RETIMING AND RESYNTHESIS: OPTIMIZING SEQUENTIAL NETWORKS WITH COMBINATIONAL TECHNIQUES

by

Sharad Malik, Ellen M. Sentovich, Robert K. Brayton,
and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M89/28

10 March 1989

RETIMING AND RESYNTHESIS: OPTIMIZING SEQUENTIAL NETWORKS WITH COMBINATIONAL TECHNIQUES

by

Sharad Malik, Ellen M. Sentovich, Robert K. Brayton,
and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M89/28

10 March 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

RETIMING AND RESYNTHESIS: OPTIMIZING SEQUENTIAL NETWORKS WITH COMBINATIONAL TECHNIQUES

by

Sharad Malik, Ellen M. Sentovich, Robert K. Brayton,
and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M89/28

10 March 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Retiming and Resynthesis: Optimizing Sequential Networks with Combinational Techniques

Sharad Malik Ellen M. Sentovich Robert K. Brayton
Alberto Sangiovanni-Vincentelli
Department of Electrical Engineering and Computer Science
University of California, Berkeley, CA 94720

February 23, 1989

Abstract

Sequential networks contain combinational logic blocks separated by registers. Application of combinational logic minimization techniques to the separate logic blocks results in improvement that is restricted by the placement of the registers; information about logical dependencies between blocks separated by registers is not utilized. Temporarily moving all the registers to the periphery of a network provides the combinational logic minimization tools with a global view of the logic. We propose an algorithm for optimizing a sequential network by moving the registers to the boundary of the network using an extension of retiming [1,2], resynthesizing the combinational logic between the registers using existing logic minimization techniques, and replacing the registers throughout the network using retiming algorithms.

1 Introduction

Over the past decade, combinational logic optimization has attained a significant level of maturity. The problems and approaches in combinational logic synthesis are well understood: almost fully for the two-level logic case (e.g. [4]), and to a lesser extent in the multi-level logic case (e.g. [3,5]). In comparison, sequential synthesis is just beginning to be recognized as a problem domain in its own right. Most existing efforts in sequential synthesis can be classified into three categories. The first approach is to consider the portions of combinational logic between register boundaries and use combinational logic optimization techniques on these separate blocks. However, this is restrictive inasmuch as it does not permit the interactions between gates separated by register boundaries to be examined in the optimization process. The second approach ([1,2]) involves moving registers across portions of combinational logic in order to minimize the cycle time and the number of registers used. This procedure, termed **retiming**, does not change any of the combinational logic blocks. Thus, it does not consider further optimizations that could have been obtained with that option. The third approach considers only a special class of sequential circuits, *viz.* finite state machines (FSMs). Operations on state transition graphs (STGs) and results from automata theory have been used to optimize implementations of STGs. One drawback with this approach is that all the manipulations and optimizations are attempted at the STG level and it is not clear how this will be reflected in the final gate-level implementation of the machine. Researchers have proposed different cost criteria such as the number of edges and the number of states in the STG as a metric for operations at the STG level. Unfortunately, none of these is a consistent reflection of the gate-level complexity.

In this paper we describe a new approach towards optimizing sequential circuits. As in [1,2] we assume a synchronous implementation with edge triggered registers. We characterize the maximal sub-network in the sequential network for which the registers can effectively be ignored and the sub-network be considered as a combinational block. This permits existing combinational logic optimization techniques to be used on this block. This approach is more powerful than the first of the approaches stated above, since it examines interactions between portions of logic separated by registers. As a result, the optimization process makes full use of dependencies between gates. Moreover, we guarantee that it is complete, i.e., we find the largest sub-network for which we can do this operation. This ensures that we do not miss any optimization that we could obtain by considering interactions between

gates. Converting this sub-network to a combinational logic block can be viewed as a retiming process in which all the registers have been pushed to the periphery of the sub-network. However, our technique is more powerful than conventional retiming. We permit **negative registers** to be pushed to the periphery. This is equivalent to temporarily "borrowing" registers from the environment, and is a legitimate operation as long as these registers are "returned" to the environment at the end of the optimization process. This additional allowance is very powerful since it permits a larger portion of the logic to be viewed as a single block than was permitted by conventional register movements and retiming. The resynthesis phase operates on this combinational logic block and resynthesizes it according to a specified cost function. This could be minimizing the area, the delay or meeting a particular area/delay tradeoff. Finally, we re-distribute the registers in this combinational block. We guarantee that there will be some legal re-distribution of the registers even with the negative registers, i.e., we will be able to return the registers that were borrowed from the environment. The re-distribution can be done while satisfying constraints such as minimizing the number of registers subject to a specified cycle time (if these constraints are satisfiable) by using the algorithms described in [2].

The sequential circuit corresponding to a finite state machine offers an interesting case to apply our theoretical formulation. In this case any cut through the combinational logic that breaks the feedback cycle, results in a sequential network that satisfies the conditions under which all the registers can be ignored. Thus, depending on the cut we make, we can consider several different combinational blocks for combinational optimization. This is significantly different than the way combinational optimization has been used thus far in state machine optimization, *viz.* considering only the cut at the registers. We propose a **Sliding Window Optimization** technique that considers a large number of these different combinational blocks. The efficacy of this technique is demonstrated through an example. Next, the relationship between this optimization process and sequential redundancy is examined. Finally, we look at the relationship between operations at the STG level, the state assignment problem and the operations permitted by this resynthesis and retiming process.

