Precomputation-Based Sequential Logic Optimization for Low Power

Mazhar Alidina, José Monteiro, Srinivas Devadas, Member, IEEE, Abhijit Ghosh, Member, IEEE, and Marios Papaefthymiou

Abstract—We address the problem of optimizing logic-level sequential circuits for low power. We present a powerful sequential logic optimization method that is based on selectively precomputing the output logic values of the circuit one clock cycle before they are required, and using the precomputed values to reduce internal switching activity in the succeeding clock cycle. We present two different precomputation architectures which exploit this observation. The primary optimization step is the synthesis of the precomputation logic, which computes the output values for a subset of input conditions. If the output values can be precomputed, the original logic circuit can be “turned off” in the next clock cycle and will have substantially reduced switching activity. The size of the precomputation logic determines the power dissipation reduction, area increase and delay increase relative to the original circuit. Given a logic-level sequential circuit, we present an automatic method of synthesizing precomputation logic so as to achieve maximal reductions in power dissipation. We present experimental results on various sequential circuits. Up to 75% reductions in average switching activity and power dissipation are possible with marginal increases in circuit area and delay.

I. INTRODUCTION

AVERAGE POWER DISSIPATION has recently emerged as an important parameter in the design of general-purpose and application-specific integrated circuits. Optimization for low power can be applied at many different levels of the design hierarchy. For instance, algorithmic and architectural transformations can trade off throughput, circuit area, and power dissipation [4], and logic optimization methods have been shown to have a significant impact on the power dissipation of combinational logic circuits [12].

In CMOS circuits, the probabilistic average switching activity of the circuit is a good measure of the average power dissipation of the circuit. Average power dissipation can thus be computed by estimating the average switching activity. Several methods to estimate power dissipation for CMOS combinational circuits have been developed (e.g., [6], [10]). More recently, efficient and accurate methods of power dissipation estimation for sequential circuits have been developed [9], [13].

In this paper, we are concerned with the problem of optimizing logic-level sequential circuits for low power. Previous work in the area of sequential logic synthesis for low power has focused on state encoding [11] and retiming [8] algorithms. We present a powerful sequential logic optimization method that is based on selectively precomputing the output logic values of the circuit one clock cycle before they are required, and using the precomputed values to reduce internal switching activity in the succeeding clock cycle.

The primary optimization step is the synthesis of precomputation logic, which computes the output values for a subset of input conditions. If the output values can be precomputed, the original logic circuit can be “turned off” in the next clock cycle and will not have any switching activity. Since the savings in the power dissipation of the original circuit is offset by the power dissipated in the precomputation phase, the selection of the subset of input conditions for which the output is precomputed is critical. The precomputation logic adds to the circuit area and can also result in an increased clock period.

Given a logic-level sequential circuit, we present an automatic method of synthesizing the precomputation logic so as to achieve a maximal reduction in power dissipation. We present experimental results on various sequential circuits. For some circuits, 75% reductions in average switching activity are possible with marginal increases in circuit area and delay.

In Section II, we briefly describe our model for power dissipation. In Section III we describe two different precomputation architectures. Algorithms that synthesize precomputation logic so as to achieve power dissipation reduction are presented in Section IV. In Section V we describe a method for multiple-cycle precomputation. Experimental results are presented in Section VI. In Section VII we describe additional precomputation architectures which are the subject of ongoing research.

II. A POWER DISSIPATION MODEL

Under a simplified model, the energy dissipation of a CMOS circuit is directly related to the switching activity.
In particular, the three simplifying assumptions are:
- The only capacitance in a CMOS logic gate is at the output node of the gate.
- Either current is flowing through some path from $V_{DD}$ to the output capacitor, or current is flowing from the output capacitor to ground.
- Any change in a logic-gate output voltage is a change from $V_{DD}$ to ground or vice versa.

All of these are reasonably accurate assumptions for well-designed CMOS gates [7], and when combined imply that the energy dissipated by a CMOS logic gate each time its output changes is roughly equal to the change in energy stored in the gate’s output capacitance. If the gate is part of a synchronous digital system controlled by a global clock, it follows that the average power dissipated by the gate is given by:

$$P_{avg} = 0.5 \times C_{load} \times \left( \frac{V_{dd}^2}{T_{cycle}} \right) \times E(\text{transitions})$$

where $P_{avg}$ denotes the average power, $C_{load}$ is the load capacitance, $V_{dd}$ is the supply voltage, $T_{cycle}$ is the global clock period, and $E(\text{transitions})$ is the expected value of the number of gate output transitions per global clock cycle [10], or equivalently the average number of gate output transitions per clock cycle. All of the parameters in (1) can be determined from technology or circuit layout information except $E(\text{transitions})$, which depends on both the logic function being performed and the statistical properties of the primary inputs.

Equation (1) is used by the power estimation techniques such as [6], [10] to relate switching activity to power dissipation.

III. PRECOMPUTATION ARCHITECTURES

We describe two different precomputation architectures and discuss their characteristics in terms of their impact on power dissipation, circuit area and circuit delay.

