PRESERVING DON'T CARE CONDITIONS DURING RETIMING

by

Ellen M. Sentovich and Robert K. Brayton

Memorandum No. UCB/ERL M91/2

15 January 1991

PRESERVING DON'T CARE CONDITIONS DURING RETIMING

by

Ellen M. Sentovich and Robert K. Brayton

Memorandum No. UCB/ERL M91/2

15 January 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

PRESERVING DON'T CARE CONDITIONS DURING RETIMING

by

Ellen M. Sentovich and Robert K. Brayton

Memorandum No. UCB/ERL M91/2

15 January 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Preserving Don't Care Conditions During Retiming

Ellen M. Sentovich and Robert K. Brayton
Department of Electrical Engineering and Computer Science
University of California, Berkeley, CA 94720

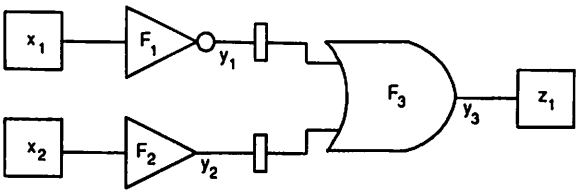
Abstract

The use of computed "don't care" conditions during combinational logic optimization has been shown to be effective in producing smaller circuit implementations. Combinational optimization techniques can be applied in the same fashion to the combinational logic blocks between registers in sequential circuits. Don't care conditions may be specified by the user, computed from the structure of the circuit, or, in the case of sequential circuits, extracted from a given, corresponding state transition graph (STG) of the sequential circuit. These conditions may be time-consuming to recompute, or impossible to re-extract if the information is invalidated by a subsequent modification of the circuit. For this reason, it is important to be able to preserve the don't care information during the application of optimization algorithms. Retiming is a technique in which the cycle time or the number of registers is minimized by determining optimal register positions while preserving the behavior of the circuit. The structure of the logic is unchanged while the registers are moved, making it possible to preserve the don't care information across a retiming operation. While not all the don't care information can be retained, in this paper, a method for preserving the **maximal** subset of the don't care conditions during retiming is proposed.

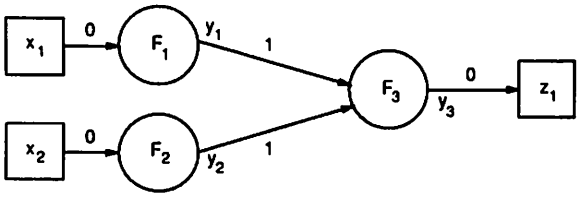
1 Introduction

Recently, sequential circuit optimization has emerged as an important problem in logic synthesis. The techniques used in the more well-understood combinational logic synthesis domain can be applied directly to the combinational logic blocks between registers in a sequential circuit. One such technique is the use of "don't care" information during boolean function simplification [1, 8]. Don't care conditions can be specified externally by the user or computed from the structure of a multi-level network which represents the circuit of interest. These conditions are stored as functions of the inputs and the internal nodes of a combinational logic network, and used to simplify the two-level boolean functions at each node of the multi-level network. The don't care functions associated with a multi-level network can be large and expensive to compute [7], making it desirable to preserve the computed functions after a modification to the network has been made. In addition, the user-specified don't care conditions are impossible to recompute after a modification, and hence it is imperative that this type of don't care is preserved as much as possible. Recently, work has been done to characterize the effects that modifications to a network representing a combinational logic circuit have on the don't care functions that arise from the structure of the network, so that their complete recomputation can be avoided whenever possible [5]. The problem of preserving don't care functions during the application of sequential optimization techniques has not been explored thus far.

Retiming [3, 4] is an operation on a sequential circuit that determines optimal positions for the registers within the circuit such that the cycle time or number of registers is minimized. While the logic functions at each node in the network are unchanged during retiming, the logic blocks, which are composed of interconnected logic nodes between registers and can be thought of as individual combinational logic networks separated by registers, are modified as the register positions change. The set of input and output signals to each block change as the registers at their boundaries are shifted. As a result, preserving the don't care information requires operations involving the inputs and outputs of the block: the don't care functions within the block must be re-expressed in terms of the new inputs to the block, and don't care information for the new outputs must be computed based on the don't care information for the old outputs before the old outputs are shifted to a new block and this information is lost. These two operations represent the minimum amount of computation that must be done to preserve the existing don't care information. While the don't care information might not be complete for each node following these operations (e.g., the new information at the outputs may need to be propagated back through the network), the complete set of don't care conditions can be computed from the preserved information in the block.



(a) Sequential Circuit



(b) Circuit Graph

Figure 1: Sequential Circuit Representation

In the sequel, a method is proposed for maximally preserving the don't care information across a retiming operation. In Section 2, some basic definitions are given involving sequential circuits and boolean functions. The formulation for preserving the don't cares is developed in Section 3, and the current directions for this work given in Section 4.

2 Background

A sequential circuit is modeled by a directed graph where each vertex i represents either

a) a primary input x_i $(i = 1, \ldots, p)$

b) a primary output z_i $(i = 1, \ldots, p)$ or

c) a variable y_i and a representation F_i of a logic function $(i = 1, \ldots, m)$.

An edge connects vertex i to vertex j if the function associated with vertex j, F_j depends explicitly on the variable y_i. Each edge e has a nonnegative integer label $w(e)$ representing the number of registers between the two logic gates it connects. Each cycle in the graph must contain at least one edge of strictly positive weight (this restriction is placed to model synchronous circuits only, thereby avoiding asynchronous problems such as race conditions). A sequential circuit is shown in Figure 1(a), with its graph in Figure 1(b). The terms circuit, network, and graph are used interchangeably whenever there is no ambiguity, as are the terms node and vertex.

