Retiming Sequential Circuits with Multiple Register Classes *

Klaus Eckl Christian Legl
Institute of Electronic Design Automation
Technical University of Munich
80290 Munich, Germany
Klaus.Eckl@ei.tum.de Christian.Legl@ei.tum.de

Abstract

Retiming is an efficient technique for redistributing registers in synchronous circuits in order to improve the circuit performance. However, the traditional retiming approaches cannot handle circuits whose registers are controlled by different clock, reset, and load enable signals. We present basic theory and a comprehensive retiming approach for circuits with multiple clock, reset, and load enable signals. We retime these circuits having multiple register classes without explicitly modeling the reset or the load enable by additional logic. The presented concepts can be combined with a wide range of existing retiming approaches. Experimental results from retiming real designs for clock period minimization show the efficiency of the new approach.

1. Introduction

In modern VLSI design the optimization of synchronous sequential circuits is a key issue. In order to optimize synchronous circuits, combinational optimization techniques are usually applied in synthesis systems while sequential optimization techniques are only rarely used.

A very promising sequential optimization technique is called *retiming*. Retiming moves registers across combinational logic blocks without changing the logic function inside the blocks. Leiserson and Saxe [9, 10] showed that retiming can be used for minimizing the clock period as well as for minimizing the number of registers under a clock period constraint. It can be applied to circuits with level-sensitive latches [11] or edge-triggered flip-flops [10, 16]. Several extensions to retiming [15, 2, 3] as well as efficient implementations [16, 13] have been proposed which show that retiming is even applicable to large circuits. In this paper, we focus on sequential circuits having edge-triggered flip-flops as registers. However, the concepts and approaches of this paper can also be transferred to circuits with level-sensitive latches.

Most retiming approaches assume that edge-triggered sequential circuits have only a single clock. Industrial designs, however, are usually *multiple-clock* systems, i.e., different registers may be connected to different clocks. This fact complicates the retiming problem because additional constraints are imposed on register movements in order to preserve temporality [17]. Lockyear and Ebeling [11], and Ishii et al. [7] already tackled the problem of retiming multiphase level-clocked circuits. However, their approaches are restricted to the special class of *well-formed* circuits in which the sequence of clock phases has the same order along any path through the circuit. Thus, these approaches need not impose additional constraints on register movements. In general multiple-clock circuits, however, these constraints must be taken into account explicitly.

Most of the existing retiming approaches assume that the circuit has a single reset signal for all registers which is activated only once at the beginning of circuit operation [19, 5, 14]. Thus, they only tackle the problem of computing an *initial state* for the retimed circuit that is equal to the initial state of the original circuit. Singhal et al. [18] generalized the *retimed initial state* problem to the *retimed reset state* problem by allowing, e.g., different reset lines to be activated during different clock cycles. They showed that register moves may be constrained due to the assumption about reset activation and due to different reset signals connected to different registers. However, they only looked at a single retiming move and did not present an overall retiming algorithm that finds an optimal retiming in accordance with the retimed reset state problem.

Registers may have a load enable (also called clock enable) capability. It is desirable that retiming preserves the load enables because modeling a load enable by a simple register with extra logic increases area and delay costs. E.g., many FPGAs offer the load enable capability for free, making its use imperative for good FPGA design. Thus, a retiming approach should be able to move registers together with their load enables across logic blocks. This problem was addressed in [1] and [6]. Camposano and Plöger [1] stated several conditions for a valid retiming step involving load enable registers. However, they did not present an overall approach for computing a retiming solution that satisfies these constraints. A first idea for solving the retiming problem in circuits with multiple load enable and clock signals was introduced in [8], but no practical implementation and no experimental results were shown.

In this paper, we present basic theory and an efficient

*This work was supported in part by DFG under grant AN 125/17.

approach for retiming synchronous circuits where different registers can be connected to different clocks, asynchronous and synchronous reset signals, and load enable signals. After some background information on traditional retiming in Section 2, we show in Section 3 how to partition the registers into *classes*. On the basis of these classes we define a circuit transformation called *multiple-class retiming* which guarantees that the retimed circuit is a *sufficiently old replacement* [9] of the original circuit. In Section 4 some theoretical aspects of multiple-class retiming are shown which lead to an efficient implementation.

Section 5 presents an efficient *multiple-class retiming approach*. It transforms the problem of retiming a multiple-class sequential circuit into a traditional single-class retiming problem. Then, any state-of-the-art retiming approach can be used which has been presented, e.g., for *clock period minimization* or *register minimization under a clock period constraint*. Finally, the multiple-class sequential circuit is updated according to the solution for the single-class retiming problem. This also includes the computation of equivalent asynchronous and synchronous reset states for the retimed circuit.

