Reducing Switching Activity on Datapath Buses with Control-Signal Gating

Hema Kapadia, Luca Benini, and Giovanni De Micheli, *Fellow, IEEE*

***Abstract*— This paper presents a technique for saving power dissipation in large datapaths by reducing unnecessary switching activity on buses. The focus of the technique is on achieving effective power savings with minimal overhead. When a bus is not going to be used in a datapath, it is held in a quiescent state by stopping the propagation of switching activity through the module(s) driving the bus. The "observability don't-care condition" of a bus is defined to detect unnecessary switching activity on the bus. This condition is used to gate control signals going to the bus-driver modules so that switching activity on the module inputs does not propagate to the bus. A methodology for automatically synthesizing gated control signals from the register-transfer-level description of a design is presented. The technique has very low area, delay, power, and designer effort overhead. It was applied to one of the integer execution units of a 64-bit, two-way superscalar RISC microprocessor. Experimental results from running various application programs on the microprocessor show an average of 26.6% reduction in dynamic switching power in the execution unit, with no increase in critical path delay and negligible area overhead.**

***Index Terms*— Clock gating, control signal gating, data buses, datapaths, logic synthesis, low power, power management, switching activity.**

I. Introduction

POWER dissipation continues to grow as an important challenge in deep submicron chip design. Power management is crucial for reliability, packaging, and cooling costs of high-performance systems, and battery life of portable devices. Achieving low average power dissipation in a complex chip calls for employing a combination of various low-power techniques at all levels of design abstraction [1]–[3]. However, low power is usually a secondary design goal after high performance and, in some cases, even after designer effort. This calls for low-power techniques that give significant power savings with low overhead.

A large fraction of the total power dissipation on a chip today is typically due to clocks, memory, and datapaths [4], [5]. Circuit design techniques are used to reduce the power consumption of active elements in memory cells, clock latches, and datapath modules. Heavily loaded wires in each of these design areas need special attention since the dynamic switching power due to high capacitive loading on the wires can dominate the total power dissipation. Low-swing buses are used extensively in memories to reduce power consumption very effectively. Using low swings on datapath buses also gives good power savings, but it requires significant effort in the design and layout of bus receivers [5].

The technique presented in this paper, control-signal gating, targets the dynamic switching power of heavily loaded buses in large datapaths. The emphasis is on taking advantage of redundant switching activity on each bus within a datapath to save power, while keeping the timing, area, and computational overhead low. Datapaths have a regular bit-slice structure of logic modules connected by buses. Some of the logic modules are controlled by control-signal inputs that are generated by random logic outside the datapath, called control logic. Control-signal gating modifies the control logic to reduce unnecessary switching activity on datapath buses. Conditions that imply that a bus is not going to be used to compute any of the primary outputs of the datapath are detected in the control logic. These conditions are used to gate control signals going to the modules driving the bus, so that no switching activity propagates through those modules. Thus, the switching activity on a bus is turned off when it is detected to be unused.

A brief discussion of the existing low-power techniques most applicable to datapaths, and their tradeoffs, is given in Section II. The formulation of control-signal gating is presented in Section III, where formal definitions are given for conditions when a bus is unused and when switching activity does not propagate through a module. These definitions are used to derive gated control signals that reduce unnecessary switching activity on buses. A methodology for implementing this technique on real designs is presented in Section IV. Section V discusses overheads and corner cases of the technique. In Section VI, experimental results of applying control-signal gating to a large datapath in a microprocessor show an average of 27.7% reduction in dynamic switching activity, resulting in 26.6% average reduction in the dynamic switching power of the datapath, with negligible overhead.

II. Low-Power Techniques for Datapaths

Clock gating turns off the clock signal going to large functional units when they are not needed [6], [7]. This saves dynamic switching power on the clock line, as well as power dissipated in the functional unit. High-level conditions that indicate that the output of a functional unit would be unused, such as a global stall in a microprocessor, are used to disable the output bus drivers of the functional unit [8]. This saves dynamic switching power on heavily loaded buses. However,

Manuscript received August 7, 1998; revised October 23, 1998. This work was supported in part by the National Science Foundation under Grant NIP9421129, in part by Toshiba Corp., and in part by the Advanced Research Project Agency under Grant DABT63-94-C-0054-P00008.

H. Kapadia and G. De Micheli are with the Computer Systems Laboratory, Stanford University, Stanford, CA 94305 USA.

L. Benini is with DEIS, University of Bologna, Bologna Italy.

Publisher Item Identifier S 0018-9200(99)01646-7.

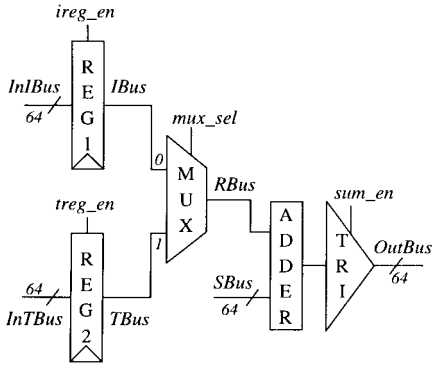


Fig. 1. Example datapath.

clock gating or disabling bus drivers at this level does not take advantage of situations when one part of a functional unit is in use while other parts are unused. Control-signal gating takes advantages of low-level redundancies within the datapath of a functional unit to save power.

Two techniques exist for grouping flip-flops within a functional unit for clock gating: hold-condition detection [9] and redundant-clocking detection [10]. In hold-condition detection, flip-flops that share common hold conditions are grouped together for clock gating. This technique is not applicable when enabled flip-flops are used, which is common in large datapaths. Redundant-clocking detection collects and analyzes simulation traces to group flip-flops for clock gating. The grouping is trace dependent, and generation of the clock-gating conditions is not automated. Also, clock gating at low levels of granularity, such as applying different gating conditions on the clock inputs of two different modules within a datapath, increases local clock skew. This is an issue in high-performance designs, since controlling the clock skew is critical to performance.

Example 1: Fig. 1 shows part of a 64-bit datapath generating a primary output bus, *OutBus*. If clock gating was applied to this datapath, the clock would be turned off when none of the primary outputs of the datapath were going to be used. However, even when other parts of the datapath are computing useful output, *TBus*, *IBus*, and *RBus* are unused when *sum_en* is zero. Also, *IBus* is unused when *mux_sel* is one, and *TBus* is unused when *mux_sel* is zero. Using only one gated clock for the entire datapath would allow the buses to switch unnecessarily under these conditions. If clock gating was applied at a lower level, the clock line going to *Reg*1 would be turned off if *sum_en* was going to be zero or *mux_sel* was going to be one. Similarly, the clock line going to *Reg*2 would be turned off if *sum_en* or *mux_sel* was going to be zero. This would stop unnecessary switching activity on *TBus* and *IBus*. However, *RBus* would switch unnecessarily if *mux_sel* changed while *sum_en* was zero. Also, the skew between the clocks going to *Reg*1 and *Reg*2 would pose an additional constraint on the arrival time of *mux_sel.*

Precomputation-based methods [11] precompute the output of a sequential circuit one clock cycle early to reduce power dissipation in the following cycle. A few bits of the inputs going to the circuit are used to precompute the output. The circuit is selectively turned off based on these precomputed output values. Logic duplication of parts of the target circuit is required for multioutput circuits. When applied to modules in a datapath, another concern would be the extra wiring complexity added in the otherwise regular structure of the datapath in order to route bits of buses to precomputation logic.

