Improving Initialization through Reversed Retiming

Leon Stok, Ilan Spillinger
IBM TJ Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598

Guy Even
Technion
Haifa, 32000 Israel

Abstract

Despite the fact that retiming circuits has a large potential (especially in automatically synthesized circuits from higher-level descriptions) it has not been widely included in the current design methodologies. One of the main problems is finding an equivalent initial state for the retimed logic. In this paper we introduce a new *reverse retiming* algorithm which will find a retiming for a given cycle time, if one exists. This new algorithm minimizes the effort required to find equivalent initial states and reduces the chance that the network needs to be modified to find an equivalent initial state. This algorithm is the kernel of a new efficient retiming method, which searches for optimal retimings preserving the initial state condition.

1 Introduction

Retiming is a transformation to relocate the registers in sequential circuits. It has been shown [6] that (under certain restrictions) this transformation preserves the functionality of the design. Retiming may be applied for several optimization goals e.g. minimizing the cycle time, minimizing the area, minimizing the number of registers or improving testability.
An example of retiming a circuit is shown in figure 1.a). The initial circuit has three registers ($r1, r2$ and $r3$ and a maximum combinatorial delay of three units through gates $g2$, $g3$ and $g4$. (Assuming unit delay model, no fan-in fan out delays etc.). In order to reduce the cycle time to two units and minimize the registers to two we can retime gate $g4$. The two registers $r2$ and $r3$ at its outputs are moved to the input and replaced by register $r4$. The retimed circuit is shown in figure 1.b).
A retiming can be described by an integer function $L()$ (called the lag) of all nodes in the network. This function represents the number of registers that are to be moved from each output of node v to each of its inputs. In figure 1, one register is removed from each output and one register is inserted in the input. Therefore, the lag of gate $g4$ equals one, $L(g4) = 1$. Note that a positive lag causes registers to move backward in the network while a negative lag moves them forward. The direction forward is defined as the direction in which the data flows through the nets.
The initial state of a circuit is determines by the initial values of the registers in the circuit. For the cases where the initial state of the sequential circuit is meaningful, it is necessary to find an equivalent initial state for the retimed circuit. It can easily be shown that it is not always possible to find the initial state of the retimed circuit. For example, let us assume that in figure 1 the initial value of $r2$ is 1 and of $r3$ is 0. The retimed circuit cannot be initialized to have the same behavior as the original circuit. One cannot find an initial value for the new register $r4$.
A retimed circuit has an initial state equivalent to an initial state in the original circuit if for any input sequence applied to both circuits (one circuit started in the initial state, the other in the equivalent one) the same sequence of outputs is produced.
One way to assure that a corresponding initial state can be found in the retimed network is to only move registers forward in the network [3]. Let us define this as *simple forward retiming.* In this case the initial state can be propagated to the new register positions by a simple simulation (i.e. forward implication) of the values in the network.

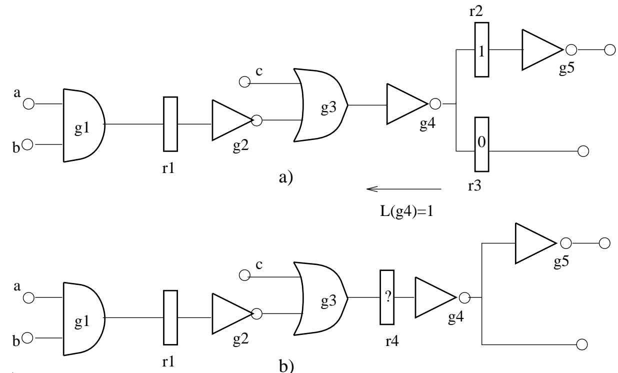

Figure 1: a) Original circuit. b) Retimed circuit

Forward retiming through the primary outputs/ primary inputs (OIs) removes a register from each path to a primary output and inserts one on each path from a primary input [8]. Notice that the number of registers on each path remains unchanged before and after such a retiming [7]. In the simple forward retiming, retiming through OIs is not allowed, since one will loose track of the initial values. As shown is figure 2 this imposes a significant restriction on the retiming and prohibits various retimings from being found. The circuit in figure 2 cannot be retimed to obtain a delay of two units by allowing only simple forward moves and not allowing moves through the outputs and inputs.
Eliminating the constraint on retiming through the OIs makes forward retiming a general retiming. Every backward retiming can be obtained by applying a sequence of forward retimings through OIs.