A. First Precomputation Architecture

Consider the circuit of Fig. 1. We have a combinational logic block $A$ that is separated by registers $R_1$ and $R_2$. While $R_1$ and $R_2$ are shown as distinct registers in Fig. 1 they could, in fact, be the same register. We will first assume that block $A$ has a single output and that it implements the Boolean function $f$.

In Fig. 2 the first precomputation architecture is shown. Two Boolean functions $g_1$ and $g_2$ are the predictor functions. We require:

$$g_1 = 1 \Rightarrow f = 1$$

$$g_2 = 1 \Rightarrow f = 0$$

Therefore, during clock cycle $t$ if either $g_1$ or $g_2$ evaluates to a 1, we set the load enable signal of the register $R_1$ to be 0. This means that in clock cycle $t+1$ the inputs to the combinational logic block $A$ do not change. If $g_1$ evaluates to a 1 in clock cycle $t$, the input to register $R_2$ is a 1 in clock cycle $t+1$, and if $g_2$ evaluates to a 1, then the input to register $R_2$ is a 0. Note that $g_1$ and $g_2$ cannot both be 1 during the same clock cycle due to the conditions imposed by (2) and (3).

A power reduction in block $A$ is obtained because for a subset of input conditions corresponding to $g_1 + g_2$ the inputs to $A$ do not change implying zero switching activity. However, the area of the circuit has increased due to additional logic corresponding to $g_1 + g_2$, the two additional gates shown in the figure, and the two flip-flops marked FF. The delay between $R_1$ and $R_2$ has increased due to the addition of the AND-OR gate. Note also that $g_1$ and $g_2$ add to the delay of paths that originally ended at $R_1$ but now pass through $g_1$ or $g_2$ and the NOR gate before ending at the load enable signal of the register $R_1$. Therefore, we would like to apply this transformation on noncritical logic blocks.

The choice of $g_1$ and $g_2$ is critical. We wish to include as many input conditions as we can in $g_1$ and $g_2$. In other words, we wish to maximize the probability of $g_1$ or $g_2$ evaluating to a 1. In the extreme case this probability can be made unity if $g_1 = f$ and $g_2 = f$. However, this would imply a duplication of the logic block $A$ and no reduction in power with a twofold increase in area! To obtain reduction in power with marginal increases in circuit area and delay, $g_1$ and $g_2$ have to be significantly less complex than $f$. One way of ensuring this is to make $g_1$ and $g_2$ depend on significantly fewer inputs than $f$. This leads us to the second precomputation architecture of Fig. 3.
B. Second Precomputation Architecture

In the architecture of Fig. 3, the inputs to the block A have been partitioned into two sets, corresponding to the registers \( R_1 \) and \( R_2 \). The output of the logic block A feeds the register \( R_3 \). The functions \( g_1 \) and \( g_2 \) satisfy the conditions of (2) and (3) as before, but \( g_1 \) and \( g_2 \) only depend on a subset of the inputs to \( f \). If \( g_1 \) or \( g_2 \) evaluates to a 1 during clock cycle \( t \), the load enable signal to the register \( R_2 \) is turned off. This implies that the outputs of \( R_2 \) during clock cycle \( t+1 \) do not change. However, since the outputs of register \( R_1 \) are updated, the function \( f \) will evaluate to the correct logical value. A power reduction is achieved because only a subset of the inputs to block A change implying reduced switching activity.

As before, \( g_1 \) and \( g_2 \) have to be significantly less complex than \( f \) and the probability of \( g_1 + g_2 \) begin a 1 should be high in order to achieve substantial power gains. The delay of the circuit between \( R_3/R_2 \) and \( R_1 \) unchanged, allowing precomputation of logic that is on the critical path. However, the delay of paths that originally ended at \( R_1 \) has increased.

The choice of inputs to \( g_1 \) and \( g_2 \) has to be made first, and then the particular functions that satisfy (2) and (3) have to be selected. Methods to perform this selection for this second precomputation architecture are described in Section IV.

C. Precomputation for Finite State Machines

As mentioned in Section III-A, these precomputation architectures are not restricted to pipeline circuits. We present in Fig. 4 an example of precomputation for a finite state machine using this second precomputation architecture.

D. An Example

We give an example that illustrates the fact that substantial power gains can be achieved with marginal increases in circuit area and delay. The circuit we are considering is a \( n \)-bit comparator that compares two \( n \)-bit numbers \( C \) and \( D \) and computes the function \( C > D \). The optimized circuit with precomputation logic is shown in Fig. 5. The precomputation logic is as follows:

\[
\begin{align*}
g_1 &= C(n-1) \cdot \overline{D(n-1)} \\
g_2 &= C(n-1) \cdot D(n-1)
\end{align*}
\]

Clearly, when \( g_1 = 1 \), \( C \) is greater than \( D \), and when \( g_2 = 1 \), \( C \) is less than \( D \). We have to implement

\[
g_1 + g_2 = C(n-1) \cdot \overline{D(n-1)}
\]

where \( \otimes \) stands for the exclusive-nor operator.

