Sequential Logic Optimization with Implicit Retiming and Resynthesis*

S. Bommu, M. Ciesielski, N. O'Neill, P. Kalla
University of Massachusetts
Department of Electrical & Computer Engineering
Amherst, MA 01003, USA
Tel: (413)-545-0401 Fax: (413)-545-1993.
email: ciesiel@ecs.umass.edu

Abstract
This paper* introduces a new logic transformation that integrates retiming with algebraic and Boolean transformations at the technology-independent level. It offers an additional degree of freedom in sequential network optimization resulting from implicit retiming across logic blocks and fanout stems. The application of this transformation to sequential network synthesis results in the optimization of logic across register boundaries. We have implemented our new technique within the SIS framework and demonstrated its effectiveness in terms of cycle-time minimization on a set of sequential benchmark circuits.

Keywords
Sequential Logic Synthesis, Logic Optimization, Retiming

1 INTRODUCTION

Over the years, sequential circuit synthesis has been a subject of intensive investigation. Though synthesis of combinational logic has attained a significant level of maturity, *sequential* circuit synthesis is lagging behind. In current state of affairs, sequential networks are first optimized by applying combinational network transformations to the logic between the register boundaries, and mapped into the gate-level network. The resulting network is then often optimized by applying retiming transformation [8].

Retiming is the process of relocating the registers across logic gates without affecting the underlying combinational logic structure. It can be used to minimize cycle-time or the number of registers under the cycle-time constraint.

*This work has been supported in part by a grant from NSF under contract No. MIP-9613864, and in part by the Healey Endowment Grant from the University of Massachusetts. S. Bommu is now with NEC C&C Research Labs, Princeton, N.J.

While in principle retiming can be applied at various levels of synchronous system design, it has been traditionally used as a *structural transformation* in gate-level circuit optimization. As such, gate-level retiming exploits only one degree of freedom in circuit optimization, namely, the relocation of registers. Furthermore, gate-level retiming does not take into account the prospective logic simplification. Potential for the optimization by subsequent re-synthesis is very limited, as it is typically applied to the logic between register boundaries.

In this paper we investigate the issue of retiming at the technology independent level. We introduce a novel and efficient approach to synthesis and optimization of synchronous sequential circuits, in which retiming is performed *implicitly* during logic optimization.

There have been several attempts to combine retiming with algebraic network transformations in the quest to optimize the logic across register boundaries. *Peripheral retiming* introduced by Malik *et. al.* [10] optimizes the underlying combinational logic after a temporary relocation of registers to the periphery of the circuit. It suffers from a limited mobility of registers during the peripheral movement phase. DeMicheli [2] introduced a concept of *synchronous divisors* and used it in local logic optimization across the register boundaries. Lin [9] formalized the theory for synchronous extraction to detect potential common divisors. Both methods operate on the structural specification of a synchronous circuit, and do not take into account the prospective logic simplification during synchronous division. Dey et al [3] proposed a method to improve effectiveness of retiming by attempting to eliminate *retiming bottlenecks*. Chakradhar et al. [1] introduced special timing constraints which are used to resynthesize the circuit. The modified circuit is subsequently retimed, and the constraints (if satisfied by the delay optimizer) guarantee that the circuit is retimable and meets the desired cycle time.

Retiming has been also used in the context of minimizing latency (rather than clock period) in pipelined circuits. A number of papers addressed a problem of combining retiming with architectural and structural transformations to minimize the latency and/or throughput. The scheme proposed by Potkonjak et al. [11] uses retiming to enable algebraic transformations that can further improve latency/throughput. Hassoun et al [5] introduced a concept of *architectural retiming* which attempts to increase the number of registers on a latency-constrained path, without increasing the overall latency. These seemingly contradicting goals are achieved by implementing "negative" registers using precomputation and prediction techniques. In the process, the circuit is structurally modified to preserve its functionality.

Most of the techniques mentioned above operate on a *structural* representation of the network. The cost function that guides retiming in network optimization does not take into account the potential for subsequent logic simplification. In contrast, our approach takes into account the effect of retiming on logic simplification. It operates directly on a *functional* specification given in terms of synchronous Boolean expressions. It is an iterative synthesis process which integrates retiming with extraction, collapsing, and node simplification, into one synchronous transformation. It efficiently handles retiming across fanout stems while preserving initial state. It also provides a simple method to compute initial state of the resynthesized circuit, consistent with the original network specification.

