* example, three different ways of partitioning the system was done. The *gate count* column shows the gate count for the unpartitioned and partitioned system. In the *partitioned* column, the gate count for the individual modules are further broken down. The *function calls* column shows the number of times each part is called via the communication bus. For example, for the *Fac1* example, part one is called one time and parts two and three are called n times to denote that it is dependent on the input value. The *loops* column shows whether there is a loop in that part and if so how many times it loops around. Again, n denotes that it is dependent on the input value.

The results are summarized in Table 8.2. The table shows the gate count, the execution time, the average and total power used as a ratio of the partitioned to

Table 8.1. Procedural functional partitioning example statistics.

Examples	Gate Count		Function Calls	Loops
	Unpartitioned	Partitioned= P1+P2+P3+...		
Fac 1	15251	17172=8918+3051+5203	1/n/n	1/2/2
Fac 2	15251	15802=9260+1349+1695+3498	1/n/n/n	1/1/1/1
Chinese 1	19766	32460=14965+3679+13816	1/1/1	4/3/n
Chinese 2	19766	34111=11116+5499+3679+13817	1/1/1/1	1/3/3/n
Chinese 3	19766	29551=11694+2345+1695+13817	1/3/3/1	1/1/1/n
Diffeq	11487	12245=1340+10905	1/12	1/n
Volsyn	11193	13163=10798+2365	1/n	n/n
NLoops	2622	3307=1811+1496	1/n	n/n

unpartitioned examples. Columns six and seven are the absolute power for the partitioned

system. Since the switching frequency is dependent on the inputs, the results shown are

averages from several runs. The average power column shows the average power used in

one clock period and the total power is the total power used for the entire execution.

In all cases, both the average and total power is reduced. The reduction in average

power ranges from 27% to as much as 78% as in the case for *Fac2*. The reduction in total

power ranges from 12% to 66%. Even with this drastic power reduction, the tradeoff is

not too terrible. The gate count is increased by 32% on average. If we consider only the

best partitioning for the *Fac* and *Ch* examples, then the gate count is only increased by

21% on average and 49% in the worse case. Execution time is increased by 22% on

average and 53% in the worse case. The reason for the size and execution time increase is

because of the extra communication overhead. The execution time overhead should be

Table 8.2. Procedural functional partitioning power reduction results.

Examples	% Overhead		Absolute Partitioned		% Power Savings	
	Area %	Time %	Average Power (μW)	Total Power (μJ)	Average Power %	Total Power %
Fac 1	13%	23%	22.65	2242.95	64%	55%
Fac 2	4%	53%	13.84	1694.67	78%	66%
Chinese 1	64%	2%	16.03	3604.99	45%	43%
Chinese 2	73%	5%	15.16	3352.01	48%	47%
Chinese 3	49%	16%	13.12	3035.78	55%	52%
Diffeq	7%	30%	40.76	7771.52	27%	15%
Volsyn	18%	24%	10.38	2587.22	29%	12%
Nloops	26%	20%	3.70	2157.51	59%	50%
Average	32%	22%	16.96	3305.83	**51%**	**42%**

less than reported because we used a fixed clock in the calculation, but the critical path for the parts are actually less. See section 8.4 for more detail on this.

It is interesting to note that in the *Fac2* example, the gate count for the partitioned system is increased by only 4%. This increase is very insignificant. In fact, a decreased was observed in [3]. A possible reason is that NSYN can optimize a small design much better than a large design.

From *Chinese1* to *Chinese2*, the total power is reduced by 4% and the gate count is increased by 9%. The difference between *Chinese1* and *Chinese2* is that part one in *Chinese1* is divided into two parts in *Chinese2*. Parts two and three in *Chinese1* are the same as parts three and four in *Chinese2* respectively. By adding an extra part, we have increased the total size because of the communication overhead. However, the individual size of each part is smaller. This causes the switching activity to be even more localized and confined within fewer gates. Thus, a reduction in the power is seen.

The total power is further reduced by another 5% from *Chinese2* to *Chinese3*. The main difference between *Chinese2* and *Chinese3* is that parts two and three are further reduced in size. The tradeoff is that more communication is added as can be seen from the size increase in part one and the longer execution time. However, the total size did not increase, but rather decreased by 13%.