Assuming a uniform probability for the inputs\(^1\), the probability that the XNOR gate evaluates to a 1 is 0.5, regardless of \( n \). For large \( n \), we can neglect the power dissipation in the XNOR gate, and therefore, we can achieve a power reduction of close to 50%. The reduction will depend upon the relative power dissipated by the vector pairs with \( C(n-1) \otimes D(n-1) = 1 \) and the vector pairs with \( C(n-1) \otimes D(n-1) = 0 \). If we add the inputs \( C(n-2) \) and \( D(n-2) \) to \( g_1 \) and \( g_2 \) it is possible to achieve a power reduction close to 75%.

IV. SYNTHESIS OF PRECOMPUTATION LOGIC

A. Introduction

In this section, we will describe methods to determine the functionality of the precomputation logic, and then describe methods to efficiently implement the logic.

We will focus on the second precomputation architecture (Section III-B) illustrated in Fig. 3. In order to ensure that the precomputation logic is significantly less complex than the combinational logic in the original circuit, we will restrict ourselves to identifying \( g_1 \) and \( g_2 \) such that they depend on a relatively small subset of the inputs to the logic block A.

B. Precomputation and Observability Don’t-Cares

Assume that we have a logic function \( f(X) \), with \( X = \{x_1, \ldots, x_n\} \), corresponding to block A of Fig. 1. Given that the logic function implemented by block A is \( f \), then the observability don’t-care set for input \( x_i \) is given by:

\[
\text{ODC}_i = f_{x_i} \cdot \overline{f_{x_i}} + \overline{f_{x_i}} \cdot f_{x_i}
\]

\( ^1 \)The assumption here is that each \( C(i) \) and \( D(i) \) has a 0.5 static probability of being a 0 or a 1.
where $f_x$ and $f_{\bar{x}}$ are the cofactors of $f$ with respect to $x_i$, and similarly for $\bar{f}$. (The cofactors can be obtained simply by setting $x_i$ to a 1 or 0 in $f$ or $\bar{f}$.)

If we determine that a given input combination is in ODC, then we can disable the loading of $x_i$ into the register since that means that we do not need the value of $x_i$ in order to know what the value of $f$ is. If we wish to disable the loading of registers $x_{m+1}, \ldots, x_n$, we will have to implement the function

$$g = \prod_{i=m+1}^{n} \text{ODC}_i$$

and use $g$ as the load enable signal for the registers corresponding to $x_{m+1}, \ldots, x_n$.

For more details regarding observability don’t-cares the reader is referred to [5] (p. 179).

C. Precomputation Logic

Let us now consider the architecture of Fig. 3. Assume that the inputs $x_1, \ldots, x_m, x_{m+1}, \ldots, x_n$, with $m < n$ have been selected as the variables that $g_1$ and $g_2$ depend on. We have to find $g_1$ and $g_2$ such that they satisfy the constraints of (2) and (3), respectively, and such that $\text{prob}(g_1 + g_2 = 1)$ is maximum.

We can determine $g_1$ and $g_2$ using universal quantification on $f$. The universal quantification of a function $f$ with respect to a variable $x_i$ is defined as:

$$U_{x_i} f = f_{x_i} \cdot f_{\bar{x}_i}$$

This gives all the combinations over the inputs $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$, that result in $f = 1$ such that the value of $x_i$ does not matter.

Given a subset of inputs $S = \{x_1, \ldots, x_m\}$, set $D = X - S$.

We can define:

$$U_D f = U_{x_{m+1}} \cdots U_{x_n} f$$

Theorem 4.1: $g_1 = U_D f$ satisfies (2). Further, no function $h(x_1, \ldots, x_m)$ exists such that $\text{prob}(h = 1) > \text{prob}(g_1 = 1)$ and such that $h = 1 \Rightarrow f = 1$.

Proof: By construction, if for some input combination $g_1, \ldots, g_m$ causes $g_1(a_1, \ldots, a_m) = 1$, then for that combination of $x_1, \ldots, x_m$ and all possible combinations of variables in $x_{m+1}, \ldots, x_n$, $f(a_1, \ldots, a_m, x_{m+1}, \ldots, x_n) = 1$.

We cannot add any minterm over $x_1, \ldots, x_m$ to $g_1$ because for any minterm that is added, there will be some combination of $x_{m+1}, \ldots, x_n$ for which $f(x_1, \ldots, x_n)$ will evaluate to 1. Therefore, we cannot find any function $h$ that satisfies (2) and such that $\text{prob}(h = 1) > \text{prob}(g_1 = 1)$.

Similarly, given a subset of inputs $S$, we can obtain a maximal $g_2$ by:

$$g_2 = U_D f = U_{x_{m+1}} \cdots U_{x_n} f$$

We can compute the functionality of the precomputation logic as $g_1 + g_2$.