2 PRELIMINARIES

A *Boolean function*, F, of n variables is a mapping $f : B^n \longrightarrow B$, where $B = \{0, 1\}$. A *literal* is a Boolean variable or its complement. A *cube* is defined as a product of literals. The *support* of a Boolean function is defined as a set of all variables that appear in the function. An expression is said to be *cube-free* when it cannot be factored by a cube. A *kernel* of an expression is a cube-free quotient of the expression divided by a cube. *Extraction* is the process of factoring out a subexpression from one or more logic functions of a network and creating a new node for the extracted expression. *Collapsing or elimination* is the process of (re)expressing a Boolean function representing a node in the logic network in terms of the support variables of its fanin node.

Forward retiming is the operation of shifting the registers from the inputs to the outputs of a node in a Boolean network; *backward retiming* is the reverse operation. A node in the network can represent an arbitrary Boolean function. It has been shown that such a transformation preserves the behavior of the circuit [8]. Forward and backward retiming transformations are illustrated in Fig. 1 a). A node is said to be forward (backward) *retimable* if each of its input (output) edges contain a register. *Retiming across a fanout stem* is the operation of forward retiming of a multiple-fanout register across its fanout stem. Retiming across a fanout stem imposes an *equivalence relation* on the fanout registers. All network transformations and *initial state* computation, must take into account this register equivalence. An expression is called a *retimable expression* if all the variables in its support set are register variables.

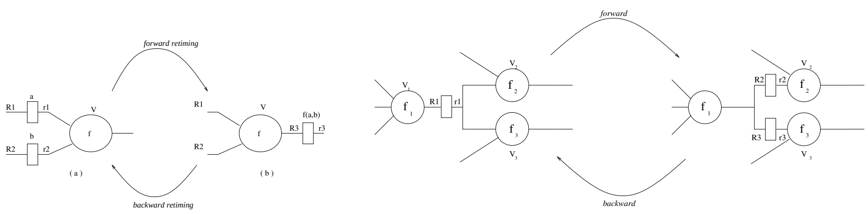

Figure 1 a) Retiming of a logic node b) Retiming across a fanout stem

Associated with each register is a pair of variables (R_i, r_i), where R_i is the input to the register and r_i is its output, referred to as a *register variable.*

The variables r_i and R_i can be viewed as inputs and outputs, respectively, of the combinational part of the sequential network, with registers providing feedback paths.

3 THEORY AND ALGORITHMS

Traditional retiming across a logic gate in a gate-level network (or across a node in a Boolean network) can be extended to a retiming across an arbitrary subexpression of the original logic function. Such a retiming, combined with the extraction of a suitable expression, forms the basis of our new sequential transformation. We shall refer to it as the *logic retiming* transformation, for lack of a better term. The following sections describe the operations involved in logic retiming.

3.1 Retime Extraction

Example 1: Consider the sequential logic network represented by the following equations and shown in Fig. 2a):

$$\begin{aligned} O_1 &= i_2 + r_3 i_1 + r_1 r_2 i_1 \qquad (1) \\ R_1 &= r_1 r_2 i_2 + r_3 i_2 \\ R_2 &= i_1 r_2 \\ R_3 &= i_2 + i_1 r_3 \end{aligned}$$

In these equations i_i denotes a primary input and r_i denotes a register variable (present state variable). O_i is a primary output function and R_i is a register function (next state function).