From this result, we can see that it is better to have more smaller parts rather than few bigger parts, as long as the size and performance overhead is kept within the limit. The

rationale is that smaller parts will have less switching activities when the part is active. The rest of the dormant parts do not contribute any dynamic power due to capacitive charging and discharging because there are no switching activities. Of course, more parts mean more communication on the communication bus and longer execution time.

8.2. Power Reduction for FSMD Functional Partitioning

We implemented the FSMD functional partitioning technique described in Chapter 5 and applied it to seven examples. We start by describing the system using the FSMD model with VHDL. After applying our FSMD partitioning technique, the system is synthesized and simulated to obtain the switching activity data. NSYN [1], a behavioral synthesizer, was used to synthesize the partitioned and unpartitioned systems from the behavioral level to the gate level. Purespeed [2], an event-driven simulator, was used to collect the switching frequency data for the power calculation. Loading capacitance was obtained from the synthesized gate level netlist and a low-power technology library. Power results are calculated using the switching activity and the loading capacitance for

Table 8.3. FSMD functional partitioning example statistics.

Examples	Unpartitioned	Partitioned		
	Size	Size	States	Bit Width
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	5307=3890+1417	101	98
DSP	278	386=131+255	13	12

each node. The unpartitioned and partitioned systems are compared in terms of their average and total power usage, area, and execution time.

Table 8.3 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 8.4. Columns 2 and 3 show the percent increase in area and execution time respectively. The absolute average and total power for the

Table 8.4. FSMD functional partitioning power reduction results.

	% Overhead		Absolute Partitioned		% Power Savings	
Examples	Area %	Time %	Average Power (μW)	Total Power (μJ)	Average Power %	Total Power %
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%**

partitioned examples are shown in columns 4 and 5. The percent average and total power savings are shown in the last two columns. 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 ranges 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 because of frequent communication. 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, thus compensating for the critical path lengthening from communication. The 24% increase in gates is not as significant because chip capacities continue 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 calculation, we have used a bus capacitance that is four times the internal capacitance.

Table 8.5. Breakdown of power consumption by components.

Processor Components	Unpartitioned		Partitioned		
	Power (uW)	%	Power (uW)	%	% Savings
FUs + muxes	28.0	87.0%	11.8	71.7%	57.8%
Registers	1.7	5.3%	2.1	13.0%	-23.5%
Controller	2.5	7.7%	1.4	8.7%	44.0%
Communication	0.0	0.0%	1.1	6.6%	-
Total	32.2	100.0%	16.4	100.0%	**49.1%**

8.3. Power Usage Breakdown by Processor Components

We evaluated the power usage of major components in a processor. The breakdown of components includes the functional units and multiplexers, registers, controller, and communication. Table 8.5 shows the breakdown results. The unpartitioned and partitioned power columns are averages of power usage from the examples from section 8.2 except that the *Diffeq* example was not included in these averages. The two percentage columns reflect the percentages of power usage by the different components. The percent savings column shows the percentage of power saved as a result of partitioning for the different components. Figure 8.1 shows the power usage breakdown graphically.

Clearly, functional units and multiplexers consume the most power in both the unpartitioned and partitioned system. However, in the partitioned system it is decreased by more than 57%. Power consumed by the registers is greater in the partitioned system
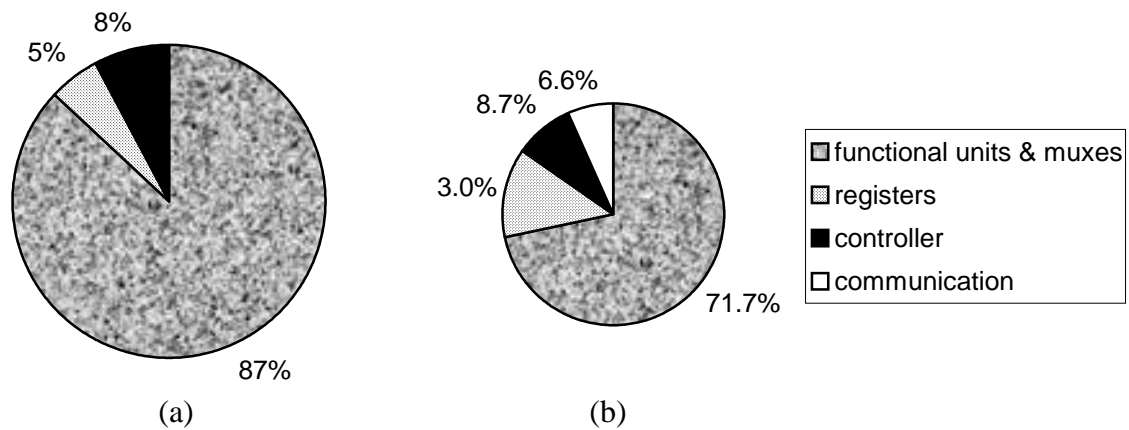