Selecting a Subset of Inputs: Exact Method: Given a function $f$ we wish to select the “best” subset of inputs $S$ of cardinality $k$. Given $S$, we have $D = X - S$ and we compute $g_1 = U_D f$, $g_2 = U_D f$. In the sequel, we assume that the best set of inputs corresponds to the inputs which result in $\text{prob}(g_1 + g_2 = 1)$ being maximum for a given $k$. We know that $\text{prob}(g_1 + g_2 = 1) = \text{prob}(g_1 = 1) + \text{prob}(g_2 = 1)$ since $g_1$ and $g_2$ cannot both be 1 on the same input vector. The above cost function ignores the power dissipatated in the precomputation logic, but since the number of inputs to the precomputation logic is significantly smaller than the total number of inputs this is a good approximation.

In the sequel we describe a branching algorithm that determines the optimal set of inputs maximizing the probability of the $g_1$ and $g_2$ functions. This algorithm is shown in pseudocode in Fig. 6.

The procedure SELECT_INPUTS receives as arguments the function $f$ and the desired number of inputs $k$ to the precomputation logic. SELECT_INPUTS calls the recursive procedure SELECT_RECUR with five arguments. The first two arguments correspond to the $g_1$ and $g_2$ functions, which are initially $f$ and $\bar{f}$. A variable is selected within the recursive procedure and the two functions are universally quantified with respect to the selected variable. The third argument $D$ corresponds to the set of variables that $g_1$ and $g_2$ do not depend on. The fourth argument $Q$ corresponds to the set of “active” variables, which may be selected or discarded. Finally, the argument $l$ corresponds to the number of variables that have to be universally quantified in order to obtain $g_1$ and $g_2$ with $k$ or fewer inputs.

If $|D| + |Q| < l$ it means that we have selected too many variables in the earlier recursions and we will not be able to quantify with respect to enough input variables. The functions $g_1$ and $g_2$ will depend on too many variables ($> k$).
We calculate the probability of $g_1 + g_2$. If this probability is less than the maximum probability we have encountered thus far, we can immediately return because of the following invariant which is true because $f$ contains $U_x, f$.

$$\text{prob}(U_x, f) = \text{prob}(f_x, f_x) \leq \text{prob}(f) \quad \forall x, f$$

Therefore as we universally quantify variables from a given $f_x$ and $f_x$ function pair the $pr$ quantity monotonically decreases.

We store the selected set corresponding to the maximum encountered probability.

Selecting a Subset of Inputs: Approximate Method: The worst-case running time of the exact method is exponential in the number of input variables and, although we have a nice pruning condition, there are many examples for which we cannot use it. Thus we have also implemented an approximate algorithm that looks at each input individually and chooses the $k$ most promising inputs.

For each input we calculate:

$$p_i = \text{prob}(U_x, f) + \text{prob}(U_x, \overline{f})$$

$p_i$ is the probability that we know the value of $f$ without knowing the value of $x_i$. If $p_i$ is high then most of the time we do not need $x_i$ to compute $f$. Therefore we select the $k$ inputs corresponding to smaller values of $p_i$.

Implementing the Logic: The Boolean operations of OR and universal quantification required in the input selection procedure can be carried out efficiently using reduced, ordered Binary Decision Diagrams (ROBDDs) [3]. We obtain a ROBDD for the $g_1 + g_2$ function. A ROBDD can be converted into a multiplexor-based network (see [1]) or into a sum-of-products cover. The network or cover can be optimized using standard combinational logic optimization methods that reduce area [2] or those that target low power dissipation [12].

D. Multiple-Output Functions

In general, we have a multiple-output function $f_1, \ldots, f_m$ that corresponds to the logic block $A$ in Fig. 1. All the procedures described thus far can be generalized to the multiple-output case.

The functions $g_{i1}$ and $g_{i2}$ are obtained using the equations below.

$$g_{i1} = U_P f_i$$
$$g_{i2} = U_P \overline{f_i}$$

where $D = X - S$ as before. The function $g$ whose complement drives the load enable signal is obtained as:

$$g = \prod_{i=1}^{m} (g_{i1} + g_{i2})$$

The function $g$ corresponds to the set of input conditions where the variables in $S$ control the values of all the $f_i$’s regardless of the values of variables in $D = X - S$.

Selecting a Subset of Outputs: Exact Method: We describe an algorithm, which given a multiple-output function, selects a subset of outputs and a subset of inputs so as to maximize a given cost function that is dependent on the probability of the precomputation logic and the number of selected outputs. This algorithm is described in pseudocode in Fig. 7.

The inputs to procedure SELECT.OUTPUTS are the multiple-output function $F$, and a number $k$ corresponding to the number of inputs to the precomputation logic.

The procedure SELECT.ORECUR receives as inputs two sets $G$ and $H$, which correspond to the current set of outputs that have been selected and the set of outputs which can be added to the selected set, respectively. Initially, $G = \phi$ and $H = F$. The cost of a particular selection of outputs, namely $G$, is given by $prG \times gates(F - H)/total\ gates$, where $prG$ corresponds to the signal probability of the precomputation logic, $gates(F - H)$ corresponds to the number of gates in the logic corresponding to the outputs in $G$ and not shared by any output in $H$, and total.gates corresponds to the total number of gates in the network (across all outputs of $F$).

