AN EFFICIENT VERIFICATION METHOD FOR A CLASS OF MULTI-PHASE SEQUENTIAL CIRCUITS

François-R. Boyer[1], El Mostapha Aboulhamid[1], and Yvon Savaria[2]

[1] DIRO, Université de Montréal, 2920 Chemin de la Tour,
C.P. 6128, Succ. Centre-Ville, Montréal, Québec, Canada, H3C 3J7
{boyerf, aboulhamid}@IRO.Umontreal.CA
[2] DGEGI, École Polytechnique de Montréal, Québec, Canada
savaria@VLSI.PolyMtl.CA

ABSTRACT: Currently, many optimizations of sequential circuits, even as simple as retiming, are avoided due to the lack of verification tools that support them. Doing general sequential equivalence to compare the circuits is impractical for circuits of a reasonable size. On the other hand, combinational optimization is part of the design process, because tools and methods are available to ensure correctness and verify combinational circuits. We present a practical method to verify sequential circuits equivalence using combinational equivalence on a transformed circuit of the same size, for a class of circuits. The constraint imposed is that for each loop in the circuit, there must be a point in both circuits that are in correspondence. The circuits can have a different number of clock phases, and they can be transformed by other scheduling algorithms than retiming and multi-phase retiming.

1. INTRODUCTION

Synthesis and optimization of sequential circuits derive an implementation by a transformation of the initial specification, this transformation has to be verified to assert the conformance of the implementation to the specification. If no information is known about the transformation, a general sequential equivalence method must be used, and this can be impractical for circuits of a reasonable size. This means that transformations must have some known properties to be able to verify them in a practical time. On the other hand, to allow the best optimization we would like to impose as few constraints as possible on these transformations.

Currently, many optimizations of sequential circuits, even as simple as retiming, are avoided due to the lack of verification tools that support them. On the other hand, combinational optimization is part of the design process, because tools and methods are available to ensure correctness and verify combinational circuits.

There are many proposed solutions to the sequential equivalence problem. Some methods try to solve the general problem [1][2], but their complexity limits the size of circuits they can process. Some other methods will perform verification with constraints on the transformations, to have a smaller complexity, but they do not permit to verify multi-phases circuits, as those produced by the methods presented in [3] and [4].

The method presented by Ashar [5] exposes some flip-flops in the original circuit, to make the FSM representing the circuit complete-1-distinguishable, before the optimizations are applied. Then, to compare the two FSMs, instead of checking the reachable states in the product machine (in $O(2^{n+m})$), it checks the reachable states of individual machines (in $O(2^n+2^m)$), where m and n are the number of flip-flops in both FSMs. This is still impractical.

Huang [6] uses an approach based on ATPG, reducing the search space to a practical size by finding equivalence between internal signal pairs. Their goal was to prove equivalence after retiming only, so the signal pairs are easy to find, but it may not handle circuits after resynthesis.

Bischoff [7] presents a method composed of three conservative algorithms. First, the circuit is cut into slices manually, then, on each slice, the algorithms are used from the fastest to the slowest until a proof, or a counter proof, is found. The simplest algorithm is normal combinational equivalence checking, which does not allow any sequential elements. The second algorithm uses Timed Ternary BDD to represent function of inputs delayed by some signal events. It allows latches with any function as clock input, but feedback loops are not supported. In addition, different signals are considered independent, so it cannot be used for multi-phases circuits, as most of these circuits are incorrect under those assumptions. If the first two methods fail, general FSM equivalence is used.

Stoffel [2] simplifies the general FSM equivalence using a decomposition of the circuit to simplify the reachable states function. As this decomposition can take a long time, in practice, an approximation is used which will give a superset of the reachable states. Checking with this superset can lead to false negatives.

Aït Mohamed [1] applies model checking to compare circuits. The idea is to use MDGs instead of ROBDDs, which permits symbols and uninterpreted functions instead of converting everything to binary representation. This may help to avoid the state explosion problem, but if the number of variables is large, or if the design is at gate level and cannot be

extracted at a higher level of abstraction, the method can still take an excessive processing time.