Example 2: In Fig. 1, precomputation is not applicable since it is not possible to precompute the output of the adder or the multiplexer without using all bits of the input buses. The amount of logic added for precomputation would be comparable to the existing logic of those modules.

Guarded evaluation [12] places enabled transparent latches on inputs of the modules that need to be selectively turned off, using existing signals in the design as latch enables. Automatic selection of the enable signals is based on logical implication, which is complex to compute for large designs. The area and power overhead of placing guard latches in wide datapaths would be quite high.

Example 3: If the adder in Fig. 1 was to be selectively turned off through guarded evaluation, two 64-bit transparent latches would be placed on *RBus* and *SBus*, both disabled when *sum_en* was zero.

Although precomputation and guarded evaluation are quite effective for random logic and small datapaths, they have significant overhead in large datapaths. Also, both of the techniques focus on stopping switching activity in the circuits of datapath modules. In comparison, control-signal gating focuses on saving dynamic switching power on datapath buses. There is no logic or wiring added in the datapath structure, resulting in very low overheads. Also, structural information available at the register-transfer (RT) level is used to synthesize gated control signals, giving high computational efficiency.

Example 4: If control-signal gating was applied to Fig. 1, control signals *mux_sel* and *sum_en* would be used to determine the conditions when the internal buses *IBus*, *TBus*, and *RBus* are unused. These conditions would be used to gate *ireg_en*, *treg_en*, and *mux_sel* so that no new values are driven unnecessarily on *IBus*, *TBus*, and *RBus*, respectively.

III. Control-Signal Gating

The rationale of control-signal gating is that buses that are not used by the environment should be frozen in a quiescent state by stopping the propagation of switching activity through their drivers. This is achieved by using the "observability don't-care condition" (ODC) to detect when a bus would be unused and to stop the propagation of switching activity through the module(s) driving the bus.

A. Observability Don't-Care Conditions

In logic synthesis, the condition under which a Boolean variable in a combinational circuit is not observed by the environment is called the ODC of the variable [13]. The ODC of a variable is computed by traversing back through its fanout cone from primary outputs of the circuit to the variable, while incrementally computing ODC's of variables from the ODC's of their immediate successors. For a variable x with

one immediate successor y that is expressed by a logic function $y = f(x)$, the ODC is computed as shown in (1)

$$\text{ODC}(x) = (f(x)|_{x=1} \,\overline{\oplus}\, f(x)|_{x=0}) + \text{ODC}(y). \quad (1)$$

In this paper, "+" is used to represent the logical OR, "&" is used to represent the logical AND, "$\oplus$" is used to represent the exclusive OR, "|" is used to represent restriction condition, and overline is used to represent the complement of Boolean functions and variables.

The first term in (1) is the complement of the Boolean difference of y with respect to x, which denotes the condition under which x is not observed at y. The second term denotes the condition under which y itself is not observed by the environment.

In this work, we redefine the ODC to deal specifically with datapath buses. A datapath is treated as a logic network graph with vertices representing datapath modules and primary input/outputs and directed edges representing interconnections of the modules. Both sequential and combinational modules are included in the graph. The value of a Boolean function or variable in the current clock cycle is given a "$_{@T}$" suffix, the same value in the previous clock cycle is given a "$_{@(T-1)}$" suffix, and that value in the next clock cycle is given a "$_{@(T+1)}$" suffix. Thus, if a signal *CurSig* was the output of a flip-flop, the value of $\textit{CurSig}_{@(T+1)}$ would be the same as the value of the input of that flip-flop in the current clock cycle. If *CurSig* was the input of a flip-flop, the value of $\textit{CurSig}_{@(T-1)}$ should have been the same as the value of the output of that flip-flop in the current clock cycle. Unless otherwise specified, Boolean functions and variables are referred to in the current clock cycle. We define two types of ODC's of a bus in two environmental contexts: Bus ODC (ODC_B) and Module ODC (ODC_M.)

Definition 1: The ODC of a datapath bus D with respect to primary outputs of the datapath is called the bus ODC of D, or $\mathbf{ODC_B(D)}$.

The ODC_B of a bus gives the condition under which the bus is unused in the datapath. Hence bus transitions when the ODC_B is *True* are useless and can be eliminated to save power. We can eliminate bus transitions by stopping the propagation of switching activity through bus drivers.

Definition 2: The ODC of an input bus D of a datapath module with respect to the module outputs is called the module ODC of D, or $\mathbf{ODC_M(D)}$ through that module.

When the ODC_M is *True* for all inputs of a module, no switching activity propagates through the module. Thus, the rationale of control-signal gating can be realized if, when the ODC_B of a datapath bus is *True*, the ODC_M of all inputs of datapath modules driving that bus is also made *True*. This is illustrated in Example 5.

Example 5: For the datapath in Fig. 1, assuming that the ODC of *OutBus* is zero (always observed), ODC_B(*IBus*) in (2) captures the fact that *IBus* is not used in the datapath either when the multiplexer is not selecting it or when the tristate driver is disabled. ODC_M(*InIBus*) captures the fact that no switching activity is going to propagate to the output of *Reg*1 in the current clock cycle (T) if the register enable was zero in the previous clock cycle ($T-1$). Unnecessary switching activity on *IBus* can be reduced by making ODC_M(*InIBus*) *True* when ODC_B(*IBus*) is *True*. This gives the equation for $\textit{ireg\_en\_gated}_{@(T-1)}$. Converting this to the equation for *ireg_en_gated* in the current clock cycle, we get the condition that *ireg_en* should be gated in the current clock cycle if ODC_B(*IBus*) is going to be *True* in the next clock cycle. This reduces, but may not stop, unnecessary switching activity on *IBus*, since the unobservability due to the adder is ignored

$$\begin{aligned}
\text{ODC}_B(\textit{IBus}) &= \textit{mux_sel} + \overline{\textit{sum}}\textit{_en} \\
\text{ODC}_M(\textit{InIBus})_{@(T)} &= \overline{\textit{ireg_en}}_{@(T-1)} \\
\textit{ireg_en_gated}_{@(T-1)} &= \textit{ireg_en}_{@(T-1)} \,\&\, \overline{\text{ODC}_B(\textit{IBus})} \\
\textit{ireg_en_gated} &= \textit{ireg_en} \,\&\, \overline{\text{ODC}_B(\textit{IBus})}_{@(T+1)}.
\end{aligned} \quad (2)$$

B. Computing Low-Overhead Module ODC (ODC_M)

Depending on the type of datapath module, the ODC_M of an input bus of the module can range from quite complex to quite simple. An input bus of a 64-bit multiplier is unobservable at the output when the other input bus is zero. Generating this ODC_M condition in hardware would require a 64-bit NOR gate for each input bus. This adds significant area, delay, power, and design effort overhead for introducing new elements in the regular datapath structure. On the other hand, as seen in Example 5, the ODC_M of a register input bus is simply the complement of a one-cycle late version of the register enable. To use this disparity among ODC_M complexities through different datapath modules, we classify datapath modules into two categories: *computational modules* and *steering modules*.

