Design Replacements for Sequential Circuits

by

Vigyan Singhal

B.Tech. (Indian Institute of Technology, Kanpur) 1989
M.S. (University of California at Berkeley) 1994

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Robert K. Brayton, Chair
Professor Alberto L. Sangiovanni-Vincentelli
Professor Dorit L. Hochbaum

1996

The dissertation of Vigyan Singhal is approved:

Chair Date

Date

Date

University of California at Berkeley

1996

Design Replacements for Sequential Circuits

by

Vigyan Singhal

Abstract

Design Replacements for Sequential Circuits

by

Vigyan Singhal
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Robert K. Brayton, Chair

In this dissertation we study the problem of design replacements for synchronous sequential circuits. There have been previous efforts to characterize the criterion for replacement for such circuits. However, all previous attempts either make implicit or explicit assumptions about the design or the environment of the design (for example, all latches have a hardware reset line and the design operation starts in a unique designated initial state). We present a notion for replacement which works for any arbitrary environment, and without making these assumptions. We also observe that if the design's output is not used for a certain number of cycles after power-up then allows additional flexibility is available for replacement. We have used these notions of design replacement for logic optimization and we report encouraging results. We also prove the complexity of the verification problem for these notions as well as heuristics for this this problem.

We study the effect of retiming transformations on circuit behavior in detail. We show how such transformations produce acceptable replacements, both with respect to the actual circuit behavior as well as the behavior of the circuit as seen through conservative logic simulators. We also show that the problem of retiming the initial state of circuits can always be solved simply by modeling reset circuitry explicitly.

Professor Robert K. Brayton
Dissertation Committee Chair

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my research advisor, Professor Bob Brayton, for his unwavering support of my research and for suggesting many ideas. I have always been amazed by how quickly he is able to tune in to and give directions on new or underdeveloped ideas. The research direction for this dissertation was mostly due to Dr. Carl Pixley. He has been a source of constant encouragement and many wonderful insights in this research. Most of the ideas in this thesis have been developed jointly with Carl. About half-way into graduate school, my good friend Adnan Aziz (now Prof. Aziz) did me a great favor by persuading me to change my specialization from database systems to computer-aided design. We have worked on many problems together, and his vigor and brilliance never ceases to impress me. I would also like to thank Professors Alberto Sangiovanni, Alan Smith and Gary Hachtel for their advice and encouragements during my stay here. I am also grateful to Professors Sangiovanni and Dorit Hochbaum for reading my thesis and approving my research.

I have been very lucky to spend the last few years in the CAD-group at Berkeley, probably the most productive research environment anywhere. Different portions of this thesis are joint research with Adnan Aziz, Bob Brayton, Carl Pixley, Rick Rudell, Shaz Qadeer and Sriram Krishnan. I have also had extremely fruitful research collaborations with many other CAD-group members– Alex Saldanha, Ellen Sentovich, Felice Balarin, Gitanjali Swamy, Luciano Lavagno, Ramin Hojati, Tom Shiple and Yoshi Watanabe. I am fortunate to have been in a position to research with so many brilliant people.

I am grateful to Flora Oviedo and Kathryn Crabtree for helping me with all the admistrative problems I faced.

My stay in the Bay Area has been enriched by numerous friends– Adnan, Alex and Avril, Amit Gupta, Amit Narayan, Anurag and Suneeta, Barathi and Rajeev, Bharat and Manveen, Des, Edo and Tokiko, Ellen, Flora, Gambhir, Gitanjali and Sanjay, Inder, Jacqueline, Jagesh and Alpa, Kumud, KJ, linuS, Luca, Luciano and Paola, Nari, Paolo and Ana, Paolo Giusto, Rafael, Raghuram, Rajat and Vandana, Rajeev and Renu, Rajeev Murgai, Ricki Blau, Sandy and Eric, Satyajit and Bratati, Shaz, Shing Kong, Sriram Krishnan, S. Sriram, Tami, Tanvi, Tom and Suzanne, Vineet and Mudita, Zach and Sarah. I am sure I forgot many other names in my hurry to file this dissertation. Thank you for making days here so enjoyable.

I would like to thank my parents and Sangeeta and Ramesh for always believing in me and supporting me throughout my education. I have very much enjoyed the love and blessings of my other family– Barbara, Woody, Rani and Essi, Teri and Larry, Robin and Laz, Tami, Jessica and Mike, Josh, Jasmine, Sally, Kaye and the rest of the Busse family.

Lastly, I am forever grateful to my sweetheart Sonali for her wonderful love and patience during the last few months I tried to finish this thesis.

The work in this thesis has been supported by NSF/DARPA Grant MIP-8719546, SRC Contracts and Grants 94-DC-324, 95-DC-324, 96-DC-324, a summer intership from Motorola and various other grants from BNR, California Micro Program, DEC, Intel, AT&T and Motorola.