Ranjan et al. [8] presented a verification technique which permits retiming combined with combinational optimization sequential synthesis for a class of circuits. In particular, they require certain constraints to be met on the feedback paths of the latches involved in the retiming process. For a general circuit, these constraints can be satisfied by forcing some latches to be immovable. Equivalence checking after performing repeated retiming and resynthesis on this class of circuit reduces to a combinational verification problem. This paper shows that this class can be extended and that no latches have to be immovable. The proposed method is mainly an extension of [8], with the following contributions:

- The class of circuits is extended to cover those obtained by general scheduling methods, as those used in software pipelining, instead of retiming. The class of retimed circuits being a strict subset of the class we cover.
- The method permits to verify multi-phase circuits, as opposed to single phased sequential circuits.
- The new method does not require any constraints on the position of registers, thus allowing more optimization.
- Rational time can be handled instead of integer time. Not handling it would require that schedules with rational times be converted to integer time, which may slow the circuit or demand to use loop unrolling.

However, some point must be observable for each loop in the circuit graph. That is, the correspondence between a signal in the original circuit and in the circuit to verify must be known at one point in each sequential loop of the circuit. If the method is applied more than once, we should keep a set of correspondence points between successive iterations, otherwise we may be unable to complete the verification. Nevertheless, there may not be any known correspondence between the original circuit and the final one.

Section 2 introduces the circuits and the class of transformations we want to process. Section 3 presents the method for single-phase circuits, proposed by Ranjan et al. [8]. A similar method is then developed for multi-phase circuits in section 4, and a method to consider gate delays, which needs more attention when there is more than one clock phase, is developed in section 5. A complete example of our verification method applied to a multi-phase circuit is presented in section 6.

2. PRELIMINARIES

A circuit is modeled as a set of gates (any calculation element without memory) interconnected by wires that can have memory elements on them. The memory elements are edge-triggered flip-flops and are often called registers. All registers have clocks with the same period, which is considered to be the time unit.

2.1 Combinational Optimization

Between registers, the combinational circuit can be changed for an equivalent one. This transformation does not change the state space, but changes the topology of the circuit. In addition, the intermediate results are not the same, and the signals may not be equivalent anymore, except for those going in and out of registers.

2.2 Retiming

For circuits with a single clock, the retiming technique, presented by Leiserson and Saxe [9], permits to move registers.

Retiming can be seen as a displacement assigned to each gate, which removes registers from one side and puts them on the other. As shown in Fig. 1, retiming of +1 for some gate means "remove one register of each output of the gate and add one to each of its inputs". This transformation does not change the behavior of the circuit, but it changes the time at which calculations are made, moving them from one clock cycle to the other. This also changes the state space, which makes some verification methods unusable.

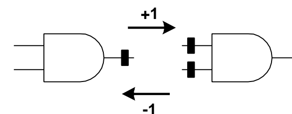

Fig. 1. Primitive retiming operations.

2.3 Fractional "Retiming" and Software Pipelining

It is not possible to move a fraction of register, but we can have multiple clock phases and apply a retiming of one phase instead of a whole cycle [4]. As an alternative, we can place registers and then activate them at the right time using different phases [3]. These methods permit, in some cases, to have circuits with higher throughput than with optimal retiming.

The method we presented in [3] uses modulo scheduling, known in software pipelining, to find an optimal schedule for the operations in the circuit. As this schedule can have fractional time, multiple clock phases are used to implement it. The method is illustrated using the simple circuit of Fig. 2.

The maximal throughput (minimal period) is found using a known algorithm to solve minimal cost-to-time ratio cycle problem, the cost being the number of registers and the time being the combinational delays. That cycle is shown in gray on Fig. 2, and has a throughput of $^1/_{10}$ which gives a period of 10. Note that the best possible period that can be obtained by retiming for this example is 13, which is the optimal period if a single-phase single clocked circuit implementation is targeted.

Then, a valid schedule with that period (10 in our example) is found using Bellman-Ford's longest path algorithm, where the weights on the edges are the delay

of the source vertex minus the number of registers times the period ($w_l(e_{ij}) = d(v_i) - P\, w(e_{ij})$), as shown on Fig. 3. The schedule of a vertex relative to the origin used for the longest path is the length of the path. The As Late As Possible and As Soon As Possible schedules relative to v1 are in Table 1.