Figure 8.1. Breakdown of power consumption by parts for: (a) unpartitioned, and (b) partitioned system.

91

than the unpartitioned system because in the partitioned system, registers have to be duplicated across the parts. Therefore, there are more registers in the partitioned system. Power consumed by the controller is 44% less in the partitioned system than in the unpartitioned system because the controllers in each part of the partitioned system are smaller and only one has to be active at a time. Overall, the partitioned system has a power savings of 49.1%.

8.4.   Critical Path

Table 8.6 compares the critical path and execution time between the unpartitioned system with the partitioned system. For the partitioned system, the results for the two partitioning methods, cycle-by-cycle behavior preservation (cbc) and critical path preservation (cp), described in section 5.4 and section 5.6 respectively are shown. The

Table 8.6. Critical path and execution time results.

Examples	Clock Cycles				Critical Path (ns)				Execution Time (ns)			
	Part1	Part2	Part3	Part4	Part1	Part2	Part3	Part4	Max CP		Var CP	
									(ns)	ratio	(ns)	ratio
Fac cp	35	44	53		9.4	4.1	6.9		1244	1.50	883	1.07
Fac cbc	23	29	36		9.9	4.5	7.7		869	1.05	629	0.76
Fac unpart	88				9.4				827	1.00	827	1.00
Ch cp	13	11	72	169	11.9	7.9	7.8	11.4	3150	1.19	2736	1.03
Ch cbc	11	9	60	141	12.3	8.3	8.4	12.8	2829	1.07	2528	0.96
Ch unpart	221				12.0				2646	1.00	2646	1.00
Dif cp	91	125			8.0	6.1			1740	1.24	1498	1.06
Dif cbc	69	95			8.2	6.3			1345	0.96	1160	0.82
Dif unpart	164				8.6				1407	1.00	1407	1.00
Vol cp	103	142			5.5	18.7			4579	1.21	3228	0.86
Vol cbc	84	117			7.0	18.9			3793	1.01	2789	0.74
Vol unpart	201				18.8				3773	1.00	3773	1.00
NL cp	207	458			4.4	4.6			3027	1.29	2998	1.27
NL cbc	151	334			4.5	4.6			2209	0.94	2203	0.94
NL unpart	485				4.8				2353	1.00	2353	1.00

Clock Cycles columns show the number of clock cycles required to execute each part. The *Critical Path* columns show the critical path period for each part. The *Max CP* column shows the total execution time if the slowest clock period (i.e. longest critical path) is used for all the parts. The *Max CP* column is further broken down to the actual execution time in nano-seconds (*ns*) and as a ratio with that of the unpartitioned system (*ratio*). The *Var CP* column shows the total execution time if each part uses a clock period that is equal to the critical path for that part. Hence the clocks for each part might be different. Again it is further broken down into the actual execution time and as a ratio with that of the unpartitioned system.

8.5. Partitioning Algorithm and Heuristic

We implemented the branch-and-bound optimal algorithm and the simulated annealing heuristic. The results are shown in Table 8.7. The first column shows the examples used. *Fac* is a factorization program. *Chinese* evaluates the Chinese Remainder Theorem. *Diffeq* is an example from the HLSynth MCNC benchmark. *Volsyn* is a

Table 8.7. Results from Branch-and-Bound and Simulated Annealing partitioning.