In computational modules, the ODC_M of one input bus depends on the values of other input buses. Arithmetic and logic modules such as adders, multipliers, logic gates, etc., fall into this category. Making use of the unobservability through a computational module to save power would require inserting new logic in the datapath, resulting in prohibitive overheads. Hence for the purpose of control-signal gating, we will assume that inputs of a computational module are always observed at the outputs. Thus, the ODC_M through computational modules is zero by definition.

Steering modules selectively steer one or none of the input data buses to the module output, depending on the values of control-signal inputs. The ODC_M of a data input of a steering module only depends on control signals. We consider three commonly used steering modules: *tristate drivers*, *registers*, and *multiplexers*. All registers are assumed to be enabled registers, and all multiplexer select lines are assumed to be one-hot. Equation (3) summarizes the ODC_M of input buses through different steering modules shown in Fig. 2

$$\begin{aligned}
\textit{Tri-State}: &\quad \text{ODC}_M(\textit{TriIn}) = \overline{\textit{tri_en}} \\
\textit{Register}: &\quad \text{ODC}_M(\textit{RegIn}) = \overline{\textit{reg_en}}_{@(T-1)} \\
\textit{Multiplexer}: &\quad \text{ODC}_M(\textit{MuxIn}0) = \overline{\textit{sel}}0, \\
&\quad \text{ODC}_M(\textit{MuxIn}1) = \overline{\textit{sel}}1, \cdots \\
&\quad \text{ODC}_M(\textit{MuxIn}N) = \overline{\textit{selN}}.
\end{aligned} \quad (3)$$

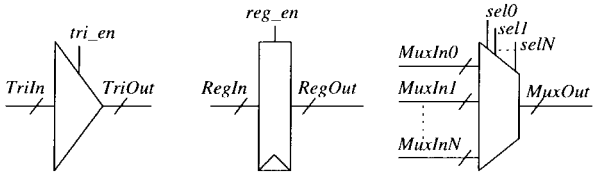


Fig. 2. Steering module types: tristate, register, and multiplexer.

The input bus of a tristate driver is unobservable at the output when the tristate enable is false. A register input bus is unobservable at the output in the current clock cycle (T) if the register enable signal was false in the previous clock cycle ($T-1$). A multiplexer input bus is unobservable at the output when the corresponding select line is false. Thus, the ODC_M of a bus through a steering module is simply the complement of the control signal that steers the bus to the module output.

C. Computing Bus ODC (ODC_B)

The ODC_B of a bus is computed by traversing its fanout cone backward from primary outputs to the bus while incrementally computing the ODC_B at each vertex from its immediate successors in the cone. Starting with the ODC of primary outputs, new terms are added to the bus ODC at each steering module along each fanout path of the bus.

In its traditional definition, the ODC of a variable with multiple fanouts is computed by first computing ODC's along each fanout independently and then combining them. Combining the ODC's of a bus at a multifanout point would be quite complex if the fanout ODC's depended on the bus itself [13]. Such dependencies are likely to be introduced by conditional branch mechanisms where the output of comparison of two buses controls a steering module. Taking advantage of such dependencies would also require expensive routing of wires from the datapath to the control-signal gating logic. This is simplified in control-signal gating by assuming that there are no data-dependent control signals controlling the steering modules. This allows for ODC's along different fanouts of a bus to be combined simply by intersection.

If a bus has N fanouts $\{Fanout_1, Fanout_2, \cdots, Fanout_N\}$, it is not observed at the primary outputs of the datapath if and only if none of the fanouts are observed. The ODC_B of the bus is the intersection of the ODC_B of all fanouts, as shown in (4)

$$\mathrm{ODC}_B(bus) = \&_{i=1}^{N}\ \mathrm{ODC}_B(Fanout_i). \tag{4}$$

If a bus is connected to one of the inputs of a module that has N outputs $\{Out_1, Out_2, \cdots, Out_N\}$, it is not observed at the primary outputs of the datapath either when it is not observed at the module output or when none of the module outputs are observed at primary outputs of the datapath. If the module loading the bus is combinational, the ODC_B of the bus is given by (5)

$$\mathrm{ODC}_B(bus) = \mathrm{ODC}_M(bus) + \left(\&_{i=1}^{N}\ \mathrm{ODC}_B(Out_i)\right). \tag{5}$$

If the module loading the bus is sequential, the current value of the bus will not be observed at the primary outputs of the datapath if it does not propagate to the module output in the next clock cycle, or if the module output in the next clock cycle is not observed at the primary outputs of the datapath. The ODC_B of the bus is given by (6) in this case

$$\mathrm{ODC}_B(bus) = \left(\mathrm{ODC}_M(bus) + \left(\&_{i=1}^{N}\ \mathrm{ODC}_B(Out_i)\right)\right)_{@(T+1)} \tag{6}$$

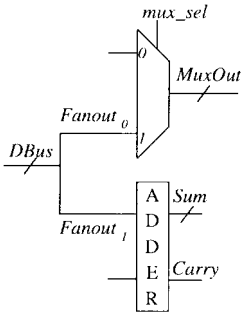


Fig. 3. Computation of ODC_B of *DBus*.

Equations (4)–(6) are used repeatedly in the fanout cone of a bus to compute its ODC_B. Example 6 illustrates the computation of the ODC_B a bus.

Example 6: The fanout cone of *DBus* is shown in Fig. 3. If the ODC's of primary outputs *MuxOut*, *Sum*, and *Carry* were given, the ODC_B of *DBus* would be computed as shown in (7). Here, since the adder is a computational module, it does not contribute to the ODC_B of $Fanout_1$

$$\begin{aligned}\mathrm{ODC}_B(Fanout_1) &= \mathrm{ODC}(Sum)\ \&\ \mathrm{ODC}(Carry)\\ \mathrm{ODC}_B(Fanout_0) &= \overline{mux\_sel} + \mathrm{ODC}(MuxOut)\\ \mathrm{ODC}_B(DBus) &= \mathrm{ODC}_B(Fanout_0)\ \&\ \mathrm{ODC}_B(Fanout_1).\end{aligned} \tag{7}$$

Reconvergent fanout on a bus introduces redundant terms in its ODC_B. Hence computed ODC_B's of buses should be simplified using logic optimization to remove these redundant terms. This is illustrated in Example 7.

