Equivalence Checking of Retimed Circuits

by

Karolína Netolická

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the
Massachusetts Institute of Technology

June 2005

Author_____
            Department of Electrical Engineering and Computer Science
                                               May 19, 2005

Certified by_____
                                              Phillip Baraona
                                VI-A Company Thesis Supervisor

Certified by_____
                                                       Arvind
                                     M.I.T. Thesis Supervisor

Accepted by_____
                                            Arthur C. Smith
                  Chairman, Department Committee on Graduate Theses

Equivalence Checking of Retimed Circuits

by

Karolína Netolická

ABSTRACT

This thesis addresses the problem of verifying the equivalence of two circuits, one or both of which have undergone register retiming as well as logic resynthesis. The aim of the thesis is to improve the ability of Formality, an equivalence checking tool written at Synopsys, to handle retimed circuits. At the beginning of this project Formality already had an implementation of peripheral retiming, an algorithm that can handle a large set of retimed circuits. In this thesis, I explain the performance, usability and special case coverage problems found in the original implementation. I review other retiming verification algorithms and conclude that none of them would perform satisfactorily in Formality. Finally, I explain the modifications made to peripheral retiming in order to solve some of the identified issues and propose partial solutions for the problems that have not been solved yet.

Thesis Supervisor: Arvind

Title: Professor of Computer Science and Engineering

Acknowledgements

1. Introduction

This thesis addresses the problem of verifying the equivalence of two circuits, one or both of which have undergone register retiming as well as logic resynthesis. The aim of the thesis is to improve the ability of Formality to handle retimed circuits.

Formality is an equivalence checking tool written at Synopsys that takes as input two digital designs and proves or disproves their functional equivalence. To accomplish this task without having to perform sequential verification, Formality first finds a one-to-one matching between the registers in the two designs and then carries out combinational verification of the logic between them. This approach works when the changes between the two designs do not span across registers.

However, Synopsys' Design Compiler and other synthesis tools have been improving their capabilities to handle register retiming. Register retiming is a circuit optimization technique that aims to shorten the clock cycle or reduce circuit area by moving registers forward or backward in a circuit design. Unlike most circuit optimization techniques, the changes made by retiming span across register boundaries, thus confusing the register matching algorithms. The purpose of register retiming in Formality is to rearrange the registers in such a way that Formality's matching algorithms can identify matching registers.

Prior to the beginning of this project, Formality already included an algorithm that could handle a large set of retimed circuits. However, there was room for improvement both in terms of performance and the ability to handle certain special cases. The aim of this thesis project was to identify and implement ways to improve the algorithm.

In the rest of this thesis, I first give some background on Formality and combinational verification. After that, I explain register retiming and Formality's retiming algorithm. After comparing the algorithm to other retiming verification algorithms, I conclude that the one used by Formality is the one that is best suited to Formality's needs. Finally, I explain the modifications we made to the algorithm as part of this project and the improvements we achieved.

2. Formality

Synopsys is an EDA (Electronic Design Automation) company whose product line ranges from logic synthesis and verification tools to layout tools. These tools are designed to work together to facilitate a unified design flow. One of Synopsys' products is Formality, an equivalence checking tool that uses formal verification methods to prove or disprove the functional equivalence of two circuit designs.

A circuit usually undergoes many transformations as it evolves from a high-level description in a register transfer level language such as Verilog or VHDL to a gate-level design. Many of the steps in this process are carried out automatically by synthesis software. Formality's role in the design cycle is to ensure that no bugs were introduced into the circuit during any of these steps. Formality's ability to perform RTL-to-RTL, RTL-to-gate and gate-to-gate verification makes it applicable to many stages of the design cycle.

Formality takes as input two versions of a circuit designs. The first version is the *reference design,* a "golden design" which is assumed to meet the design specification. The second design is called the *implementation design.* In most cases, the implementation design is obtained by applying a series of synthesis or optimization steps to the reference design. Formality's goal is to prove or disprove the functional equivalence of these two designs. Two circuits are functionally equivalent if for every input sequence there exists an initial state for registers in each of the designs such that both designs produce identical output sequences.

As part of Synopsys' suite of tools, Formality must be able to successfully verify any operation done by Synopsys' Design Compiler (DC). DC is a logic synthesis tool that translates a circuit from a register transfer level (RTL) description to a gate-level description and performs logic optimization. Previously, Formality's retiming algorithm was sufficiently powerful to support most of DC's retiming operations. However, DC has recently enhanced its register retiming capabilities, and as a result Formality now must be able to verify a wider range of retimed designs.

The next section explains Formality's verification flow and why retiming poses a challenge to combinational verification tools.

2.1 Formality's Verification Strategy

Formality uses combinational verification techniques to carry out the equivalence proof. To see how Formality transforms the verification of a sequential circuit into a combinational problem, consider the Mealy model of a sequential circuit shown in Figure 1 [1, page 85]. Every sequential circuit can be thought of as being composed of a register bank with one data input and one data output per each register, and a block of combinational logic with m primary inputs, n primary outputs, p inputs coming from the p registers and p outputs that serve as inputs to the registers. To verify equivalence of two of these circuits, we first try to find a one-to-one correspondence between the registers in the two designs and between the primary inputs and outputs of the two designs. If such a correspondence is found, it is sufficient to prove the equivalence of the combinational logic connecting all of these matched components in order to prove the equivalence of the entire circuits. This is a very important simplification because simulating a circuit as a finite state machine is very expensive.
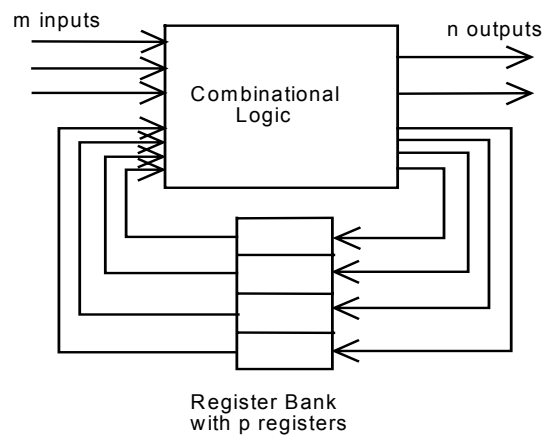

Figure 1: Mealy model of sequential circuits

However, this method is applicable only when the registers in the two designs can be matched. Register retiming changes the number, names and location of the registers in the circuit, thereby eliminating the one-to-one correspondence between the registers in the two designs. Therefore, algorithms for verification of retimed circuits must either not require register matching, or must somehow modify the circuit to enable the registers to be matched.

Figure 2 summarizes how register retiming fits into Formality's verification flow. Formality supports the concept of hierarchical verification, where components of a design, called cells, may themselves be complex circuit designs. To use Formality on a retimed design, the user must specify which cell in which design has been retimed, using the following command:

```
set_parameter –retimed i:/TOP/add32
```

The retimed cell could be the top-level design, or a cell further down the hierarchy. Formality calls the retiming algorithm on both the retimed cell and the cell that matches it in the other design; therefore, it is necessary to match the cells in the reference and implementation design prior to carrying out register retiming. After the cells have been matched, Formality prepares the appropriate cells for retiming by "flattening" them, which means recursively substituting each cell within the retimed cell with its lower-lever description, until the entire retimed cell is described using only primitive components such as gates and registers. After this step, Formality calls the register retiming algorithm on each of the two cells, and then proceeds with regular matching and verification.

| Matching of Cells | → | Flattening of retimed cells | → | Register Retiming | → | Matching of inputs, outputs and registers | → | Verification |

Figure 2: Formality verification flow with retiming

The next section gives an overview of register retiming and some of the subtleties involved.

3. Register Retiming

Register retiming is a circuit optimization technique that moves registers forward or backward across combinational elements in a circuit. The aim of this procedure is to shorten the clock cycle or reduce circuit area.

