Precomputation-Based Sequential Logic Optimization for Low Power

Mazhar Alidina*, José Monteiro, Srinivas Devadas
Department of EECS
MIT, Cambridge, MA

Abhijit Ghosh
MERL
Sunnyvale, CA

Marios Papaefthymiou
Department of EE
Yale University, CT

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.

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. Upto 75% reductions in average switching activity and power dissipation are possible with marginal increases in circuit area and delay.

1 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 [5], 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 a 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., [7, 10]). More recently, efficient and accurate methods of power dissipation estimation for sequential circuits have been developed [9, 13].

In this work, 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 (e.g., [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 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 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 switching activity. We present experimental results on various sequential circuits. For some circuits, 75% reductions in average power dissipation are possible with marginal increases in circuit area and delay.

The model we use to relate switching activity to power dissipation can be found in [7]. In Section 2 we describe two different precomputation architectures. An algorithm that synthesizes precomputation logic so as to achieve power dissipation reduction is presented in Section 3. In Section 4 we describe a method for multiple-cycle precomputation. In Section 5 we describe additional precomputation architectures which are the subject of ongoing research. Experimental results are presented in Section 6.

2 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.

2.1 First Precomputation Architecture

Consider the circuit of Figure 1. We have a combinational logic block **A** that is separated by registers R_1

*Currently at AT&T Bell Laboratories, Allentown, PA

Permission to copy without fee all or part of this material is granted, provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

©1994 ACM 0-89791-690-5/94/0011/0074 $3.50

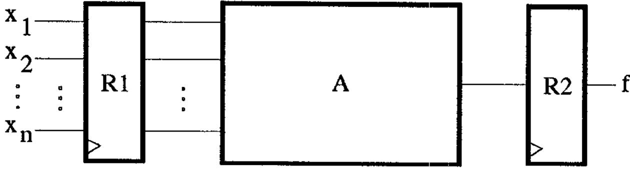

Figure 1: Original Circuit

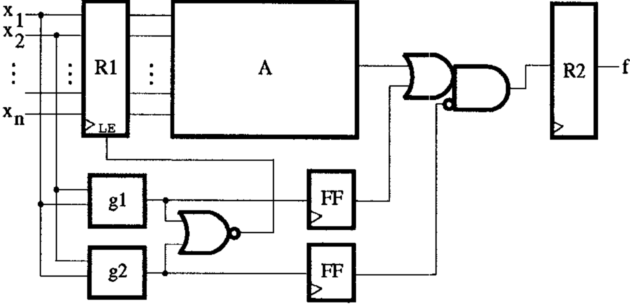

Figure 2: First Precomputation Architecture

and R_2. While R_1 and R_2 are shown as distinct registers in Figure 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.

The first precomputation architecture is shown in Figure 2. Two Boolean functions g_1 and g_2 are the *predictor* functions. We require:

$$g_1 = 1 \Rightarrow f = 1 \quad (1)$$

$$g_2 = 1 \Rightarrow f = 0 \quad (2)$$

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 Equations 1 and 2.

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 non-critical 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 $g_2 = \overline{f}$. 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 Figure 3.

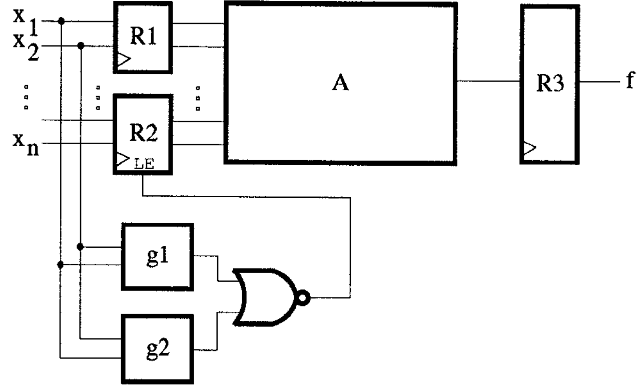

Figure 3: Second Precomputation Architecture

2.2 Second Precomputation Architecture

In the architecture of Figure 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 Equations 1 and 2 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 which should produce reduced switching activity in most cases.

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

The choice of inputs to g_1 and g_2 has to be made first, and then the particular functions that satisfy Equations 1 and 2 have to be selected. A method to perform this selection is described in Section 3.