To place back registers, and find their clock phase, we use a graph for which the weights on the edges are the time between the start of an operation and the start of the operation that needs its result. More formally, the weight $w_s(e_{ij}) = s(v_j) - s(v_i)$, where $s(v_i)$ is the schedule of vertex v_i. To minimize the number of registers, they are only placed to cut any path that is longer than the period. We give a linear time method that provides good register placement, although the number of registers is not necessarily minimal.

The register placement is shown on Fig. 4. The phase is the distance of the register from the reference point, divided by the period. The phase relative to register 'e' is the fraction besides each register on Fig. 4.

An attractive feature of this method is that it permits to place registers almost anywhere without changing the speed of the circuit (if we neglect the propagation delay through latches). In fact, in the context where clock skews and registers delays are not yet considered, registers can always be added, but, of course, they can't always be removed. The added registers will be used only during the proof, so they can be considered perfect. If real registers are to be added, more precautions should be taken. To add a register, you simply place it on the input of an operation. Then, to find the phase, you add to the phase of some register the length of the path from that register to this new one, divided by the period, and take only the fractional part of the result. For example, in Fig. 4, to place a register between v3 and v5, you place it just before v5 and the distance from 'a' being 3, the period being 10 and the phase of 'a' being $^6/_{10}$, the resulting phase will be $^9/_{10}$.

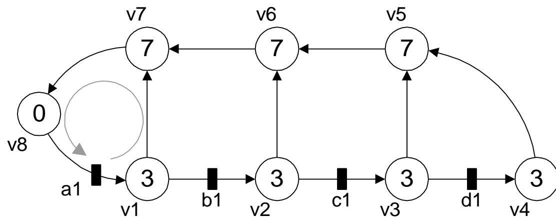

Fig. 2. A simple circuit. Delays of "gates" are shown on them.

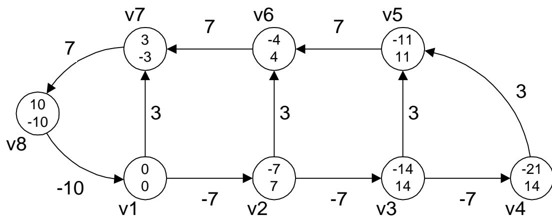

Fig. 3. Longest paths from/to v1 which are the ASAP and (minus) ALAP schedules.

Table 1. Schedules and mobility relative to v1.

Vertex	1	2	3	4	5	6	7	8
ASAP	0	-7	-14	-21	-11	-4	3	10
ALAP	0	-7	-14	-14	-11	-4	3	10
Mobility	0	0	0	7	0	0	0	0
Interval	0	-7	-14	[-21,-14]	-11	-4	3	10

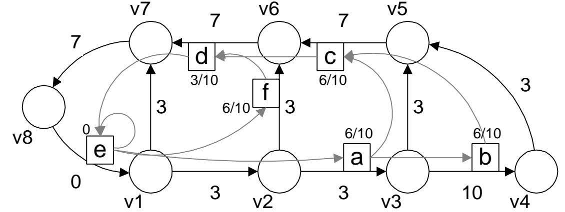

Fig. 4. Register placement for the schedule, with their clock phase.

3. SINGLE-PHASE CIRCUITS

For circuits where all registers are controlled by a single clock, an efficient verification method has been presented by Ranjan [8]. That method permits to verify equivalence with the original circuit after optimization by retiming and resynthesis (combinational optimization). In this section, we summarize the principal concepts presented in [8], which will then be extended to multi-phase circuits in the next section.

The idea is to first transform the circuit to an acyclic representation, then to apply the optimizations on that representation. Finally, to verify the equivalence, both the original and the optimized circuit, in acyclic representation, are transformed to combinational representations and are compared by a combinational equivalence checker.

3.1 Acyclic Sequential to Combinational

In a circuit without loop, the outputs depend only on the inputs. However, because the circuit can contain registers, the output will depend on the inputs at multiple, but finite, different moments. As there is only one clock phase, the time is an integer (the number of cycles).

Clocked Boolean Functions (CBF)

The CBF is a combinational representation of a sequential circuit. It gives the expression for the output in function of time.