In order to show the applicability, we implemented our new approach and applied it to the clock period minimization problem. Experimental results in Section 6 show the efficiency of our multiple-class retiming approach.

2. Background

In this section, we briefly review the terminology and the graph model for single-class sequential circuits.

Leiserson and Saxe [9, 10] model a single-class sequential circuit as a vertex-weighted, edge-weighted, directed multigraph $G = \langle V, E, d, w \rangle$, called *retiming graph*. Each combinational logic block and each input and output of the sequential circuit is represented by a unique vertex $v \in V$. Additionally, a host vertex v_{host} is introduced to model the environment of the circuit. There exists an edge $e_{uv} \in E$ if an output of the logic block represented by vertex u is connected to an input of the logic block represented by vertex v. Moreover, there are edges from v_{host} to all input vertices and from all output vertices to v_{host}.

The vertex weight $d(v)$ is the propagation delay of the logic block of vertex v. The edge weight $w(e_{uv})$ indicates the number of registers along the interconnection from vertex u to vertex v. A path $p : u \rightsquigarrow v$ in the graph is a sequence of edges from vertex u to vertex v. The path weight $w(p)$ is the sum of the edge weights along the path.

A *retiming* is an integer-valued vertex labeling $r : V \to Z$. A retiming (or lag) value $r(v)$ denotes the number of registers that are moved from the outputs to the inputs of v. The retimed graph $G_r = \langle V, E, d, w_r \rangle$ can be derived from the original graph G by computing the retimed edge weights,

$$w_r(e_{uv}) = w(e_{uv}) + r(v) - r(u). \tag{1}$$

Similarly, the retimed path weight $w_r(p)$ of a path $p : u \rightsquigarrow v$ is computed by $w_r(p) = w(p) + r(v) - r(u)$. A retiming $r : V \to Z$ is *legal* [10] if for each edge e_{uv} the retimed edge weight $w_r(e_{uv})$ is nonnegative, i.e., $w_r(e_{uv}) \geq 0$.

A retiming $r' : V \to Z$ with $r'(v_{host}) \neq 0$ moves registers across inputs and outputs. However, an equivalent retiming r for which $r(v_{host}) = 0$ holds can always be obtained from r' by the *normalization* [5]

$$r(v) = r'(v) - r'(v_{host}). \tag{2}$$

In the sequel we will always assume that $r(v_{host}) = 0$ holds.

3. Multiple-class retiming

Before looking at the problem of multiple class retiming, we present the types of registers we are going to support and how to model these different types in a retiming graph.

The left hand side of Fig. 1 shows a generic register l between some gates u and v. Each register l is at least connected to a data input and output signal and a clock signal *clk*. A register may be connected to further control signals which are

- an asynchronous reset signal *ars* causing an asynchronous set or clear behavior,
- a synchronous reset signal *srs* causing a synchronous set or clear behavior,
- a load enable (or clock enable) signal *le*.

Contrary to the circuits handled by most of the existing retiming approaches we now also consider circuits containing multiple types or *classes* of registers.

Definition 1 (Register class) *A class C of registers is characterized by a tuple* (*clk, le, ars, srs*) *of signals. A register l is contained in class C iff each of its control signals is identical to the corresponding signal in the tuple of class C.*

The case that some register does not use, e.g., a load enable is modeled by connecting a constant 0 signal to the input *le*. Two clock signals are identical if they have the same period and phase offset. Two load enable or reset signals are identical if the signals are logical equivalent. The (a)synchronous reset state is determined by the reset signal being connected to either a set or clear input of the register. However, the class of a register does not depend on the reset state itself. Registers that belong to the same class are denoted *compatible registers*.

In the sequel, a circuit containing different classes of registers is called a *multiple-class circuit* in contrast to a *single-class circuit* which contains only one class of registers. For a multiple-class circuit it is no longer sufficient to store the number of registers $w(e_{uv})$ on an edge e_{uv} of the retiming graph, as each register may belong to a different class. This information needs to be modeled in the retiming graph.