2.1 Retiming

Retiming is an operation on a sequential circuit whereby registers are moved across logic gates in order to minimize the clock cycle or the number of registers while maintaining the behavior of the circuit. Retiming algorithms were first proposed by Leiserson *et al* [3, 4]. The movement of registers can be quantified by an integer $r(v)$ for each vertex v, which represents the number of registers that are to be moved in the graph from each out-edge of vertex v to each of its in-edges. The resulting edge weight for an edge from vertex u to vertex v is $w_r(e) = w(e) + r(v) - r(u)$.

A **legal retiming** is the assignment of an integer $r(v)$ to each vertex such that the resulting edge weights are all nonnegative. A legal retiming has been proven to generate a circuit that is functionally equivalent to the original circuit [3]. The circuit shown in Figure 1 can be retimed by selecting $r(F_3) = -1$ and $r(v) = 0$ for all other vertices. The resulting retimed circuit is shown in Figure 2.

In [4] the external interface to a synchronous circuit is modeled by a single node called the host node (v_h), to which all inputs and outputs are connected. While any legal retiming may have $r(v_h) \neq 0$ (implying a change in the external interface which is not allowed), all legal retimings with $r(v_h) \neq 0$ have equivalent legal retimings with $r(v_h) = 0$. For example,

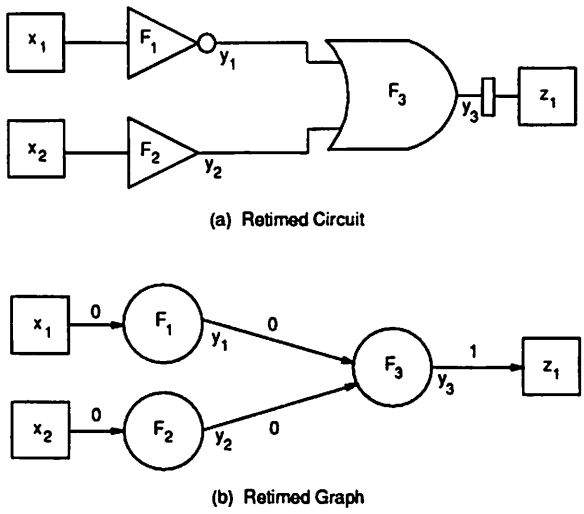


Figure 2: Retimed Sequential Circuit

the graph in Figure 3(a) can be retimed to produce that of Figure 3(b) by either assigning $r(v_h) = 1$ and $r(F_1) = 0$ or by equivalently assigning $r(v_h) = 0$ and $r(F_1) = -1$. In this paper, a retiming is always assumed to be a legal retiming with $r(v) = 0$ for all vertices v that represent I/O pins.

During a legal retiming, the registers move in a limited fashion. Lemma 1 in [4] asserts that $w_r(p) = w(p) + r(v) - r(u)$, where p is a path from u to v in the circuit, and $w(p)$ is the sum of the edge weights along that path. Since $r(v) = 0$ for all I/O pins, the weight of each path from an input pin to an output pin is unchanged during retiming. Given a particular circuit with its associated retiming, a register on a particular path can be mapped to a new position on that path in the retimed circuit because the total path weight is unchanged during retiming. As a result, each register in the circuit can be thought of as having moved backward (to some edge in its transitive fanin) or forward (to some edge in its transitive fanout). This classification of the movement of registers during retiming is essential to the method for preserving don't cares across retiming presented in Section 3.

2.2 Don't Care Sets

Three different types of don't care sets have been introduced for multi-level logic optimization [1]:

External DC set is comprised of user-specified don't care conditions, such as input combinations that never occur, or input combinations that occur but are not important or have no effect on the outputs. In general, each output j may have a different external DC set, denoted by d_j. A property of the external don't care set, not often explicitly stated but assumed by logic optimization programs, is that each don't care for each output must remain a don't care independent of how the other don't cares at other outputs are used. This arises from the way this information is used in logic optimization. A set of don't care functions is called "compatible" if it has this independence property.

Satisfiability DC set (SDC) arises because variables at the intermediate vertices of the graph are not independent of the primary inputs, and represents all the inconsistent conditions in the network. The SDC is given by the following equation:

$$SDC = \sum_{i=1}^{m}(y_i \neq F_i) = \sum_{i=1}^{m}(\bar{y}_i F_i + y_i \bar{F}_i)$$

Observability DC set (ODC) is defined for each vertex v_i. It is derived by looking at the fanout of the variable y_i and seeing how specifically y_i is used. In [6], two sets based on observability don't care conditions were defined: MSPF's (maximum sets of permissible functions), which include for each node the on-set of the node function and the largest possible set of observability don't cares for that node, and CSPF's (compatible sets of permissible functions), which include the on-set of the node function and a maximal set of compatible observability don't cares for that node, which

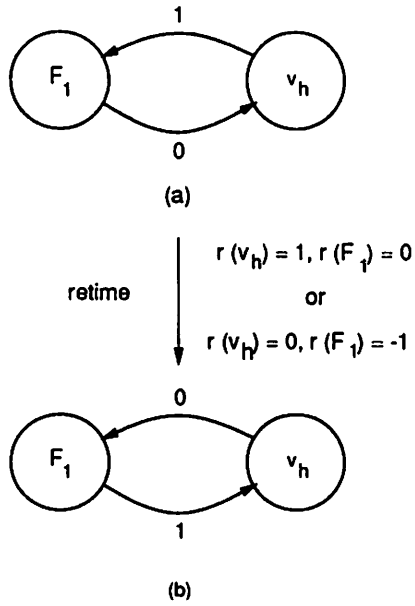


Figure 3: Any legal retiming can be specified with $r(v_h) = 0$

means that the don't cares can be used simultaneously in simplifying all the nodes without the need for recomputing the don't cares at any other nodes. These ideas were extended in [8], applied to general multi-level networks, and algorithms for computing a maximal set of CSPF's were given. Beginning at the outputs with the external don't cares and working towards the inputs, CSPF's are computed at each node and can be used in the simplification of that node simultaneously with simplifying other nodes.