The basic problem is to determine the initial values for the registers inserted in the primary input paths.

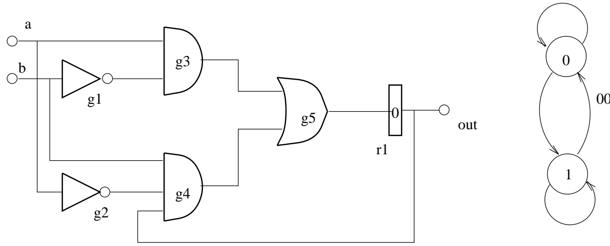

Figure 2: No simple forward retiming

For example, the register $r1$ in figure 2 can be duplicated. One of the duplicates is directly connected to the last input of gate $g4$, the other directly feeds the primary output. Removing the register from the output and inserting two new registers at the inputs a and b produces a valid retiming. But we can not easily calculate the initial values for these new registers by forward implication.

Touati [9] describes a method which finds a sequence of input values to be inserted at the inputs to find the appropriate initial values of the registers in the retimed circuit. Given a particular legal retiming one can derive the number of forward moves through the OIs to obtain this retiming. Let us call this number of forward OI moves k. A sequence of k input values is needed, which prescribes the values inserted in each OI move. This sequence can be obtained by inspecting the state machine extracted from the circuit. In this state machine a sequence of k transitions must be found which will bring the machine into the initial state. Any state may be the starting point of such a sequence. Each time the inputs are retimed (i.e. registers are inserted in the input paths), they are initialized with the values from this input sequence.

Let us apply this method to the example of figure 2. The state machine for this network has two states. Assume that we want to find an initial state for the retiming of gate $g5$ by one, $L(g5) = +1$, by repeated forward moves. This requires one forward OI move. Therefore, the length of the input sequence to initialize the new registers at the inputs equals one, $k = 1$. In the partial state diagram of figure 2 we have to search for a sequence of one transition that leads to state '0'. The transition $(a = 0, b = 0)$ brings us from state '1' to state '0' and can be used. When the register is removed from the output *out*, two new registers are inserted at the inputs a and b (both initialized with a 0). These new registers can be moved forward through gates $g1, ..., g4$ by simple forward moves to their final positions at the inputs of $g5$.

To be able to execute this method to find an initialization, the state machine is required to have a sequence of state transitions of length k leading to the initial state. If this is not the case the circuit must be *modified* to include such a sequence.

However, another retiming may exist which enables one to find an initial state without modifications to the network. The major contribution of this work is that we explore the existence of such retimings. The *reverse retiming* algorithm is introduced, which has the same complexity as that of the best known retiming algorithm [6]. The algorithm finds a retiming such that the number of registers that move backward through a single combinational block is the minimal between all possible retimings. Especially, whenever there exists a simple forward retiming, the modified algorithm will find it. No published methods known to the authors do explore the existence of such alternative retimings.

The rest of this paper is organized as follows. Section 2 describes our reverse retiming algorithm. Section 3 explains the implication and justification procedure used in BooleDozer to calculate the initial states. Section 4 presents the retiming method which preserves the initial state condition. Finally, section 5 shows that the results of our experiments are in accordance with the claims on the retiming algorithm.

2 Reverse Retiming

In a retiming, backward moves should be avoided as much as possible. The basic difficulty with backward moves is the existence of a mapping for the initial state. Finding this mapping is known as an NP-hard problem, as it is similar to the phase of justification in the process of automatic test pattern generation [5]. So whenever possible, a good criteria is to minimize the number of registers that move backward through each functional (combinational) block.

A circuit graph $G = (V, E, w, d)$ consists of a directed graph (V, E) with non-negative integer weights $w(e)$ on the edges and non-negative real delays $d(v)$ on the vertices. The weights on the edges model the number of registers along the edge, the delays of the vertices model the propagation delay of the nodes. Given a path $p = [v_0, ..., v_k]$, its weight is defined as $w(p) = \sum_{i=0}^{k-1} w(e_i)$, where e_i is the edge from v_i to v_{i+1}. The delay is defined as $d(p) = \sum_{i=0}^{k} d(v_i)$. The minimum feasible clock period of G, $\Phi(G)$ is defined as: $\Phi(G) = \max\{d(p) | w(p) = 0\}$.