Therefore, we introduce a modified retiming graph $G^{mc} = \langle V, E, d, \underline{l} \rangle$ which we call *mc-retiming graph* or short *mc-graph* (an example is depicted right hand side of Fig. 1). Instead of the edge values $w(e_{uv})$ we place an ordered list $\underline{l}(e_{uv}) = [l_1, \ldots, l_{w(e_{uv})}]$ on each edge. This list indicates a sequence of $w(e_{uv})$ registers on this edge. l_1 corresponds to the register closest to the source of the edge, while $l_{w(e_{uv})}$ is the register closest to the sink of the edge. An optional

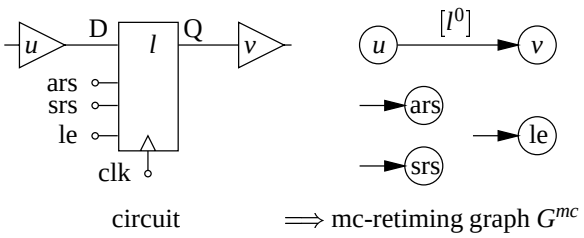

Figure 1. Modeling registers in a mc-retiming graph.

superscript j at some register l_i^j denotes the index of the corresponding register class C_j. Additionally, for each load enable and reset signal we introduce an output vertex in the mc-retiming graph.

For an mc-graph we keep the notion of edge weights which are now given by the length of the lists: $w(e_{uv}) = |\underline{l}(e_{uv})|$. Similarly to the single-class retiming graph (sc-graph) we apply the concept of register lists also to paths: The sequence of registers on path $p : u \rightsquigarrow v$ is given by the list $\underline{l}(p)$ and $l_i(p)$ denotes the i-th register on that path.

A basic *mc-retiming step* for a vertex v can now be performed as depicted in Fig. 2. E.g., for one forward mc-retiming step at vertex v, there must be a complete *layer* of registers at the incoming edges of v which comprises the last registers in each list. During the forward retiming step, the registers of the layer are removed from their list, and a new layer is inserted at the fanout edges of v by prepending a register to the list of each outgoing edge.

Of course, for multiple-class circuits certain restrictions apply to the *validity* of such a mc-retiming step. A mc-retiming step is valid if it yields a circuit which is a *sufficiently old replacement* [9] of the original circuit. It has been shown in [1] for load enable signals that such a retiming step is valid if all registers of the layer to be moved are connected to the same load enable signal. The same condition applies to the clock signal in order to preserve the temporal equivalence of the circuit [17]. Registers with reset signals can be moved if the corresponding signals are identical [18]. Since our definition of register classes is based on the same conditions, we can relate the validity of a mc-retiming step to the classes of the registers to be moved:

Theorem 1 (Validity of a mc-retiming step)
A mc-retiming step as shown in Fig. 2 is valid if all registers to be moved belong to the same class.

The proof of this theorem directly follows from [17, 1, 18]. Note that by examining the state space or the logic functions we may find few cases where registers of different classes can be moved across a vertex while still preserving circuit equivalence. However, such extensive analysis would deteriorate the efficiency of retiming. Therefore, we do not consider such cases.

As in traditional single-class retiming, we can also define a retiming for a multiple-class circuit as an integer valued vertex labeling $r : V \to Z$. The validity of such a retiming for a multiple-class circuit is given by the following theorem.

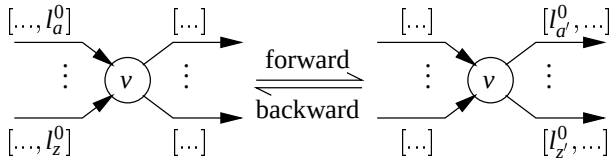

Figure 2. Performing a basic mc-retiming step

Theorem 2 (Legal mc-retiming) *A retiming $r : V \to Z$ generating a circuit which is a sufficiently old replacement of the original circuit is a legal retiming for a multiple-class circuit, short a **legal mc-retiming**, if it can be implemented by a sequence of valid mc-retiming steps.*

The proof of this theorem follows from the validity of each single basic mc-retiming step. In the following sections we will show how a legal mc-retiming can be efficiently computed.

4. Theory of multiple-class retiming

We will now derive some properties of mc-retiming and we show its relation to sc-retiming. This relation will lead to an efficient implementation of mc-retiming. The proofs of the theorems can be found in [4].

Let a legal retiming $r : V \to Z$ be given for a sc-retiming problem where no register is moved across the inputs/outputs of the circuit, i.e., $r(v_{host}) = 0$. It was shown in [12] that the retiming values are bounded from above and from below as given by the following theorem.

Theorem 3 *For each vertex v of G, let $p_I(v)$ and $p_O(v)$ be minimum weighted paths from the host vertex v_{host} to v and from v to v_{host}, respectively. Then, any legal sc-retiming $r : V \to Z$, with $r(v_{host}) = 0$, satisfies*

$$r_{min}(v) \leq r(v) \leq r_{max}(v), \tag{3}$$

where $r_{min}(v) = -w(p_I(v))$, and $r_{max}(v) = w(p_O(v))$.