Example 7: If *MuxOut* in Fig. 3 was connected to the unconnected input of the adder, the ODC_B of *MuxOut* would be the same as the ODC_B of *Fanout*1 in (7). This would change the ODC_B of *DBus* to (8)

$$\begin{aligned}\mathrm{ODC}_B(Fanout_0) &= \overline{mux\_sel} + \mathrm{ODC}_B(Fanout_1)\\ \mathrm{ODC}_B(DBus) &= \mathrm{ODC}_B(Fanout_0)\ \&\ \mathrm{ODC}_B(Fanout_1)\\ &= \mathrm{ODC}_B(Fanout_1).\end{aligned} \tag{8}$$

D. Stopping Propagation of Switching Activity Through a Module

As discussed in Section III-A, propagation of switching activity through a module should be stopped when the ODC_B of its output(s) is *True*. Depending on the type of module, this can be done either by forcing the ODC_M of all inputs to be *True* or by holding all inputs unchanged.

Since the ODC_M through a computational module is zero, stopping the propagation of switching activity through a computational module requires that none of the inputs change. This would require placing enabled latches on wide input buses in the datapath, which can be quite expensive. Hence control-signal gating is not applied to computational modules.

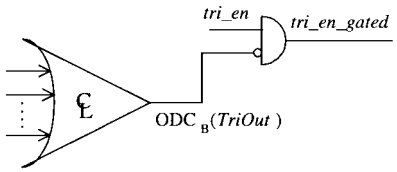


Fig. 4. Gating a tristate driver.

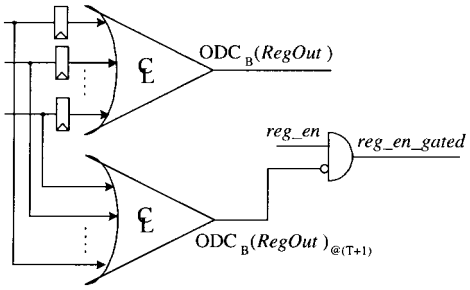


Fig. 5. Gating a register.

In a tristate driver, the ODC_M of the input bus is simply the complement of the tristate enable (3). Thus, the propagation of unnecessary switching activity through a tristate driver is stopped by gating the tristate enable by the ODC_B of its output bus, as shown in Fig. 4.

In a register, the ODC_M of the input bus is the complement of the register enable in the previous clock cycle (3). Hence the register enable needs to be gated one cycle early in order to stop the propagation of switching activity through the register when the ODC_B of its output is *True*. This is done by gating the register enable by a one-cycle early version of the output ODC_B, $\text{ODC}_{B@(T+1)}$, as shown in Fig. 5. The fanin cone of the ODC_B is backtracked up to flip-flops, and the logic of the cone is reconstructed from the inputs of those flip-flops instead of the outputs to get $\text{ODC}_{B@(T+1)}$.

In a multiplexer, since the ODC_M of each input bus is the complement of its corresponding select line (3), the intersection of ODC_M's of all input buses is empty. Thus, in order to stop the propagation of unnecessary switching activity through a multiplexer, the select lines and the selected input bus need to be held unchanged when the ODC_B of its output is *True*. As shown in Fig. 6, a select line is held unchanged by backtracking its fanin cone up to flip-flops and gating the enables of these flip-flops with the $\text{ODC}_{B@(T+1)}$ of the multiplexer output. Similar to gating register enables, the flip-flop enables are gated one cycle early in order to stop the select lines from changing in the current cycle if the ODC_B of the output bus is *True*. If the fanin cone of a select line has fanouts leaving the cone, some logic needs to be duplicated to preserve the functionality of those fanouts. However, the required duplication is likely to be small since heavily loaded select lines are likely to have only a few levels of logic after flip-flops in order to meet timing constraints.

To hold the multiplexer input buses unchanged, enabled latches would have to be added in the datapath. We chose not to do this to maintain low overheads at the expense of giving up power savings. However, control-signal gating applied to steering modules in the fanin cone of a multiplexer

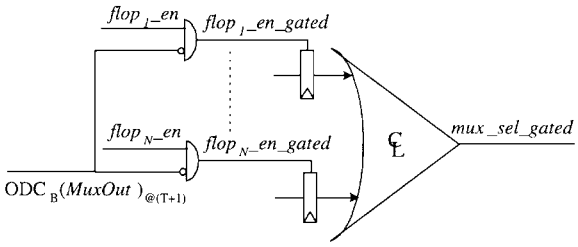


Fig. 6. Gating a multiplexer.

input bus would hold the bus unchanged in some fraction of all the clock cycles when the output ODC_B is *True* and the input bus is selected by the select lines. We save power on the multiplexer output bus in only these clock cycles. In the remaining clock cycles, the multiplexer output would switch unnecessarily, mirroring the activity on the selected input bus. The fraction of clock cycles in which power can be saved on the output bus of a multiplexer depends on other fanouts of the input buses, as illustrated in Examples 8 and 9.

Example 8: In Fig. 1, the multiplexer output is *RBus*, which is unused in the datapath when *sum_en* is zero. Both multiplexer input buses *TBus* and *IBus* have no other fanouts. As shown in (9), the ODC_B of *RBus* logically implies the ODC_B's of *IBus* and *TBus*. Using these ODC_B's to stop the propagation of switching activity would give (10) for gated control signals

$$\begin{aligned} \text{ODC}_B(\textit{IBus}) &= (\textit{mux\_sel} + \overline{\textit{sum\_en}}) \\ \text{ODC}_B(\textit{TBus}) &= (\overline{\textit{mux\_sel}} + \overline{\textit{sum\_en}}) \\ \text{ODC}_B(\textit{RBus}) &= \overline{\textit{sum\_en}} \\ \Longrightarrow \text{ODC}_B(\textit{RBus}) &\rightarrow \text{ODC}_B(\textit{TBus}), \text{ODC}_B(\textit{IBus}) \end{aligned} \tag{9}$$

$$\begin{aligned} \textit{ireg\_en\_gated} &= \textit{ireg\_en} \,\&\, \overline{\text{ODC}_B(\textit{IBus})}_{@(T+1)} \\ \textit{treg\_en\_gated} &= \textit{treg\_en} \,\&\, \overline{\text{ODC}_B(\textit{TBus})}_{@(T+1)} \\ \textit{mux\_sel\_gated}_{@T} &= \textit{mux\_sel\_gated}_{@(T-1)}, \\ &\quad \textit{if } (\text{ODC}_B(\textit{RBus})_{@(T+1)} == \textit{True}) \end{aligned} \tag{10}$$

Thus, if *sum_en* is going to be zero in the next clock cycle, both *ireg_en_gated* and *treg_en_gated* are zero in the current cycle, and *mux_sel_gated* is held unchanged. Unnecessary switching activity on *RBus* is stopped in 100% of all clock cycles when it is unused in the datapath, since switching activity is also stopped on both *TBus* and *IBus* in those clock cycles.