2.3 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 Figure 4. The precomputation logic is as follows.

$$g_1 = C\langle n-1\rangle \cdot \overline{D\langle n-1\rangle}$$

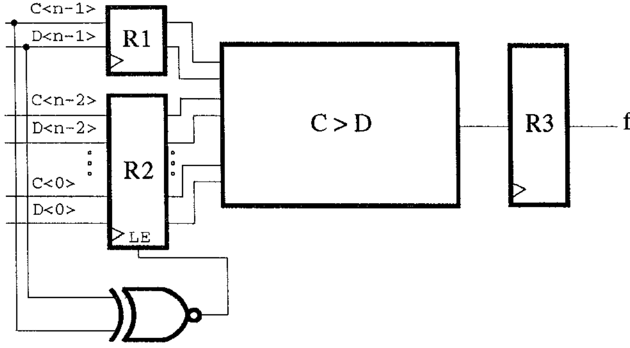

Figure 4: A Comparator Example

$$g_2 = \overline{C\langle n-1\rangle} \cdot D\langle n-1\rangle$$

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

$$\overline{g_1 + g_2} = C\langle n-1\rangle \otimes D\langle n-1\rangle$$

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 expect to achieve a power reduction of close to 50%. The reduction will depend upon the relative power dissipated by the vector pairs with $C\langle n-1\rangle \otimes D\langle n-1\rangle = 1$ and the vector pairs with $C\langle n-1\rangle \otimes D\langle n-1\rangle = 0$. If we add the inputs $C\langle n-2\rangle$ and $D\langle n-2\rangle$ to g_1 and g_2 we expect to achieve a power reduction close to 75%.

3 Synthesis of Precomputation Logic

3.1 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 primarily on the second precomputation architecture illustrated in Figure 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**.

3.2 Precomputation and Observability Don't-Cares

Assume that we have a logic function $f(X)$, with $X = \{x_1, \cdots, x_n\}$, corresponding to block **A** of Figure 2. 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:

$$ODC_i = f_{x_i} \cdot f_{\overline{x_i}} + \overline{f}_{x_i} \cdot \overline{f}_{\overline{x_i}}$$

where f_{x_i} and $f_{\overline{x_i}}$ are the *cofactors* of f with respect to x_i, and similarly for $\overline{f}$.

[1]The assumption here is that each $C\langle i\rangle$ and $D\langle i\rangle$ has a 0.5 static probability of being a 0 or a 1.

If we determine that a given input combination is in ODC_i then we can disable the loading of x_i into the register. If we wish to disable the loading of registers $x_m, x_{m+1}, \cdots, x_N$, we will have to implement the function:

$$g = \prod_{i=m}^{N} ODC_i$$

and use $\overline{g}$ as the (active low) load enable signal for the registers corresponding to $x_m, x_{m+1}, \cdots, x_N$.

3.3 Precomputation Logic

Consider the architecture of Figure 3. Assume that the inputs $x_1, \cdots, x_m$, 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 Equations 1 and 2, respectively, and such that $prob(g_1 + g_2 = 1)$ 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:

$$U_{x_i} f = f_{x_i} \cdot f_{\overline{x_i}}$$

Given a subset of inputs $S = \{x_1, \cdots, x_m\}$, set $D = X - S$. We can define:

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

Theorem 3.1 *$g_1 = U_D f$ satisfies Equation 1. Further, no function $h(x_1, \cdots, x_m)$ exists such that $prob(h = 1) > prob(g_1 = 1)$ and such that $h = 1 \Rightarrow f = 1$.*

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

We cannot add any minterm over $x_1, \cdots, x_m$ to g_1 because for any minterm that is added, there will be some combination of $x_{m+1}, \cdots, x_n$ for which $f(x_1, \cdots, x_n)$ will evaluate to a 0. Therefore, we cannot find any function h that satisfies Equation 1 and such that $prob(h = 1) > prob(g_1 = 1)$. ∎