2.3 Smoothing, Consensus and Observability

The smoothing operator S for a boolean function f is defined as follows:

$$S_x f = f_x + f_{\bar{x}}$$

where f_x is the function f evaluated at $x = 1$ (which is the cofactor of f with respect to x, and is sometimes denoted $f\,|_{x=1}$), and $f_{\bar{x}}$ is the function f evaluated at $x = 0$ (the cofactor of f with respect to $\bar{x}$). The smoothing operator can be interpreted as producing a function that is 1 when f is 1 for either value of x; that is, $S_x f$ is a function that is 1 whenever there exists a value of x such that f is true (i.e., it is the existential quantifier). It is the minimum set containing f and independent of x. Smoothing a function f with respect to a set of inputs $I = \{x, y, z\}$ is denoted $S_x S_y S_z f$ or simply, $S_I f$.

The consensus operator C for a boolean function f is defined as follows:

$$C_x f = f_x \cdot f_{\bar{x}}$$

The consensus operator can be interpreted as producing a function that is 1 when f is 1 for both values of x (i.e., it is the universal quantifier). It represents the maximum set contained by f and independent of x. Taking the consensus of a function f with respect to a set of inputs $I = \{x, y, z\}$ is denoted $C_x C_y C_z f$ or simply, $C_I f$.

An input signal x to a particular gate f is said to be **observable** at f if $\frac{\delta f}{\delta x} = 1$, where

$$\frac{\delta f}{\delta x} = f_x \oplus f_{\bar{x}}$$

That is, the signal x is observable at f if setting x to 1 gives a different result at f than setting x to 0.

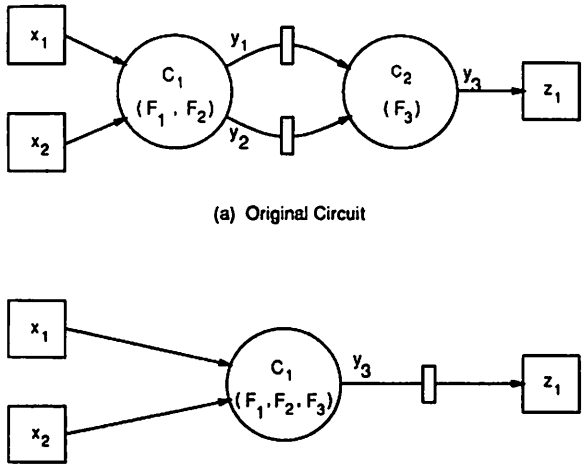


Figure 4: Logic Block View of a Sequential Circuit

3 Preserving Don't Cares after Retiming

A sequential circuit can be viewed as a set of combinational logic blocks separated by registers. The inputs to each block are primary inputs of the circuit and some register outputs, and the outputs of each block are primary outputs of the circuit and some register inputs. The don't care function for each node in a particular block, if it has been computed, is expressed in terms of the inputs to the block and internal logic nodes within the block. The logic block view of the sequential circuit in Figure 1(a) is shown in Figure 4(a). The block C_1 has as inputs the primary inputs x_1 and x_2, and outputs the signals y_1 and y_2 which are register inputs. The don't care functions for the nodes in block C_1 will depend on x_1, x_2, and the internally generated signals y_1 and y_2.

After the circuit is retimed, the registers acting as boundaries of the block may have been moved forward or backward (the new position of a register in a retimed circuit is on an edge in the transitive fanin or the transitive fanout of the original register position, see Section 2 for an explanation). As a result, the block has different inputs and outputs, and the don't care functions within the block must be updated to reflect this change.

If the registers at the outputs of the block have been moved forward, the block contains new internal nodes and outputs, and the don't care functions for these new nodes should be updated before the nodes are simplified. These new nodes bring with them don't care functions from their previous block which should be updated to include don't care information from the current block. For example, if the circuit represented in Figure 4(a) is retimed to produce that in Figure 4(b) in which the two registers have been moved forward across block C_2, the node F_3 becomes a new node of block C_1. It has a don't care function from block C_2 which can be augmented by propagating don't care information forward from the nodes in block C_1. If the registers at the outputs have been moved backward, internal nodes within the block become new outputs, and don't care information from the old outputs in addition to don't cares based on structural information between the old outputs and new outputs must be passed to the new outputs before the old outputs become part of a different logic block. If the circuit represented in Figure 4(b) is retimed to produce that in Figure 4(a), the register is moved back across node F_3 and the block C_1 has new outputs, signals y_1 and y_2. Don't care information from node F_3 should be propagated back to nodes F_1 and F_2 before node F_3 becomes a part of block C_2 and block C_1 loses access to that information.

If the registers at the inputs of a block are moved, the don't care functions at each node within the block must be re-expressed in terms of the new input signals. In addition, the don't care functions for the nodes that change blocks must be updated. If the registers at the inputs to a block are moved backward, new nodes become part of the block, and the don't care functions for these nodes should be updated before they are simplified. These new nodes have don't care functions that are based on information from the previous block; the functions should be augmented with information from the current block. Similarly, if the registers at the inputs are moved forward, some nodes are shifted out of the block. Don't care information for these nodes should be computed based on information from the current block before the nodes are shifted to the next block.

Thus, there are two operations needed for updating the don't care information in a block: re-expressing the don't care functions in the block in terms of the new inputs, and propagating don't care functions between nodes within a block.