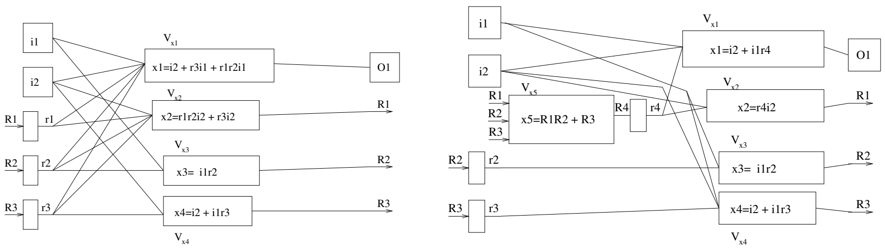

Figure 2 a) The original network, b) Retime-extraction

Consider subexpression $k_r = r_1 r_2 + r_3$, common to O_1 and R_1. This subexpression can be extracted from the expressions for O_1 and R_1 and used to create a new node in the network, V_{x5}. Since all the inputs to k_r are register variables, this expression is forward *retimable.* Forward retiming across V_{x5} leads to the creation of a new register represented by variables (R_4, r_4). After the retiming, the expression for R_4 is then given in terms of register input variables R_i, as illustrated in Fig. 2 b).

This transformation can be expressed as a new operation, called *retime-extraction*, which is the basis of our logic retiming transformation. For a given retimable expression k_r, the following steps implement retime-extraction:

1. For every node f_i of the network, containing expression k_r, substitute the expression with a variable r_k.
2. Introduce a new node corresponding to k_r expressed in terms of register input variables, R_i. Represent it by register function R_k.
3. Introduce a new register (R_k, r_k).

It should be emphasized that, whenever the register variables in the support of retimable expression k_r fan out to other functions, the retime-extract operation involves *implicit retiming across fanout stems.* In our example this applies to registers R_2, R_3 which have multiple fanouts. Consequently, a set of equivalence relations will be imposed on these registers and used in the subsequent logic simplification. On the other hand, if a register involved in the retime-extraction fans out only to the retimable expression, it will be rendered redundant by the transformation and subsequently removed. In our example, R_1 fans out only to the retime-extracted expression, and can be removed along with the associated logic function (Fig. 4).

3.2 Collapsing and Simplification

The next step is to collapse the node represented by a new variable R_k into its fanin nodes, as shown in Fig. 3. The resulting expression is then simplified. Notice the implicit duplication of logic, necessary to perform the collapsing and simplification. This ensures that the functionality of the rest of the network remains unchanged. In our case, logic for R_1, R_2, R_3 is duplicated (see the area marked by the dotted line). The simplification is possible, in effect, due to register equivalence imposed on fanout registers. For simplicity, in all the figures, we use the same variable name for each of the registers obtained after retiming across a fanout.

In our case the collapsing and simplification lead to the following expression:

$$R_4 = R_1 R_2 + R_3 = (r_4 i_2)(i_1 r_2) + (i_2 + i_1 r_3) = i_2 + i_1 r_3 \tag{2}$$

The simplified Boolean expression for R_k is also referred to as a *retime-expression* $RE(k_r)$. It can be calculated for every retimable kernel or cube k_r using the above procedure. The computation of $RE(k_r)$ is central to the logic

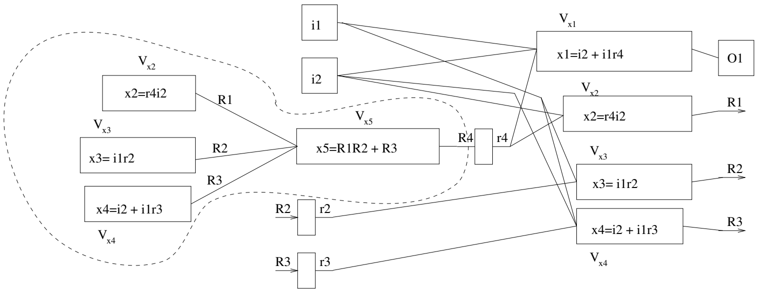

Figure 3 Collapsing of R_4 into its fanin nodes

retiming transformation. In our example, the expressions associated with V_{x5} and V_{x4} are the same $(i_2+i_1r_3)$ and hence V_{x5} can be removed, as shown in Fig. 4(a). Finally, notice that register function R_1 is not used. This is because the register disappeared as a result of retime extraction across $r_1r_2+r_3$. Therefore, the combinational logic function associated with the register function can be deleted. The resulting network is shown in Fig. 4(b). Furthermore, since the register functions R_3, R_4 are identical, the two registers could be merged into one, provided that their initial states are identical, i.e., $r_3^0 = r_4^0$. Whether this is possible or not, depends on the initial conditions imposed on the network (see the next section on initial state computation).