For a signal s, the CBF $s(t)$ at time t is :

- If s is the output of a gate G with inputs $y_1 \ldots y_n$:
$$s(t) = f_G(y_1(t), \ldots, y_n(t))$$
- If s is the output of a register : $s(t) = y(t-1)$
- If s is a primary input : $s(t)$ is independent of $s(t')$ for $t \neq t'$

For example, to get the CBF of the circuit in Fig. 5, we proceed as follows. The CBF of each part is:

$$o(t) = c(t) \cdot d(t)$$
$$d(t) = c(t-1)$$
$$c(t) = b(t) \oplus a(t)$$
$$b(t) = a(t-1)$$

Then we substitute everything in $o(t)$ to have $o(t) = (a(t-1) \oplus a(t)) \cdot (a(t-2) \oplus a(t-1))$.
The CBF gives the behavior of the circuit at steady-state, which is when all registers have correct values (after the initialization phase).

THEOREM 1. Canonicity of CBF. [8]
If C_1 and C_2 are acyclic sequential circuits, where all registers are activated at the same time, and F_1 and F_2 are their CBFs, then we have that $F_1 \equiv F_2 \Leftrightarrow C_1 \equiv C_2$. The equivalence between F_1 and F_2 being combinational while the equivalence between C_1 and C_2 is at steady-state.

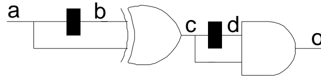

Fig. 5. An acyclic circuit.
Its CBF is $o(t) = (a(t-1) \oplus a(t)) \cdot (a(t-2) \oplus a(t-1))$.

3.2 Circuits with loops

When a circuit contains feedback loops, they must be broken to make the circuit acyclic before its CBF can be computed. The method does not accept purely combinational loops, so there is at least one register in every loop. A certain number of these registers are chosen as cut points, and made observable. The outputs of the chosen registers are now considered as primary inputs to the circuit. The cut points must be the same in the original circuit and the optimized one, so the cut is done before applying optimization transformations. This will put constraints on the possible transformations, as we cannot move a register used as a cut point. Therefore, we may want to use a minimal cut, to reduce the constraints. Once the acyclic version of the circuit has been optimized and verified (with the CBFs), it is "glued back" to have an optimized and verified cyclic circuit.

4. MULTI-PHASE CIRCUITS

Our objective is to verify sequential circuits with multiple clocks, but all having the same period. Some reference time is arbitrarily fixed (often the activation time of some register) and each register has an activation time, in the clock period, relative to that reference. Without loss of generality, the clock period is considered to be the time unit, so the activation time, which is the clock phase, will be a fraction in the interval [0,1[. For example, a register can be activated at time 0 and another at time $^1/_2$. Because events are periodic, and the period is 1, the register activated at time $^1/_2$ is also activated at time $1\,^1/_2$, and at time $2\,^1/_2$...
Registers are modeled as instantaneous, that is, if a register is activated at time 4, it's content will have the old value until time 4, excluded, and the new value from time 4, included. The new value being the value of its input just before the activation time. Note that using the period as the time unit will make it easier to compare different implementations with different period lengths.

4.1 Acyclic Sequential to Combinational

Using the same idea as for single-phase circuits, output signals can be defined as functions of time, but here the time will be a rational number instead of an integer.

Fractional CBF

From the register model described above, a register that has y as input, and that is activated by the clock phase $\boldsymbol{f}$, will have as output:

$$s(t) = y(\lfloor t - \boldsymbol{f} \rfloor + \boldsymbol{f} - \boldsymbol{e})$$

This means that $s(t) = y(t - \boldsymbol{e})$ if the fractional part of t is $\boldsymbol{f}$, and it will keep that value for a whole cycle, until $t + 1$, where it will take the new input value. The value $\boldsymbol{e}$ can be seen as a positive rational smaller than the time resolution in the system under consideration. In other words for every possible time t in the system $\lfloor t - \boldsymbol{e} \rfloor = \lceil t \rceil - 1$.
For the combinational elements, it is exactly as in the single-phase CBFs described in section 3.1. For inputs, it is slightly different, because the phase at which the input changes must be known. Like for the registers, the inputs are changing only once per cycle, at a specified phase. The following can be said of an input s which changes on phase $\boldsymbol{f}$:

$$s(t) = s(\lfloor t - \boldsymbol{f} \rfloor + \boldsymbol{f})$$

and $s(t)$ is independent of $s(t')$ for $\lfloor t - \boldsymbol{f} \rfloor \neq \lfloor t' - \boldsymbol{f} \rfloor$