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

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

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

3.3.1 Selecting a Subset of Inputs

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 $g_1 = U_D f$, $g_2 = U_D \overline{f}$. In the sequel, we assume that the best set of inputs corresponds to the inputs which result in $prob(g_1 + g_2 = 1)$ being maximum for a given k. We know that $prob(g_1 + g_2 = 1) = prob(g_1 = 1) + prob(g_2 = 1)$ since g_1 and g_2 cannot

```
SELECT_INPUTS( f, k ):
{
    /* f = function to precompute */
    /* k = # of inputs to precompute with */
    BEST_PROB = 0 ;
    SELECTED_SET = φ ;
    SELECT_RECUR( f, f̄, φ, X, |X| - k ) ;
    return( SELECTED_SET ) ;
}

SELECT_RECUR( f_a, f_b, D, Q, l ):
{
    if( |D| + |Q| < l )
        return ;
    pr = prob(f_a = 1) + prob(f_b = 1) ;
    if( pr ≤ BEST_PROB )
        return ;
    else if( |D| == l ) {
        BEST_PROB = pr ;
        SELECTED_SET = X - D ;
        return ;
    }
    choose x_i ∈ Q such that i is minimum ;
    SELECT_RECUR( U_{x_i} f_a, U_{x_i} f_b,
                        D ∪ x_i, Q - x_i, l ) ;
    SELECT_RECUR( f_a, f_b, D, Q - x_i, l ) ;

    return ;
}
```

Figure 5: Procedure to Determine the Optimal Set of Inputs

both be 1 on the same input vector. The above cost function ignores the power dissipated 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.

A branch and bound algorithm is used to determine the optimal set of inputs maximizing the probability of the g_1 and g_2 functions. This algorithm is shown in pseudo-code in Figure 5 and is described in detail in [1].

3.3.2 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) [4]. We obtain a ROBDD for the $g_1 + g_2$ function. A ROBDD can be converted into a multiplexor-based network (see [2]) or into a sum-of-products cover. The network or cover can be optimized using standard combinational logic optimization methods that reduce area [3] or those that target low power dissipation [12].

3.4 Multiple-Output Functions

In general, we have a multiple-output function $f_1, \cdots, f_m$ that corresponds to logic block **A** in Figures 2 and 3. All the procedures described thus far can be generalized to the multiple-output case.

The functions g_{1i} and g_{2i} are obtained using the equations below.

$$g_{1i} = U_D f_i$$
$$g_{2i} = U_D \overline{f_i}$$

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

$$g = \prod_{i=1}^{m} (g_{1i} + g_{2i})$$

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 $D = X - S$.

3.4.1 Selecting a Subset of Outputs

In general, it is hard to find a set of inputs for which every output of a multiple-output function is precomputable. We have developed 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 pseudo-code in Figure 6 and is described in detail in [1].

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 precomputed depends on an input that is being disabled, then the output will be incorrect.

Once a set of outputs $G \subset F$ and a set of precomputation logic inputs $S \subset X$ have been selected, we need to duplicate the registers corresponding to $(support(G) - S) \cap support(F - G)$. The inputs that are being disabled are in $support(G) - S$. Logic in the $F - G$ outputs that depends on the set of duplicated inputs has to be duplicated as well. It is precisely for this reason that we maximize $prG \times$ gates(G)/total_gates rather than prG in the output-selection algorithm as we want to reduce the amount of duplication as much as possible.

4 Multiple Cycle Precomputation

4.1 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. We give an example illustrating multiple-cycle precomputation.

Consider the circuit of Figure 7. The function f computes $(C + D) > (X + Y)$ in two clock cycles. 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.

$$g_1 = C\langle n-1 \rangle \cdot D\langle n-1 \rangle \cdot \overline{X\langle n-1 \rangle} \cdot \overline{Y\langle n-1 \rangle}$$

```
SELECT_OUTPUTS( F = {f1, ···, fm}, k ):
{
    /* F = multi-output func. to precompute */
    /* k = # of inputs to precompute with */
    BEST_COST = 0 ;
    SEL_OP_SET = φ ;
    SELECT_OREC( φ, F, 1, k ) ;
    return( SEL_OP_SET ) ;
}

SELECT_OREC( G, H, proldG, k ):
{
    lf = gates(G ∪ H)/total_gates × proldG ;
    if( lf ≤ BEST_COST )
        return ;
    BEST_PROB = total_gates/gates(G ∪ H)
                × BEST_COST ;
    if( G ≠ φ )
        if( SELECT_INPUTS( G, k ) == φ )
            return ;
    prG = BEST_PROB ;
    cost = prG × gates(G)/total_gates ;
    if( cost > BEST_COST) {
        BEST_COST = cost ;
        SEL_OP_SET = G ;
    }
    choose fi ∈ H such that i is minimum ;
    SELECT_OREC( G ∪ fi, H − fi, prG, k ) ;
    SELECT_OREC( G, H − fi, prG, k ) ;

    return ;
}
```