The remainder of this paper is organized as follows. Section 2 gives the theoretical formulation and results on which our approach is based. In Section 3 we consider the application of this formulation to the problem of optimizing a circuit implementing a finite state machine. Finally, in Section 4, we conclude by summarizing our approach and stating the future directions of our research in this area.

2 Theoretical Formulation

We model a sequential circuit by a directed acyclic graph called a **communication graph**[1] where each vertex v represents either

a) an input/ouput pin or

b) a combinational logic block.

The vertices in the graph are connected by directed edges. We place the restriction that each input pin has no incoming edges and exactly one outgoing edge, and that an output pin has no outgoing edges and exactly one incoming edge. An **internal edge** connects vertex u to vertex v if both u and v represent combinational logic blocks, and the logic represented by v explicitly depends on the value computed at u. A **peripheral edge** connects either an input pin to the logic block that represents that input or connects a logic block that computes the value of an output to the corresponding output pin. Each edge e has a corresponding weight $w(e)$ representing the number of registers between the two vertices it connects. An example of a communication graph is shown in Figure 1. We use the terms circuit, network, and graph interchangeably whenever there is no ambiguity.

A **path** is a sequence of edges and vertices that are connected in the graph. The weight of a path is the sum of the weights of all the edges along the path. In Figure 1, the path from input i_1 to output o_1 has weight 2, while the path from input i_2 to o_1 has weight 3.

2.1 Retiming: An Overview

Retiming is an operation on a communication graph whereby registers are moved across logic blocks in such a way as to minimize the clock cycle or minimize the number of registers, while maintaining the behavior of the circuit.

[1] This is related to the definition of a communication graph presented in [1]

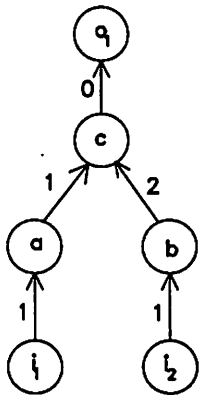


Figure 1: Communication Graph

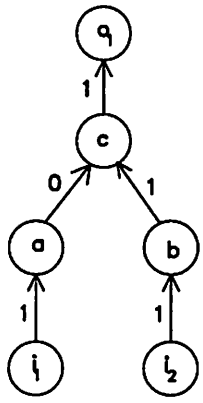


Figure 2: Retimed Circuit

Retiming algorithms were first proposed by Leiserson *et al* [1,2]. The movement of registers can be quantified by an integer $L(v)$ (called the lag of v) for each vertex v, which represents the number of registers that are to be moved in the retimed circuit from the out-edges of vertex v to its in-edges.

Definition 1 *A* **legal retiming** *is the assignment of an integer $L(v)$ to each vertex such that*

a) $L(v) = 0$ if v is an I/O pin and

b) $w(e) + L(v) - L(u) \geq 0$ where e is the edge from vertex u to vertex v.

The edge weights of the retimed circuit, $w_r(e) = w(e)+L(v)-L(u)$, must be nonnegative for all edges e, representing a nonnegative number of registers connecting the two logic blocks. A legal retiming has been proven [1] to generate a circuit that is functionally equivalent to the original circuit. The circuit shown in Figure 1 can be retimed by assigning a lag of -1 to vertex c ($L(c) = -1$) and a lag of 0 to all other vertices. The resulting retimed circuit is shown in Figure 2. Note that the path weights from the inputs to the outputs are unchanged.

2.2 Extensions to Retiming

The retiming operation can be extended by introducing the concept of a "negative" register, that is, an edge weight in the graph that is negative. Allowing a negative edge weight n on a peripheral edge is equivalent to "borrowing" n registers from the environment. The registers are "returned" by a subsequent retiming step whereby n registers are forced to each edge with weight $-n$. Negative edge weights are allowed on the peripheral edges only. The