A circuit graph can be derived from an actual circuit $C(B, R, N)$ by replacing the combinational blocks B by nodes V, the nets N by edges E between the nodes and the registers R by weights w on these edges [6]. In the graph model a special node with zero delay called the host and denoted by h is added. For every primary output of the network an edge with zero weight is inserted from the output to the host. For every primary input an edge is added from the host to the input.

Let $L(v)$ be the lag function for all nodes $v \in V$, $e(u, v)$ an edge from u to v, $w(e)$ the weight of the edges before retiming and $w_L(e)$ the weights of the edges after retiming. w_L can be determined from w and the lags by: $w_L(e) = w(e) + L(v) - L(u)$.

The normalized lag $L^*(v)$ of a node v is defined as the difference between the lag of the node and the lag of the host, i.e. $L^*(v) = L(v) - L(h)$. Since the retimed weight $w_L(e)$ is only dependent on the difference of the lags between two nodes and not on the absolute value of the lags itself, retiming with the normalized lags will result in the same circuit as retiming with the original lags.

For each node v a required time $t_r(v)$ is defined. For the simplicity of the discussion, we assume all primary

Algorithm 1. reverse_retiming(G,c)

```
foreach v ∈ V do
    L(v) = 0;
k = 1;
do
    Compute retimed circuit G_L;
    Compute t_r for each vertex v ∈ V;
    M = {v|t_r < 0}
    foreach v ∈ M do
        L(v) = L(v) − 1;
    k = k + 1;
while k ≤ |V| and M ≠ ∅;
```

outputs and registers to be synchronized to the clock and their required time is equal to the required cycle time c. For any other node v the required time is defined as the difference between the smallest required time for its successors and the propagation delay of the node itself: $t_r(v) = \min_{(v,u)\in E}(t_r(u)) - d(v)$. Notice, the generality of our reverse retiming algorithm holds also under other delay models and different required times for the outputs and registers.
The *reverse retiming* algorithm, described in algorithm 1, first sets the lags of all nodes to zero. In the outer loop it retimes the circuit according to the lags L and recomputes the set M of nodes whose outputs are not in time to meet a required time. The lags of these nodes are decreased by one. If none of the nodes violates a constraint the iteration can be stopped. If $|V|$ iterations have been tried no solution which meets the requirements is possible for G..
All claims for the retiming algorithms (FEAS in [6] and retime in [8]) hold. In addition reverse retiming finds the retiming with the minimal normalized lag, as expressed in the following theorem:

Theorem 1 *Let $G = (V, E, w, d)$ be a circuit and c a required clock cycle. Let $L(v), v \in V$ denote the retiming computed by the algorithm reverse_retiming(G, c). Then:*

1. *The algorithm reverse_retiming(G, c) finds a retiming L such that $\Phi(G_L) \le c$, if such a retiming exists,*
2. *if the graph G is strongly connected, and if $\Phi(G_L) \le c$, then for every retiming L' for which $\Phi(G'_L) \le c$ holds:*

$$\max_{v\in V} L(v) - L(h) \le \max_{v\in V} L'(v) - L'(h)$$

A complete proof for this theorem is described in [4]. This proof relies heavily on the proof of the FEAS algorithm [6] and a variant of the Bellman-Ford algorithm. Result 1 from the above theorem, shows that a retiming, for a given cycle time, will be found if one exists, similar to the FEAS algorithm. Result 2 expresses that the retiming which is found is the one with the smallest maximum normalized lag value.
Consider the example given in figure 3.a). The delay of the circuit is three (through the gates $g7$, $g8$ and $g9$). The goal for retiming is to obtain a circuit with