Figure 6: Procedure to Determine the Optimal Set of Outputs

$$g_2 \;=\; \overline{C\langle n-1\rangle} \cdot \overline{D\langle n-1\rangle} \cdot X\langle n-1\rangle \cdot Y\langle n-1\rangle$$

where g_1 and g_2 satisfy the constraints of Equations 1 and 2, respectively. Since $prob(g_1 + g_2) = \frac{2}{16} = 0.125$, we can disable the loading of registers $C\langle n-2:0\rangle$, $D\langle n-2:0\rangle$, $X\langle n-2:0\rangle$, and $Y\langle n-2:0\rangle$ 12.5% of the time, which results in switching activity reduction. This percentage can be increased to over 45% by using $C\langle n-2\rangle$ through $Y\langle n-2\rangle$. We can additionally use single-cycle precomputation logic (as illustrated in Figure 4) to further reduce switching activity in the $>$ comparator of Figure 7. More examples of this technique can be found in [1].

5 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. This was the case for the architectures shown in Section 2. We also extend precomputation so that it can be used in combinational logic circuits.

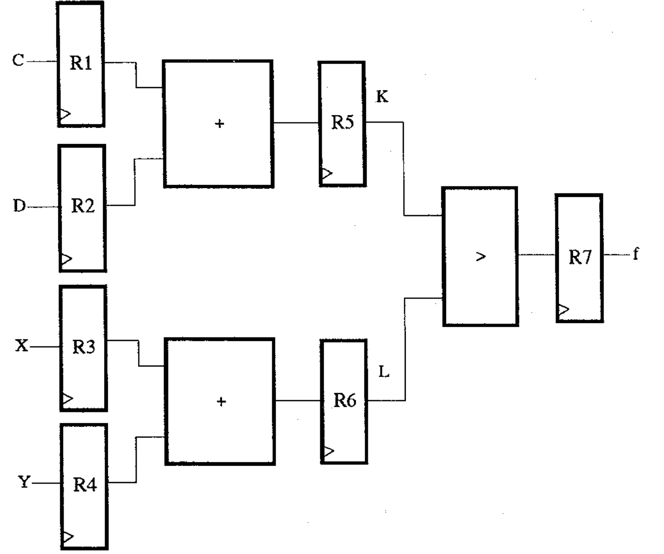

Figure 7: Adder-Comparator Circuit

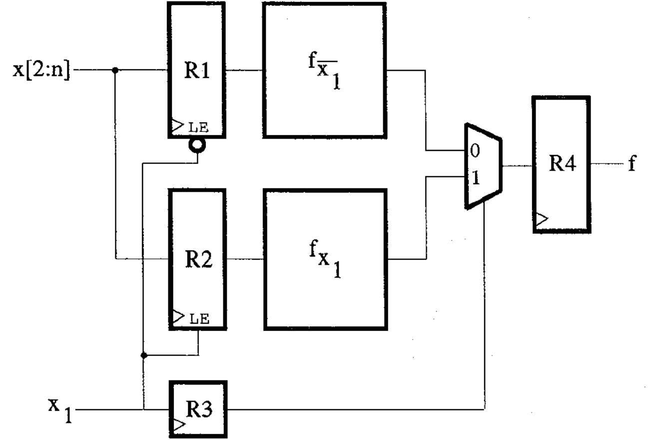

Figure 8: Precomputation Using the Shannon Expansion

5.1 Multiplexor-Based Precomputation

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

$$f \;=\; x_1 \cdot f_{x_1} \;+\; \overline{x_1} \cdot f_{\overline{x_1}} \tag{3}$$

where f_{x_1} and $f_{\overline{x_1}}$ are the cofactors of f with respect to x_1.