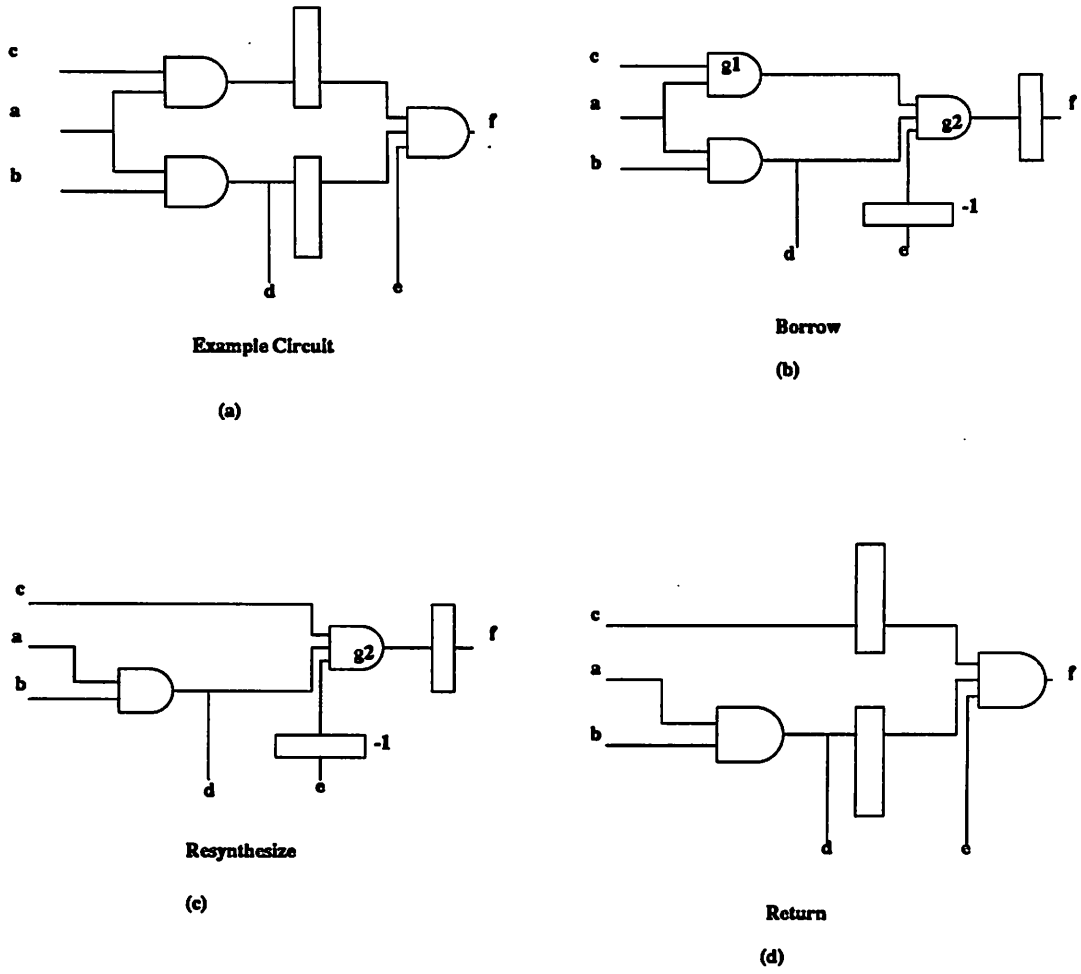


Figure 3: Example: Use of a Negative Register

observation that the peripheral edge weights can temporarily take on negative values allows retiming operations and subsequent optimizations that otherwise would not be possible. An example circuit is shown in Figure 3(a) (in schematic drawings, combinational logic blocks are represented by conventional gate symbols or circles, and registers by rectangles). If a lag of -1 is assigned to the gate $g2$, the edge between input e and gate $g2$ would have weight -1, as in Figure 3(b). This is equivalent to borrowing a register at input e. During subsequent combinational resynthesis, the redundant connection from a to $g1$ allows the removal of gate $g1$ (Figure 3(c)). Finally the circuit is retimed with $L(g2) = 1$. This returns the register borrowed at input e resulting in the circuit shown in Figure 3(d). This smaller implementation could not be obtained without allowing the edge weight to temporarily take on a negative value.

We define in addition to legal retiming, a specific type of retiming that exploits the negative register concept while pushing the registers to the boundaries of a network.

Definition 2 *A* **peripheral retiming** *is a retiming such that*

a) $L(v) = 0$ *where* v *is an I/O pin and*

b) $w(e) + L(v) - L(u) = 0$ *where* e *is an internal edge from vertex* u *to vertex* v.

A peripheral retiming moves all registers to the peripheral edges, leaving a purely combinational logic block between two sets of registers. For example, by assigning a lag of 1 to vertex b in Figure 2, we obtain the circuit in Figure 4,

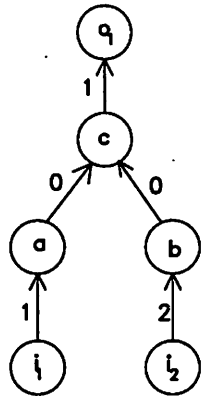


Figure 4: Peripherally Retimed Circuit

which is a peripheral retiming of both the circuit in Figure 1 and that in Figure 2.

Note that the definition of a peripheral retiming permits negative edge weights on the peripheral edges, which corresponds to the negative register concept presented at the beginning of this section. Permitting negative registers on peripheral edges is a legitimate operation.

Proposition 1 *A circuit that undergoes a peripheral retiming and a subsequent legal retiming is equivalent to the original circuit.*

The proof of this has been omitted for brevity.

2.3 Conditions for Peripheral Retiming

Not all circuit structures permit a peripheral retiming: the circuit in Figure 5 has no peripheral retiming because the register cannot be is pushed to the output or to the input without leaving one of the internal edges with a negative weight. For example, if the register is moved toward output o_2, a negative weight is forced on the edge between vertices c and e; if this negative edge is pushed toward the inputs, a negative weight is forced on the edge between vertices b and e. A similar problem occurs when the register is pushed toward the inputs.

It is important to characterize the circuit structure that allows a peripheral retiming since these circuits can undergo a resynthesis optimization on their entire combinational logic block. For this purpose we define the **path weight matrix** of a network.

Definition 3 *A* **path weight matrix, W,** *of a sequential network is an $m \times n$ matrix, where*

1) m is the number of inputs

2) n is the number of outputs

3) $W_{ij} = *$ if no path exists between input i and output j

4) $W_{ij} = \sim$ if two paths between input i and output j have different weights

5) $W_{ij} = \sum_{path\ i_i \rightarrow o_j} w(e)$ if all paths between input i and output j have the same weight.