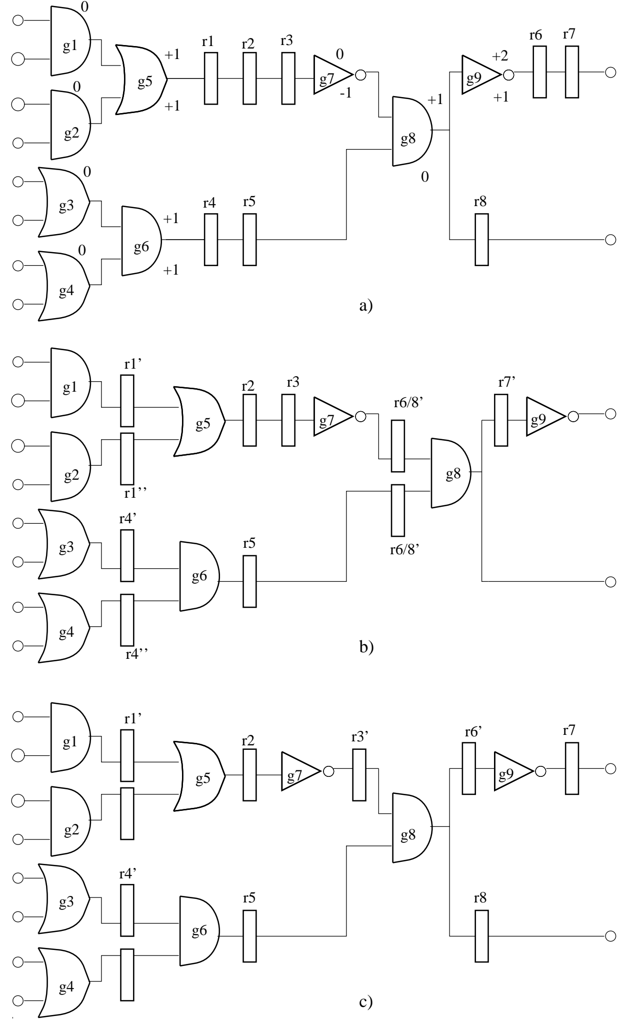

Figure 3: a) Circuit b) Circuit after FEAS c) Circuit after reversed retiming

cycle time one. If the original retiming [6] is applied to this circuit the lag values displayed above the gates are calculated. Retiming using these values results in the circuit in figure 3.b). Assume that the initial state of the circuit requires all registers to be initialized to zero. No initial value can be calculated for the registers resulting from the merge of $r6$ and $r8$ ($r6/8'$). Touati's method can be used to modify the network such that these registers can be initialized.
No simple forward retiming exists for this example. However, there exists another retiming which requires no modification of the circuit and enables one to find an initial state. This is the retiming as found by our reverse retiming algorithm. The lags for the reverse retiming are displayed below the gates. Retiming with these lags gives the circuit in 3.c). The initial state has a one in $r3'$ and $r6'$ and a zero in all other registers.
This example shows that there is a whole class of circuits which does not adhere to the reachable initial state condition, but for which a retiming with an equivalent initial state exists. For this class of circuits

Algorithm 2. Update_Registers($C(B,R,N), L()$)

```
do
I = J = ∅;
foreach r ∈ R do
    foreach b ∈ {b|∃(r,b) ∈ N} do
    if (L(b) < 0) then
        Build the forward cone FC from r;
        Stop and insert a register when you
            reach a node u such that L(u) ≥ 0
            or reach a register pr;
        Insert new register nr with value
            don't care, before u or pr;
        I = I + r;
        foreach u ∈ FC do
            L(u) = L(u) + 1;
        endforeach
    endif
    endforeach
    foreach b ∈ {b|∃(b,r) ∈ N do
    if (L(b) > 0) then
        Build the backward cone BC from r;
        Stop and insert a register when you
            reach a node u such that L(u) ≤ 0
            or reach a register pr;
        Insert new register nr with value
            don't care, after u or pr;
        J = J + r;
        foreach u ∈ BC do
            L(u) = L(u) − 1;
        endforeach
    endif
    endforeach
endforeach
if (I ∪ J = ∅) then return (success);
Forward implication for all values of registers in I
and justification of all values in J using justify [5];
if (justify fails) then
    return (failure);
else
    Remove all registers reg ∈ I ∪ J from C;
endif
while TRUE;
```

no additional logic has to be added if the proper retiming can be found. The reverse retiming algorithm is likely to find such a retiming and thus requires no logic to be added.