There are two pruning conditions that are checked for in the procedure SELECT.ORECUR. The first corresponds to assuming that all the outputs in $H$ can be added to $G$ without decreasing the probability of the precomputation logic. This is a valid condition because the quantity $proldG$ in each recursive call can only decrease with the addition of outputs of $G$. We then set a lower bound on the probability of the precomputation logic prior to calling the input selection procedure. Optimistically assuming that all the outputs in $H$ can be added to $G$ without lowering the precomputation logic probability, we are not interested in a precomputation logic probability for $G$ that would result in a cost that is equal to or lower than BEST_COST.

Logic Duplication: Since we are only precomputing a subset of outputs, we may incorrectly evaluate the outputs that we are not precomputing as we disable certain inputs during particular clock cycles. If an output that is not being precom-
SELECT_OUTPUTS_APPROX( F = \{ f_1, \ldots, f_m \}, k ):
{ 
  BEST_COST = 0 ;
  foreach x_i \in X \{ /* Output selection */
    foreach f_j \in F \{ 
      g_j = U_{x_i} f_j + U_{x_i} \overline{f_j} ;
    \}
  foreach f_j \in F \{ 
    G = \{ f_j \} ;
    H = F - \{ f_j \} ;
    probG = prob(g_j) ;
    curr.cost = probG \times \text{gates}(F - H) / \text{total.gates} ;
  /* Add any outputs that make the cost increase */
  g = g_j ;
  foreach f_j \in F \{ 
    G = G \cup \{ f_j \} ;
    probG = prob(g \cdot g_j) ;
    cost = \text{probG} \times \text{gates}(F - H) / \text{total.gates} ;
    if( cost > curr.cost ) 
    { curr.cost = cost ;
    g = g \cdot g_j ;
    } else 
    G = G - \{ f_j \} ;
  } 
  if( curr.cost > BEST_COST ) 
  { 
    BEST_COST = curr.cost ;
    SEL.OP.SET = G ;
  } 
} 
foreach x_i \in X \{ /* Input selection */
  g = 1 ;
  foreach f_j \in SEL.OP.SET 
    g = g \cdot ( U_{x_i} f_j + U_{x_i} \overline{f_j} ) ;
  p_i = prob(g) ;
  select the k x_i's corresponding to smaller p_i's
}

In this algorithm we first select the set of outputs that will be precomputed and then select the inputs that we are going to precompute those outputs with. When we are selecting the outputs we still do not know which inputs are going to be selected, thus we select those outputs that seem to be the most precomputable. Universally quantifying just one of the inputs, we start with one output and compute the same cost function as in the first method, \text{probG} \times \text{gates}(F - H) / \text{total.gates}. Then we add outputs that make the cost function increase. We repeat this process for each input. At the end we keep the set of outputs corresponding to the maximum cost.

Once we have a set of promising outputs to precompute we can use the approximate algorithm described in Section IV-C-2 to select the inputs. This algorithm runs in polynomial time in the number outputs times the number of inputs.

V. MULTIPLE CYCLE PRECOMPUTATION

A. Basic Strategy

It is possible to precompute output values that are not required in the succeeding clock cycle, but required 2 or more clock cycles later.
Consider the topology of Fig. 10. If the register outputs of \( R_3 \) are not used except to compute \( f \), then we can precompute the value of the function \( f \) using a selected set of inputs, namely those corresponding to register \( R_1 \). If \( f \) can be precomputed to a 1 or a 0 for a set of input conditions, then for these inputs we can turn off the load enable signal to \( R_2 \). This will reduce switching activity not only in logic block \( A \), but also in logic block \( B \), because there will be reduced switching activity at the outputs of \( R_3 \) in the clock cycle following the one where the outputs of \( R_2 \) do not change.

**B. Examples**

We give examples illustrating multiple-cycle precomputation.

Consider the circuit of Fig. 11. The function \( f \) computes \( (C + D) > (X + Y) \) in two clock cycles\(^2\). Attempting to precompute \( C + D \) or \( X + Y \) using the methods of the previous section do not result in any savings because there are too many outputs to consider. However, 2-cycle precomputation can reduce switching activity by close to 12.5% if the functions below are used.

\[
g_1 = C(n-1) \cdot D(n-1) \cdot X(n-1) \cdot Y(n-1)
\]

\[
g_2 = C(n-1) \cdot D(n-1) \cdot X(n-1) \cdot Y(n-1)
\]

where \( g_1 \) and \( g_2 \) satisfy the constraints of (2) and (3), respectively. Since \( \text{prob}(g_1 + g_2) = \frac{15}{16} = 0.125 \), we can disable the loading of registers \( C(n-2:0), D(n-2:0), X(n-2:0), \) and \( Y(n-2:0) \) 12.5% of the time, which results in switching activity reduction. This percentage can be increased to over 45% by using \( C(n-2) \) through \( Y(n-2) \). We can additionally use single-cycle precomputation logic (as illustrated in Fig. 5) to further reduce switching activity in the \( > \) comparator of Fig. 11.