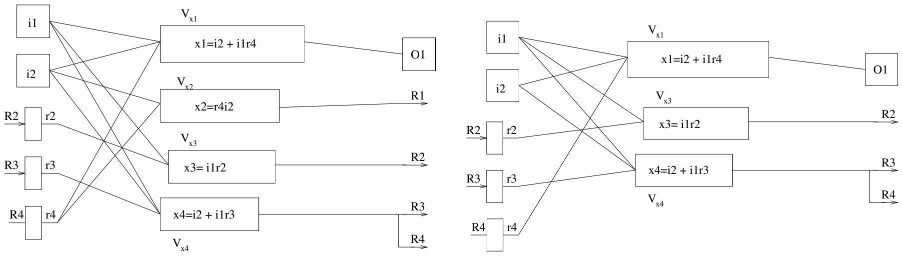

Figure 4 a) Network after simplification b) Final network with redundant logic removed

This network is the direct result of our logic retiming transformation. The retime-extraction, collapsing and simplification transformations are performed implicitly through the computation of the retime-expression.

3.3 Initial State Computation

The initial state computation upon forward retiming across an arbitrary logic expression, as formally given in [12], is straightforward. Let r_i^0 be the initial value of a register (R_i, r_i). For a retimable expression $k_r(r_1, r_2, ..., r_n)$, the initial value of the register (R_k, r_k), added by the retime-extraction, is given by $r_k^0 = k_r(r_1^0, r_2^0, ..., r_n^0)$. For the example above, with retimable expression $k_r = r_1 r_2 + r_3$, the initial value of register (R_4, r_4) is then given by $r_4^0 = r_1^0 r_2^0 + r_3^0$.

3.4 Cost Modeling

Logic retiming is characterized by several important properties, which can be illustrated conceptually in Fig. 5. First, it can be shown that logic retiming does not degrade the overall cycle-time under the unit-delay model. Since the retime-expression node $RE(k_r)$ is obtained by means of collapsing and simplification, it will always be appended to the network at the same (last) level as the nodes that are collapsed into it. By definition, the arrival time at the output of this node will be no greater than the latest arrival time in the rest of the network. Hence adding a retime-expression node will not increase the topological longest path under this model.

Realistically, since retime-extraction may increase fanout on some of the nodes (for example node i_1 in Fig. 4), the critical path delay could actually increase. This may happen, for example, when a node on a critical path fans out to the newly created node, $RE(k_r)$ (see node V_1 in the figure). This problem can be identified by considering an augmented delay model which takes the fanout factor into consideration.

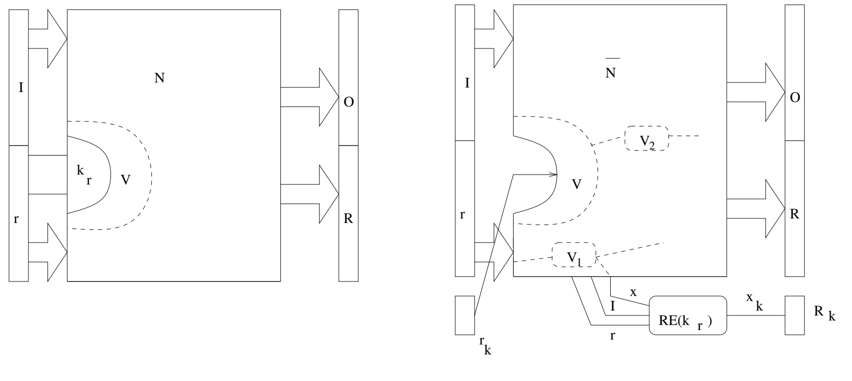

Figure 5 Conceptual view of logic retiming

Finally, observe that the complexity of a node (measured e.g. in the number of literals) that is affected by retime-extraction will always be reduced by the extraction of the retimable expression (see node V in the figure). Since it can be argued that the complexity of a node reflects to a certain degree its delay, the delay of the critical path will be reduced, provided that retime-extraction targets the expressions along that path (for example, if V_2 is on a critical path) and that the fanout increase does not offset the gain due to extraction.