Chapter 1

Introduction

Over the last few years we have seen a tremendous growth in the complexity of digital systems being designed. The top of the line microprocessor designs consist of millions of transistors and hundreds of thousands of simple gates. To make the design process more manageable, the role of design automation methods is becoming increasingly more important. In this dissertation we will concentrate on two of these methods– logic synthesis and verification.

Logic synthesis is the process of transforming the logic implementation of a circuit so that the implementation achieves the desired behavior of the circuit. There are many components of logic synthesis– transforming a register-transfer level (RTL) design into a gate-level design, restructuring a gate-level design to achieve an optimum combination of performance (timing), area, low-power and testability, mapping a gate-level design to a library of technology-dependent components, restructuring a transistor-level design after layout and placement for more optimal use of area, etc.

Verification is the process of verifying that a given design does "what it is supposed to do". Verification too appears at many stages in the design flow– verifying that the high-level architecture specification achieves the desired properties, verifying equivalence between a pair of designs which are at the same or adjacent levels of abstraction (e.g. gate-level vs. gate-level or RTL-level vs. transistor-level), etc. It should be the case that if one design is obtained from another via logic synthesis, then, by the definition of logic synthesis, the verification between this pair of designs should be unnecessary. However, in practice, this is rarely the case. This is because of many reasons– commercial and academic logic synthesis tools have bugs, designers who use these tools cannot afford or are unwilling to trust the correctness of the tools, the tools may assume a different criterion for equivalence than what the designer has in mind, etc. Many times small local design changes are made by hand, and it difficult to keep track of these changes and thus the problem of design verification between pairs of designs manifests itself very often.

It should be clear that a notion of design equivalence is crucial for logic synthesis as well as for verification. For synthesis we would like to be able to take an arbitrary piece of logic from a design and replace it with another, optimized, piece of logic. For verification, we want to find if there is any environment which can distinguish two given designs. The focus of this dissertation will be on notions of design equivalence as well as how to use these notions for the problems of synthesis and verification. We will restrict ourselves to synchronous sequential designs which can be modeled with a single clock. The sequential nature of these designs is characterized by a state space which the design traverses. At any clock cycle, for a given input and a given state of the design, the design produces an output and moves to a new state. In a gate-level implementation of such a design, the states are implemented by memory elements (like latches or flip-flops) which are driven by the same global clock; the other gates in the design implement the combinational part which determines the output and the next state for a given present state and an input.

Our primary focus will be to address the problem of sequential design replacement[1] Even though there has been much research in the areas of sequential synthesis and verification over the last few years, it is surprising to see little transfer of the technology to industry. For example, when a gate-level netlist is passed on to a synthesis tool for optimization, such a netlist is almost always[2] combinational, i.e. without any memory elements; the design is often cut at latch boundaries before passing to a synthesis tool [9]. For example, one problem with some sequential synthesis methods in the literature is the assumption of a designated initial state for a design; often it is not possible to live with such as assumption for the optimization of an arbitrary sequential netlist in an industrial setting. We will discuss this further in Chapter 2.

For arbitrary sequential netlists, one notion that can used for replacement is the

[1] Actually there is much debate over the notion of design replacement for combinational design (see, for example, [13, 52, 47, 77]). However, that controversy has its roots in designs which have cycles in the combinational parts of the circuit. Almost all real synchronous designs require at least one memory element in every cyclic path, and so we will restrict ourselves to this class of circuits. We will discuss this issue further in Section 2.3.

[2] Retiming is probably the only exception to this. There are few other automatic sequential synthesis techniques used by designers, or available in commercial CAD tools.

concept of FSM equivalence. However, this is often too strict, and it is often possible to use weaker notions to get a safe replacement. We will discuss the other alternative notions, proposed by others, in Chapter 3. We will discuss our objections to all these notions. Then we present our notion of *safe replacement* in Chapter 4. We will argue that if we wish to make a replacement in an arbitrary piece of logic, this is the exact notion which we want to use, if we know nothing about the environment. Then, in Chapter 5, we will present techniques to exploit this flexibility for synthesis. We will also discuss the verification problem for safe replaceability by analyzing the complexity of this problem and giving a heuristic solution.

Then, in Chapter 6 we will see that we can extract additional flexibility for resynthesis if we know additional information about the environment of the design, for example, if we know that the behavior of a design during the first few clock cycles after the design is powered up is not important. This will lead us to the more flexible notion of replacement, *delay replacement*. We will investigate logic synthesis techniques so that we can optimize gate-level circuits using this notion.