3.1 Basics of Register retiming
There are two basic types of register retiming: Forward retiming and backward retiming. Forward retiming refers to removing a fixed number of registers from each input or a gate and inserting the same number of registers at the output. Conversely, in backward retiming we remove a fixed number of registers from the output of a gate and insert the same number of registers in front of each input [2].

Let us first consider an example of forward retiming in a circuit with simple edge-triggered flip-flops. In the first part of figure 3, each input of the OR gate is driven by a q-pin of a register. Furthermore, both of these registers have identical clock signals. Therefore, it is possible to remove one register from each input of the OR gate and instead insert one register on the output. Notice that the two registers in the original version became one register in the retimed version; therefore, the new circuit has a smaller area. Even more importantly, if we resynthesize the logic of the retimed circuit, it is possible to eliminate the first two gates, thereby making the circuit even smaller.
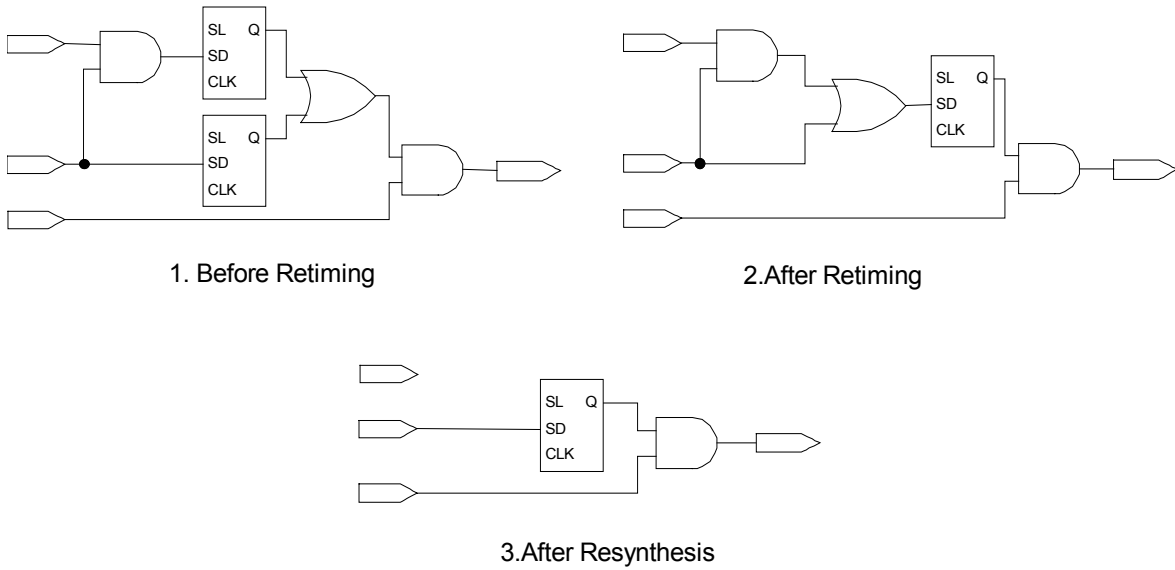

1. Before Retiming   2. After Retiming

3. After Resynthesis

Figure 3: Forward Retiming

Conversely, in backward retiming, a register is deleted from all the direct fanouts of a gate and a new register is inserted in front of every input of the gate. Figure 4 illustrates this concept and also shows how retiming can be used to shorten the clock cycle. Notice that before retiming, the critical path of the circuit in Figure 4 went through three gates, whereas after retiming, this path has been shortened to two gates. This allows us to increase the clock speed.
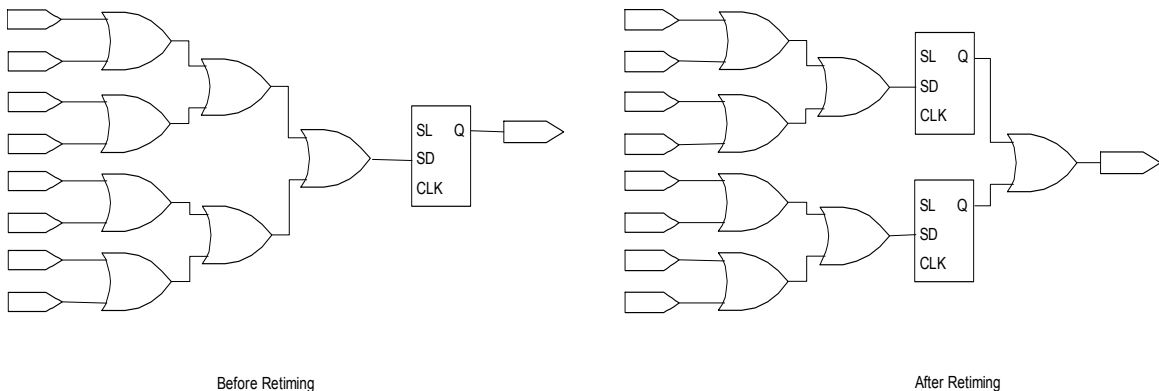

Before Retiming   After Retiming

Figure 4: Backward retiming

In the next section, I present a more formal definition of retiming.

3.2 Formalizing the Problem

To simplify the handling of the circuit, we use notation developed by Leiserson et. al in [3]. As shown in Figure 5, each circuit can be represented as a directed graph $G = (V, E)$, where each combinational element, each primary input and each primary output corresponds to a node v and each connection between two elements translates to an edge $e(u,v)$. Each edge has a weight $w(e) \geq 0$ which represents the number of sequential elements on that connection. We use the term *isomorphic circuits* to denote a pair of circuits whose graphs are isomorphic if we ignore edge weights. This is the case when the only differences between the two circuits are caused by register retiming; in other words, neither of the two circuits underwent logic resynthesis or any further optimization after register retiming.
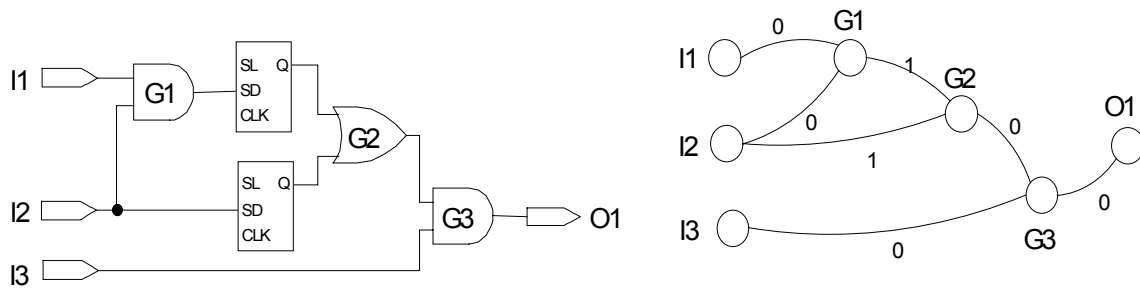

Figure 5: Representing a circuit as a graph

A "retiming" of the graph is an assignment of integers $r(v)$ for each vertex v such that for each edge $e(u,v)$,

$$w'(e) = w(e) + r(v) - r(u);$$
$$w'(e) \geq 0$$

Where the new set of weights, $w'(e)$, represents the new locations of the registers after retiming.

Intuitively, $r(v)$ represents the number of latches shifted backward through element v (from the outputs to the inputs) as a result of the retiming. There are usually additional constraints on the weights imposed by clock period requirements and combinational delays. This system of equations and constraints is an integer linear program for which there exist efficient algorithms.

3.3 Register Classes

Until now the discussion has been restricted to D flip-flops. For the rest of this thesis, consider a register with the following signals:

D: synchronous data
Q: output
CLK: clock
SL: synchronous load, also known as load enable

AL: asynchronous load
AD: asynchronous data (loaded when AL is set to 1 – level-sensitive)

This general sequential element can be made into a simple edge-triggered flip-flop, an edge-triggered flip-flop with a load-enable signal or a level-sensitive latch simply by setting the unnecessary signals to the appropriate constant values.