Example 9: On the other hand, consider modifying the datapath of Fig. 1 to put additional fanouts on multiplexer input buses, as shown in Fig. 7. Tristate enables *logic_en*, *sum_en*, and *shift_en* are mutually exclusive, and the ODC of *OutBus* is zero. Unlike Example 8, the ODC_B of *RBus* does not imply the ODC_B of *TBus* or *IBus*, as shown in (11)

$$\begin{aligned} \text{ODC}_B(\textit{IBus}) &= ((\textit{mux\_sel} + \overline{\textit{sum\_en}}) \,\&\, \overline{\textit{logic\_en}}) \\ \text{ODC}_B(\textit{TBus}) &= ((\overline{\textit{mux\_sel}} + \overline{\textit{sum\_en}}) \,\&\, \overline{\textit{shift\_en}}) \\ \text{ODC}_B(\textit{RBus}) &= \overline{\textit{sum\_en}} \end{aligned} \tag{11}$$

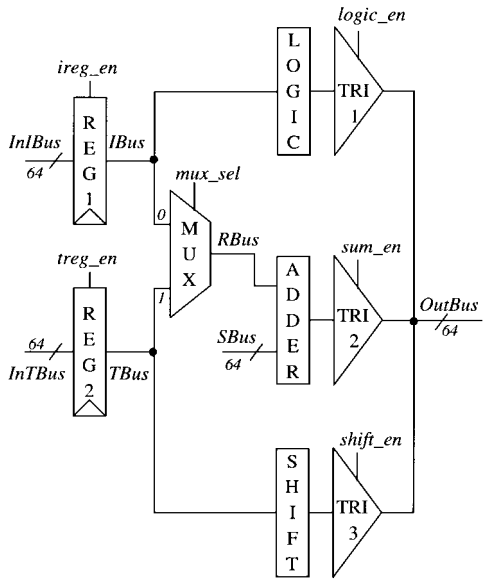


Fig. 7. Modified example datapath.

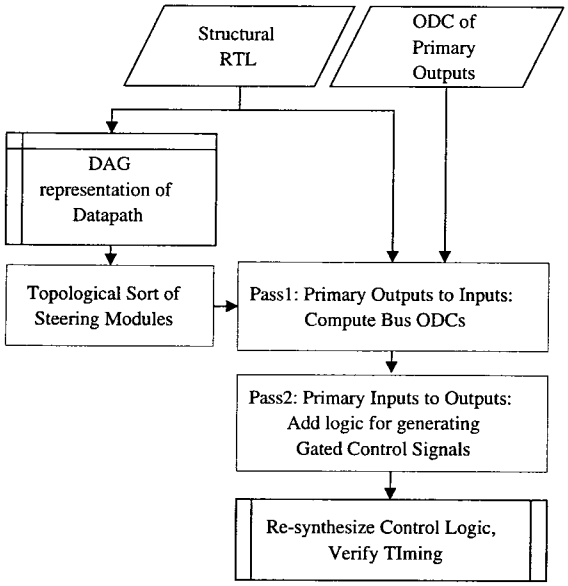


Fig. 8. Methodology for control-signal gating.

Gated control signals are generated by using (11) in (10). When *mux_sel* is zero (selecting *IBus*) and *sum_en* is zero, *RBus* is unused in the datapath and its switching activity depends on *IBus*. In this case, if *logic_en* was one, *IBus* would continue switching since it is used by the logic module. This would cause *RBus* to switch unnecessarily. However, if *shift_en* was one instead, switching activity would be stopped on *IBus* by *ireg_en_gated* and *RBus* would not switch unnecessarily. If all three tristate enables had equal probability of being one, unnecessary switching activity would be stopped on *RBus* in 50% of the clock cycles when *sum_en* was zero.

Equation (12) summarizes the method to stop the propagation of switching activity through different types of steering modules shown in Figs. 5–7

$$\begin{aligned}
&\textit{Tristate:}\\
&\quad \textit{tri_en_gated} = \textit{tri_en}\ \&\ \overline{\mathrm{ODC}_B(\textit{TriOut})}\\
&\textit{Register:}\\
&\quad \textit{reg_en_gated} = \textit{reg_en}\ \&\ \overline{\mathrm{ODC}_B(\textit{RegOut})}_{@(T+1)}\\
&\textit{Multiplexer:}\\
&\quad \textit{if}\ (\mathrm{ODC}_B(\textit{MuxOut})_{@(T+1)} == \textit{True}),\\
&\quad \textit{mux_sel_gated}_{@(T+1)} = \textit{mux_sel_gated}_{@(T)}. \qquad (12)
\end{aligned}$$

The logic added to stop the propagation of switching activity in all three types of steering modules uses and modifies control signals only. Hence control-signal gating does not require any modifications in the datapath.

IV. Implementation Methodology

Fig. 8 shows the tool flow for applying control-signal gating to real designs. The inputs to the tool are 1) a structural RT-level description of the design that has been partitioned into datapath and control-logic units and 2) information about ODC's of primary outputs. Using a netlist database manipulation system, steering modules in the datapath are topologically sorted and traversed twice to generate gated control signals that are resynthesized with the existing control logic using a synthesis tool.

A. Topological Ordering of Steering Modules

For the purpose of topological ordering, the logic network graph representation of a datapath (discussed in Section III-A) is converted to a directed acyclic graph (DAG) by removing feedback edges. Also, computational modules are removed from the set of vertices of this DAG and are treated as fanout points from each input to all outputs of the module. This DAG is sorted [13], [15] in increasing topological order from primary inputs of the datapath to primary outputs. The complexity of topological sort is $O(N+E)$, where N is the number of vertices and E is the number of edges in the DAG [16]. Even for large datapaths with wide buses, the DAG representation used here is quite simple since it only has as many vertices as the number of steering modules in the datapath and individual bits of buses are collapsed in the edges. This results in insignificant computational times for topological ordering.

B. First Pass

The first pass visits steering modules in the datapath netlist in decreasing topological order (from primary outputs to primary inputs). The logical expression of the ODC_B of the output bus of each steering module is computed as shown in the pseudocode of *ComputeBusODC* in Fig. 9.

For a given steering module driving an output bus, *ComputeBusODC* starts with an ODC_B of one (never observed) for the bus and follows each fanout of it in the datapath netlist. If the fanout is a primary output, its ODC is assigned to the fanout ODC_B. If the ODC of the primary output is not given, it is assumed to be zero (always observed). If the module loading the fanout is a steering module, the ODC_B of the fanout is computed from its ODC_M through the loading module and the ODC_B of the outputs of the loading module. Due to sequential feedback loops in the datapath, the steering module loading the

```
ComputeBusODC(OutputBus, DrivingModule) {
  ODC_B(OutputBus) = 1;
  foreach (Fanout of OutputBus) {
    ODC_B(Fanout) = 1;
    if (Fanout is a Primary Output) {
      ODC_B(Fanout) = ODC(PrimaryOutput) if given,
                    = 0 otherwise;
    }
    /* Fanout is input of a LoadingModule */
    elsif (LoadingModule is a Steering Module) {
      Get ODC_M(Fanout) (Equation 3);
      if (TopoOrder(LoadingModule) >
          TopoOrder(DrivingModule))
        Compute ODC_B(Fanout) (Equations 6 and 7);
      else
        ODC_B(Fanout) = ODC_M(Fanout);
    }
    else {
      /* LoadingModule is a computational module with
       * N outputs ComOut_1,...,ComOut_N */
      foreach ComOut_i (i = 1,2,...,N)
        ODC_B(Fanout) &= ComputeBusODC(ComOut_i, LoadingModule);
    }
    ODC_B(OutputBus) &= ODC_B(Fanout);
  }
  return(ODC_B(OutputBus));
}
```

Fig. 9. Pseudocode for computing bus ODC.

fanout could have a lower topological order number than that of the driving module. In this case, the ODC_B of the outputs of the loading module is not yet computed, and hence it is assumed to be zero. This is a conservative simplification that gives up some redundant switching activity on the fanout due to steering modules in the feed-forward path from the loading module to the driving module.