Intuitively, we can not move more registers forward or backward across v than the minimum number of registers available on any path from v_{host} to v or from v to v_{host}, respectively. Theorem 3 guarantees for sc-retiming that, e.g., up to $r_{max}(v)$ registers can be moved backward across v. However, for mc-retiming this may not be possible since different register classes can prohibit some retiming steps. This is illustrated by the following example.

Figure 3 shows a part of a mc-retiming graph. For vertex v_1 a minimum weighted path to the host vertex is given by $p_O(v_1) = (v_1, v_3, v_{host})$. It carries the register list $\underline{l}(p_O) = [l_a^0, l_b^0]$ and has weight $w(p_O) = 2$. Thus, for sc-retiming, $r_{max}(v_1) = w(p_O) = 2$ registers could be moved backward across v_1. For mc-retiming, the first backward mc-retiming step would be valid because the corresponding layer is comprised by compatible registers l_a^0 and l_c^0. For the second backward move at v_1 we first move register l_d^1 across vertex v_2 which is a valid step. However, now the registers l_b^0 and l_d^1 at the fanout edges of v_1 belong to different classes C_0 and C_1 which prohibits any further backward move.

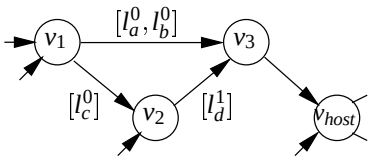

Figure 3. Part of a mc-retiming graph G^{mc}.

We will now investigate the condition under which a retiming value is a valid mc-retiming value.

Definition 2 (Valid mc-retiming value) *A retiming value $r(v)$ is a* valid *mc-retiming value iff $r(v)$ can be implemented by a sequence of valid mc-retiming steps.*

The validity of $r(v)$ is directly related to the compatibility of the registers in the layers that can be reached on the incoming and outgoing paths of vertex v.

Theorem 4 (Conditions for a valid mc-retiming value)
A retiming value $r(v) > 0$ is a valid mc-retiming value iff
a) $r(v) \leq r_{max}(v)$,
b) $r(v) - 1$ *is a valid mc-retiming value, and*
c) all registers $l_{r(v)}(p)$ on all paths starting at vertex v are compatible.
A retiming value $r(v) < 0$ is a valid mc-retiming value iff
a) $r(v) \geq r_{min}(v)$,
b) $r(v) + 1$ *is a valid mc-retiming value, and*
c) all registers $l_{w(p)+1+r(v)}(p)$ on all paths ending at vertex v are compatible.
Trivially, $r(v) = 0$ is always a valid mc-retiming value.

From Theorem 4 it follows that there exist a unique lower and upper bound for the valid mc-retiming values of each vertex which we will denote *forward mc-retiming bound* $r^{mc}_{min}(v)$ and *backward mc-retiming bound* $r^{mc}_{max}(v)$, respectively. Because of Theorem 4 these bounds never exceed the sc-retiming bounds, i.e.,

$$r_{min}(v) \leq r^{mc}_{min}(v) \leq r(v) \leq r^{mc}_{max}(v) \leq r_{max}(v). \quad (4)$$

By Theorem 3, a legal sc-retiming only guarantees that $r_{min}(v) \leq r(v) \leq r_{max}(v)$. If we also want to guarantee a legal mc-retiming solution, we must additionally incorporate the mc-retiming bounds (4) into the mc-retiming problem formulation. This leads to the following pivotal theorem which states the conditions for a legal mc-retiming solution.

Theorem 5 (Legal mc-retiming) *A retiming $r : V \to Z$ is a legal mc-retiming iff the following conditions hold:*

$$\forall e_{uv} \in E \quad : \quad r(u) - r(v) \leq w(e_{uv}) \quad (5)$$

$$r(v_{host}) = 0 \quad (6)$$

$$\forall v \in V \quad : \quad r^{mc}_{min}(v) \leq r(v) \leq r^{mc}_{max}(v). \quad (7)$$

Conditions (5) and (6) are the same as for sc-retiming and simply reflect the demand for positive retimed edge weights and prohibition of input/output moves, respectively. Additionally, condition (7) allows only valid mc-retiming values. These values can be implemented by valid mc-retiming steps. Thus, r represents a legal mc-retiming according to Theorem 2.

Corollary 1 *A mc-retiming $r : V \to Z$ is legal iff r is a legal sc-retiming and all retiming values fulfill the constraints given by the mc-retiming bounds.*

Corollary 1 directly leads to an efficient approach for mc-retiming. We can use any sc-retiming approach during the search for an optimal mc-retiming solution as long as we satisfy the additional mc-retiming bounds for the retiming values. The following section will show that these mc-retiming bounds can be incorporated into the traditional sc-retiming problem formulation in a natural way.