Next, consider the circuit of Fig. 12. The multiple-output function \( f \) computes \( \text{MAX}(C + D, X + Y) \) in two clock cycles. We can use exactly the same \( g_1 \) and \( g_2 \) functions as those immediately above, but \( g_1 \) is used to disable the loading of registers \( X(n-2:0) \) and \( Y(n-2:0) \), and \( g_2 \) is used to disable the loading of \( C(n-2:0) \) and \( D(n-2:0) \). We exploit the fact that if we know that \( C + D > X + Y \), there is no need to compute \( X + Y \), and vice versa. Finally, we can implement the \( \text{MAX} \) function as shown in Fig. 13, duplicate registers and use single-cycle precomputation on the \( > \) operator (as illustrated in Fig. 5) to achieve switching activity reduction.

\(^2\) + in the figure stands for addition.

**VI. EXPERIMENTAL RESULTS**

We first present results on datapath circuits such as carry-select adders, comparators, and interconnections of adders and comparators in Table I. In all examples the precomputation architecture of Fig. 3 was used and all the outputs of each circuit were precomputed. For each circuit, the number of
TABLE I

<table>
<thead>
<tr>
<th>CKT</th>
<th>Original</th>
<th>Precompute logic</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>literals</td>
<td>Levels</td>
<td>Power</td>
<td>Bins</td>
</tr>
<tr>
<td>comp16</td>
<td>286</td>
<td>7</td>
<td>1283</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>682</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>12</td>
<td>550</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>16</td>
<td>516</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>20</td>
<td>526</td>
</tr>
<tr>
<td>add.comp16</td>
<td>3026</td>
<td>8</td>
<td>6941</td>
</tr>
<tr>
<td></td>
<td>4/8</td>
<td>24</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8/0</td>
<td>51</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8/8</td>
<td>67</td>
<td>6</td>
</tr>
<tr>
<td>max16</td>
<td>350</td>
<td>9</td>
<td>1744</td>
</tr>
<tr>
<td>add.max16</td>
<td>3200</td>
<td>9</td>
<td>7270</td>
</tr>
<tr>
<td></td>
<td>4/8</td>
<td>24</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8/0</td>
<td>51</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8/8</td>
<td>67</td>
<td>6</td>
</tr>
<tr>
<td>csa16</td>
<td>975</td>
<td>10</td>
<td>2045</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>18</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>25</td>
<td>5</td>
</tr>
</tbody>
</table>

TABLE II

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Original</th>
<th>Precompute logic</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>literals</td>
<td>Levels</td>
<td>Power</td>
<td>Bins</td>
</tr>
<tr>
<td>s挹進</td>
<td>28</td>
<td>3</td>
<td>258</td>
</tr>
<tr>
<td></td>
<td>47</td>
<td>36</td>
<td>157</td>
</tr>
<tr>
<td>#282*</td>
<td>8</td>
<td>8</td>
<td>31</td>
</tr>
<tr>
<td>#219*</td>
<td>21</td>
<td>1</td>
<td>61</td>
</tr>
<tr>
<td>#216*</td>
<td>16</td>
<td>4</td>
<td>02</td>
</tr>
<tr>
<td>#18</td>
<td>12</td>
<td>2</td>
<td>196</td>
</tr>
<tr>
<td>#25*</td>
<td>22</td>
<td>2</td>
<td>194</td>
</tr>
<tr>
<td>#25</td>
<td>22</td>
<td>2</td>
<td>194</td>
</tr>
<tr>
<td>#52</td>
<td>10</td>
<td>10</td>
<td>133</td>
</tr>
<tr>
<td>#32</td>
<td>75</td>
<td>36</td>
<td>1067</td>
</tr>
<tr>
<td>#222*</td>
<td>22</td>
<td>22</td>
<td>445</td>
</tr>
<tr>
<td>#55</td>
<td>65</td>
<td>55</td>
<td>250</td>
</tr>
<tr>
<td>#203</td>
<td>1</td>
<td>220</td>
<td>3</td>
</tr>
<tr>
<td>#203</td>
<td>1</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>#186</td>
<td>25</td>
<td>18</td>
<td>113</td>
</tr>
<tr>
<td>#25</td>
<td>25</td>
<td>18</td>
<td>113</td>
</tr>
<tr>
<td>#24</td>
<td>24</td>
<td>1</td>
<td>54</td>
</tr>
<tr>
<td>#52</td>
<td>19</td>
<td>9</td>
<td>717</td>
</tr>
<tr>
<td>#25</td>
<td>27</td>
<td>27</td>
<td>95</td>
</tr>
<tr>
<td>#203</td>
<td>10</td>
<td>4</td>
<td>270</td>
</tr>
<tr>
<td>#212</td>
<td>42</td>
<td>39</td>
<td>1794</td>
</tr>
<tr>
<td>s挹進</td>
<td>16</td>
<td>16</td>
<td>634</td>
</tr>
<tr>
<td>s挹進</td>
<td>10</td>
<td>10</td>
<td>635</td>
</tr>
<tr>
<td>s挹進</td>
<td>38</td>
<td>3</td>
<td>491</td>
</tr>
<tr>
<td>s挹進</td>
<td>36</td>
<td>16</td>
<td>144</td>
</tr>
</tbody>
</table>

*Precompute logic calculated using the exact algorithm.