Examples	States	Energy (uJ)			Time (s)		% Savings	
		Unpart	B&B	SA	B&B	SA	B&B	SA
Fac	20	3,503	1,454	1,454	2	1	58.5%	58.5%
Chinese	44	4,831	1,614	1,672	10hr	1	66.6%	65.4%
Diffeq	58	7,928	4,083	4,106	10hr	2	48.5%	48.2%
Volsyn	16	2,465	1,634	1,634	1	1	33.7%	33.7%
NLoops	12	4,275	1,116	1,120	1	1	73.9%	73.8%
MP	101	728	-	-	14+hr	5	-	-
DSP	13	52	44	45	1	1	13.8%	12.7%
Average							**49.2%**	**48.7%**

volume-measuring medical instrument controller. *NLoops* is an example with nested loops. *MP* is a small microprocessor. *DSP* is a digital signal processor. The *St* column shows the number of states for the examples. The *energy* columns show the energy for the unpartitioned, the branch-and-bound partitioned system, and the simulated annealing partitioned system respectively. The *time* column shows the execution time in seconds for the branch-and-bound and simulated annealing. The *% Savings* columns show the percent energy savings obtained from the branch-and-bound and simulated annealing partitioning respectively. There is no result for the MP example because the CPU run time took more than 14 hours.

An average of 49.2% energy reduction was achieved using the branch-and-bound algorithm, and 48.7% using the simulated annealing heuristic. We see that the solution obtained by the simulated annealing heuristic is on average only 0.5% worst than that obtained by the branch-and-bound algorithm, yet it is an order of magnitude faster. The same random seed for the simulated annealing heuristic was used for all the examples. These results compare favorably with the 41% average energy savings shown in Table 8.4.

Table 8.8 shows the statistics for the branch-and-bound algorithm. The first and second columns show the number of states and the total number of nodes in the binary search tree respectively. The third column shows the percentage of nodes pruned. The *Branches Pruned* columns show the number of branches pruned as a result of satisfying conditions one and two respectively as discussed in section 7.1. Finally, the *Time* column

shows the execution time for the algorithm. Although more than 99.9% of the nodes are pruned for the 50 state example, the execution time is still quite long.

8.6. Internal to External Energy Ratios

Table 8.9 compares the effect of different internal and external capacitance ratios. The six columns labeled *E/I=* show the percent energy reduction achieved for the external to internal energy ratio of 10, 50, 100, 200, and 500 respectively. The percentages show the energy reduction from the unpartitioned FSMD. Depending on the external to internal energy ratio, the average energy reduction can range from 10.9% to as much as 49.2%.

The amount of power reduction is greatly affected by the external to internal energy ratio. As the ratio increases, it becomes harder to find a partitioning that reduces the overall power. Although our sizable power savings are dependent on this ratio, there are two factors that may help to keep this ratio small. First, the new copper technology for integrated circuits will slow down the increase of this ratio. Second, the bus between the partitioned FSMDs may be very short because it exists within a component, rather than

Table 8.8. Branch-and-Bound statistics.

States	Total Nodes	% Nodes Pruned	Branches Pruned		Time (sec)
			Condition 1	Condition 2	
20	1.0×10^6	99.7%	0	21	1
25	3.3×10^7	99.8%	81	28,551	281
50	1.1×10^{15}	99.968%	1,137	495,915	36,719

the typical on-chip busses that connect large numbers of components and must span large portions of the chip.

8.7. Power Saving Techniques Compared

Figure 8.2 shows a comparison of average power savings between our FSMD partitioning technique with the guarded evaluation [4] and selectively-clocked [5] 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.

Table 8.9. Effects of different external to internal energy ratios.

Examples	E/I=10	E/I=50	E/I=100	E/I=200	E/I=500
FAC	58.5%	43.2%	30.5%	15.9%	0.0%
Chinese	66.6%	60.8%	57.7%	52.2%	19.5%
Diffeq	48.5%	41.9%	0.0%	0.0%	0.0%
Volsyn	33.7%	0.0%	0.0%	0.0%	0.0%
NLoops	73.9%	71.6%	68.7%	63.0%	45.7%
DSP	13.8%	0.0%	0.0%	0.0%	0.0%
Average	49.2%	36.2%	26.1%	21.8%	10.9%

In the guarded evaluation technique, all the savings come from the reduced switching activity of the functional units. This is accomplished by adding latches in front of the functional units. Hence the power consumption for the functional units is reduced by 41% but the power consumption for the registers (which includes latches) is increased by 74% over the unoptimized technique. The power usage by the controller is about the same as the unoptimized technique.