Simplification of Floor Operators

If the construction just described is applied blindly, there will be many floor operators everywhere in the expression.
Here are formally defined rules sufficient to simplify the generated expressions. The *Simp* function does the simplification using the following rules:

$Simp(\lfloor \lfloor x \rfloor + y \rfloor) = Simp(\lfloor x \rfloor) + Simp(\lfloor y \rfloor)$	
$Simp(\lfloor -\boldsymbol{e} \rfloor) = -1$	
$Simp(\lfloor x - \boldsymbol{e} \rfloor) = \lceil x \rceil - 1$	When x is a constant.
$Simp(f) = Map(Simp, f)$	For each argument of a function.

These simple rules will be used to simplify the description of a circuit. At the end of the description of both circuits (original and transformed one) the expressions at the cut points will be single-phased. The problem is then completely reduced to comparison using integer time, which is the work presented in [8].

4.2 Circuits with Loops

As with single-phase circuits, loops are broken to have an acyclic circuit before its CBF is computed. However, with multi-phase circuits, it is possible to cut anywhere, if verification in multiple steps is permitted. Having multi-phase circuits, as those presented in [3], makes it possible to add registers anywhere, as explained in section 2.3. This permits to add a register where a cut is requested, and then prove locally that this

does not change the behavior of the circuit. The cut points can then be chosen anywhere, but we may want to minimize the number of cuts, to minimize constraints on the transformations, or choose some place that seems to be optimized as much as we can.

Local CBF and Local Proofs

To do local proofs, instead of making the CBFs going from outputs to the primary inputs of the circuit, a CBF is made from a register to its predecessors. A register *a* is considered the predecessor of *b* if there exists a path from *a* to *b* without going through a register. The local CBF of each register can be found even if there are loops in the circuit, as there are no combinational loops. To prove that adding, or removing, a register does not change the behavior of the circuit, the local CBFs can be used. The local CBF for the concerned register and the local CBFs of all registers of which it is the predecessor are found, when the register is there and when it is removed. Then we prove that the CBFs are equivalent.

Multiple Steps Proof

Performing proofs in multiple steps, instead of a single global proof, gives a better choice for cut points. The circuit can be cut anywhere and then optimized without necessarily having a register on the cut point, neither in the original circuit nor in the optimized one. In that case, no register has a fixed position. This gives a greater freedom for optimization, and may result in better circuits. In addition, single-phase circuits can also be considered as multi-phase circuits, to be able to do local proofs at the beginning and at the end of the whole proof, even if the final circuit has to be single-phased. Local proofs at the beginning show that adding registers at the cut points does not change the behavior of the circuit, while the proofs at the end are there to show that removing those registers is also correct. In certain cases, depending on the optimizations done, the added registers cannot be removed and must be kept in the final circuit.

5. CONSIDERING GATE DELAYS

All the discussions so far assume that the clock period gives enough time for combinational circuits to compute the next values of registers. Of course, a complete proof should also check the delays and we present the simple delay model we are using.

For a combinational circuit that computes $s = F(y_1, y_2, \ldots)$. If the short path takes at least *min* units of time, and the long path takes less than *max* units of time, *s* can be given in function of time as follows:

$$s(t) = \begin{cases} F(y_1(t-min), y_2(t-min), \ldots) & \text{If } y_1 \ldots \text{ are steady on }]t-max, t-min]. \\ \text{Undefined} & \text{Otherwise.} \end{cases}$$

With a delay model where $min = 0$, we can say that $s(t) = F(y_1(t), y_2(t), \ldots)$ if inputs are steady from time $t-max$. That is exactly the $s(t)$ which we used in the CBFs, so it is only required to prove, for each local CBF, that the inputs are steady for at least *max* units of time before the clock phase. This also shows that our CBFs are not always correct in a delay model where $min > 0$. That is, it may not be able to prove equivalence of wave-pipelined circuits. The general case $s(t)$, shown above, must be used in the CBFs if those circuits are to be verified.