If the loading module is a computational module, *ComputeBusODC* is called recursively for each of its outputs. From (4), the ODC_B of the fanout is computed as the intersection of all computational module output ODC_B's. Similarly, in the outer loop, the ODC_B of the bus is computed as the intersection of all fanout ODC_B's. For registers and multiplexers, the logical expression of the $\text{ODC}_{B@(T+1)}$ of the output bus is computed from the fanin cone of the corresponding ODC_B computed by *ComputeBusODC*, as described in Section III-D.

C. Second Pass

The second pass traverses steering modules in the datapath netlist in increasing topological order, from primary inputs to primary outputs. Depending on the type of steering module visited, the logic required for gating control signals is computed using (12). In the netlist of the control-logic unit, new logic is added for generating the gated control signals and bus ODC's, and steering-module control signals are replaced with the gated control signals.

D. Postprocessing

Last, the control-logic unit needs to be resynthesized to optimize and map control-signal gating logic along with the rest of the unit. Timing analysis should be performed to make sure that timing constraints are met after applying control-signal gating.

V. Overheads and Corner Cases

While applying control-signal gating to a datapath, the overheads and corner cases of the technique need to be considered in order to decide its scope and applicability. These are discussed in the next two subsections, along with proposals for design-specific solutions to overcome some of them.

A. Overheads

Throughout the derivation of control-signal gating in Section III, tradeoffs were made to keep the overhead of the technique to a minimum. Gates are added in the control-logic unit for generation of ODC_B's and gated control signals, and some control logic is duplicated in order to preserve the functionality of fanouts leaving fanin cones of multiplexer select lines, as discussed in Section III-D. This comprises the area overhead and also results in additional power dissipation in control logic.

There are extra fanouts added to control signals to generate the ODC_B of buses. This could slow down the control signals and increase short-circuit power due to slower rise and fall transitions on these signals.

An extra gate is added in the paths of register and tristate enables. If an enable signal is in the critical path, its fanin cone should be reoptimized after adding the gating logic in order to absorb the gating delay into the existing logic. If such a signal has a lot of other fanouts (such as a microprocessor *Stall* signal) and is in the critical path, a small amount of logic duplication should be considered.

The ODC_B of a bus that is topologically close to the primary inputs of a datapath is more complex than that of a bus that is close to the primary outputs, since the fanout cones grow as we go from primary outputs toward primary inputs. In topologically deep datapaths, the ODC_B of a bus close to primary inputs could get quite complex. This provides opportunity for maximum removal of unnecessary switching activity on the bus looking ahead over multiple levels of steering logic. However, if the logic for generation of the ODC_B of such a bus ends up in the critical path, the fanout cone should be restricted to a smaller size. This can be done by traversing backward through the fanout cone of the bus and removing steering modules at the leaves of the cone until the resulting ODC_B meets timing constraints.

B. Corner Cases

Signals that are primary inputs of the chip or are fetched from on-chip memory in the current clock cycle are just-in-time signals. It is not possible to find a one-cycle early version of such a signal.

Control-signal gating uses a one-cycle early version of the ODC_B of register and multiplexer output buses to gate the control-signal inputs of those modules. It is not possible to generate the ODC_B one cycle early if its fanin cone depends on a just-in-time signal. Hence the ODC of the primary output or the ODC_M of the steering module input that depends on a just-in-time signal is assumed to be zero when computing the ODC_B of a register or a multiplexer output. This case is more likely to occur in a datapath with a pipeline of registers.

Example 10: For a P-stage pipeline of registers, the register enable of the first register in the pipeline would have to be

TABLE I
POWER SAVINGS WITH CONTROL-SIGNAL GATING

Benchmarks			Switching	Dyn. Switching Power (W)		
Name	Runtime	Occupancy	Reduction	Ungated	Gated	Reduction
ProtcolProc	212,798	16.2	21.8%	3.0	2.3	22.1%
Sumup	7,312	19.1	19.9%	3.8	3.0	20.8%
Saxpy	10,016	20.3	24.9%	3.7	2.8	25.0%
QuickSort	23,388	28.1	37.1%	4.3	2.8	35.7%
Sparse	30,710	40.8	36.1%	6.5	4.3	33.8%
BubbleSort	98,680	46.36	24.3%	6.8	5.3	22.2%

gated by a $(P-1)$-cycles early version of the output ODC_B of the last register.

If the fanin cone of a multiplexer select line contains a just-in-time signal, it cannot be gated to remain unchanged when the ODC_B of the multiplexer output is *True*. This can be fixed by inserting an enabled transparent latch in the path of the just-in-time signal going to the fanin cone. If enabled latches are not present in the library, or if placing a latch causes the corresponding select line to end up in the critical path, the select lines of that multiplexer should not be gated.

Another scenario is when an input bus of a multiplexer is just-in-time or its switching activity cannot be stopped by control-signal gating. Applying control-signal gating to the multiplexer select lines would be worthwhile in a wide datapath if switching activity on at least one input bus of the multiplexer could be controlled. If none of the input buses can be controlled by control-signal gating, the select lines of that multiplexer should not be gated.

If the fanout of a bus drives a control input of a steering module, it should be disconnected for the purpose of deriving gated control signals to preserve the assumption that there are no data-dependent control signals. Such a fanout should be assumed to have zero ODC_B, and the steering module controlled by it should be treated as a computational module.

VI. EXPERIMENTAL SETUP AND RESULTS

A. Experimental Setup

The protocol processor (PP) of the MAGIC chip designed by the FLASH multiprocessor team [17] formed the basis of our experiments. The PP is a two-way superscalar RISC microprocessor, designed in a 0.5-μm CMOS technology. Control-signal gating was applied to one of the two integer execution units of PP: the BExecuteUnit, which consists of control logic and datapath. The area of the control section is 0.37 mm^2 with approximately 1.6 K transistors, and the datapath is 3.3 mm^2 with approximately 45 K transistors.

The datapath of BExecuteUnit is 64 bits wide, with three input and two output buses, seven computational modules and 11 steering modules. Gated control signals were generated for three registers and a multiplexer that drive four internal buses *IBus*, *TBus*, *SBus*, and *RBus*. The remaining seven steering modules drive the two output buses. One of the output buses drives flip-flops without enables in a state machine, and the other output bus has an ODC that depends on the value of the instruction that is not yet fetched from the instruction cache. Hence, the ODC's of both output buses were assumed to be zero.