Figure 8 shows an architecture based on Equation 3. 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 multiplexor 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 of illustration. In fact, any

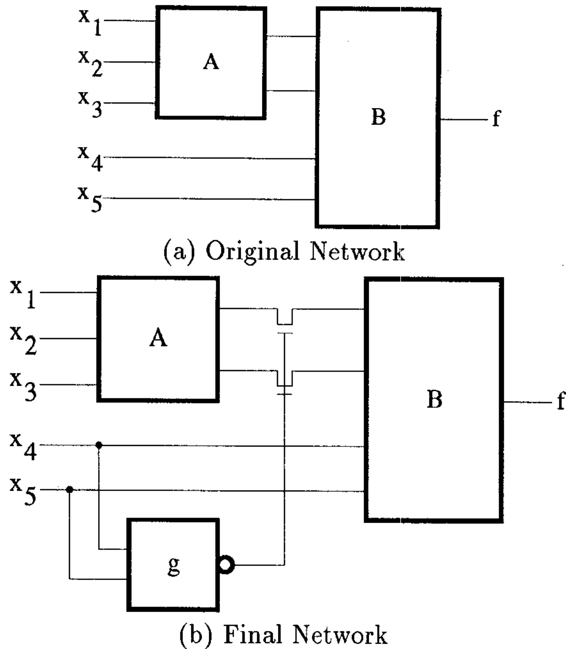

Figure 9: Combinational Logic Precomputation

input $x_1, \cdots, x_n$ 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.

5.2 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 sub-functions **A** and **B** as shown in Figure 9(a). Suppose we also want to precompute this function with the inputs x_4 and x_5. Figure 9(b) shows how this can be accomplished. For simplicity, pass transistors, instead of transmission gates, are shown. The function g with inputs x_4 and x_5 drives the gates of the pass transistors. As in the previous architectures, $g = \overline{g_1 + g_2}$. Hence, when g is a 0, the pass transistors 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 pass transistors 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 best-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 [6]. 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**.

6 Experimental Results

At first we present results on datapath circuits such as carry-select adders, comparators, and interconnections of adders and comparators in Table 1. The precomputation architecture of Figure 3 was used in all examples and the selection of outputs and inputs to use for precomputation was done manually for examples **csa16**, **add_comp16** and **add_max16** and automatically (using the algorithms outlined in Figures 5 and 6) for the rest. For each circuit, the number of literals, levels of logic and power of the original circuit, the number of inputs, literals and levels of the precompute logic, the final power and the percent reduction in power are shown. All power estimates are in micro-Watts and are computed using the techniques described in [7]. A zero delay model and a clock frequency of 20MHz was assumed. The rugged script of **sis** was used to optimize the precompute logic.

Power dissipation decreases for almost all cases. For circuit **comp16**, a 16-bit parallel comparator, the power decreases by as much as 60% when 8 inputs are used for precomputation. Multiple-cycle precomputation results are given for circuits **add_comp16** and **add_max16**. The circuit **add_comp16** is shown in Figure 7, and the circuit **add_max16** is the same circuit with the comparator replaced by a maximum function. For circuit **add_comp16**, for instance, the numbers 4/8 under the fifth column indicates that 4 inputs are used to precompute the adders in the first cycle and 8 inputs are used to precompute the comparator in the next cycle.

Results on random logic circuits are presented in Table 2. The random logic circuits are taken from the MCNC combinational benchmark sets. We have presented results for those examples where significant savings in power was obtained. Once again, the same precomputation architecture and input and output selection algorithms are used as in Table 1 and the columns have the same meaning, except for the second and third columns which show the number of inputs and outputs of each circuit. It is noteworthy that in some cases, as much as 75% reduction in power dissipation is obtained.

The area penalty incurred is indicated by the number of literals in the precomputation logic and 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 do 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 certain classes of datapath circuits.

7 Conclusions 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 Figure 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 Figures 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 5, 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.

8 Acknowledgements

Thanks to Anantha Chandrakasan for providing us with information regarding power dissipation in registers. J. Monteiro and S. Devadas were supported in part by the Defense Advanced Research Projects Agency under contract N00014-91-J-1698 and in part by a NSF Young Investigator Award with matching funds from Mitsubishi Corporation.

References

[1] M. Alidina. Precomputation-Based Sequential Logic Optimization for Low Power. Master's thesis, Massachusetts Institute of Technology, May 1994.

[2] P. Ashar, S. Devadas, and K. Keutzer. Path-Delay-Fault Testability Properties of Multiplexor-Based Networks. *INTEGRATION, the VLSI Journal*, 15(1):1–23, July 1993.

[3] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A Multiple-Level Logic Optimization System. In *IEEE Transactions on Computer-Aided Design*, volume CAD-6, pages 1062–1081, November 1987.