The key element to the efficiency of logic retiming is accurate estimation of the cost associated with a given retimable expression. Fig. 6 illustrates the idea of cost estimation based on simple literal count. It is important to note that the two candidate nodes, k_r and $RE(k_r)$, are not yet part of the network. Two gains are computed: Δx for standard extraction, and Δr for retime-extraction.

$$\begin{aligned} \Delta x &= max(lit\_count(V1),\ lit\_count(V2),\ lit\_count(V3)) \\ \Delta r &= lit\_count(RE(K_r)) \end{aligned} \tag{3}$$

The literal counts of nodes $V1$,$V2$,$V3$ are computed before extraction or retime-extraction; these include the literals of k_r. Retime-extraction (which results in the addition of node $RE(k_r)$) is performed if $\Delta r < \Delta x$.

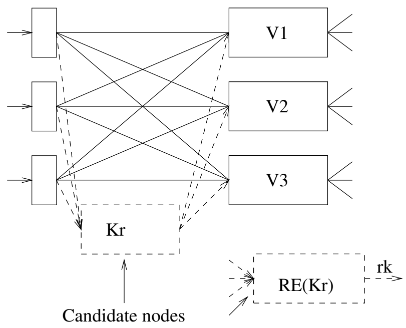

Figure 6 Delay gain estimation based on literal count

Also note that while this approach emphasizes the delay, it can also be used to target the logic area (approximated by the total number of literals). The gain in area can be computed by comparing Δr with $\Delta y = lit\_count(k_r)$. Depending on the depth of collapsing and the amount of logic simplification, Δr may be greater or smaller than Δy.

The initial experiments have shown that even this simplistic gain metric can result in cycle-time reduction. A more accurate approach, currently being considered, involves a fast node decomposition using Time-Driven Cofactoring (TDC) [4].

3.5 Logic Retiming Algorithm

Logic retiming is an iterative operation comprised of the following steps:

1. Select a set of *candidate subexpressions* to be extracted.
2. For each candidate subexpression, check if it is *retimable*.
3. If retimable, *estimate the delay gain* of retime-extraction (Δr) and regular extraction (Δx). It should be emphasized that the gain Δr for the retime-expression k_r is based on all the transformations involved: retime-extraction, collapsing and simplification.
4. If retime-extraction gives better gain, perform retime-extraction. Otherwise, perform regular extraction.

3.6 Comparison with Extraction and Gate-level Retiming

The following example illustrates that logic retiming can lead to circuit optimization (both in terms of delay and area) that is not possible with conventional multi-level synthesis or gate-level retiming alone, see Fig. 7.

Example 2: Consider again logic network for Example 1 (1). Recall that retime-extraction of expression $k_r = (r_1 r_2 + r_3)$ resulted in a new variable $R_4 = R_1 R_2 + R_3 = (r_4 i_2)(i_1 r_2) + (i_2 + i_1 r_3) = i_2 + i_1 r_3$

Extraction:
extract k_r, introduce variable $x_5 = r_1 r_2 + r_3$

$$
\begin{aligned}
O_1 &= x_5 i_1 + i_2\\
R_1 &= x_1 i_2\\
R_2 &= i_1 r_2\\
R_3 &= i_2 + i_1 r_3\\
x_1 &= (r_1 r_2 + r_3)
\end{aligned}
$$

Logic Retiming:
retime_extract across $R_4 = R_1 R_2 + R_3$, collapse and simplify.

$$
\begin{aligned}
O_1 &= r_4 i_1 + i_2\\
R_2 &= i_1 r_2\\
R_3 &= i_2 + i_1 r_3\\
R_4 &= i_2 + i_1 r_3 = R_3
\end{aligned}
$$

Retiming:
structural operation, retime across $(r_1 r_2)$

$$
\begin{aligned}
O_1 &= (r_5 + r_3) i_1 + i_2\\
R_2 &= i_1 r_2\\
R_3 &= i_2 + i_1 r_3\\
R_5 &= R_1 R_2
\end{aligned}
$$