5. Efficient algorithm for mc-retiming

In the last section we showed that a mc-retiming solution can be obtained by performing sc-retiming with additional constraints on the retiming values. In order to do so we now propose an efficient mc-retiming approach which performs the following three steps:

1. Compute backward and forward mc-retiming bounds.
2. Compute sc-retiming which fulfills the mc-retiming constraints using any generic retiming approach.
3. Move the registers in the graph G^{mc} to their final positions according to retiming values obtained in step 2.

The following subsections will explain these steps in more detail.

5.1. Computing the mc-retiming constraints

We will only show how to compute the backward mc-retiming bounds $r^{mc}_{max}(v)$. All results derived for this case can be easily transfered to the computation of the forward mc-retiming bounds $r^{mc}_{min}(v)$.

From Section 4 we know that a unique value r^{mc}_{max} exists for each vertex of the graph G^{mc}. We call a retiming r^* where $r^*(v) = r^{mc}_{max}(v)$ for each vertex v a *maximal backward retiming* because it moves the registers as far as possible backward toward the primary inputs. We can compute a maximal backward retiming, and thus the values r^{mc}_{max}, by applying valid backward mc-retiming steps as long as possible while counting the number of registers which move across each vertex. The next theorem shows that no matter in which order we apply the basic retiming steps, we always end up with the unique maximal backward retiming r^*.

Theorem 6 (Maximal backward retiming) *Let r be a legal mc-retiming which has already been implemented by valid mc-retiming steps. Then, as long as there exists a vertex u with $r(u) < r^{mc}_{max}(u)$, there exists at least one vertex v for which a valid mc-retiming step can be applied.*

Proof : Let u be a vertex with $r(u) < r^{mc}_{max}(u)$. Then, each outgoing path $p : u \rightsquigarrow v_{host}$ has a register $l_1(p)$ on it, and all these registers are compatible. By an iterative procedure [4], we can show that either u or some vertex v in the transitive fanout of u is retimeable by a valid mc-retiming step. □

Algorithm 1 Compute_Backward_Bounds($G^{mc}(V,E)$)

```
1:  Q ← ∅
2:  for all v ∈ V do
3:     r(v) ← 0
4:     if v is backward mc-retimeable then
5:        Q ← Q ∪ {v}
6:     end if
7:  end for
8:  while Q ≠ ∅ do
9:     remove a vertex v from Q
10:    while v is backward mc-retimeable do
11:       backward_mcretime_step(v)
12:       r(v) ← r(v) + 1
13:    end while
14:    for all u ∈ FANIN(v) do
15:       if u is backward mc-retimeable then
16:          Q ← Q ∪ {u}
17:       end if
18:    end for
19: end while
20: return r^mc_max(v) ← r(v)
```

Algorithm 1 shows an efficient procedure which computes the backward mc-retiming bounds r^{mc}_{max}. First, all vertices that are amenable for a backward mc-retiming step are added to the set Q. In the main loop one vertex v is removed from Q at a time and retimed backward as often as possible while increasing its retiming value accordingly. Then, each fanin vertex u of v which became backward retimeable due to the retiming of v is inserted into the set Q. The loop terminates when the set Q is empty. Then, each retiming value is equal to the backward mc-retiming bound r^{mc}_{max}.

The runtime of this algorithm is $O(k(V+E))$ where $k = \max\{r^{mc}_{max}(u) \mid u \in V\}$. This runtime complexity is strictly less than the complexity of existing sc-retiming approaches. E.g., the *FEAS* algorithm used for minimum clock period retiming takes a runtime of $O(VE \cdot \lg V)$ [10]. Algorithms for register minimization under a clock period constraint have even higher runtime complexities. Thus, the computation of the mc-retiming bounds has only a small impact on the runtime of the overall mc-retiming approach. This will also be shown by the results in Section 6.

5.2. SC-Retiming under mc-retiming constraints

The sc-retiming problem is an integer linear program (ILP) with difference constraints only. Thus, it can be solved efficiently, e.g., by shortest-path algorithms for minimum period retiming and by minimum-cost flow algorithms for register minimization [10].