[4] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[5] A. Chandrakasan, T. Sheng, and R. W. Brodersen. Low Power CMOS Digital Design. In *Journal of Solid State Circuits*, pages 473–484, April 1992.

[6] S. Devadas, A. Ghosh, and K. Keutzer. *Logic Synthesis.* McGraw Hill, New York, NY, 1994.

[7] A. Ghosh, S. Devadas, K. Keutzer, and J. White. Estimation of Average Switching Activity in Combinational and Sequential Circuits. In *Proceedings of the 29th Design Automation Conference*, pages 253–259, June 1992.

[8] J. Monteiro, S. Devadas, and A. Ghosh. Retiming Sequential Circuits for Low Power. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 398–402, November 1993.

[9] J. Monteiro, S. Devadas, and B. Lin. A Methodology for Efficient Estimation of Switching Activity in Sequential Logic Circuits. In *Proceedings of the 31st Design Automation Conference*, pages 12–17, June 1994.

[10] F. Najm. Transition Density, A Stochastic Measure of Activity in Digital Circuits. In *Proceedings of the 28th Design Automation Conference*, pages 644–649, June 1991.

[11] K. Roy and S. Prasad. SYCLOP: Synthesis of CMOS Logic for Low Power Applications. In *Proceedings of the Int'l Conference on Computer Design: VLSI in Computers and Processors*, pages 464–467, October 1992.

[12] A. Shen, S. Devadas, A. Ghosh, and K. Keutzer. On Average Power Dissipation and Random Pattern Testability of Combinational Logic Circuits. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 402–407, November 1992.

[13] C-Y. Tsui, M. Pedram, and A. Despain. Exact and Approximate Methods for Switching Activity Estimation in Sequential Logic Circuits. In *Proceedings of the 31st Design Automation Conference*, pages 18–23, June 1994.

Circuit	Original			Precompute Logic			Optimized	
	Lits	Levels	Power	I	Lits	Levels	Power	% Reduction
comp16	286	7	1281	2	4	2	965	25
				4	8	2	683	47
				6	12	2	550	57
				8	16	2	518	60
				10	20	2	538	58
add_comp16	3026	8	6941	4/0	8	2	6346	9
				4/8	24	4	5711	18
				8/0	51	4	4781	31
				8/8	67	6	3933	43
max16	350	9	1744	8	16	2	1281	27
csa16	975	10	2945	2	4	2	2958	0
				4	11	4	2775	6
				6	18	4	2676	9
				8	25	5	2644	10
add_max16	3090	9	7370	4/0	8	2	7174	3
				4/8	24	4	6751	8
				8/0	51	4	6624	10
				8/8	67	6	6116	17

Table 1: Power Reductions for Datapath Circuits

Circuit	Original					Precompute Logic			Optimized	
	I	O	Lits	Levels	Power	I	Lits	Levels	Power	% Reduction
apex2	39	3	395	11	2387	4	4	1	1378	42
cht	47	36	167	3	1835	1	1	1	1537	16
cm138	6	8	35	2	286	3	3	1	153	47
cm150	21	1	61	4	744	1	1	1	574	23
cmb	16	4	62	5	620	5	10	1	353	43
comp	32	3	185	6	1352	6	13	2	627	54
cordic	23	2	194	13	1049	10	18	2	645	39
cps	24	109	1203	9	3726	7	26	3	2191	41
dalu	75	16	3067	24	11048	5	12	2	7344	34
duke2	22	29	424	7	1732	9	24	3	1328	23
e64	65	65	253	32	2039	5	5	1	513	75
i2	201	1	230	3	5606	17	42	5	1943	65
majority	5	1	12	3	173	1	1	1	141	19
misex2	25	18	113	5	976	8	16	3	828	15
misex3	25	18	626	14	2350	2	2	1	1903	19
mux	21	1	54	5	715	1	0	0	557	22
pcle	19	9	71	7	692	3	3	1	486	30
pcler8	27	17	95	8	917	3	3	1	571	38
sao2	10	4	270	17	1191	2	2	1	422	65
seq	42	35	1724	11	6112	2	1	1	2134	65
spla	16	46	634	9	2267	4	6	1	1340	41
term1	34	10	625	9	3605	8	14	3	2133	41
too_large	38	3	491	11	2718	1	1	1	1756	35
unreg	36	16	144	2	1499	2	2	1	1234	18

Table 2: Power Reductions for Random Logic Circuits