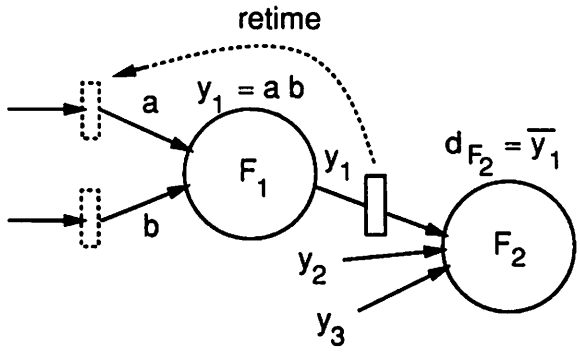


Figure 5: Inputs change as the register moves backward

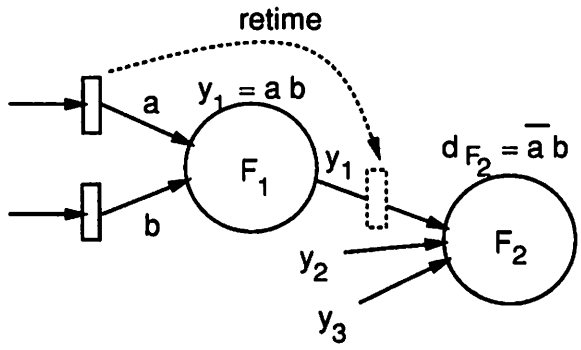


Figure 6: Inputs change as the registers move forward

3.1 Re-expressing Don't Care Functions

If the registers at the inputs to a particular block are moved, the don't care functions within that block must be re-expressed in terms of the new inputs so these functions can be used in a meaningful way by the node simplification algorithm. This function re-expression operation can be divided into two cases: one in which the registers are moved backward, and one in which the registers are moved forward.

If the registers are moved backward, the old inputs become internal nodes. The node simplification algorithm may allow the don't care functions to be expressed in terms of internal nodes, in which case no modification is necessary. (In MIS [2], the SDC is used in tandem with the ODC and external don't cares during node simplification, so there is no need to express the don't cares in terms of inputs to the logic block.) If the node simplification algorithm requires that the don't cares be functions of the primary inputs, some modification must be made. When the registers move backward, the old inputs are functions of the new inputs and the don't care functions can be re-expressed by collapsing the logic between the old and new inputs. This operation is illustrated by the circuit represented in Figure 5. The don't care function for node F_2 is expressed as $d_{F_2} = \bar{y}_1$. If node F_1 is retimed to move the register to its inputs, $r(F_1) = 1$, then the don't care function at node F_2, $d_{F_2} = \bar{y}_1$, can be re-expressed as $d_{F_2} = \overline{ab} = \bar{a} + \bar{b}$.

The case in which the registers at the inputs are moved forward requires more consideration. The don't care functions must be re-expressed in terms of the new inputs, which are functions of the old inputs. For example, in the circuit represented in Figure 6, node F_1 is retimed to move the registers from its inputs to its output, $r(F_1) = -1$. The function d_{F_2} must be expressed in terms of y_1 rather than a and b, since a and b are no longer inputs to the block containing F_2.

This re-expression problem can be formulated in a more general way (see Figure 7). Given a function f, with inputs $S = \{s_1, s_2, \ldots, s_m\}$ and $T = \{t_1, t_2, \ldots, t_n\}$, and a set of functions represented by the signals $Y = \{y_1, y_2, \ldots, y_p\}$, which are each functions of S, the problem is to express f as a function of the signals in Y and T, that is:

Given: $f(S,T)$ and $Y(S)$

Determine: $f_{new}(Y,T)$

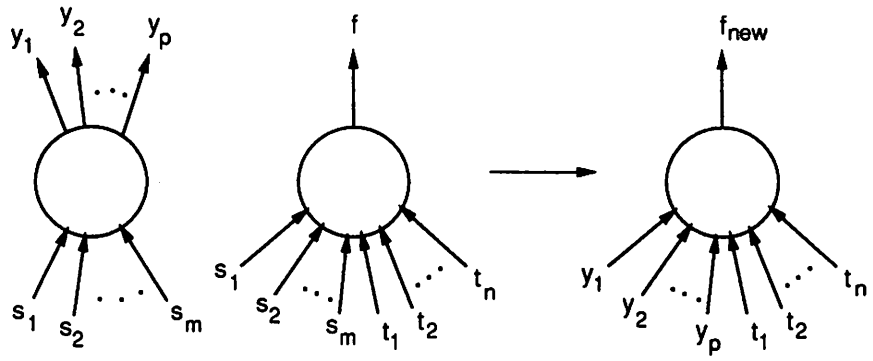
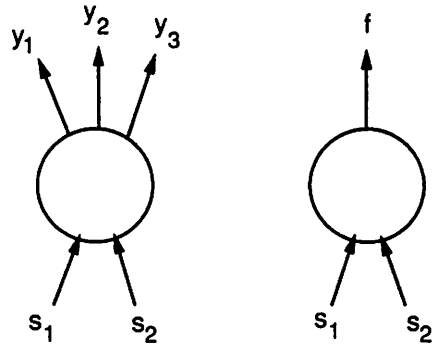


Figure 7: Re-expressing a Function



s_1	s_2	y_1	y_2	y_3	f
0	0	0	0	0	1
0	1	1	0	1	1
1	0	1	0	1	0
1	1	0	1	1	0

Figure 8: Example: Re-expressing f in terms of y_1, y_2, y_3

In relation to retiming, the signals in S represent all the register outputs for the registers that have been moved forward while the signals in T represent primary inputs to the block or register outputs for registers that have not been moved or have been moved backward. The signals in Y represent the new positions of the registers that have been moved forward.