Dynamic switching power was estimated using (13), where V_{DD} is the power-supply voltage and f is the clock frequency. For a design with N nodes, C_i is the capacitive loading and α_i is the switching activity on node i

$$P = V_{\text{DD}}^2 \times f \times \left(\sum_{i=1}^{N} C_i \times \alpha_i \right). \tag{13}$$

The power supply for our design was 3.3 V, and the clock frequency was 100 MHz. Benchmark programs were run on the structural RT-level netlist of PP. Capacitive loading on each node was extracted from the chip layout. The capacitive loading of new nodes added by control-signal gating was estimated after resynthesizing the control logic. Node toggle count, the number of clock cycles in which a node toggles during a benchmark run, was collected for each node in both control and datapath units. For each benchmark run, the switching activity (α) of a node was calculated using (14)

$$\alpha = \frac{\text{Node Toggle Count}}{\text{Total Clock Cycles in Benchmark}}. \tag{14}$$

B. Results

Table I shows results of running five integer benchmarks and a multiprocessor protocol test on PP. ProtocolProc runs a cache-coherent multiprocessor memory protocol. Sumup adds elements of an array of 100 integers. Saxpy multiplies such an array by a constant and adds that with another array. QuickSort and BubbleSort sort the elements of such arrays. Sparse adds the sum of each row of a 10×64 matrix to each element in the row. The second column shows the runtime of each program in number of clock cycles. The third column shows the occupancy of BExecuteUnit, which is the percentage of clock cycles in which BExecuteUnit is executing instructions. The benchmarks are listed in order of increasing occupancy. Higher occupancy causes higher switching activity on internal buses in the BExecuteUnit, potentially resulting in more conditions of unnecessary switching activity that can be saved by control-signal gating. The fourth column gives percentage net reduction in the total switching activity after applying control-signal gating, which accounts for reduced switching activity in the datapath and increased activity in the control logic. The next two columns give dynamic switching power dissipation in watts in the original (ungated) netlist and the (gated) netlist after applying control-

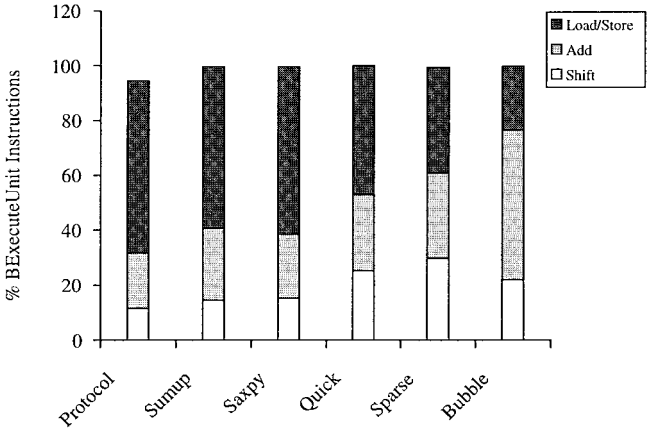


Fig. 10. Profile of instructions in BExecuteUnit.

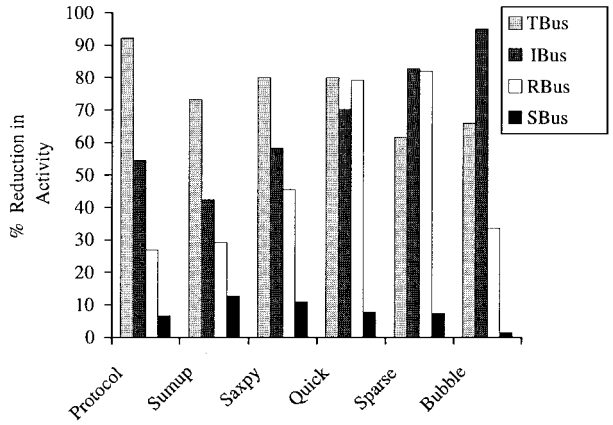


Fig. 11. Reduction in switching activity on major buses.

signal gating. The last column gives the percentage reduction in dynamic switching power. We get an average of 27.4% reduction in switching activity, resulting in 26.6% average reduction in dynamic switching power in the BExecuteUnit.

Fig. 10 shows a profile of BExecuteUnit instructions for each benchmark, classified into three major categories: Shift, Add, and Load/Store. The x-axis displays the benchmarks and the y-axis gives the occurrence of each type of instruction as a percentage of total BExecuteUnit instructions. Fig. 11 shows the percentage reduction in switching activity on each bus. Of the four buses controlled by control-signal gating, Shift instructions use *TBus*, Add instructions use *SBus*, *TBus*, and *RBus*, and Load/Store instructions use *SBus*, *IBus*, and *RBus* as operands. The total reduction in switching activity in the datapath reflects the cumulative effect of reduced switching activity on these four buses and the resulting reduction in switching activity in the datapath modules and other buses in the fanout cones of these buses. *SBus* has very little unnecessary switching activity on it because it is used by both Add and Load/Store instructions. Hence the amount of switching activity reduced on *SBus* is consistently low in all benchmarks.

ProtocolProc, Sumup, and Saxpy have the lowest BExecuteUnit occupancy, resulting in low occurrence of unnecessary switching activity on the internal buses. The majority of the unnecessary switching activity is on *TBus* only since the instruction profile of these benchmarks in Fig. 10 is dominated by Load/Store instructions. This results in a high reduction in switching activity on *TBus* in Fig. 11 but overall low power savings in Table I.

QuickSort and Sparse have a higher occupancy, and evenly distributed instruction profiles in Fig. 10. This gives many conditions of redundant switching activity, resulting in a significant reduction in switching activity on all four buses in Fig. 11. Therefore, these benchmarks give high power savings in Table I.

BubbleSort also has a high occupancy of the execute unit, but its instruction profile is strongly dominated by Add instructions in Fig. 10. This leaves very little avenue for reducing unnecessary switching activity on *SBus*, *TBus*, and *RBus*. Even though low occurrence of Load/Store instructions results in high reduction in switching activity on *IBus* in Fig. 11, the overall power saving is low in Table I.

TABLE II
AREA OVERHEAD IN CONTROL-LOGIC UNIT

	UnGated	Gated	% Overhead
Cells	217	225	3.7
Nets	162	171	5.6
Total Area	2206	2301	4.3

Table II shows the area overhead. The area was compared from synthesis area estimates before and after applying control-signal gating. Control-signal gating added only a few gates and nets, resulting in less than 5% increase in the area of the control-logic unit. Since the size of the control logic is 10% the size of the datapath, this overhead is insignificant.

Enable signals for the three-input registers had an extra NOR gate delay in their path after applying control-signal gating. After resynthesis of the control-logic unit, we were able to keep the critical path delays of these signals the same as they were before control-signal gating was applied.