In addition, we define the satisfiability condition on the path weight matrix, which is intimately related to the existence of a peripheral retiming.

Definition 4 *A matrix* **W** *is* **satisfiable** *if*

a) $W_{ij} \neq \sim$, $\forall i$, $\forall j$

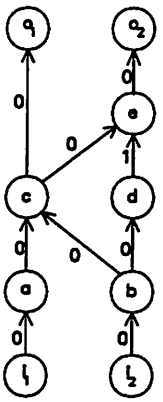


Figure 5: Circuit with no valid peripheral retiming

b) $\exists \alpha_i, \exists \beta_j, 1 \leq i \leq m, 1 \leq j \leq n, \alpha_i, \beta_j \in I$ *such that for each* $W_{ij} \neq *$, $W_{ij} = \alpha_i + \beta_j$.

Finally, we state the relationship between a satisfiable path weight matrix and the existence of peripheral retiming.

Theorem 1 *A sequential network has a peripheral retiming if and only if its path weight matrix is satisfiable.*

Note the significance of this result: it gives a complete characterization of the class of sequential circuits for which all the registers can be pushed to the boundary allowing resynthesis on the combinational block.

A peripheral retiming involves finding a set of $\alpha's$ and $\beta's$ that satisfy the path weight matrix, and moving the registers accordingly. The path weight matrix contains information about the number of registers between each input and each output. α_i and β_j dictate how many registers will appear at the i^{th} input and the j^{th} output edge respectively, in the peripherally retimed circuit. A matrix that is satisfiable has no $\sim$ entries, and has at least one set of $\alpha_i's$ and $\beta_j's$ such that $\alpha_i + \beta_j = W_{ij}$. For the circuit in Figure 1, the path weight matrix is as follows:

	o_1
i_1	2
i_2	3

and can be satisfied by choosing, for example, $\alpha_1 = 1$, $\alpha_2 = 2$, $\beta_1 = 1$, resulting in the circuit shown in Figure 4.

Note the the path weight matrix for the circuit in Figure 5, which had no peripheral retiming, is as follows:

	o_1	o_2
i_1	0	0
i_2	0	1

Applying the conditions necessary to satisfy the matrix, we obtain:

$$\alpha_1 + \beta_1 = 0 \tag{1}$$

$$\alpha_1 + \beta_2 = 0 \tag{2}$$

$$\alpha_2 + \beta_1 = 0 \tag{3}$$

$$\alpha_2 + \beta_2 = 1. \tag{4}$$

Subtracting Equation 1 from Equation 2:

$$\beta_2 - \beta_1 = 0.$$

Subtracting Equation 3 from Equation 4 we obtain

$$\beta_2 - \beta_1 = 1.$$

The contradiction implies that the path weight matrix is not satisfiable.

2.4 Legal Resynthesis Operations

Permitting negative registers on the peripheral edges is a legitimate operation as long as the resynthesized circuit has a legal retiming. This leads us to ask the following question: can we guarantee that the resynthesized circuit always has a legal retiming? To examine this further we need to define a synchronous communication graph[2].

Definition 5 *A synchronous communication graph is one in which each path between an input pin and an output pin has a non-negative path weight.*

The following theorem precisely states the conditions in which a legal retiming exists.

Theorem 2 *A communication graph has a legal retiming if and only if it is synchronous.*

Note that since the initial communication graph had no negative edges (it represents a real circuit) it is synchronous. Peripheral retiming preserves the synchronous property since retiming does not change the path weight between an input and an output pin. However, resynthesis can change the communication graph and hence it may destroy the synchronous property.

Let us see how this can happen. Let G_1 be the communication graph before resynthesis and G_2 be the graph after resynthesis. If there was a path between input i and output j in G_1 and there is a path between them in G_2, then the path weight for this path in G_2 is $\alpha_i + \beta_j$. This is the same as the path weight W_{ij} in G_1. Since G_1 was synchronous, this path weight is non-negative. Now consider the case in which no path existed in G_1 between input i and output j and resynthesis creates a path. The path weight for this path in G_2 is $\alpha_i + \beta_j$. Since α_i and β_j may be negative and G_1 did not force a non-negativity constraint on $\alpha_i + \beta_j$ (since no path existed between input i and output j), it is possible that $\alpha_i + \beta_j$ may be negative, thus destroying the synchronous property. Note that output j does not actually depend on input i; however, resynthesis created a *pseudo-dependency* between the two.

An example is shown in Figure 6(a). This circuit has a peripheral retiming shown in Figure 6(b). Resynthesis discovers that the three-input OR gate g_1, can be replaced by a two-input OR gate g_2 (Figure 6(c)). The communication graph for this circuit is not synchronous since there exists a path of negative weight (-1) between input a and output *out*1. By Theorem 2 we know that this circuit has no legal retiming.