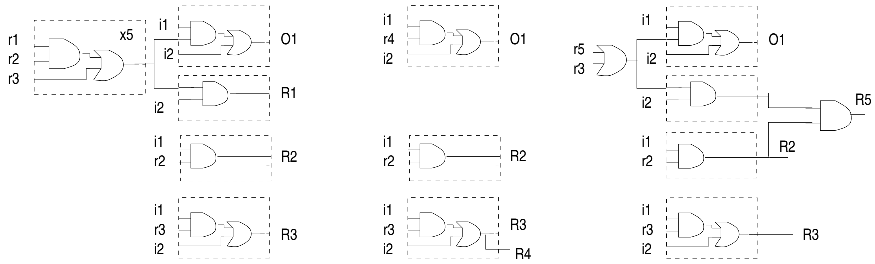

Figure 7 Comparison of logic retiming with extraction and retiming (feedback loops $R_i \rightarrow r_i$ are omitted for simplicity).

4 IMPLEMENTATION AND EXPERIMENTAL RESULTS

We implemented the logic retiming transformation within the SIS framework. The implementation of the logic retiming algorithm involves the generation of common subexpressions of the combinational part of the network; these common subexpressions are generated using the rectangle intersection algorithm used in SIS. Only those kernels whose value exceeds the user-defined threshold are selected. For each of the selected kernels we compare the regular extraction value with the retime-extraction value using the gain estimation technique.

The cost function, used in our preliminary experiments, is the number of literals in the SOP form, as discussed in section 3.4. Although simplistic, this cost function allowed us to quickly validate the theory. This version of logic retiming yielded delay improvements over the regular extraction transformation. Research is now focused on the application of the concept of retime-extraction to the transformations used in *script.delay* and other delay optimization techniques such as *speed_up*. We are also investigating the application of logic retiming to area minimization under cycle-time constraints.

We have tested our technique on a number of sequential circuits from the ISCAS'91 benchmark set. The circuits were input as logic networks in *blif* format, its local functions (nodes) were collapsed into SOP form. Each circuit was then optimized using logic retiming and independently synthesized with standard SIS multi-level optimization. The circuits were resynthesized and mapped into the standard SIS lib2.genlib library. The script used for logic retiming is identical to the script with conventional SIS transformations, except that the *gkx* command has been replaced by the "retime kernel extract" (*rkx*) command of logic retiming. The general structure of the scripts used in our experiments is given below:

script.rkx	*script.gkx*
sweep	*sweep*
collapse or eliminate <*threshold*>	*collapse or eliminate* <*threshold*>
simplify	*simplify*
rkx <*options*>	*gkx* <*options*>
resub -a	*resub -a*
sweep	*sweep*
simplify	*simplify*

The results are reported in Table 1, which compares the clock-cycle delay, number of registers, and area overhead of the circuits obtained by the two flows. The delay was computed using the mapped delay model. Those circuits which did not contain any retimable kernels are not shown in the table.

Even though our initial implementation of logic retiming used a simplistic

Table 1 *gkx* vs *rkx*: Comparison of mapped circuits

Ckt	rkx			gkx			% change		
	Area	Clk	Reg	Area	Clk	Reg	Area	Clk	Reg
s298	167040	9.59	25	145232	10.95	14	15	-12	79
s344	198592	13.50	17	187456	17.06	15	6	-21	13
s444	223648	9.51	24	203232	13.09	21	10	-27	14
s526	228288	10.10	25	208336	13.64	21	10	-26	19
s400	266800	11.05	28	211120	12.95	21	26	-15	33
s9234	1156752	31.90	147	1101536	38.28	135	5	-17	9
s5378	1316832	25.29	189	1286672	26.31	162	2	-4	17
s510	245920	24.49	8	223184	28.20	6	10	-13	33
s15850	3912912	104.1	538	3802480	108.23	504	3	-4	7
s1488	629648	42.72	13	607840	39.67	6	4	7	117
s382	329904	10.63	33	215760	13.82	21	53	-23	57

figure of merit based on literal count, most of the circuits synthesized with this technique showed a significant reduction in delay. We expect that with a better estimation scheme and more selective retime-extraction the transformation will give better and consistent improvement in delay. The algorithm and library used for mapping would also have a significant impact on the delay of the optimized circuit; applying retime-extraction with explicit knowledge of the above information could improve the effectiveness of logic retiming.