To take the additional control signals into account, we define the term "register class" as a set of sequential elements that have functionally equivalent clock, AL and SL signals. This allows us to formulate the following rule: a set of registers can only be moved across a combinational element if all of the registers belong to the same register class. Figure 6 illustrates why this restriction is necessary. Consider the case where we have an OR gate with a register on each if its inputs, and at time t R1's enable signal is set to 0 and R2's enable signal is 1. If we tried to retime these registers across the OR gate, we would not be able to find a value for SL of the new register that would preserve the behavior of the original circuit because no such value exists.
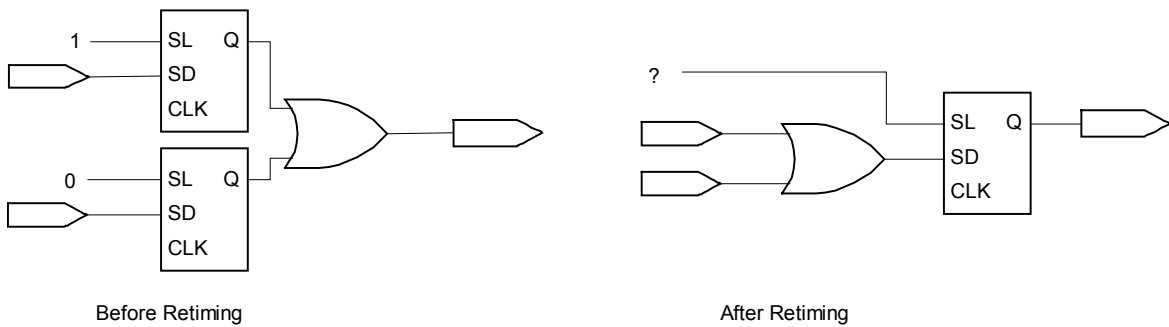

Before Retiming   After Retiming

Figure 6: Registers must belong to the same class in order to be retimed

3.4 Reset State

Another issue we must consider is reset state. Many circuits include an external asynchronous load input to reset the registers to a known initial state when the circuit is switched on. After every retiming operation, it is necessary to compute the reset state of the new register so that the new circuit behaves equivalently to the old one after being reset. For forward retiming, the reset state of the new register(s) is computed by simulating the reset state forward across the gate. For backward retiming, however, it is not always possible to find an equivalent reset state. Consider the example shown in Figure 7. No combination of AD values for the two retimed registers can make each of the fanouts of the OR gate have a different value after one clock tick.
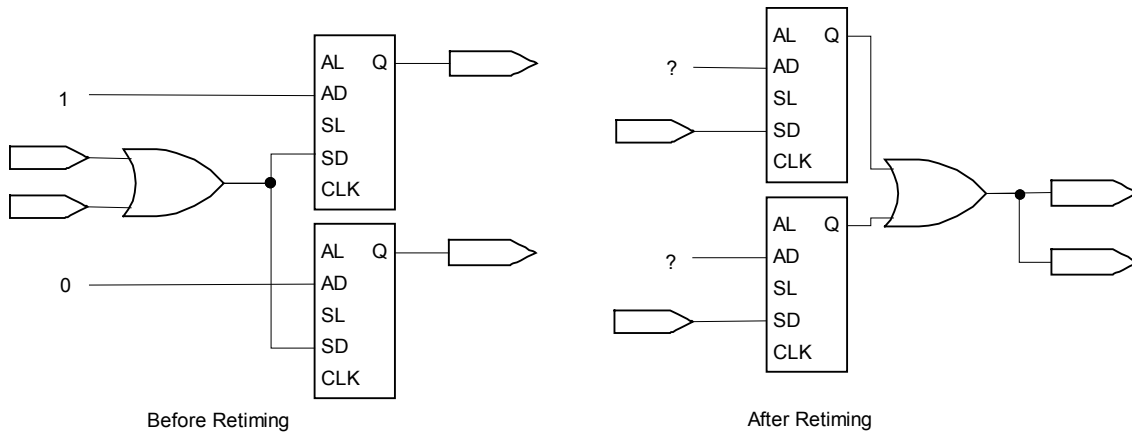
Before Retiming   **After Retiming**

Figure 7: No reset state

This section explained how register retiming works and what it is used for. In the next few sections, I talk about how to verify circuits that have been retimed. Although some of the verification algorithms have the word "retiming" in them, they are not algorithms that optimize circuits using register retiming; rather, they somehow "undo" the retiming so the circuits can be verified.

4. Peripheral Retiming

To carry out the "register retiming" step in the flowchart of Figure 2, Formality uses an algorithm called peripheral retiming [4]. The original version of this algorithm moves all registers to the periphery of the circuit, that is, stacks all registers in the circuit at either the primary inputs or primary outputs. The algorithm can be summarized as follows:

1. Mark all registers inside sequential loops as fixed
2. Retime all non-fixed registers as far forward as possible
3. For all non-fixed registers that did not reach the output,
    retime them as far backward as possible
4. While there are non-peripheral non-fixed registers left,
    i. Duplicate their input logic cones
    ii. Retime them backward as far as possible

An example of how this algorithm works is shown in Figure 8. In part a), the top register is marked as fixed and will not be retimed; the bottom register is retimed forward. In part b) we see that the top copy of the bottom register did not reach the output; therefore, in part c) its input logic is duplicated and in part d) the register is retimed backward to the inputs of the circuit.

The algorithm does not require the circuits to be isomorphic and so it handles designs which have been resynthesized after retiming. However, the original algorithm and its implementation had several shortcomings. These shortcomings can be broken down into three categories: performance, usability, and coverage.
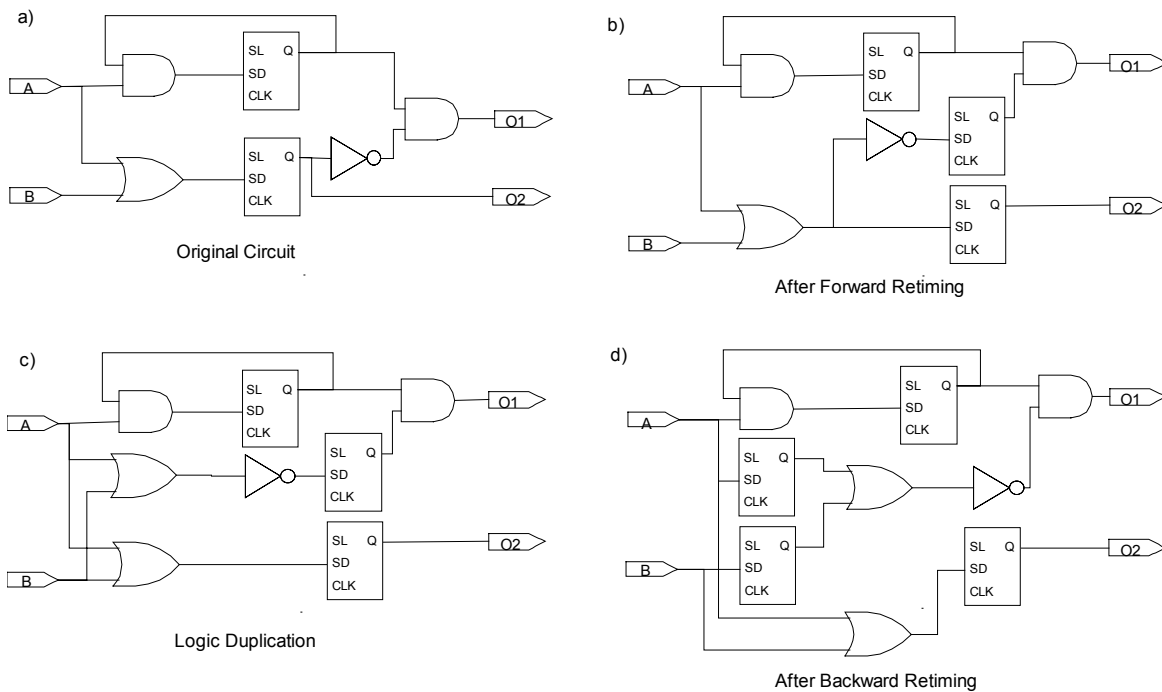
10

a) Original Circuit

b) After Forward Retiming

c) Logic Duplication

d) After Backward Retiming

Figure 8: An example of peripheral retiming

4.1 Performance Issues

Performance issues cause the verification to take a long time or use a lot of memory. The peripheral retiming algorithm suffered from two performance issues:

- By moving all registers to the periphery, the algorithm removed all opportunities for breaking up the logic cones at register boundaries and thus simplifying the problem. As a result, the subsequent combinational verification step was very slow because it had to verify the entire circuit as one huge logic cone.