3 Initial State Calculation

Given a retiming for a circuit graph $G(V,E,w,d)$, a retiming function can be defined for the circuit $C(B,R,N)$. By construction there is a one-to-one correspondence between a node $v \in V$ and a combinational block $b \in B$. The lag of a block b is defined equal to the lag of the corresponding node v. A retimed circuit C' for a lag function $L()$ is constructed using the *Update_Registers* algorithm described in algorithm 2. This algorithm simultaneously calculates the new positions and the initial values for the registers, such that the initial state of the retimed circuit is equivalent to the initial state of the original circuit. All registers in the original circuit C have contents zero, one or don't care, as the initial state. The outer loop of the *Update_Registers* algorithm checks all registers in the design. For each register r we check whether the lag of one of its outputs is negative. If true, insert this register r in the implication list I. In figure 4 register $r1$ is inserted in list I. Also, traverse the design forward (towards the primary outputs). If you reach a node u which was not been visited in this iteration, and its lag value is negative then increment its lag by one, and keep traversing the design in the forward direction. Whenever you reach node u with lag $L(u) >= 0$ or you reach a register pr, introduce a new register nr before u or pr and stop traversing the design in this direction. In figure 4, $g1$ its lag is incremented. The forward cone stops at gate $g2$, so a new register $nr1$ is inserted in front of $g2$.

In a similar way, for each register in the design it is checked if one of its inputs has a positive lag. If so, traverse the design backward, decrement the lags of the appropriate nodes, insert new registers and update the justification list J. In our example (figure 4), J will contain register $r2$ and two new registers $nr2$ and $nr3$ are inserted before gate $g2$. All new registers inserted in the circuit have a don't care initial value.

Only if I and/or J are non-empty, retiming moves have been done. For all values of registers in I a *forward implication procedure* is called. For all values of registers in J, a *backward justification* is used. Both of these procedures are described in [5]. If justification fails then there is no possible equivalent retimed initial state. If it succeeds, the appropriate register values are updated. All registers in the lists I and J (they have been replaced, by new registers) can be deleted, and the Update_Registers algorithm proceeds to the next iteration. The number of iterations required for a successful computation of the retimed equivalent initial state is the maximum absolute value of the normalized lags.

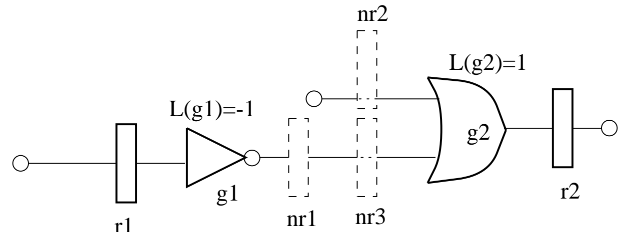

Figure 4: Register update procedure

4 Retiming preserving initial states.

Suppose that during the update of all registers, one fails in the justification process, e.g. the backtracing computation for a block b causes a conflict. In such a case, it may be helpful to bound the lag of node b, such that the backtracing computation that causes the justification to fail will not be necessary.

For every node b, that could not be backtraced during the i-th iteration of the Update_Registers algorithm, the lag is bounded by i, $L(b) < i$. To accomplish this, the circuit graph G must be modified. An edge with weight $i-1$ is added between the node v (the counterpart of b in G) and the host h. Let us call this circuit graph G^m.

In [4] we prove that modifying the graph G to the graph G^m by a construction as sketched above does

Table 1: Improvement from R-FEAS over FEAS

	Original			FEAS		Reverse Retiming		
Circuit	*CT*	*Reg*	*RCT*	L^*	*Reg*	L^*	*Reg*	*Runtime*
s344	28	15	19	1	21	0	27	1
s349	28	15	19	1	21	0	28	1
s382	17	21	12	1	39	0	33	1
s400	17	21	12	1	41	0	34	1
s444	20	21	13	1	44	0	35	1
s526	14	21	11	1	33	0	42	1
s526n	14	21	11	1	28	0	33	1
s953	27	29	23	1	39	0	33	2
s38417	65	1465	49	1	1477	0	1504	71

CT: cycle time, *Reg*: number of registers
RCT : retimed cycle time, L^*: maximum normalized lag
Runtime : runtime in seconds

not influence the optimality of the result of the reverse retiming algorithm. In other words, if a retiming is found for the circuit graph G^m for a cycle time c, this is also a valid retiming for the original circuit G with a cycle time c. In addition, the normalized lags of all those unjustifiable nodes are less then their bounds.
Our method does not require the implicit enumeration of all reachable states, a process which in itself may be very expensive. The most expensive part in our method is the justification step. These steps are applied locally to a small subset of the network.