REFERENCES

[1] S.T. Chakradhar, S. Dey, M. Potkonjak, S.G. Rothweiler, "Sequential Circuit Delay Optimization using Global Path Delays", *Proc. 1993 Design Automation Conf.*, 1993, pp. 483-489.

[2] G. De Micheli, "Synchronous Logic Synthesis: Algorithms for Cycle-Time Optimization", *IEEE Trans. on CAD*, Vol.10, No.1, Jan. 1991, pp. 63-73.

[3] S. Dey, M. Potkonjak, S. Rothweiler, "Performance Optimization of Sequential Circuits by Eliminating Retiming Bottlenecks", *Intl. Conference on CAD*, 1992, pp. 504-509.

[4] P. Gutwin, P. McGeer, R. Brayton, "Delay Prediction for Technology-Independent Logic Equations", *Proc. IEEE Intl. Conference on Computer Design*, 1992, pp. 468-471

[5] S. Hassoun, C. Ebeling, "Architectural Retiming: Pipelining Latency-Constrained Circuits", *Proc. 1996 Design Automation Conf.*, 1996.

[6] G.D. Hachtel, M. Hermida, A. Pardo, M. Poncino, F. Somenzi, "Re-

encoding Sequential Circuits to Reduce Power Dissipation", *Intl. Conference on CAD*, 1994, pp. 70-73.

[7] S. Iman and M. Pedram "Logic Extraction and Factorization for Low Power", *Proc. 1995 Design Automation Conf.*, 1995, pp. 248-253.

[8] C. Leiserson, F. Rose, J. Saxe, "Optimizing Synchronous Circuitry by Retiming, *Third Caltech Conference on VLSI*, 1983, pp. 87-116.

[9] B. Lin, "Restructuring of Synchronous Logic Circuits", *Proc. European Design Automation Conference*, 1993, pp. 205-209.

[10] S. Malik, E. Sentovich, R. Brayton, A. Sangiovanni-Vincentelli, "Retiming and Resynthesis: Optimizing Sequential Networks with Combinational Techniques", *IEEE Trans. on CAD*, Vol.10, No.1, Jan. 1991, pp. 74-84.

[11] M. Potkonjak, S. Dey, Z. Iqbal, A. Parker. "High Performance Embedded System Optimization Using Algebraic and Generalized Retiming Techniques", *Proc. IEEE Intl. Conference on Computer Design*, 1993, pp.498-504.

[12] H.J. Touati and R.K. Brayton, "Computing the Initial States of Retimed Circuits", *IEEE Trans. on CAD*, (12)1, Jan. 1993, pp. 157-162.

[13] C.Y. Tsui, M. Pedram, C-A Chen, and A. Despain, "Technology Decomposition and Mapping Targetting Low Power Dissipation", *Proc. 1994 Conference on CAD*, 1994, pp. 82-87.

5 BIOGRAPHY

Surendra Bommu received his B.Tech degree in Electrical and Electronics Engineering from Indian Institute of Technology, Madras, India in 1993. He received his M.S. degree from the department of Electrical and Computer Engineering, University of Massachusetts, Amherst in 1996. He is currently working as a research associate at C&C Research Labs, NEC USA Inc., Princeton. His research interests include Logic Synthesis, High Level Synthesis, and Hardware/Software Codesign.

Maciej Ciesielski received his M.S. in Electrical Engineering from Warsaw Technical University, Poland, in 1974, and Ph.D. in Electrical Engineering from the University of Rochester in 1983. From 1983 to 1986 he was a Senior Member of Technical Staff at GTE Laboratories. Currently he is Associate Professor in the Department of Electrical and Computer Engineering at the University of Massachusetts, Amherst. His research interests include: CAD for VLSI systems, architectural, logic and physical synthesis and performance optimization of IC's. He has served on the technical program committees for ICCAD, ICCD, IWLS, IFIP/IWLAS, and others. He is a member of the IEEE.

Niall O'Neill and Priyank Kalla are graduate students at the University of Massachusetts, Department of Electrical & Computer Engineering.