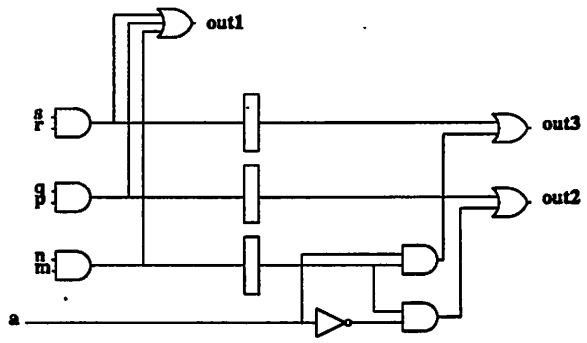
Thus, resynthesis must ensure that it does not introduce a pseudo-dependency with a negative path weight; this is the only condition that the resynthesis must satisfy. This will not significantly reduce the usefulness of the retiming and resynthesis approach for two reasons. First, based on existing combinational logic optimizations, it seems relatively difficult (and hence less probable) for this to occur. Second, this condition can easily be checked after resynthesis and the resynthesis rejected if this does happen.

2.5 Summary of the Algorithm

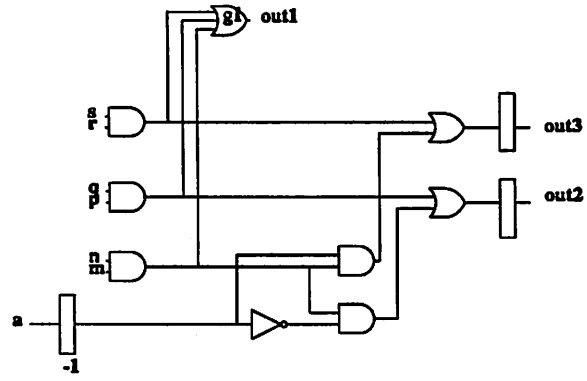
Given a synchronous circuit with a satisfiable path weight matrix, we can optimize it by pushing the registers to the boundary, resynthesizing the interior logic, and finally replacing the borrowed registers. The algorithm can be summarized as follows:

1. Formulate the path weight matrix for the circuit.
2. Compute α_i and β_j for $1 \leq i \leq m$, $1 \leq j \leq n$.
3. Place α_i registers after each input i and β_j registers before each output j; remove (replace by wires) all other registers.
4. Resynthesize the interior combinational logic block using standard techniques.
5. Formulate the path weight matrix for the retimed circuit.
6. If the path weight matrix has no negative entries, find a legal retiming for the circuit according to a cost criterion (minimize clock cycle, minimize state).

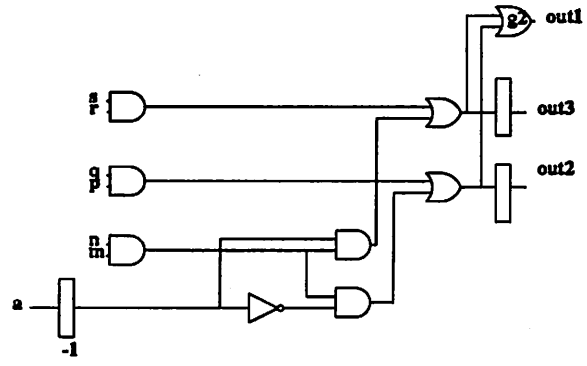
[2]This is related to the definition of a synchronous circuit presented in [1]



(a)



(b)



(c)

Figure 6: Introducing Pseudo-dependencies with Negative Path Weight

3 Optimizing FSM Implementations

We now focus our attention at optimizing perhaps the most prevalent and important class of sequential circuits: FSM implementations. We will show how our general formulation applies in this case and the resulting optimization algorithms that develop from this. We first consider a specific example and then give the general optimization procedure.

3.1 An Example

Figure 7(a) shows a gate level schematic of an FSM implementation. We break the feedback cycle by cutting the net p. This results in a pseudo-input *p_in* and a pseudo-output *p_out*. The circuit is then redrawn with the signal flow unidirectional (Figure 7(b)). A peripheral retiming of this circuit is shown in Figure 7(c). The optimization of the combinational block discovers that *out*2 is always equal to **0** and thus the AND gate $g1$ is not needed. Also, x_1 may be replaced by the constant **0** without changing the logic. Thus, g_2 and g_3 are not needed either; these gates can be removed (Figure 7(d)). The circuit is retimed to a legal retiming (Figure 7(e)). The feedback connection is made and the final circuit is shown in Figure 7(f). This circuit has one less register and two fewer gates than the initial circuit; this represents a significant gain.

Conventional optimization techniques would have only considered breaking the feedback cycle at the registers. Optimizing the resulting combinational circuit leads to no improvement in this case.

3.2 The Generic Optimization Procedure

Figure 8(a) shows the block diagram of a generic FSM implementation. Let us consider any cut (a breaking of a net) that breaks all the cycles (Figure 8(b)). The resulting logic can then be re-drawn so that the signal flow is unidirectional (Figure 8(c)). PSI and PSO are the pseudo-inputs and pseudo-outputs introduced in the circuit because of the cut. I_1 and I_2 are the sets of primary inputs feeding the two blocks and O_1 and O_2 are the sets of primary outputs available from the two blocks. If any primary input feeds both logic blocks, then it must be given different names in the two blocks. Thus, $I_1 \cap I_2 = \phi$. The path weight matrix for this network is:

	O_1	O_2	PSO
PSI	1	0	1
I_2	1	0	1
I_1	0	*	0

This matrix is satisfiable. The following is an assignment of the α's and β's that satisfies it: $\alpha = [0 \quad 0 \quad -1]$ and $\beta = [1 \quad 0 \quad 1]$. (This is not a unique assignment, several satisfying assignments exist). This corresponds to the peripheral retiming shown in Figure 8(d). Thus, we see that for any such cut, there always exists a peripheral retiming.