- For a large circuit, the duplication of input logic often made the circuit area grow disproportionately. This greatly increased the amount of memory the verification consumed, as well as lengthened the verification time.

4.2 Usability Issues

Usability issues make the verification difficult to set up or difficult to debug if it fails. The original implementation of peripheral retiming had the following two usability issues:

- The duplication of input logic, apart from slowing down the verification, also made it much more difficult to debug a failing verification. When a verification fails, Formality displays schematics of both the reference and implementation designs with suggestions of where the problem areas might be. However, the excessive logic

duplication made it very difficult to relate these schematics to the original design the user submitted.

- Formality propagates information about port equivalence only after retiming. As a result, when several input pins on one of the retimed cells were equivalent to each other because they either had a common driver outside the retimed cell or because their drivers were set to be equivalent on the top level, register retiming did not infer this information and the user had to explicitly state all such equivalences in the verification setup.

4.3 Coverage Issues

These are special cases that were either overlooked in the initial implementation effort or that are very difficult to solve theoretically.

- It is impossible to retime a register that is part of a cycle to the periphery of the circuit. The original algorithm handled this issue by not moving registers inside sequential loops at all. This approach worked for cases where none of the loops in either the reference or implementation design had been retimed; however, when those loops had been retimed, the algorithm failed.

- A special case of the loop problem occurred when a register with an enable signal in one of the designs was replaced with a combination of a D-flip flop and a feedback multiplexer in the other design. The original code did not handle this case, although it could be solved by replacing all D-flip flops with feedback multiplexers by registers with an enable signal.

- Backward retiming introduced problems with reset state. While in most cases Formality was calculating the reset state correctly, in section 3.4 I showed that there are cases when the reset state is impossible to determine.

- The original implementation featured a very simplistic view of register classes. First, two registers were considered to be in the same class if and only if each control signal was equivalent to the corresponding control signal on the other register. However, when clock gating is introduced, two registers can be transparent at the same time and thus be in the same class even if the pairwise comparison of their control signals does not suggest so.

- Furthermore, two signals were regarded as equivalent only if they had a common driver, instead of if they computed the same function. This proved to be a big problem because Formality converts all registers in a design to a canonical form by putting a set of gates in front of each register. As a result, two latches were never identified as being in the same class.

- Unmatched test outputs caused registers to be retimed to non-equivalent locations. The implementation design often has extra outputs inserted whose purpose is to aid in testing the circuit. These unmatched outputs often caused situations when a

register in one design needed to be retimed backward while a register in the other did not.

Many of these issues could be handled by reimplementing the relevant part of the algorithm. However, some of these problems, such as the loop problem or the logic duplication, seemed fundamental enough that I decided to first investigate other retiming algorithms to determine whether there existed any other retiming verification algorithms that would perform better. The results of this investigation are detailed in the next section.

5. Other retiming verification algorithms

In this section, I review several retiming verification algorithms to determine whether the there exists an algorithm that does not suffer from the problems outlined in section 4.

As I have explained in section 2, retiming breaks Formality's combinational solvers by making it impossible to establish a one-to-one matching between the registers in the two designs. This suggests two ways one could conceivably solve this problem:

1. Move the registers back to where they were before retiming, or to other locations where they can be matched (peripheral retiming falls under this category)
2. Use sequential verification techniques which do not require register matching

The rest of this section looks at these approaches in turn, outlines the most widely known algorithms in each category and explains their advantages and disadvantages.

5.1 Algorithms to match retimed registers
These algorithms fall into two main categories. The first category consists of algorithms that work only on isomorphic circuits and use Leiserson and Saxe's retiming equation [3] to move registers in the two circuits to equivalent locations. The second category of algorithms consists of variations on peripheral retiming [4]. Because peripheral retiming was already explained in detail, I omit it in this section.

5.1.1 Algorithms based on the retiming equation
Consider two designs, one of which is a retimed version of the other (in other words, no modifications except retiming were done on the first design to obtain the second). Recall the retiming equation presented in section 3.2:

$$w'(e) = w(e) + r(v) - r(u),$$
$$w'(e) \geq 0$$

$$\text{or:} \quad r(u) = r(v) + w(e) - w'(e)$$

We know the values of $w(e)$ and $w'(e)$ from the graphs of the two designs. That means that if we can find a set of values $r(v)$ for all vertices v in one of the designs such that the retiming equation holds, we have proven that the second design is the result of a retiming

transformation performed on the first design. Because retiming does not change functionality, this also proves that the circuits are functionally equivalent.

While this method is simple and many efficient algorithms have been developed to find the values of $r(v)$ [5,6], it has the disadvantage of only working when neither of the circuits was resynthesized after retiming. This is an unacceptable restriction for Formality.

5.2 Sequential Verification Algorithms

There are two major techniques to carry out sequential verification: finite state machine (FSM) traversal and temporal logic. These algorithms are general enough to work for any two sequential circuits and do not take advantage of any retiming invariants. The next two sections explain these two techniques.

5.2.1 FSM Traversal

Any sequential circuit can be thought of as a finite state machine, where the state corresponds to the aggregate state of all the registers inside the circuit and the transitions between the states are determined by the Boolean logic connecting the registers. In addition, each state is associated with an output vector that is simply a vector of all the primary outputs.

Two finite state machines are equivalent if for every sequence of input vectors they produce the same sequence of output vectors.

Given this definition, the simplest way to check FSM equivalence is to exhaustively check each reachable state with all possible input vectors to make sure both circuits transition to an equivalent next state. Unfortunately, the runtime of this algorithm is exponential in both the number of primary inputs and number of registers in the circuits.

To alleviate this problem, implicit FSM traversal algorithms represent the states of the FSM not as bit vectors, but as Boolean functions. More specifically, on each iteration, the algorithm computes a Boolean function b_k that evaluates to true for all states that can be reached in k iterations. This approach shortens the maximum runtime to the sequential depth of the FSM (the maximum number steps needed to reach a state, over all states). While this is an improvement, in the worst case the sequential depth can still grow exponentially with the number of registers. For this reason, FSM traversal is not practical for large circuits and has not been widely deployed commercially [1].

5.2.2 Temporal Logic

Like FSM traversal, temporal logic is another method for solving general sequential circuits and does not rely on any retiming invariants. One algorithm using temporal logic was developed by Ranjan et. al. in [7]. The essence of the algorithm lies in treating Boolean variables at different instances in time as independent variables. For example, the function computed by circuit shown in Figure 9 can be represented by a Boolean function $o(t_0) = a(t_0) + a(t_{-1})$.
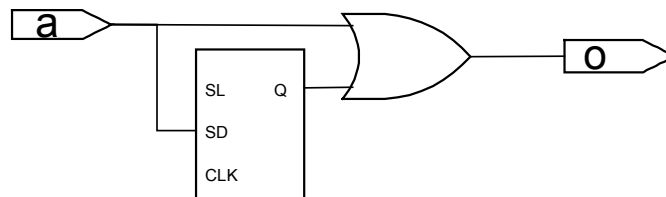

Figure 9: Temporal logic

The advantage of temporal logic is that it converts the sequential verification problem into a combinational one that can be handled efficiently with Formality's current combinational logic solvers. The disadvantage is that temporal logic is difficult to adapt to all the different kinds of registers and to feedback loops with multiple registers.