In the selectively-clocked technique, most of the savings are from the FSM and is about 45%. However, because the power consumption for the FSM accounts for less than 14% of the total power consumption, the overall power reduction is very small. The power usage by the registers and functional units are about the same as the unoptimized technique.

In the FSMD partitioning technique, we have 58% power savings for the functional units and muxes, and 42% for the controller. This is offset by an increase of 24% for the
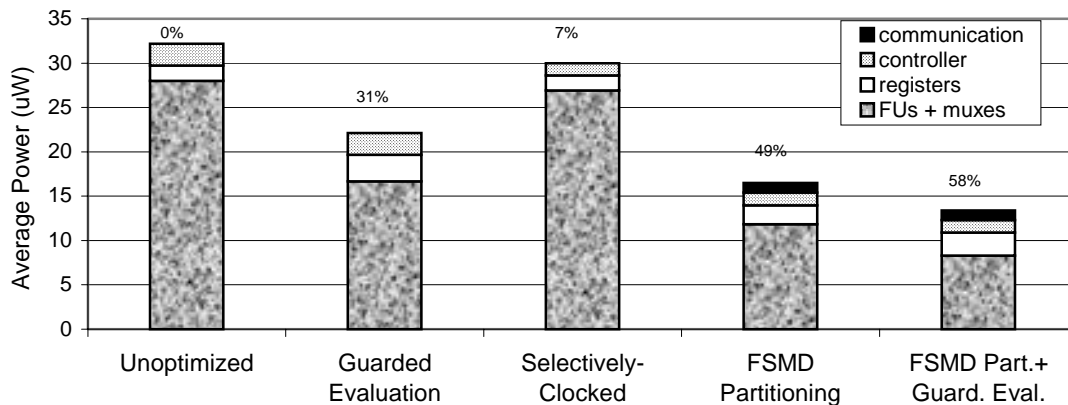
Figure 8.2. Average power savings compared. Percentages show power savings. Shorter bars are better.

registers and 10% by communication. The power usage by the functional units and muxes is less for the FSMD partitioning technique than for the guarded evaluation technique because there is power savings from the muxes for the former technique but not the latter technique. The power usage by the registers is more than the unoptimized technique because some registers have to be duplicated. However, it is slightly less than that of the guarded evaluation technique 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 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 8.2.

In our analysis, we count the maximum number of functional units used in any clock cycle and the actual total number of functional units synthesized. From this, we get the ratio of functional units that are doing useful and useless work. This is done for both the unpartitioned and partitioned circuits. We found that in the unpartitioned circuit approximately 1/3 of the functional units are doing useless work and approximately 1/5 for the partitioned circuit.

8.8.   Summary

In this chapter, we have presented our results from various experiments that we performed on testing our functional partitioning technique. We found that our FSMD functional partitioning technique can reduce power by about 50%.

References

[1]   Y. Hsu, T. Liu, F. Tsai, S. Lin, & C. Yu, "Digital design from concept to prototype in hours," in *Asia-Pacific Conference on Circuits and Systems*, December 1994.

[2]   PureSpeed Simulator, An event-driven simulator from FrontLine Design Automation, Inc.

[3]   F. Vahid, "I/O and Performance Tradeoffs with the FunctionBus during Multi-FPGA Partitioning," *International Symposium on FPGA,* pp. 27-34, February, 1997.

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

[5]   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.

Chapter 9. Conclusion

Power reduction is a critical metric for circuit design. We have introduced a new functional partitioning technique for reducing power consumption either at the procedural behavioral level or at the finite-state machine with datapath behavioral level. Unlike previous power reduction shutdown techniques that focus only on either the datapath or the controller, our approach partitions the entire processor to shut down both the controller and the datapath.

An optimal branch-and-bound algorithm and a near optimal simulated annealing heuristic for performing FSMD functional partitioning for low power were presented. The algorithm and heuristic make use of our power estimation model and the theoretical energy bounds for functional partitioning. The branch-and-bound algorithm and simulated annealing heuristic were both able to achieve roughly a 49% energy savings. This compares favorably with our simulated average energy savings of 41%.

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.