The resulting combinational block (concatenation of N_2 and N_1) to be resynthesized depends on the choice of the cut. Since each cut results in a different combinational block, we can consider a series of such cuts and optimize each of the resulting combinational blocks. In fact, this forms the basis of the **Sliding Window Optimization** technique. This is better explained by looking at Figure 9. The combinational logic can be considered to be wrapped around a circle to reflect its feedback nature. The first cut (C_1) is made at the initial placement of registers. This represents one *window* or *view* of this circular combinational logic. The next cut (C_2) can be made by sliding this window around to look at a different view of the logic. Combinational resynthesis can be used at each view. The optimization process continues by sliding this window around the circle for at least one revolution. This may be continued until the resynthesis does not yield any improvement for a complete revolution. Note that if only cut C_1 was used (as in conventional FSM logic minimization) no information is used about the feedback nature of the logic. In the *Sliding Window* procedure, each window conveys some of this feedback information. We believe this aspect of the logic interaction has not been considered so far in optimization techniques and is significantly more powerful than conventional techniques for optimizing FSM implementations. We would like to

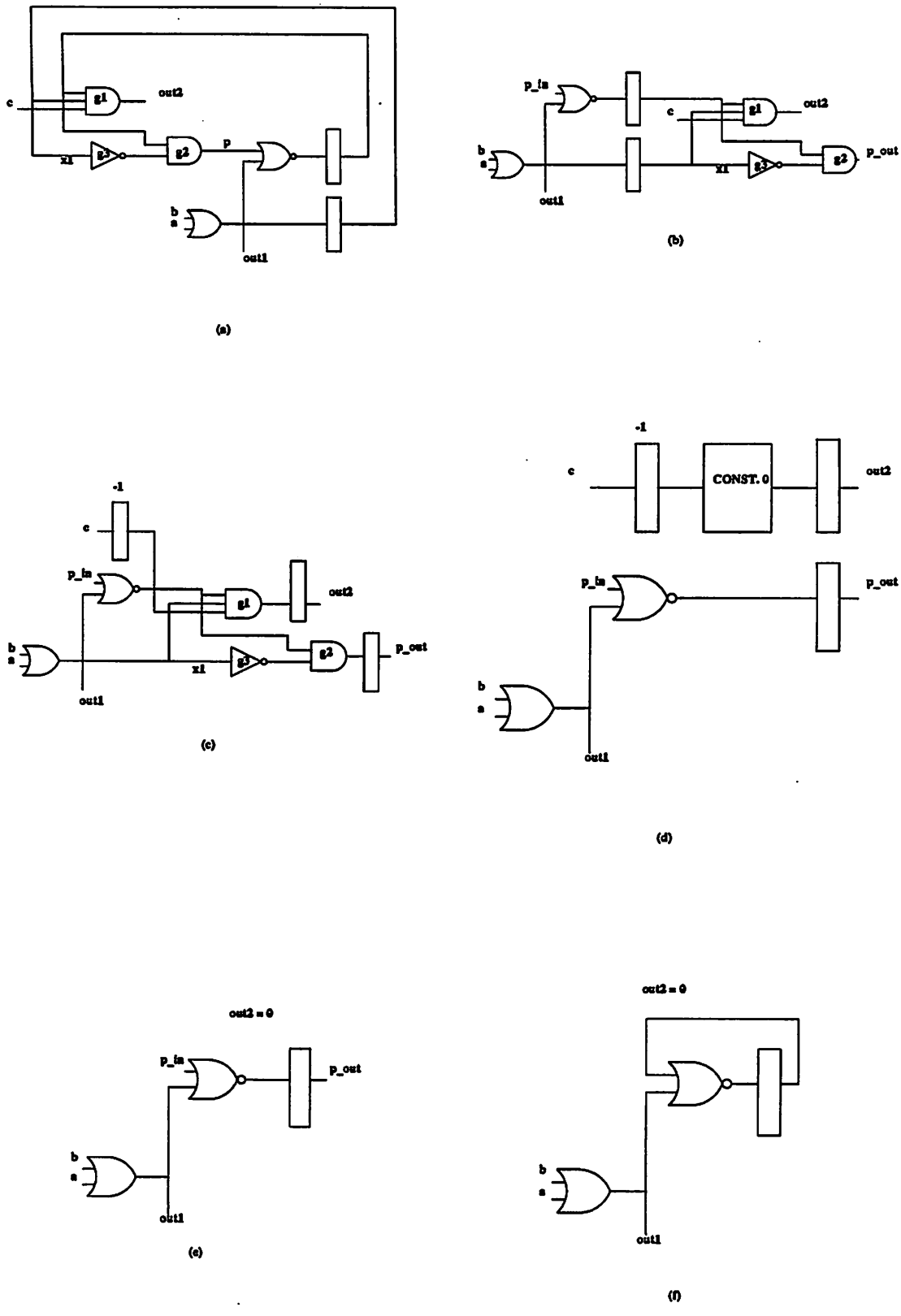


Figure 7: Example FSM Optimization

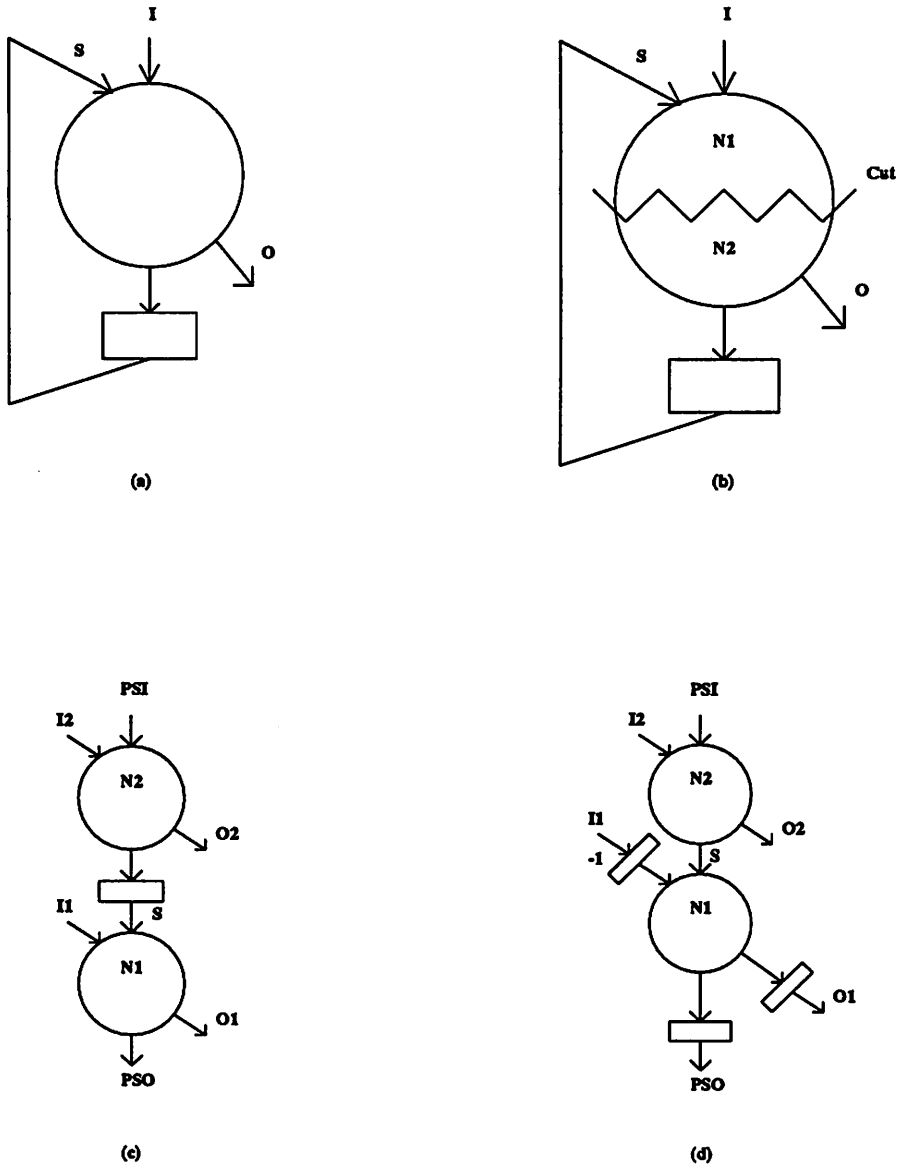


Figure 8: Generic FSM Optimization

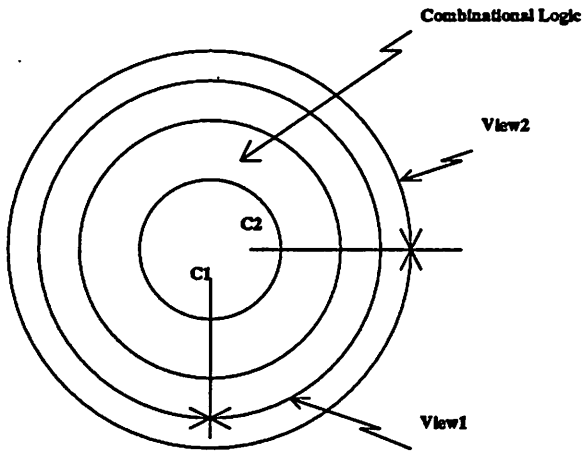


Figure 9: Sliding Window Optimizations

reiterate the importance of permitting the peripheral retiming to allow negative registers at the periphery. Without this, arbitrary cuts would not be possible [3].

3.3 Relationship to Sequential Redundancy

Let us look again at the example in Figure 7. The gate g_1 and the signal x_1 can be shown to be sequentially redundant. It is interesting to note that these were exposed by combinational resynthesis (in this case by checking for combinational irredundancy). In fact, if we start with an initial combinational logic block of the circuit (corresponding to the cut at the registers) that is irredundant, and for some view in the sliding window process the combinational block has a redundancy, then this must be a sequential redundancy in the original sequential circuit. As a result, we have a technique by which sequential redundancy can be removed by removing combinational redundancy. This leads us to ask the more important question: is this method complete? That is, if we make each view of the circuit combinationally irredundant, then is the final FSM implementation sequentially irredundant? We are currently working on resolving this question.