5.3: Summary

The ideal retiming verification algorithm would be general enough to work on all retimed circuits, yet would still be reasonably fast. Unfortunately, the findings in this previous sections show that currently no such algorithm exists. After investigating all the alternatives, peripheral retiming appears to be the best compromise between speed and wide applicability.

Having established that peripheral retiming meets Formality's needs best, I move on to explain how Formality's implementation of peripheral retiming was modified during this project.

6. Changes Made to Peripheral Retiming

In section 4 I outlined the various shortcomings of the original implementation of the peripheral retiming algorithm. In this section I explain which of these issues have been resolved and how.

I worked on this project with my coworker, Muzaffer Hiraoglu, who implemented the changes made to clock gating, described in section 6.4, and the removal of multiplexer loops, described in section 6.5. I implemented the rest of the modifications, which includes removing test outputs (section 6.1), changing the definition of register pin equivalence (section 6.2), and inferring port equivalence (section 6.3). I describe the changes that I directly implemented in more detail.

6.1 Removing Test Outputs

It is common practice in digital circuit design to insert test outputs into the design. The purpose of these test outputs is to be able to observe the values of individual registers during testing; they have no effect on the functionality of the circuit and may well be present in one of the versions of the design and absent in the other.

Test outputs often used to cause the retiming transformation to fail. To understand why, consider Figure 10. Parts a) and b) show two versions of the original circuit, one with a test output and one without. Parts c), d) and e) show the result of applying peripheral retiming to each version of the circuit. The reference design did not need any modifications because the register was already at the output. In the implementation design, the extra logic between the register and the test output makes it impossible for the register to reach the test output. Therefore, the register is retimed backward, duplicating the OR gate.

Although the algorithm succeeded in moving all registers to the periphery of the circuit, we still cannot match the registers because reference design now has only one register, whereas the implementation design has three. Furthermore, the OR gate duplication was completely unnecessary, because the test output would be ignored by the verification phase anyway.
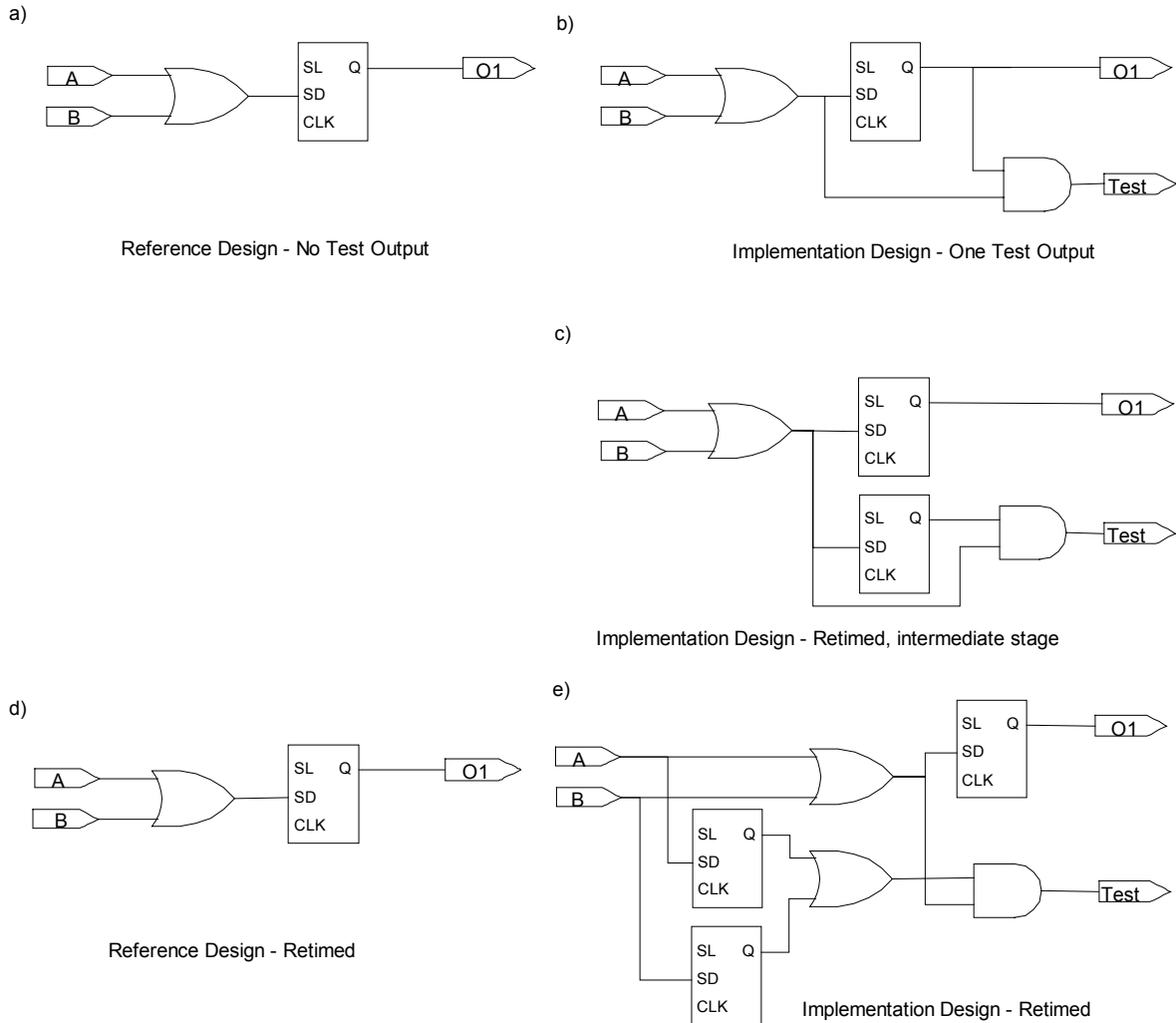
Figure 10: The effect of test outputs on peripheral retiming

To fix this problem, we need to somehow make the peripheral retiming algorithm ignore the test outputs and all the logic driving them (and not driving any other outputs). Unfortunately, there is no way for Formality to automatically identify a test output. One potential solution I considered was to treat all unmatched outputs as test outputs; however, Figure 10 shows that output port matching occurs only after retiming. Therefore, the user needs to provide information about test outputs in the setup. For the example in Figure 10, the user would type

```
set_dont_verify_point i:/TOP/test
```

to specify that the port "test" is a test output. Formality processes these user settings prior to register retiming; therefore, it is possible to obtain a list of points that have been set as don't-verify points. To ensure that register retiming ignores test outputs, I modified the flattening algorithm. Flattening is done by iterating over the outputs of the cell and flattening each of their logic cones. If an output is skipped in this process, that output and any logic driving it

that does not drive other outputs does not appear in the flat version. Therefore, the solution to this problem was to skip the test outputs in the flattening process.