In general, a function f_{new} which behaves identically to the function f may not exist because the information carried by the signals in S may not be completely captured by the signals in Y, i.e., there is not, in general, a one-to-one mapping from the information carried by the signals S to the information carried by the signals Y. The construction of f_{new} for re-expressing the don't care function f involves retaining the maximal amount of information from f and including information obtained from the SDC set for the signals in Y as a function of S. Let SDC_Y be the SDC for the signals in Y:

$$SDC_Y = y_1 \cdot \bar{f}_{y_1} + \bar{y}_1 \cdot f_{y_1} + y_2 \cdot \bar{f}_{y_2} + \bar{y}_2 \cdot f_{y_2} + \ldots + y_p \cdot \bar{f}_{y_p} + \bar{y}_p \cdot f_{y_p}$$

Then f_{new} is constructed as follows:

$$f_{new} = C_S\,(f + SDC_Y) \tag{1}$$

The consensus produces the **maximal** subset of $f + SDC_Y$ that is independent of S. The conditions on Y that set f_{new} to 1 are those that set $f + SDC_Y$ to 1 for **all** values of S. This conservative approach to constructing f_{new} may result in the loss of some don't care conditions as the next example illustrates. Note that the loss of the don't care conditions is inherent to the problem; the loss is due to the fact that the signals in S carry more information than the signals in Y, and is not due to some limitation of this method for re-expressing the functions.

An example of this computation is given in in Figure 8, where the functionality of the circuit is given by a truth table. The goal is to represent f in terms of y_1, y_2, and y_3 rather than s_1 and s_2. In this example, $y_1 = s_1 \oplus s_2$, $y_2 = s_1 s_2$, $y_3 = s_1 + s_2$, and $f = \bar{s}_1$. f_{new} is computed from Equation 1 as

$$f_{new} = y_1 y_2 + \bar{y}_1 \bar{y}_2 + \bar{y}_3$$

Some of the conditions in f_{new} arise from the SDC alone, and correspond to combinations of the Y values that can never occur: $y_1 y_2$, $\bar{y}_1 \bar{y}_2 y_3$, $\bar{y}_1 y_2 \bar{y}_3$, and $y_1 \bar{y}_2 \bar{y}_3$. One condition arises from the function f: $\bar{y}_1 \bar{y}_2 \bar{y}_3$. Finally, note that the condition

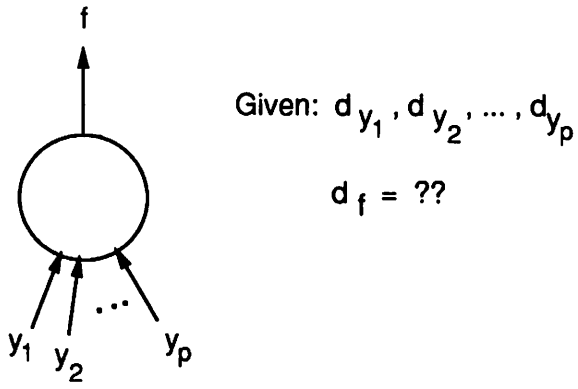


Figure 9: Propagating Don't Cares Forward

$y_1\bar{y}_2y_3$ is not included in f_{new}. This condition evaluates to 1 in f for one assignment of values to S, but evaluates to 0 in f for another assignment of values to S, where both assignments produce the same assignment for Y. When f is a don't care function, the conservative approach is taken which includes assignments to Y in f_{new} only if such assignments are don't cares in f for **all** corresponding assignments to S.

3.2 Propagating Don't Care Functions

As the registers are moved during a retiming operation, the structure of the logic blocks change, causing some nodes to become part of a new block. Such nodes may not have don't care functions associated with them, in which case these functions should be computed based on the other don't care functions in the block before the nodes are moved to other blocks. Once they are in their new blocks, the don't care functions should be augmented with don't care information from the new blocks. This is done by propagating don't care information either forward or backward within the block to the nodes of interest.

3.2.1 Propagating Don't Cares Backward

Techniques for propagating don't care information backward, that is, from a node to its fanin nodes, have been developed in [8]. In the algorithm described in that paper, the computation begins with external don't care information expressed for the outputs, and propagates that information, along with observability don't care information extracted during the computation, backward to the primary inputs. In [8], two formulae were given: one for propagating don't cares from the output of a node to each of its inputs, and another for computing the don't cares of the node based on the don't cares of all of its fanout nodes. Those formulae were proven to generate **maximal compatible** observability don't cares. It is important that these functions are compatible, so that they can be used simultaneously during node simplification.

The same computation as that given in [8] can be used to propagate don't care information backward after retiming. Furthermore, the fact that they are maximal and compatible implies that after retiming, the don't cares obtained can still be simultaneously used during node simplification, and that after retiming, the maximal don't care set with respect to the don't cares given for the unretimed circuit and such that the don't cares are still compatible, has been retained.

3.2.2 Propagating Don't Cares Forward

The problem of propagating don't care information forward is illustrated in Figure 9. In that network, f is a function of $y_1, y_2, \ldots, y_p$. The don't care functions are known for the functions y_i and denoted d_{y_i}, and the task is to compute an appropriate don't care function for f.

A don't care condition for f arises for a particular variable assignment in the network if that assignment produces the value 0(1) for f, while the same assignment, modified in such a way that the new assignment is in the don't care set, produces the value 1(0) for f. An example is illustrated in Figure 10. Suppose the don't care function at y_5 is $d_{y_5} = y_1\bar{y}_2$. In that case, the cube $y_1\bar{y}_2y_3y_4$ is a don't care for f: $y_1\bar{y}_2y_3y_4$ results in $y_5 = 0$, $y_6 = 1$, and $f = 0$, and since $y_1\bar{y}_2$ is a don't care for y_5, the value 1 can be assigned to y_5 in which case $f = 1$.