Corollary 1 shows that we can generate a legal mc-retiming by searching for a legal sc-retiming with additional retiming constraints imposed. Using $r(v_{host}) = 0$, we can rewrite the mc-retiming constraint (7) using two difference constraints,

$$r(v_{host}) - r(v) \leq -r^{mc}_{min}(v) \quad (8)$$
$$r(v) - r(v_{host}) \leq r^{mc}_{max}(v). \quad (9)$$

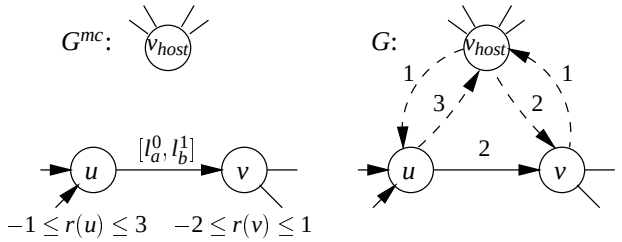

Figure 4. Transforming mc-graph to sc-graph.

If we add these constraints to the sc-retiming problem, we again have an ILP with difference constraints only. Thus, we can still use the same efficient sc-retiming approaches but now we guarantee that the mc-retiming bounds are fulfilled. Note that only the number of constraints increases but not the number of variables. In fact, in the *Minaret* approach [13] it was shown that additional constraints may be used to reduce the size of the ILP such that less time is needed to solve the problem.

Other approaches, like the *FEAS* algorithm [10] or *Reverse Retiming* [5], use the retiming graph as the basis for their computation. By the following steps, we can transform the mc-graph $G^{mc} = \langle V, E, d, \underline{l} \rangle$ into a sc-graph $G = \langle V, E \cup E_C, d, w \rangle$ such that the mc-retiming constraints for G^{mc} are inherently modeled in the sc-graph G.

1. Each edge e_{uv} in G^{mc} is translated into an edge e_{uv} in G with the register list $\underline{l}(e_{uv})$ substituted by the weight $w(e_{uv}) = |\underline{l}(e_{uv})|$.
2. We model the constraints (8) and (9) by an additional set E_C of *constraint edges*. For each vertex v we introduce an edge from the host vertex to vertex v with $w(e_{v_{host},v}) = -r^{mc}_{min}(v)$ and an edge from the vertex v to the host vertex with $w(e_{v,v_{host}}) = r^{mc}_{max}(v)$.

The constraint edges enforce that any sc-retiming solution will not move more registers across a vertex v than given by the corresponding constraint edge weight. For example, for the sc-graph G in Fig. 4 it is guaranteed for vertex u that not more than 1 register is moved forward and not more than 3 registers are moved backward. Thus, the given mc-retiming constraints for vertex u will be fulfilled.

The number of constraint edges is only $O(V)$. As the runtime of retiming algorithms is usually quadratic or cubic in the number of vertices, the overall runtime is nearly not affected by the constraint edges. In fact, as proposed in [16, 13] a closer look often reveals that many constraint edges are redundant and can be removed because they are dominated by neighboring constraints.

5.3. Updating the mc-retiming graph

The last step of our mc-retiming approach is to move the registers of the mc-graph G^{mc} to their final position according to the retiming values. Two problems must be solved. First, a proper sequence of mc-retiming steps must be found. Second, the asynchronous as well as the synchronous reset state must be computed for each moved register. In order to solve both problems, we use a modi-

fied version of the *Update_Registers* algorithm proposed by Even et al. [5]. When moving a register backward or forward, the new reset states are computed by backward justification or forward implication, respectively.

It may happen that a backward move across a vertex v is not possible because backward justification fails. Similar to [5] we handle this conflict by decreasing the backward mc-retiming bound $r^{mc}_{max}(v)$ such that the conflicting backward move is no longer allowed. Then, a new retiming solution is computed that prohibits the non-justifiable backward move.

Alternatively, like in [14] we could derive additional retiming constraints already when computing the mc-retiming bounds such that the existence of a backward justification is guaranteed. As noted in [14], this generally results in multiple sets of constraints for which multiple ILPs must be solved. Furthermore, we would have to justify a lot of backward moves which are eventually not implemented in the final retiming solution. Because of this large computational overhead, we dropped this alternative.

6. Results

In order to show the effectiveness and efficiency of our approach, we chose clock period minimization as the retiming optimization goal. For ease of implementation, we used the unit delay model to estimate gate delays. Of course, using a more realistic delay model is only a matter of having access to an appropriate timing analyzer. For the generic sc-retiming approach, we selected *Reverse_Retiming* [5]. It has the same complexity as *FEAS* [10] but on average it generates retiming solutions for which the reset state computation is easier [5]. Additionally, we used acceleration techniques for minimum period retiming as proposed in [16].

In order to solve the problems of backward justification and forward implication during reset state computation we use a straight-forward approach based on BDDs. The overall approach for updating the mc-retiming graph is implemented as described in Section 5.3.