5 Some experiments

The retiming method as described in the previous sections is implemented within the BooleDozer [1] logic synthesis system. We applied the reverse retiming algorithm to 33 sequential multi-level circuits in the MCNC (EDIF) benchmark set [2]. A unit delay model is assumed. Each gate has a propagation delay of one, no delays on the registers and no delays due to fan-in or fan-out were used. For 14 of those circuits (s27, s208, s298, s386, s420, s510, s641, s713, s820, s832, s1196, s1488,s1494, s35932) retiming could not improve their cycle time. For the remaining 19 designs, retiming reduced their cycle time up to 40%.
Our main interest in these experiments is the comparison between the retimed circuit obtained by the FEAS algorithm [6] vs. the one obtained by the reverse retiming algorithm. Notice that the minimal feasible cycle time achieved by both algorithms is the *same*, and the difference is the retiming function which reflects the feasibility to find an equivalent initial state for the retimed design. For ten designs (s208.1, s420.1, s838, s838.1, s1238, s1423, s5378, s9234.1, s15850, s38584.1) the maximal value of the normalized lag produced by both algorithms was the same being 0 or +1. In four circuits reverse retiming was not able to find a solution with $L^*=0$, because it simply does not exist. In most cases this is due to a path from a primary input to a register which is too long to meet the optimal cycle time. The only way to solve this is moving registers forward through the OIs or moving registers backward. For the remaining nine circuits a difference is found between reverse retiming and FEAS. For these nine examples, listed in table 1, the retiming function generated by FEAS has maximum normalized lag value of +1. Using the technique of [9], finding an initial state requires state enumeration and may require the addition of logic. The better retiming function achieved by reverse retiming has a maximum normalized lag of 0. Only forward implication moves are necessary to find the equivalent initial state. No expensive state enumeration nor the addition of logic is required.
For all examples, both FEAS and RFEAS (since they have the same complexity and a similar implementation) run in under 71 seconds on an IBM RS6000, model 350.

6 Conclusions

The main contributions of this paper can be summarized as follows. The new *reverse retiming* algorithm will find a retiming under a cycle time constraint which requires only forward moves if such a retiming exists. If no such retiming exists it will produce a retiming which requires a minimal number of justification steps to find an equivalent initial state. Reverse retiming has the same complexity as the best-known retiming algorithms. In 9 of the MCNC benchmarks reverse retiming produced circuits with a maximum normalized lag of 0 while FEAS produced a maximum normalized lag of 1.
Furthermore, a new procedure to update the network and find the equivalent initial state is described. This procedure does not require the implicit enumeration of the state machine to find a sequence of initial values. If the procedure fails, an iterative method is provided to produce alternative retimings. Earlier methods [9] required the addition of logic to initialize retimed circuits for which the initial state is unreachable. This paper shows that a class of circuits exists for which reverse retiming will find a retiming without any modification of the logic.

References

[1] *LogicBench: BooleDozer Synthesis User's Guide.* IBM Corporation, Hopewell Junction NY, 1994.

[2] F. Brglez, D. Bryan, and K. Kozminski. Combinational profile of sequential benchmark circuits. In *Proc. of the Int. Symp. on Circuits and Systems*, pages 1929–1934, May 1989.

[3] S. Dey, M. Potkonjak, and S.G.Rotweiler. Performance optimization of sequntial circuits by eliminating retiming bottlenecks. In *Int. conf. on comp. aided design*, pages 504–509, Santa Clara, November 1992.

[4] G. Even and I. Spillinger. Facilitating the problem of finding initializations after retiming; definitions, heuristic cost functions and algorithms. IBM Research Report, RC 19525, Yorktown Heights, March 1994.

[5] S. Kundu, et al. A small test generator for large designs. In *Proc. of International Test Conference*, pp. 30–40, Sept. 1992.

[6] C. Leiserson and J. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[7] S. Malik, et al. Retiming and resynthesis: optimizing sequential networks with combinational techniques. *IEEE Trans. CAD (USA)*, 10(1):74–84, Jan. 1991.

[8] G. D. Micheli. Synchronous logic synthesis: Algorithms for cycle-time minimization. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst*, 10(1):63–73, Jan. 1991.

[9] H. Touati and R. Brayton. Computing the initial states of retimed circuits. *IEEE Trans on CAD*, 12(1):157–162, January, 1993.