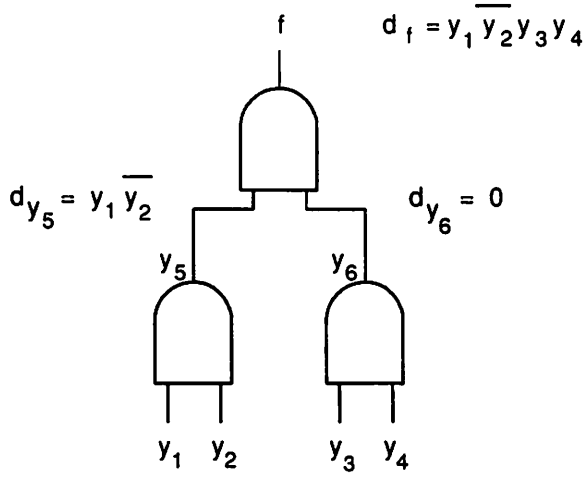


Figure 10: Example: Propagating Don't Care Functions Forward

The don't care function for a node is computed based on the the don't care functions at its inputs and the observability of the inputs. Therefore, the following definition of an observable set will aid in developing a formula for propagating don't cares forward.

Definition 1 *Let f be a Boolean function of n input variables in the set $Y = y_1, y_2, \ldots, y_n$, and let $\mathbf{y}^{\mathbf{l}} = [y_1, y_2, \ldots, y_n] \in B^n$ be some assignment of values to the input variables ($f(\mathbf{y}^{\mathbf{l}}) \in B$). Let G be a subset of Y, $G \subseteq Y$, $\mathbf{g}^{\mathbf{l}}$ an assignment to the variables in G, H be the complement of G, $H = Y - G$, and $\mathbf{h}^{\mathbf{k}}$ an assignment to the variables in H. G is an* **observable set** *with respect to f if and only if*

$$\exists\, \mathbf{g}^{\mathbf{l}}, \mathbf{g}^{\mathbf{j}}, \mathbf{h}^{\mathbf{k}} \; s.t. \; f(\mathbf{g}^{\mathbf{l}}, \mathbf{h}^{\mathbf{k}}) \neq f(\mathbf{g}^{\mathbf{j}}, \mathbf{h}^{\mathbf{k}})$$

$\mathbf{h}^{\mathbf{k}}$ is the condition under which the set is observable. The following Lemma makes use of Definition 1.

Lemma 1 *Given $Y = y_1, y_2, \ldots, y_n$, the set of input variables for a function f, and G, a subset of Y, G is an observable set at f under the conditions given by $S_G f \cdot S_G \bar{f}$.*

Proof. By definition, $S_G f$ gives the conditions under which there exists an assignment to the variables in G such that $f = 1$, i.e. it gives the conditions, $\mathbf{h}^{\mathbf{i}}$ s.t. $\exists\, \mathbf{g}^{\mathbf{1}}, f(\mathbf{g}^{\mathbf{1}}, \mathbf{h}^{\mathbf{i}}) = 1$. Similarly, $S_G \bar{f}$ gives the conditions under which there exists an assignment to the variables in G such that $f = 0$. The product of the two smoothing terms gives the conditions on the variables not in G such that there exists an assignment to the variables in G that sets f to 1, and another assignment to the variables in G that sets f to 0. By definition, each condition in $S_G f \cdot S_G \bar{f}$ is a condition under which G is an observable set at f. □

In fact, it can be shown that the observability condition given in Section 2 applied to several inputs of the function f also generates conditions under which a set is observable, and is a subset of the conditions given by Lemma 1.

Corollary 1 *Let Y be a set of elements, $Y = \{y_1, y_2, \ldots, y_n\}$, in which each element y_i represents a boolean variable in the function f. Then*

$$S_Y f \cdot S_Y \bar{f} \supseteq \frac{\delta^n f}{\delta y_1 \delta y_2 \cdots \delta y_n} \tag{2}$$

and hence $\frac{\delta^n f}{\delta y_1 \delta y_2 \cdots \delta y_n}$ generates conditions under which the set Y is observable.

Proof. See Appendix. □

The don't care conditions can be moved forward across a node according to the following theorem.

Theorem 1 *Given a Boolean function f, with n inputs, $Y = y_1, y_2, \ldots, y_n$, each with associated don't care conditions, $D = d_{y_1}, d_{y_2}, \ldots, d_{y_n}$, the* **maximal** *don't care function for f based only on the don't care conditions at the inputs, d_f, can*

be expressed as

$$d_f = \sum_{G \subseteq Y} \left(\prod_{g \in G} d_g \right) \left(\mathcal{S}_G f \cdot \mathcal{S}_G \bar{f} \right) \tag{3}$$

Proof. The proof has two parts: first it must be shown that all the conditions produced by Equation 3 are don't care conditions for f, and second, that this set of don't cares is maximal. For the first part, note that each term in the summation of Equation 3 is comprised of the intersection of don't care conditions that are common to a subset of input signals with the conditions under which that subset is an observable set with respect to f. Each of the resulting conditions can be considered a don't care for f because 1) the don't care condition at the set of inputs, $\prod_{g \in G} d_g$, allows any assignment of values to those inputs, and 2) the observability set condition $\mathcal{S}_G f \cdot \mathcal{S}_G \bar{f}$ guarantees that there exists two assignments to g, one that results in $f = 1$ and the other that results in $f = 0$. To prove that the set of don't cares produced is maximal, suppose there is a don't care condition d on f that is not included in Equation 3. d must arise from one or more of the input don't cares d_{y_i}, since that is the only don't care information given about the circuit for this problem (the function f may be embedded in a larger circuit giving rise to don't cares based on this structure, but these need not be considered for the problem of simply propagating don't cares forward). d also must be observable at the output f in order to be a don't care for f. Finally, regardless of which input(s) the don't care condition arises from, it will be transferred to f provided it is both observable and contained in some subset of the inputs. Therefore, d must be contained by d_f. □