6.2 Changing the definition of equivalence of register pins

As I explained in section 4, the original implementation of peripheral retiming considered two clock or enable signals to be equivalent if and only if they were both driven by the same port or logic component. The algorithm was able to skip over buffers and inverters; however, if two clock signals were functionally equivalent but driven by different gates, they were considered different. Figure 11 illustrates this situation: in all three cases, the registers belong to the same class; however, the old implementation considered the registers in figure 3-c to be in different classes.
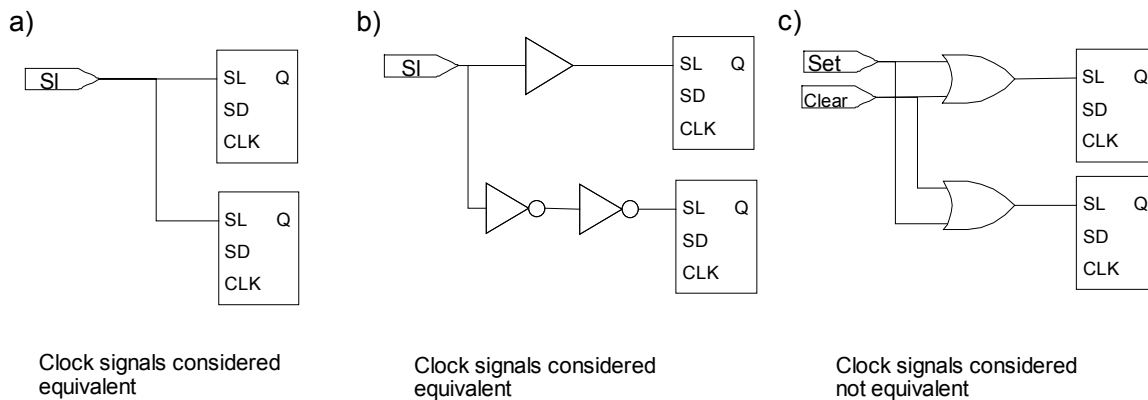

Figure 11: Original definition of clock signal equivalence

To solve this issue, I rewrote the function that identifies signal equivalence by using a BDD (Binary Decision Diagram) package to compute the logical function the signal represents. However, BDDs are expensive to build and computing the BDDs of all clock and enable signals every time we are checking the equivalence of two register classes would be very inefficient. Therefore, I used the algorithm shown in Figure 12 to check the equivalence of two clock or enable pins.

```
checkPinEquivalence(pin P1, pin P2) {
    If P1 and P2 are both constant,
        If they have the same constant value, return true;
        If they have different constant values, return false;
    Else if P1 and P2 have the same driver pin
        after skipping buffers and inverters, return true;
    Else find or build and cache BDDs for P1 and P2;
        If P1 and P2 have the same BDDs, return true;
        Else return false.
}
```
Figure 12: checkPinEquivalence function pseudocode

The BDDs are only computed for non-constant signals that do not have a common driver. A single register may be compared to many others to determine which ones are in the same class. To avoid recomputing BDDs, the function that recursively builds the BDDs

stores the BDD for the output pin of every logic component it encounters. These BDDs are stored only for the duration of peripheral retiming.

If one of the clock or enable signals is driven by a large multiplier, the BDD construction algorithm runs out of memory. In this case, the checkPinEquivalence function returns False. This could potentially produce a false negative verification result if the pins were in fact equivalent. However, it is uncommon for a clock or enable pin to be driven by a large multiplier, so failing only in this isolated case is a big improvement over the original implementation.

6.3 Inferring port equivalence

The next part of this project was identifying clock and enable signal equivalence when each signal is driven by a different port, but these ports are in fact equivalent. There are two reasons why this could happen: either the ports are driven by the same port (or some other component) outside the retimed cell, or they are driven by separate ports but these ports have been set equivalent by the user during setup.

It is very common that a clock port in the reference design corresponds to several clock ports in the implementation design. This happens when clock-tree insertion has been done on the reference design to balance the load on the clock source. Formality allows the user to specify that all the clock ports in the implementation design are equivalent to each other and to the one in the reference design by using the set_equivalence or set_user_match command. The set_equivalence command is usually used to indicate the equivalence of two ports in the same design and Formality processes this information by replacing both ports with a single port. The set_user_match command specifies equivalence across designs and helps Formality's matching algorithms establish a correct matching of the inputs and outputs.

Regardless of which cell is set as retimed, it is usually most convenient for the user to specify equivalence or a user match on the top-level design. For example, in Figure 13, the user would usually set an equivalence or a user match on the top level ports using one of the following commands:

```
set_equivalence i:/TOP/topCLK1 i:/TOP/topCLK2 i:/TOP/topCLK3

set_user_match r:/topCLK i:/topCLK1 i:/topCLK2 i:/topCLK3
```

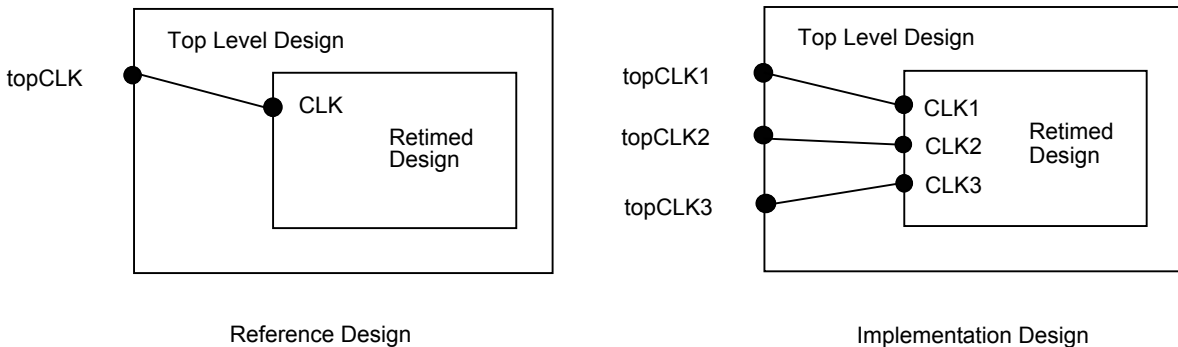rather than specifying the same properties for the inner CLK, CLK1, CLK2 and CLK3 pins.

Figure 13: If a user sets topCLK to be equivalent to topCLK1, topCLK2 and topCLK3, retiming must infer that CLK1, CLK2 and CLK3 are equivalent ports

Regardless of whether the equivalence of the clock pins of a register is determined by comparing their drivers or by comparing their BDDs, it is necessary to take this user-supplied information into account. At the time when register retiming is carried out, though, Formality has not yet propagated these user-set properties to the inner ports.

Another case where it is necessary to infer port equivalence is if several ports on the retimed cell are in fact driven by the same pin outside the retimed design. This is illustrated in Figure 14: the ports CLK1, CLK2 and CLK3 should all be considered equivalent.
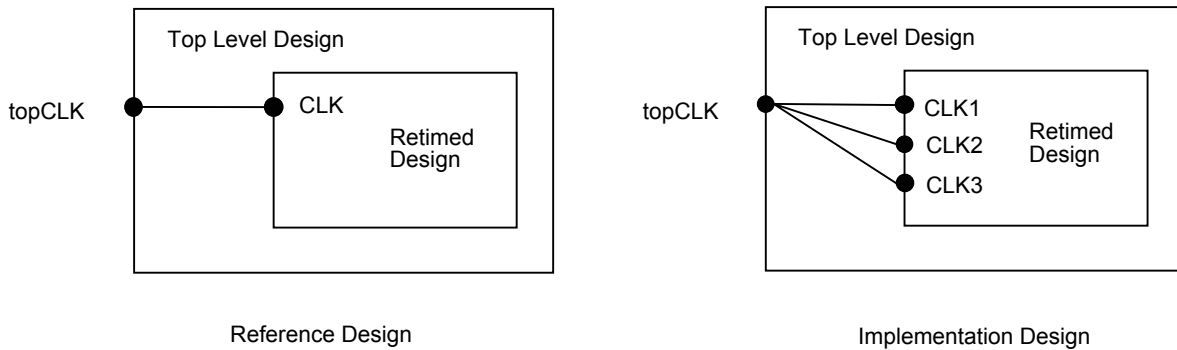

Reference Design   Implementation Design

Figure 14: Retiming must infer that CLK1, CLK2 and CLK3 are equivalent ports

To address these issues, I added a function that is called once on each retimed cell prior to moving any registers. The function finds equivalence classes of ports using a disjoint set data structure with a representative member for each set.