The number of literals in the precomputation logic and power is 3% on the average. The extra delay incurred is proportional to the number of levels in the precomputation logic and is quite small in most cases. It should be noted that it may be possible to use the other precomputation architectures for all of the examples presented here. Some of these examples are perhaps better suited to other architectures than the one we used to derive the results, and therefore larger savings in power may be possible. Secondly, the inputs and outputs to be selected and the precomputation logic are determined automatically, making this approach suitable for automatic logic synthesis systems. Finally, the significant power savings obtained for random logic circuits indicate that this approach is not restricted only to datapath circuits.

VII. OTHER PRECOMPUTATION ARCHITECTURES

In this section, we describe additional precomputation architectures. We first present an architecture that is applicable to all logic circuits and does not require, for instance, that the inputs should be in the observability don’t-care set in order to be disabled, which was the case for the architectures shown in Section III. We also extend precomputation so that it can be used in combinational logic circuits.

A. Multiplexer-Based Precomputation

All logic functions can be written in a Shannon expansion. For the function $f$ with inputs $X = \{x_1, \ldots, x_n\}$ we can
write:

\[ f = x_1 \cdot f_{x_1} + \overline{x_1} \cdot f_{\overline{x}_1} \]  \hspace{1cm} (4)

where \( f_{x_1} \) and \( f_{\overline{x}_1} \) are the cofactors of \( f \) with respect to \( x_1 \).

Fig. 14 shows an architecture based on (4). We implement the functions \( f_{x_1} \) and \( f_{\overline{x}_1} \). Depending on the value of \( x_1 \), only one of the cofactors is computed while the other is disabled by setting the load-enable signal of its input register. The input \( x_1 \) drives the select line of a multiplexer which chooses the correct cofactor.

The main advantage of this architecture is that it applies to all logic functions. The input \( x_1 \) in the example was chosen for the purpose illustration. In fact, any input \( x_1, \ldots, x_n \) could have been selected. Unlike the architectures described earlier, we do not require that the inputs being disabled should be don't-cares for the input conditions which we are precomputing. In other words, the inputs being disabled do not have to be in the observability don’t-care set. A disadvantage of this architecture is that we need to duplicate the registers for the inputs not being used to turn off part of the logic. On the other hand, no precomputation logic functions have been added to the circuit.

The algorithm to select the best input for this architecture is also quite different. We will not discuss this algorithm in detail, except to mention that in this case, we are interested in finding the input that yields the most area efficient \( f_{x_1} \) and \( f_{\overline{x}_1} \) functions.

B. Combinational Logic Precomputation

The architectures described so far apply only to sequential circuits. We now describe precomputation of combinational circuits.

Suppose we have some combinational logic function \( f \) composed of two subfunctions \( A \) and \( B \) as shown in Fig. 15(a). Suppose we also want to precompute this function with the inputs \( x_4 \) and \( x_5 \). Fig. 15(b) shows how this can be accomplished. For simplicity, pass transistors are shown, however, we have several choices as to what to use within the dotted circles instead of the pass transistors.

Transmission Gates: Assume that transmission gates are used in place of the pass transistors in Fig. 15(b). The function \( g \) with inputs \( x_4 \) and \( x_5 \) drives the transmission gates. As in the previous architectures, \( g = g_1 + g_2 \). Hence, when \( g \) is a 0, the transmission gates are turned off and the new values of logic block \( A \) are prevented from propagating into logic block \( B \). The inputs \( x_4 \) and \( x_5 \) are also inputs to the logic block \( B \) just as in the original network in order to ensure that the output is set correctly.

For the combinational architecture, there is an implied delay constraint, i.e., the transmission gates should be off before the new values of \( A \) are computed. In the example shown, the worst-case delay of the \( g \) block plus the arrival time of inputs \( x_4 \) or \( x_5 \) should be less than the worst-case delay of logic block \( A \) plus the arrival time of the inputs \( x_1, x_2, \) or \( x_3 \). The arrival time of an input is defined as the time at which the input settles to its steady state value [5]. If the delay constraint is not met, then it may be necessary to delay the \( x_1, x_2 \) and \( x_3 \) inputs with respect to the \( x_4 \) and \( x_5 \) inputs in order to get the switching activity reduction in logic block \( B \).