Equation 3 can only be applied when the given don't cares at the inputs are CSPF's. The reason for this is that the don't cares at the inputs are used **simultaneously** in the computation of Equation 3. When using MSPF's, the don't care functions can only be used one at a time, and the remaining don't care functions must be updated after using a particular don't care function during simplification. For example, suppose $f = y_1 y_2$. the CSPF calculation would yield $d_{y_1} = \bar{y}_2$, $d_{y_2} = \bar{y}_1 y_2$, while the MSPF calculation would yield $d_{y_1} = \bar{y}_2$, $d_{y_2} = \bar{y}_1$. Applying Equation 3 using CSPF's produces $d_f = 0$, while using MSPF's produces $d_f = \bar{y}_1 \bar{y}_2$ (which is erroneous because there is no condition in either the specification or the structure that indicates that $\bar{y}_1 \bar{y}_2$ should be a don't care for f).

Considering all inputs together, note that

$$\begin{aligned} \mathcal{S}_Y f \cdot \mathcal{S}_Y \bar{f} &= 1 \; if \; f \not\equiv 1 \; and \; f \not\equiv 0 \\ &= 0 \; if \; f \equiv 1 \; or \; f \equiv 0 \end{aligned}$$

This illustrates the one case in which some don't care information is lost when applying Equation 3. Consider a function f which is identically 1 (or identically 0), and suppose that it has a don't care function, d_f, which has been computed by propagating CSPF's backward from the output nodes to that node. The CSPF's for the inputs to node f will all contain the function d_f. If the don't care information at the inputs is now propagated forward to the output f, this don't care information will be lost. Equation 3 produces 0 whenever f is 0 or 1 (because there is no assignment that sets f to the opposite value). As a result, don't care information can be lost when f is a constant function, but this information is also not important for a constant function since the node simplification algorithm should be able to detect that it is a constant, and simplify it accordingly.

For the circuit in Figure 10,

$$\begin{aligned} d_f &= d_{y_5} \left(\mathcal{S}_{y_5} f \cdot \mathcal{S}_{y_5} \bar{f} \right) + d_{y_6} \left(\mathcal{S}_{y_6} f \cdot \mathcal{S}_{y_6} \bar{f} \right) + d_{y_5} d_{y_6} \left(\mathcal{S}_{y_5} \mathcal{S}_{y_6} f \cdot \mathcal{S}_{y_5} \mathcal{S}_{y_6} \bar{f} \right) \\ &= y_1 \bar{y}_2 \left(\mathcal{S}_{y_5} \left(y_5 y_6 \right) \cdot \mathcal{S}_{y_5} \left(\bar{y}_5 + \bar{y}_6 \right) \right) \\ &= y_1 \bar{y}_2 y_6 \\ &= y_1 \bar{y}_2 y_3 y_4 \end{aligned}$$

4 Conclusions

A method has been proposed for preserving a **maximal** set of don't care conditions across a retiming operation on a sequential circuit. Thus far, the problem of preserving such conditions across retiming has not been addressed in the literature, and hence has been identified as a new problem. In addition, it is important that the proposed don't care sets are **maximal** since these don't cares are beneficial in subsequent synthesis algorithms, and since some don't care conditions cannot be recomputed once they are lost.

The methods given are currently being implemented and tested, and a suitable method for evaluating these techniques is being investigated. Certainly, better results will be obtained for retaining don't care information and using it during subsequent operations than not retaining it; simply generating a table of benchmark circuits with resulting literal counts would not provide any insight into the applicability of these methods. A more appropriate investigation would involve determining how much of the don't care set is lost after retiming, and how much this lost portion can affect the final implementation. These issues, as well as further exploration of the sources of sequential don't cares, are being investigated.

As a by-product of the technique proposed in this paper, a method was discovered for moving don't care conditions forward in a network (methods for moving them backward are already known, and given in, e.g., [8]). As a result, don't care conditions may be specified for any portion of a sequential circuit, and propagated both forward and backward to other parts of the circuit. Don't cares can be moved across latches by adjusting the don't care conditions appropriately (i.e., given don't cares on latch outputs that are functions of other latch outputs and primary inputs, such conditions can be transferred to the latch inputs after taking the consensus of the don't care functions with respect to the primary inputs; the consensus will produce the maximal subset of the don't care function that is independent of the primary inputs). One issue that is being explored is the relationship of the don't cares produced by simply sweeping through the network and computing the observability don't cares, and those produced by sweeping both backward and forward through the network. In both cases, some external don't cares may be given for some parts of the network.

Some work has been done on computing the new initial state of a retimed circuit [9]. This work is based on extracting a portion of the state transition graph for a sequential circuit to determine a single state that is equivalent to the initial state in the unretimed circuit. The techniques proposed in this paper can be extended slightly to recompute the initial state. In particular, an initial state, or a set of initial states, is expressed in terms of the latch outputs in the circuit (similar to the don't care functions, which are expressed in terms of the latch outputs and the primary inputs). The techniques presented in this paper can be extended to generate a new set of initial states given the initial states for the unretimed circuit.

In the complete paper, an outline for the algorithm for applying these methods will be given. Conclusions will be drawn about the trade-offs between retiming existing don't care conditions and completely recomputing the don't cares, as well as about the limitations of retiming don't cares in terms of losing some don't care conditions.

References

[1] Karen A. Bartlett, Robert K. Brayton, Gary D. Hachtel, Reily M. Jacoby, Christopher R. Morrison, Richard L. Rudell, Alberto Sangiovanni-Vincentelli, and Albert R. Wang. Multilevel Logic Minimization Using Implicit Don't Cares. *IEEE Transactions on Computer-Aided Design*, 7(6):723–740, June 1988.