The function first traverses the logic backward from every port of the retimed cell, skipping over buffers and inverters, and for each port finds the pin driving it. If any ports have the same driver pin or port, they are put in the same equivalence class. As a second step, the function iterates through a list of all port equivalences set by the user using either of the commands, and adds these relationships to the disjoint set structure.

The CheckPinEquivalence function presented in part 3.1 queries this data structure every time it encounters a port and replaces the port with its representative member. This way, the driver pin comparison or the BDD comparison only ever encounters one port from each equivalence class. Note that the port is replaced only in data structures private to register retiming; the actual circuit is not modified.

6.4 Clock Gating

A third issue in register class identification was clock gating. Clock gating is an operation where the different register control signals are combined in such a way that the clock signal holds a constant value and only lets a rising edge through when the register is enabled. Figure 15 shows two registers, one with a gated clock and one without. Although the two registers have exactly the same functionality, they do not have equivalent clock signals and would be put into different classes by the original algorithm.

The solution to this problem was to identify registers with gated clock signals and re-wire the control logic to the register so as to remove the clock gating. Once the clock gating is

removed, a simple pair-wise comparison of the different control signals suffices to determine register class equivalence.
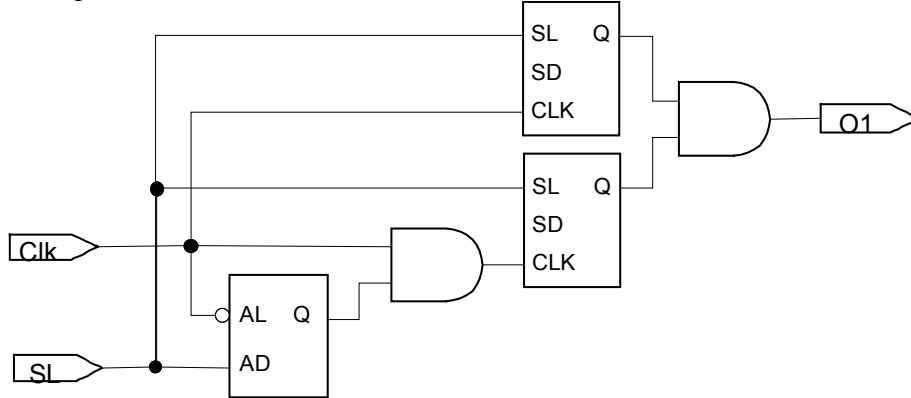

Figure 15: A circuit with two registers, one with clock gating and the other without

6.5 Mux Loops

A special case of the loop problem that we were able to satisfactorily solve is mux loops. In some cases, a register with an enable signal in one design is represented in the other design as a D-flip flop with a feedback multiplexer (see Figure 16). This creates a similar problem as the one created by test outputs: the presence of a loop in one design and its absence in the other causes the two registers to be retimed to different locations and Formality's matching algorithms would have no way of determining that these registers match.

To solve this problem, the retiming algorithm now analyzes the logic surrounding D-flip flops and if it detects a mux loop, it replaces the mux and the D-flip flop with a load-enabled register. This load enabled register is then free to move backward or forward, and thus will reach the same location as the corresponding register(s) in the other design.
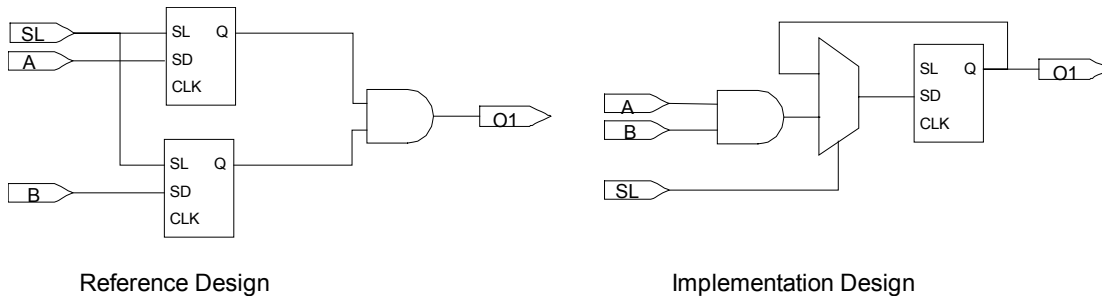

Reference Design   Implementation Design

Figure 16: The reference design has a mux loop, the implementation has a D-flip flop

7. Problems Not Solved

7.1 Logic Duplication

The second step of the peripheral retiming algorithm, which involves retiming all non-fixed registers that hadn't reached the outputs backward as far as possible, causes several problems:

20

- The duplication of the logic cones preventing registers from being pulled back causes the circuit to grow disproportionately in size. This slows down the verification, increases memory consumption and makes the circuit difficult to debug if verification fails.
- The backward retiming step causes problems with reset state.

To solve these issues, we considered modifying the retiming algorithm by eliminating the backward retiming stage. The modified peripheral retiming algorithm would go as follows:

1. Mark all registers inside sequential loops as fixed
2. Retime all non-fixed registers as far forward as possible

In this modified algorithm, most of the registers do not reach the periphery of the circuit, but rather stop in front of some gate with unequal numbers of registers on its inputs. The fact that not all registers reach the periphery of the circuit would not be a problem if all registers stopped in equivalent locations in the two circuits, so that the registers could be matched. However, in practice, this property can only be guaranteed in isomorphic circuits.

It is easy to see why the modified algorithm works for isomorphic circuits. Suppose that we have two equivalent, isomorphic designs. In both of the designs, each register keeps moving forward until it reaches some gate with unequal numbers of registers on its inputs that causes it to stop. Suppose that in one of the designs, a particular register r stops at gate g. Let g' be the gate in the other design that is analogous to g. Because the circuits are equivalent, there must exist a register r' in the other design which matches r. We know that r' must be located before g' because isomorphic circuits are just retimed versions of each other, and if r could not be retimed forward past g, then there is no way r' could have been retimed forward past g'. Therefore, r' must be located somewhere before g', and when retimed forward as far as possible, r' will stop right before g', making it easy to match r with r'.

Therefore, the modified algorithm works for isomorphic circuits. However, it produces false negatives in some cases when the circuits have been resynthesized after retiming. In particular, whenever logic optimization eliminates some of the gates where the registers would have otherwise stopped during the forward retiming step, the algorithm fails. Figure 17 shows an example of this. We can demonstrate using temporal logic that the two circuits in Figure 17 are equivalent:

| Reference design: | $O = (a \cdot b_1 + b_1) \cdot c_1 = b_1 \cdot c_1$ |
| Implementation design: | $O = b_1 \cdot c_1$ |

However, if forward retiming is applied to both circuits, in the reference design the registers stop at the first AND gate and at the OR gate, whereas in the implementation design the registers go all the way to the output. This results in each design having a different number of registers, making it impossible to match the registers. Once again, this happened because the logic optimization step removed gates in the implementation design that would have otherwise stopped the registers.
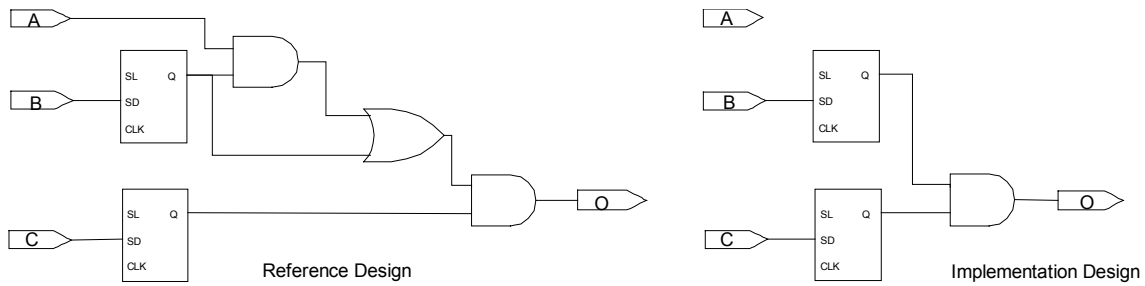
Figure 17: A case where forward retiming alone is insufficient.