We applied our mc-retiming approach to a set of mid-sized and large benchmark circuits from the LGSynth93[1] benchmark suite as well as to a set of industrial designs. Although the LGSynth93 benchmark circuits are single-class circuits, they were selected to show the efficiency of the computation of the mc-retiming bounds. The industrial circuits, denoted `ind1` to `ind9`, are multiple-class circuits.

Table 1 shows characteristic data for the computation of the mc-retiming bounds. The circuit name, the number of gates, and the number of registers are shown in columns 1 to 3. The number of register classes is given in column *#C*. Column *#back* and *#forw* show the overall number of backward and forward moves that are performed by the maximal backward and forward retiming algorithm, respectively. The last column shows the CPU time (DEC Alpha 4100 5/300) for computing the mc-retiming bounds.

For the industrial examples, the number of classes ranges from 2 to 18. CPU times for the computation of the mc-retiming bounds indicate that maximal backward and forward retiming can be performed very fast. E.g., it takes only 15.83 seconds for circuit `s38417` with 22520 gates and 1465 registers.

Table 1. Computation of mc-retiming bounds

circuit	#gate	#reg	#C	#back	#forw	CPU(s)
gcd	869	59	1	5618	621	0.63
mm9b	574	26	1	108	29	0.10
mult16a	212	16	1	1747	16	0.23
s344	217	15	1	219	164	0.07
s382	232	21	1	414	145	0.08
s400	246	21	1	448	157	0.10
s444	274	21	1	532	191	0.10
s526	277	21	1	529	221	0.12
s838.1	530	32	1	222	98	0.12
s953	621	29	1	770	108	0.18
s1238	690	18	1	121	9	0.12
s1423	796	74	1	657	175	0.18
s5378	2742	163	1	987	2412	0.70
s9234.1	2796	135	1	3793	2448	0.92
s38417	22520	1465	1	94499	40164	15.83
ind1	1426	318	18	510	122	0.43
ind2	1943	103	7	825	62	0.50
ind3	208	46	4	170	111	0.07
ind4	78	7	2	135	20	0.03
ind5	651	38	2	622	120	0.20
ind6	509	33	2	397	10422	1.45
ind7	635	18	2	595	151	0.20
ind8	311	58	3	129	36497	4.02
ind9	648	26	2	609	16067	1.62

The minimum period retiming results for the circuits of Table 1 are shown in Table 2. The initial clock period, the initial number of registers, the clock period after retiming, and the number of registers after retiming are shown in the columns denoted *initial CP*, *initial #reg*, *final CP*, and *final #reg*, respectively. Column *#steps* shows the actual number of performed mc-retiming steps. Columns titled *MINP* and *UREG* give the CPU times for *Reverse_Retiming* and updating the mc-retiming graph, respectively.

Table 2 shows a substantial reduction in the clock period for the benchmark circuits as well as for the industrial designs. This indicates that retiming is a very powerful optimization technique also for multiple-class circuits. All backward moves could be justified by our BDD-based approach such that no backtrack was necessary. Comparing CPU times it is important to notice that the impact of computing the mc-retiming bounds on the efficiency of the overall retiming approach is low. This holds even if very efficient acceleration techniques [16] are used for the sc-retiming algorithm. This stresses the theoretical results of Section 5.1. Please note that the large CPU time in column *UREG* is due to the BDD-based reset state computation, which we have implemented in a straight-forward way only.

7. Conclusion

In this paper we have presented basic theory and an approach to retiming synchronous circuits where different registers can be connected to different clock, asynchronous and synchronous reset, and load enable signals. In order to manage this variety of registers, we introduced the concept of

[1] As in [5], the initial state of each register is assumed to be zero.

Table 2. Results for clock period minimization

	initial		final			CPU(s)	
circuit	CP	#reg	CP	#reg	#steps	MINP	UREG
gcd	32	59	28	77	40	0.2	0.6
mm9b	70	26	58	27	19	0.1	0.0
mult16a	38	16	8	84	491	0.1	44.8
s344	28	15	19	25	25	0.0	0.0
s382	17	21	10	44	111	0.4	2.8
s400	17	21	10	48	116	0.9	2.6
s444	20	21	11	45	153	0.6	3.6
s526	14	21	9	84	146	0.7	8.2
s838.1	21	32	20	34	2	0.1	0.1
s953	27	29	22	34	7	0.1	0.1
s1238	31	18	30	82	121	0.1	1.9
s1423	66	74	59	160	513	0.2	170.7
s5378	32	163	27	185	63	0.6	0.3
s9234.1	55	135	51	135	32	0.6	0.3
s38417	65	1465	45	1629	952	6.6	289.5
ind1	11	318	10	322	4	0.4	6.9
ind2	19	103	14	277	687	0.7	3289.5
ind3	7	46	6	45	98	0.0	0.1
ind4	6	7	3	38	36	0.0	0.1
ind5	11	38	8	181	387	0.1	0.3
ind6	8	33	7	83	160	0.1	0.1
ind7	13	18	8	188	380	0.1	42.5
ind8	8	58	7	94	64	0.1	13.8
ind9	10	26	9	105	331	0.1	178.0

register classes. We defined a circuit transformation called mc-retiming which guarantees that the retimed circuit is a valid replacement of the original circuit. Moreover, no additional logic has to be added to the circuit in order to implement the transformation.