Even though the above argument was given with respect to an FSM implementation, it holds for any sequential circuit. In this case the different views correspond to different sub-networks with satisfiable path weight matrices.

3.4 Relationship to Operations at the STG Level and the State Assignment Problem

The approach outlined in this paper subsumes the traditional single-cut-at-the-registers and conventional retiming approaches towards sequential optimization. We now explore the relationship this has to the third approach, *viz.* operations at the STG level and also to the state assignment problem. The specific question that we would like to answer is: given an implementation M_1 corresponding to a state transition graph G_1, with a state assignment S_1, is it possible to derive a machine M_2 corresponding to an equivalent STG G_2, and a state assignment S_2 by applying only a series of resynthesis and retiming operations on M_1? There need not be a one-to-one correspondence between the states of G_1 and the states of G_2. When machine M_1 is in state s_1, machine M_2 could be in one of several states $s_{21}, s_{22}, \ldots s_{2n}$ (s_1 is said to *correspond* to $s_{21}, s_{22}, \ldots s_{2n}$). We were able to answer this question in the affirmative

[3] Without permitting a negative register, a register cannot be made to migrate across a gate that has a path to a primary output (migration against the signal flow), or across a gate that has a path from a primary input (migration in the signal flow direction). This severely restricts the motion of the registers and the corresponding permissible cuts.

for the case in which G_2 did not have greater observability than G_1. This is clarified through the following formal statements.

Definition 6 *A state transition graph G_1 has a a greater observability than an equivalent state transition graph G_2 if there exists a state s_1 in G_1 that corresponds to two different states s_{21} and s_{22} in G_2.*

Theorem 3 *Given a machine implementation M_1 corresponding to a state transition graph G_1, with a state assignment S_1, it is always possible to derive a machine M_2 corresponding to an equivalent state transition graph G_2, and a state assignment S_2 by applying only a series of resynthesis and retiming operations on M_1 if G_2 does not have greater observability than G_1.*

Note that the theorem statement does not suggest that the retiming and resynthesis cannot increase the observability, but rather that it cannot guarantee that these steps are enough to derive any specified implementation with the increased observability.

A minimal STG is defined as follows:

Definition 7 *A state transition graph, G_1, is minimal if and only if there does not exist an equivalent state graph G_2 in which there exists a state s_2 corresponding to two states s_{11} and s_{12} in G_1.*

If G_1 is a minimal state graph then no other equivalent graph can have greater observability. This leads to the following interesting corollary.

Corollary 1 *Given a machine implementation M_1 corresponding to a minimal state transition graph G_1, with a state assignment S_1, it is always possible to derive a machine M_2 corresponding to an equivalent state transition graph G_2, and a state assignment S_2 by applying only a series of resynthesis and retiming operations on M_1.*

Note the significance of this result. It states that if we start with an implementation corresponding to the minimal state graph, it is always possible to obtain any possible implementation corresponding to an equivalent state graph with any state assignment through a series of retiming and resynthesis steps. In this case it is possible to reach any point in the optimization space using just retiming and resynthesis. However, unlike operations at the STG level, this process operates at the gate level. The cost function here is a more accurate and consistent reflection of the hardware complexity. It is this property that makes this approach more appealing. Unfortunately, the theorems in this section indicate only an existence condition without suggesting a search process. We know that there exists a series of retiming and resynthesis steps that will take us from a given point to another point in the state assignment space, but we do not know how to use this to determine what the best point is.

4 Conclusions

We have presented an approach towards optimizing sequential circuits by considering the maximal sub-circuits for which all registers can effectively be ignored. This permits existing combinational techniques to be used on this sub-circuit. We have presented a complete characterization of these maximal sub-circuits and guarantee a legal retiming at the end of combinational resynthesis. The concept of "borrowing" latches from the environment is an important one, for it extends the class of circuits that we can optimize by this technique. Next, we consider the important problem of optimizing finite state machine implementations. We see how a *Sliding Window* optimization procedure can be used to consider several different combinational blocks corresponding to the same sequential circuit. This permits a greater range of optimization than that permitted by existing techniques. Currently we are implementing these ideas in SIS, the sequential optimization system being developed at U. C. Berkeley.

References

[1] C.E. Leiserson and J.B. Saxe, "Optimizing Synchronous Systems," *Twenty-Second Annual Symposium on Foundations of Computer Science,* IEEE, October 1981, pp. 23-26.

[2] C.E. Leiserson, F.M. Rose, and J.B. Saxe, "Optimizing Synchronous Circuitry by Retiming," *Proceedings of the Third Caltech Conference on VLSI,* March 1983.

[3] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A.R. Wang, "MIS: A Multiple-Level Logic Optimization System", *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, no. 6, Nov. 1987, pp. 1062-1081.

[4] R.K. Brayton, G.D. Hachtel, C.T. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston, 1984.

[5] K.A. Bartlett, D.G. Bostick, G.D. Hachtel, R.M. Jacoby, M.R.Lightner, P.H.Moceyunas, C.R.Morrison and D.Ravenscroft, "BOLD: A Multiple-Level Logic Optimization System", *Proc. IEEE Int. Conf. on CAD*, 1987.