Although the approach of eliminating the backward retiming step looked promising at first, we were unable to find a way to handle the problematic cases. Therefore, the implemented algorithm still uses backward retiming, with all the associated problems.

7.2 Loops

Registers inside loops present another difficult problem for peripheral retiming. The original implementation did not move these registers at all. This approach worked whenever there was no retiming done inside the loop and so the registers were in equivalent positions to begin with; however, we found that very frequently the registers in the loops had been retimed.

It is not possible to move a register out of a loop; however, it is possible to move it within the loop. Therefore, one approach would be to try to find equivalent locations *inside* the loops. If combinational optimizations in the circuit were done only inside or outside the loop and didn't cross the boundary of the loop, then it is be possible to find a location within the loop where the registers would match. One way to do this is to apply forward retiming to all registers, including those inside loops. The positions where the loop registers eventually stop are equivalent. As is shown in Figure 18, retiming a register forward across a loop boundary produces two registers, one that can freely move forward towards the output of the circuit, and another that comes back to the beginning of the loop. If there were more registers "stuck" at the beginning of the loop, we could combine each one in turn with the register in the loop and retime them forward one by one until they all reach the other side of the loop (except for the one register that was originally inside the loop).
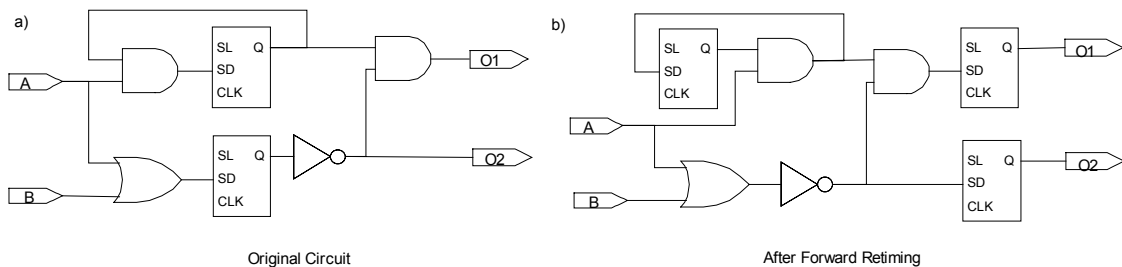

Figure 18: Forward retiming of registers in loops

This approach works in many cases; however, for some cases it actually causes a false negative. Figure 19 shows one such case. Using the original algorithm, the two registers on

the B input would be retimed backward and thus combined into one and matched with the register on the B input of the implementation design. Both registers in the loops are in equivalent locations. Therefore, this verification would succeed using the original algorithm.

However, the enhancement of moving registers through loops would cause this case to fail. In the reference design, none of the registers can be retimed forward. On the other hand, in the implementation design the register on input B can now be combined with the register in the loop and retimed to the output. As a result, even if Formality managed to match the new register on the output with the two registers on input B in the reference design, the registers are not in equivalent locations and the verification would fail.
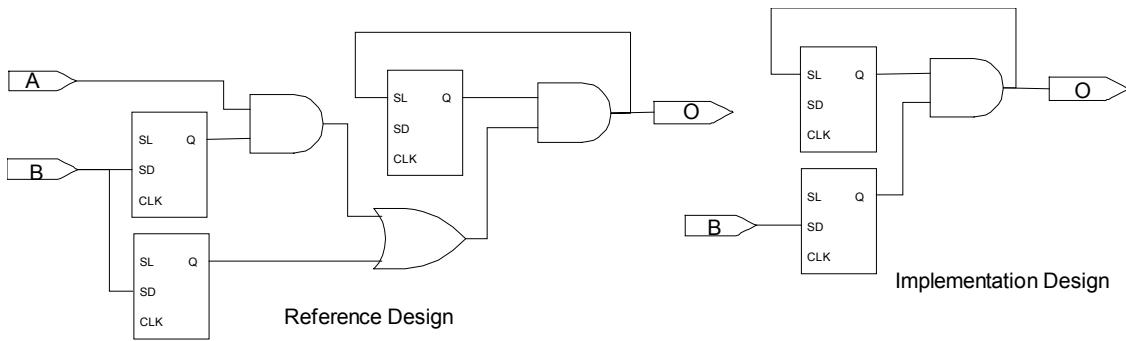

Figure 19: A case for which forward retiming through loops does not work

Therefore, the current implementation of the algorithm still keeps registers inside loops fixed.

8. Summary: Improvements Achieved

The aim of the project was primarily to increase the scope of cases that the retiming algorithm in Formality handles. This raises some difficulty in quantifying improvements. Because there were no significant improvements in speed or memory consumption as a result of this project, the best way to quantify the improvements is to summarize which of the issues presented have been solved and which have not. That summary is presented in Table 1.

Table 1: Summary of register retiming accomplishments

Problem	Solved?	Solution (Implemented or proposed)
Large logic cone to verify	No	Could be alleviated with cutpoints
Logic duplication causes large memory usage	No	Could be eliminated by removing backward retiming
Duplication makes debugging failing verifications difficult	No	Could be eliminated by removing backward retiming
Port equivalence is not inferred from common	Yes	Equivalence classes of ports,

23

driver or from equivalence set on top level		only use one port from each class
Loops	No	Most cases could be solved by retiming registers inside loops
Loops caused by Mux decomposition	Yes	Mux removal code
Reset state issues	No	Could be eliminated by removing backward retiming
Register control signals equivalent only if they are driven by the same pin	Yes	Use BDD comparison instead of comparing drivers
Clock Gating	Yes	Remove clock gating
Test outputs	Yes	User must identify these test outputs

To give an idea of the practical impact of these improvements, I summarize here the impact of the project on a sample set of 19 test cases that my coworker, Muzaffer Hiraoglu, investigated at the beginning of the project. Each of these test cases included some retimed components and was originally failing. Upon investigation of the test cases, my coworker found that 10 of the test cases were failing due to other reasons (mostly set-up issues). Out of the remaining 9 test cases, two thirds were solved by this project.

The number of test cases investigated is too low for this fraction to be statistically significant. However, the success of the project is also supported by anecdotal evidence from the marketing department.

9. Conclusion

This thesis project improved Formality's performance on retimed circuits by solving many of the issues faced by the original implementation. Furthermore, as part of this project I examined alternative retiming verification algorithms and determined that no other algorithm is better suited for use in Formality. Finally, in this thesis I have outlined partial solutions for the problems that have not yet been satisfactorily solved.

Most of the academic papers on retiming verification restrict their scope to the special case of isomorphic circuits. However, isomorphic circuits are of little practical interest and there have been very few attempts at solving this problem for the general case of retimed and resynthesized circuits. A lot of work remains to be done to find a practical solution to the retiming verification problem.

References

[1] T. Kropf, "Introduction to Formal Hardware Verification", Springer, 1999, Berlin.

[2] Design Compiler Reference Manual: "Introduction to Register Retiming", Dec. 2003, Synopsys.

[3] C. Leiserson and J. Saxe, "Retiming Synchronous Circuitry", Algorithmica, 6:5-35, 1991.

[4] B. Lockyear, "Peripheral Retiming Applied to Circuit Verification," technical memo, Synopsys Advanced Technology Group

[5] P. Kostelijk, A. van der Werf, "Functional verification for retiming and rebuffering optimization," *edac_euroasic,* 1993, IEEE Press, 1993, pp.99-104.

[6] S.-Y. Huang, K.-T. Cheng, and K.-C. Chen, "On verifying the correctness of retimed circuits." In *Proc. Of the Great Lakes Symposium on VLSI,* Ames, Iowa, 1996, pp. 277-281.

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