Transparent Latches: A violation of the delay constraint described immediately above can result in nodes in the circuit being stuck at metastable states (halfway between the supply voltages) causing excessive power dissipation. In order to ensure that this does not occur, transparent latches can be used instead of transmission gates. This results in increased overhead for precomputation. Note that a violation of the delay constraint may cause glitching in the circuit, but the nodes will settle to the supply voltages.

AND Gates: One can also replace the pass transistor with an AND gate. This will reduce switching activity, though not
as much as in the transparent latch case. This is because \( g \) may make a \( 0 \rightarrow 1 \) transition during a clock cycle, possible causing unnecessary \( 1 \rightarrow 0 \) transitions at the outputs of the AND gates. This option works best for parachorge logic.

VIII. CONCLUSION AND ONGOING WORK

We have presented a method of precomputing the output response of a sequential circuit one clock cycle before the output is required, and exploited this knowledge to reduce power dissipation in the succeeding clock cycle. Several different architectures that utilize precomputation logic were presented.

In a finite state machine there is typically a single register, whose inputs are combinational functions of the register outputs. The precomputation architectures make no assumptions regarding feedback. For instance, \( R_1 \) and \( R_2 \) in Fig. 2 can be the same register.

Precomputation increases circuit area and can adversely impact circuit performance. In order to keep area and delay increases small, it is best to synthesize precomputation logic which depends on a small set of inputs. Precomputation works best when there are a small number of complex functions corresponding to the logic block \( A \) of Figs. 2 and 3. If the logic block has a large number of outputs, then it may be worthwhile to selectively apply precomputation-based power optimization to a small number of complex outputs. This selective partitioning will entail a duplication of combinational logic and registers, and the savings in power is offset by this duplication.

Other precomputation architectures are being explored, including the architectures of Section VII, and those that rely on a history of previous input vectors. More work is required in the automation of a logic design methodology that exploits multiplexor-based, combinational and multiple-cycle precomputation.

ACKNOWLEDGMENT

The authors would like to thank A. Chandrakasan for providing us with information regarding power dissipation in registers and P. Vanbekbergen for pointing out that transparent latches should be used in Fig. 15(b).

REFERENCES


Mazhar Aldina received the B.S. degree from Lehigh University in 1992 and the S.M. degree from the Massachusetts Institute of Technology in 1994, both in electrical engineering. He is currently a member of the Technical Staff with the Signal Processing and Integrated Circuit Design Group of AT&T Bell Laboratories. His research interests are in low power design, CAD for low power, and VLSI design.

José Monteiro was born in Lisbon, Portugal. He received the Engineer's and Master's degrees in electrical and computer engineering in 1989 and 1992, respectively, from Instituto Superior Técnico at the Technical University of Lisbon. He is currently working on the Ph.D. degree at the Massachusetts Institute of Technology in the area of power estimation and synthesis for low power of VLSI circuits.

Srinivas Devadas received the B. Tech degree in electrical engineering from the Indian Institute of Technology, Madras in 1985 and the M.S. and Ph.D. degrees in electrical engineering from the University of California, Berkeley, in 1986 and 1988, respectively. Since August 1988, he has been at the Massachusetts Institute of Technology, Cambridge, and is currently an Associate Professor of Electrical Engineering and Computer Science. His research interests span all aspects of synthesis of VLSI circuits, with emphasis on optimization techniques for synthesis at the logic, layout and architectural levels, testing of VLSI circuits, formal verification, hardware/software co-design, design-for-testability methods and interactions between synthesis and testability of VLSI systems. Dr. Devadas was the Analog Devices Career Development Chair of Electrical Engineering from 1989 to 1991. He has received five Best Paper awards at CAD conferences and journals, including the 1990 IEEE TRANSACTIONS ON CIRCUITS AND DEVICES Best Paper award. In 1992, he received a NSF Young Investigator Award. He has served on the technical program committees of several conferences and workshops including the International Conference on Computer Design, and the International Conference on Computer-Aided Design. He is a member of the ACM.
Abhijit Ghosh (M'91) received the B.Tech degree in electrical and electronics engineering from the Indian Institute of Technology, Kharagpur, in 1986. He received the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California at Berkeley in 1988 and 1991, respectively.

Since 1991, he has been a Senior Engineer at Mitsubishi Electric Research Laboratories, Inc., Sunnyvale, CA, conducting research in CAD for VLSI and system design. His research interests include all aspects of CAD for VLSI with special emphasis on logic synthesis, formal verification, testing, parallel processing, fault-tolerant computing, compiler optimization, low power design and synthesis, and electronic system design automation.

Dr. Ghosh received the Best Paper award at the 27th IEEE Design Automation Conference, 1990. He is a member of ACM.

Marino Papadimitriou received the B.S. degree in electrical engineering from the California Institute of Technology in 1988 and the S.M. and Ph.D. degrees in computer science from the Massachusetts Institute of Technology in 1990 and 1993, respectively.

Currently, he is an Assistant Professor of Electrical Engineering and Computer Science at Yale University, New Haven, CT. His research interests include algorithms, parallel computation, and VLSI design.