[2] Robert K. Brayton, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert R. Wang. MIS: A Multiple-Level Logic Optimization System. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1062–1081, November 1987.

[3] C. E. Leiserson and J. B. Saxe. Optimizing Synchronous Systems. *Journal of VLSI and Computer Systems*, 1(1):41–67, Spring 1983.

[4] C. E. Leiserson and J. B. Saxe. Retiming Synchronous Circuitry. In *TM 372, MIT/LCS, 545 Technology Square, Cambridge, Massachusetts 02139*, October 1988.

[5] Patrick McGeer and Robert K. Brayton. Consistency and Observability Invariance in Multi-Level Logic Synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 426–429, November 1989.

[6] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney. The Transduction Method - Design of Logic Networks Based on Permissible Functions. In *IEEE Transactions on Computers*, October 1989.

[7] Alexander Saldanha, Albert Wang, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Multi-Level Logic Simplification using Don't Cares and Filters. In *Proceedings of the Design Automation Conference*, pages 277–282, 1989.

[8] Hamid Savoj and Robert K. Brayton. The Use of Observability and External Don't Cares for the Simplification of Multi-Level Networks. In *Proceedings of the Design Automation Conference*, pages 297–301, 1990.

[9] Herve Touati and Robert Brayton. Computing the Initial States of Retimed Circuits. Unpublished Manuscript.

Appendix

Theorem 2 *Let Y be a set of elements, $Y = \{y_1, y_2, \ldots, y_n\}$, in which each element y_i represents a boolean variable in the function f. Then*

$$\mathcal{S}_Y f \cdot \mathcal{S}_Y \bar{f} \supseteq \frac{\delta^n f}{\delta y_1 \delta y_2 \cdots \delta y_n} \tag{4}$$

Proof. The proof is constructed using induction on the number of elements in the set Y. For $n = 1$,

$$\begin{aligned} \mathcal{S}_{y_1} f \cdot \mathcal{S}_{y_1} \bar{f} &= \left(f_{y_1} + f_{\bar{y}_1}\right)\left(\bar{f}_{y_1} + \bar{f}_{\bar{y}_1}\right) \\ &= f_{\bar{y}_1}\bar{f}_{y_1} + f_{y_1}\bar{f}_{\bar{y}_1} \end{aligned}$$

$$\begin{aligned} \frac{\delta f}{\delta y_1} &= f_{y_1} \oplus f_{\bar{y}_1} \\ &= f_{y_1}\bar{f}_{\bar{y}_1} + \bar{f}_{y_1} f_{\bar{y}_1} \end{aligned}$$

So for $n = 1$, $\mathcal{S}_{y_1} f \cdot \mathcal{S}_{y_1} \bar{f} = \frac{\delta f}{\delta y_1}$. Next assume Equation 4 is true for $n-1$ elements. That is, assume

$$\mathcal{S}_{Y-y_n} f \cdot \mathcal{S}_{Y-y_n} \bar{f} \supseteq \frac{\delta^{n-1} f}{\delta y_1 \delta y_2 \cdots \delta y_{n-1}}$$

To prove Equation 4 is true for n elements, we first smooth with respect to y_n:

$$\mathcal{S}_{y_n}\left(\mathcal{S}_{Y-y_n} f \cdot \mathcal{S}_{Y-y_n} \bar{f}\right) \supseteq \mathcal{S}_{y_n}\left(\frac{\delta^{n-1} f}{\delta y_1 \delta y_2 \cdots \delta y_{n-1}}\right)$$

Using the fact that $\mathcal{S}_y f \cdot \mathcal{S}_y g \supseteq \mathcal{S}_y (f \cdot g)$ results in

$$\mathcal{S}_{y_n}\left(\mathcal{S}_{Y-y_n} f\right) \cdot \mathcal{S}_{y_n}\left(\mathcal{S}_{Y-y_n} \bar{f}\right) \supseteq \mathcal{S}_{y_n}\left(\mathcal{S}_{Y-y_n} f \cdot \mathcal{S}_{Y-y_n} \bar{f}\right) \supseteq \mathcal{S}_{y_n}\left(\frac{\delta^{n-1} f}{\delta y_1 \delta y_2 \cdots \delta y_{n-1}}\right)$$

Finally, using the fact that $y + z \supseteq y \oplus z$,

$$\begin{aligned} \mathcal{S}_{y_n}\left(\mathcal{S}_{Y-y_n} f\right) \cdot \mathcal{S}_{y_n}\left(\mathcal{S}_{Y-y_n} \bar{f}\right) &\supseteq \left(\frac{\delta^{n-1} f}{\delta y_1 \delta y_2 \cdots \delta y_{n-1}}\right)_{y_n} + \left(\frac{\delta^{n-1} f}{\delta y_1 \delta y_2 \cdots \delta y_{n-1}}\right)_{\bar{y}_n} \\ &\supseteq \left(\frac{\delta^{n-1} f}{\delta y_1 \delta y_2 \cdots \delta y_{n-1}}\right)_{y_n} \oplus \left(\frac{\delta^{n-1} f}{\delta y_1 \delta y_2 \cdots \delta y_{n-1}}\right)_{\bar{y}_n} \\ &\supseteq \frac{\delta}{\delta y_n}\left(\frac{\delta^{n-1} f}{\delta y_1 \delta y_2 \cdots \delta y_{n-1}}\right) \\ \mathcal{S}_Y f \cdot \mathcal{S}_Y \bar{f} &\supseteq \frac{\delta^n f}{\delta y_1 \delta y_2 \cdots \delta y_n} \end{aligned}$$

□