Our retiming approach is based on transforming the mc-retiming problem into the traditional retiming problem as introduced by Leiserson et al.. Thus, efficient state-of-the-art retiming algorithms can be applied for the mc-retiming problem. Experimental results show the efficiency of the presented mc-retiming approach. Being able to retime sequential circuits with multiple register classes will significantly increase the applicability of retiming to real designs.

Based on the proposed mc-retiming approach we are now working on extending our approach in order to handle timing constraints due to multiple-clock systems. A first concept has been presented in [8].

Acknowledgment

We are very grateful to Dr. Peter Vanbekbergen, Dr. Jean Christophe Madre, Dr. Peter Zepter, and Dr. Albert Wang for many valuable discussions. We also thank Synopsys Inc. for the opportunity of a summer internship. Finally, we highly appreciate the continuous support and steady interest of Prof. Kurt J. Antreich in our work.

References

[1] R. Camposano and P. G. Plöger. Retiming and high-level synthesis. In *International Workshop on High-Level-Synthesis*, pages 191–201, Nov. 1992.

[2] G. De Micheli. Synchronous logic synthesis: Algorithms for cycle-time minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(1):63–73, Jan. 1991.

[3] S. Dey, M. Potkonjak, and S. G. Rothweiler. Performance optimization of sequential circuits by eliminating retiming bottlenecks. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 504–509, Nov. 1992.

[4] K. Eckl and C. Legl. A new retiming approach for sequential circuits with multiple flip-flop classes. Technical Report TUM-LRE-98-1, Institute of Electronic Design Automation, Technical University of Munich, Apr. 1998.

[5] G. Even, I. Y. Spillinger, and L. Stok. Retiming revisited and reversed. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(3):348–357, Mar. 1996.

[6] A. T. Ishii. Retiming gated-clocks and precharged circuit structures. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 300–307, Nov. 1993.

[7] A. T. Ishii, C. E. Leiserson, and M. C. Papaefthymiou. Optimizing two-phase, level-clocked circuitry. In T. Knight and J. Savage, editors, *Advanced Research in VLSI and Parallel Systems: Proceedings of the 1992 Brown/MIT Conference*, pages 245–264. MIT Press, 1992.

[8] C. Legl, P. Vanbekbergen, and A. Wang. Retiming of edge-triggered circuits with multiple clocks and load enables. In *International Workshop on Logic Synthesis (IWLS)*, volume 1, May 1997.

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

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

[11] B. Lockyear and C. Ebeling. Optimal retiming of multi-phase, level-clocked circuits. In T. Knight and J. Savage, editors, *Advanced Research in VLSI and Parallel Systems: Proceedings of the 1992 Brown/MIT Conference*, pages 265–280. MIT Press, 1992.

[12] B. E. Lockyear. *Algorithms for Retiming Level-Clocked Circuits and their use in Increasing Circuit Robustness*. PhD thesis, University of Washington, Seattle, Washington, 1994.

[13] N. Maheshwari and S. S. Sapatnekar. An improved algorithm for minimum-area retiming. In *ACM/IEEE Design Automation Conference (DAC)*, pages 2–7, June 1997.

[14] N. Maheshwari and S. S. Sapatnekar. Minimum area retiming with equivalent initial states. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 216–219, Nov. 1997.

[15] S. Malik, E. M. Sentovich, R. K. Brayton, and A. Sangiovanni-Vincentelli. Retiming and resynthesis: Optimizing sequential networks with combinational techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(1):74–84, Jan. 1991.

[16] N. Shenoy and R. Rudell. Efficient implementation of retiming. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 226–233, Nov. 1994.

[17] N. V. Shenoy, K. J. Singh, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. On the temporal equivalence of sequential circuits. In *ACM/IEEE Design Automation Conference (DAC)*, pages 405–409, June 1992.

[18] V. Singhal, S. Malik, and R. K. Brayton. The case for retiming with explicit reset circuitry. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 618–625, Nov. 1996.

[19] H. J. Touati and R. K. Brayton. Computing the initial states of retimed circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(1):157–162, Jan. 1993.