6. EXAMPLE OF PROOF

We will verify that the optimized circuit of Fig. 4 is equivalent to the original circuit of Fig. 2. In the original circuit, the four registers (a1, b1, c1 and d1) are all activated by the same clock, and after optimization, the circuit has six registers and three phases (0, $^3/_{10}$ and $^6/_{10}$). For this proof, we will consider that each function is different and unknown. The functions will be named by the vertex name, possibly followed by a letter, to indicate outgoing edge to which it is associated, and then an ‘f’. The output of a vertex is noted as its function but without the ‘f’ and different caps are used to distinguish circuits. The output of a register is simply noted by the name of the register. Using this notation, and simplifications rules from section 4.1, we used Mathematica to find the CBF for each circuit, using register a1/e as cut point. Then we asked Mathematica to prove that they are equivalent.

The Mathematica session is presented on Fig. 6. First, the rules are entered as is. Note that the rules are not added to the “Simplify” of Mathematica, so that only the entered rules, and constants evaluation, will be used to simplify the expressions. Instead of redefining the equality, to be able to compare primary inputs according to the rule, we add a simplification rule, for each primary inputs, that maps equal inputs to the same expression. Therefore, simplification applied to a primary input will give the expression at the time it last changed. The rules for combinational elements, registers and primary inputs used for the description of the multi-phase circuit are from Section 4.1, and those for the single-phase circuit are from Section 3.1.

Now we want to prove that if ‘a1’ and ‘e’ have the same value then the next value of ‘a1’ will also be the same as the next value of ‘e’. To do that, the expressions are compared, but we must ensure to compare using the right times. As the time unit is the period of the circuit, the time is already scaled for the comparison, but there may be a phase offset. After simplification, the expression for ‘e’ has only one phase, as it is the output of a single register, but for the expression to be equal to that of ‘a1’ we must ensure that they are compared at their according phases. Once they are in phase, any combinational equivalence tools, as those used by [8], can be used to compare the expressions. Here we use Mathematica to compare the next values assuming that they are currently equal.

7. CONCLUSION

We have shown that it is possible to verify the equivalence between multi-phase circuits in a reasonable amount of time, if some points are kept observable. These points, on the other hand, do not have to be on registers if we accept to have proofs in multiple steps. All the proofs that are done are equivalence between two combinational expressions, which is much faster, in general, than equivalence proofs on sequential circuits. Since the final expressions contain only integer time, we have reduced our problem to the verification problem resolved in [8]. From that, we deduce that the experimental results will give the same execution time as in [8].
Therefore, the presented method permits more optimizations, with multi-phase circuits, and these circuits are still verifiable.
To have better optimizations, a resynthesis method specially designed for the multi-phases circuits should be developed. In addition, some work could be done on how the circuit should be cut to have the desired freedom for optimization.

8. REFERENCES

[1] O. Aït Mohamed, E. Cerny, and X. Song, "MDG-based Verification by Retiming and Combinational Transformations", *IEEE Great Lakes Symposium on VLSI*, Lafayette, Louisiana, Feb. 19-21, 1998.

[2] D. Stoffel, and W. Kunz, "Record & Play: A Structural Fixed Point Iteration for Sequential Circuit Verification", *IEEE/ACM Int. Conf. Computer-Aided Design*, San Jose, CA, Nov. 9-13, 1997.

[3] F. R. Boyer, E. M. Aboulhamid, Y. Savaria, and I. E. Bennour, "Optimal design of synchronous circuits using software pipelining techniques", *IEEE Int. Conf. Computer Design*, Austin, TX, Oct. 5-7, 1998, pp. 62-67.

[4] A. T. Ishii, C. E. Leiserson, and M. C. Papaefthymiou, "Optimizing two-phase, level-clocked circuitry", *Journal of the ACM*, vol. 44, no. 1, Jan. 1997, pp. 148-199.

[5] P. Ashar, A. Gupta, and S. Malik, "Using Complete-1-Distinguishability for FSM Equivalence Checking", *IEEE/ACM Int. Conf. Computer-Aided Design*, San Jose, CA, Nov. 10-14, 1996.