VII. CONCLUSIONS

We have presented a practical method for reducing unnecessary switching activity on datapath buses. Control-signal gating offers minimal area and timing overheads and low computational cost. This technique is particularly relevant to hardware designers, providing a methodology that can be used even without a dedicated tool to take advantage of those cases of redundant switching activity that would be otherwise cumbersome to find in large datapaths. It is easily usable by a designer to reduce unnecessary power dissipation in the datapath of their unit, without having to worry about side effects such as clock skew or changes in the datapath layout. The technique was applied to one of the integer execution units of a superscalar microprocessor. Results of benchmark simulations on the microprocessor showed an average 26.6% reduction in dynamic switching power in the execution unit, a 5% increase in the area of the control logic, and no new critical paths.

ACKNOWLEDGMENT

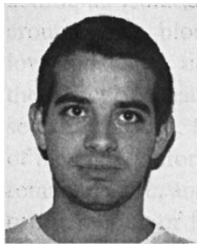
The authors would like to thank Prof. M. Horowitz for helpful discussions and suggestions and J. Bergmann, the hardware designer of PP. They also thank B. Ellersick, R. Ho, D. Ofelt, and K. DiTommaso for their help in the preparation of this manuscript.

REFERENCES

[1] M. Pedram, "Power minimization in IC design: Principles and applications," *ACM Trans. Design Automat. Electron. Syst.*, vol. 1, no. 1, pp. 3–56, 1996.
[2] T. Burd and R. Brodersen, "Processor design for portable systems," *J. VLSI Signal Process.*, vol. 13, no. 2/3, pp. 203–222, 1996.
[3] L. Benini and G. De Micheli, *Dynamic Power Management: Design Techniques and CAD Tools*. Norwell, MA: Kluwer, 1998.
[4] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez, "Reducing power in high-performance microprocessors," in *Proc. 35th Design Automation Conf.*, San Francisco, CA, June 1998, pp. 12–15.
[5] M. Gowan, L. Brio, and B. Jackson, "Power considerations in the design of the Alpha 21264 microprocessor," in *Proc. 35th Design Automation Conf.*, San Francisco, CA, June 1998, pp. 726–731.
[6] G. Gerosa, S. Gary, O. Dietz, D. Pham, K. Hoover, J. Alvarez, H. Sanchez, P. Ippolito, T. Ngo, S. Litch, J. Eno, J. Golab, N. Vanderschaaf, and J. Kahle, "A 2.2w, 80 MHz superscalar RISC microprocessor," *IEEE J. Solid-State Circuits*, vol. 29, pp. 1440–1454, Dec. 1994.
[7] G. Tellez, A. Farrahi, and M. Sarrafzadeh, "Activity-driven clock design for low power circuits," in *Proc. IEEE Int. Conf. Computer Aided Design (ICCAD)*, San Jose, CA, Nov. 1995, pp. 62–65.
[8] V. Tiwari, R. Donnelly, S. Malik, and R. Gonzalez, "Dynamic power management for microprocessors: A case study," in *Proc. 10th Int. Conf. VLSI Design*, Hydrabad, India, Jan. 1997, pp. 4–7.
[9] F. Theeuwen and E. Seelen, "Power reduction through clock gating by symbolic manipulation," in *Proc. Symp. Logic and Architecture Design*, Dec. 1996, pp. 184–191.
[10] M. Ohnishi, A. Yamada, H. Noda, and T. Kambe, "A method of redundant clocking detection and power reduction at RT-level design," in *Proc. 1997 Int. Symp. Low Power Electronics and Design*, Monterey, CA, Aug. 1997, pp. 131–136.
[11] M. Alidina, J. Monteiro, S. Devadas, A. Ghosh, and M. Papaefthymiou, "Precomputation-based sequential logic optimization for low power," in *Proc. 1994 IEEE Int. Conf. Computer Aided Design*, San Jose, CA, Nov. 1994, pp. 74–81.
[12] V. Tiwari, S. Malik, and P. Ashar, "Guarded evaluation: pushing power management to logic synthesis/design," in *Proc. Low Power Design Symp.*, Dana Point, CA, Apr. 1995, pp. 221–226.
[13] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.
[14] M. Damiani and G. De Micheli, "Don't care set specifications in combinational and synchronous logic circuits," *IEEE Trans. Computer Aided Design*, vol. 12, pp. 365–388, Mar. 1993.
[15] N. Deo, *Graph Theory with Applications to Engineering and Computer Science*. Englewood Cliffs, NJ: Prentice-Hall, 1974.
[16] A. Aho, J. Hopcroft, and J. Ullman, *Data Structures and Algorithms*. Reading, MA: Addison-Wesley, 1983.
[17] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, "The Stanford FLASH multiprocessor," in *Proc. 21st Int. Symp. Computer Architecture*, Chicago, IL, Apr. 1994, pp. 302–313.

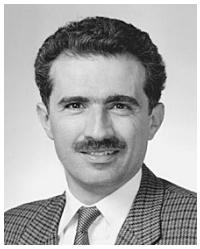
Hema Kapadia received the B.E. (Hons.) degree in electrical and electronics engineering from the Birla Institute of Technology and Science, Pilani, India, in 1992 and the M.S. degree in electrical engineering from the University of Rochester, New York, in 1993. She is currently pursuing the Ph.D. degree in the Department of Electrical Engineering, Stanford University, Stanford, CA.

After having worked on several design projects in academics and in industry, she is currently involved in computer-aided-design research to find practical solutions for some of the problems faced by designers. Her research interests are in the areas of low-power techniques for high-performance designs and bridging the gap between synthesis and layout in the standard-cell design methodology.



Luca Benini received the B.S. degree (*summa cum laude*) in electrical engineering from the University of Bologna, Bologna, Italy, in 1991 and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1994 and 1997, respectively.

Since 1998, he has been an Assistant Professor in the Department of Electronics and Computer Science at the University of Bologna. He also is a Visiting Researcher at Stanford University and at Hewlett-Packard Laboratories, Palo Alto, CA. His research interests are in all aspects of computer-aided design of digital circuits, with special emphasis on low-power applications. He has been a member of the Technical Program Committee for the Design and Test in Europe Conference and the International Symposium on Low Power Design.



Giovanni De Micheli (S'79–M'79–SM'89–F'94) is Professor of electrical engineering, and, by courtesy, of computer science at Stanford University, Stanford, CA. His research interests include several aspects of the computer-aided design of integrated circuits and systems, with particular emphasis on automated synthesis, optimization, and validation. He is author of *Synthesis and Optimization of Digital Circuits* (New York: McGraw-Hill, 1994) and coauthor of *Dynamic Power Management: Circuit Techniques and CAD Tools* (Norwell, MA: Kluwer, 1998) and of three other books. He was Codirector of the NATO Advanced Study Institutes on Hardware/Software Codesign, held in Tremezzo, Italy, in 1995 and on Logic Synthesis and Silicon Compilation, held in L'Aquila, Italy, in 1986.

Dr. De Micheli received a Presidential Young Investigator award in 1988. He received the 1987 IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS/ICAS Best Paper Award and two Best Paper Awards at the Design Automation Conference, in 1983 and in 1993. He is the Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS. He was the Program Chair (for Design Tools) of the Design Automation Conference in 1996 and 1997, and he is currently Vice General Chair. He was Program and General Chair of the International Conference on Computer Design in 1988 and 1989, respectively.