[6] S. Y. Huang, K. T. Cheng, and K. C. Chen, "On Verifying the Correctness of Retimed Circuits", *IEEE Great Lakes Symposium on VLSI*, Ames, Iowa, Mar. 22-23, 1996, pp. 277-280.

[7] G. P. Bischoff, K. S. Brace, S. Jain, and R. Razdan, "Formal Implementation Verification of the Bus Interface Unit for the Alpha 21264 Microprocessor", *IEEE/ACM Int. Conf. Computer Design*, Austin, TX, Oct. 12-15, 1997.

[8] R. K. Ranjan, V. Singhal, F. Somenzi, and R. K. Brayton, "Using Combinational Verification for Sequential Circuits", *Design Automation and Test in Europe*, Munich, Germany, Mar. 9-12, 1999.

[9] C. E. Leiserson, and J. B. Saxe, "Retiming synchronous circuitry", *Algorithmica*, vol. 6, no.1, 1991, pp. 3-35.


```
Simpl[⌊⌊x_⌋ + y_⌋] := Simpl[⌊x⌋] + Simpl[⌊y⌋];
Simpl[⌊-ϵ⌋] := -1;
Simpl[⌊x_ - ϵ⌋] := ⌈x⌉ - 1;
Simpl[f_] := Map[Simpl, f];

PrimaryInput[in_, ϕ_] := (Simpl[in[t_]] := in[Simpl[⌊t - ϕ⌋ + ϕ]]);
Register[in_, time_, ϕ_] := in[⌊time - ϕ⌋ + ϕ - ϵ];

PrimaryInput[e, 0];
v1h[t_] := v1hf[e[t]];    v1d[t_] := v1df[e[t]];
v2h[t_] := v2hf[v1d[t]];  v2d[t_] := v2df[v1d[t]];
v3h[t_] := v3hf[a[t]];    v3d[t_] := v3df[a[t]];
v4h[t_] := v4hf[b[t]];
v5[t_] := v5f[v3h[t], v4h[t]];
v6[t_] := v6f[f[t], c[t]];
v7[t_] := v7f[v1h[t], d[t]];
a[t_] := Register[v2d, t, 6/10];
b[t_] := Register[v3d, t, 6/10];
c[t_] := Register[v5, t, 6/10];
d[t_] := Register[v6, t, 3/10];
f[t_] := Register[v2h, t, 6/10];
nexte[t_] := Register[v7, t, 0];

Simpl[nexte[t]] // TraditionalForm
```

$$v7f(v1hf(e(\lfloor t \rfloor - 1)), v6f(v2hf(v1df(e(\lfloor t \rfloor - 2))), v5f(v3hf(v2df(v1df(e(\lfloor t \rfloor - 3)))), v4hf(v3df(v2df(v1df(e(\lfloor t \rfloor - 4))))))))$$

```
V1h[t_] := v1hf[a1[t]];   V1d[t_] := v1df[a1[t]];
V2h[t_] := v2hf[b1[t]];   V2d[t_] := v2df[b1[t]];
V3h[t_] := v3hf[c1[t]];   V3d[t_] := v3df[c1[t]];
V4h[t_] := v4hf[d1[t]];
V5[t_] := v5f[V3h[t], V4h[t]];
V6[t_] := v6f[V2h[t], V5[t]];
V7[t_] := v7f[V1h[t], V6[t]];
b1[t_] := V1d[t - 1];
c1[t_] := V2d[t - 1];
d1[t_] := V3d[t - 1];
nexta1[t_] := V7[t - 1];

nexta1[⌊t⌋] // TraditionalForm
```

$$v7f(v1hf(a1(\lfloor t \rfloor - 1)), v6f(v2hf(v1df(a1(\lfloor t \rfloor - 2))), v5f(v3hf(v2df(v1df(a1(\lfloor t \rfloor - 3)))), v4hf(v3df(v2df(v1df(a1(\lfloor t \rfloor - 4))))))))$$

```
Simpl[nexte[t + ϕ]] == nexta1[⌊t⌋] /. {e → a1, ϕ → 0}
```

True

Fig. 6. Mathematica session of the proof: returns True.