Retiming memory elements across combinational gates is often used as a logic synthesis step for optimizing either the area or the performance of circuits. However, retiming may cause design changes which can be detected by some environments, and in this sense, retiming may lead to design replacements which are not safe. However, these replacements are delay replacements. Conservative three-valued simulators are often used to analyze the behavior of designs. If such simulators are used, it can be shown that these simulators are conservative enough so that the original design cannot be distinguished from the retimed design as seen from the output of the simulator. We discuss all these issues in Chapter 7. The output of a simulator that the designers use to analyze a design is important. Simulators usually have to be reasonably fast to be of practical use. Often this speed is achieved at the cost of accuracy. Thus, a simulator may classify two different designs as exhibiting equivalent input-output behavior even though real implementations of the two designs show different behavior (which a more accurate, but slower, simulator may detect). However, since the simulator is used to craft the design, the design is maintained such that the behavior of the design as seen through the simulator is consistent with the desired behavior of the design. If it is not so, it is often the design that has to change, even if the inconsistency was due to the conservativeness of the simulator. For example, if an input sequence initializes a design but this cannot be certified by a conservative three-valued simulator, the designer may change the input sequence so that the initialization is certified by the conservative simulator even though the original sequence also was correct.

For designs where some latches have initial state values, when these latches are retimed, these initial state values have to be retimed also. We review the existing techniques to achieve this retiming of the initial state in Chapter 8. We also show that an alternative solution falls out of the way we model such latches in our designs, as described in Chapter 2.

Finally, in the last chapter we discuss where our work extends for the future and the lessons we learned in this research.

Chapter 2

Preliminaries

In this chapter we are first going to discuss some design modelling issues which are needed for the presentation that follows in this thesis. We will also describe the terminology we will be using in the thesis.

2.1 Memory elements

For the most part in this thesis, we are dealing with synchronous, sequential gate-level circuits. The sequential nature of such circuits is implemented by memory elements, either latches or flip-flops. We will assume that the memory elements are edge-triggered (for most of the work we present, it is possible but sometimes tedious to extend the work to level-sensitive latches). We will simply use the term "latch" to denote such an element.

Latches come in two flavors– latches with a hardware reset line (we will call them "reset latches")[1] and latches without a hardware reset line ("no-reset latches"). It is easier to reason the behavior of designs where all latches are reset latches. However, there are many reasons why many industrial designs have many no-reset latches. No-reset latches occupy less area and contribute less delay to the circuit. Also, having no-reset latches allows much saving in area by avoiding the routing of the global reset line all over the design. In this thesis, we will assume that all latches in the designs are no-reset latches (later in this section we will show how reset latches can easily be modeled with no-reset latches and some additional combinational gates). When a design consisting of t no-reset

[1] We will use the term "reset latches" to denote both kinds of latches– ones which get assigned to 0 when the global reset line is pulled and the ones which get assigned to value 1, even though it might be more appropriate to call the latter "set latches".

latches is switched on, it randomly starts up in one of the 2^t **power-up states** (a state corresponds to an assignment of binary values to each of the latches).

Many sequential synthesis and verification studies (such as [21, 80, 17, 48, 5]) rely on the supposition that all latches are reset latches by assuming that there is a designated initial state for each design; such an initial state corresponds to a designated assignment of binary values to each of the latches and it is assumed that when the global reset line is pulled in the first clock cycle of operation, all latches assume their respective values. While this assumption is applicable to some designs, it certainly does not apply to all designs. Assuming a designated initial state simplifies the analysis of many problems. This thesis does not assume a designated initial state, and in this it differs from many previous studies.

We will now argue why it is so important to analyze designs without assuming a designated initial state. In our experience, many industrial design do have no-reset latches. While most of the these latches lie in the so-called "data" part of the circuit, one can certainly find many such latches in the "control" part also. Also, often it is not possible to cleanly partition the data and the control part; in many cases it is not even known what the data part is and what the control part is, especially for gate-level designs. Another reason for not assuming a designated initial state, even if all latches are reset latches, is that we cannot assume that whenever the circuit operates, the global reset line has been pulled. Indeed, we have observed real designs where it was not the case that the global reset was activated from the first clock cycle; in some modes of operation, the design was being used even before activating the reset line. In such situations it would have been very dangerous for a synthesis tool to make the assumption that the design was used only after pulling the reset line, and the behavior of the designs in the states unreachable from the designated initial state does not matter. Another problem with the designated initial state model is caused by circuits which have more than one class of latches, each class is wired to a different reset line. It is not clear what the designated initial state is in this situation. Also, we have observed some designs where some reset lines are outputs of combinational logic and it is not clear if there is a meaningful initial state at all.

The bottom line is that (1) there is a large number of designs where some or many latches do not have hardware reset lines, and (2) even for designs where all latches are reset latches it may be incorrect to assume an initial state. In this thesis we aim to address the problems of design equivalence, logic synthesis and verification of circuits without assuming the existence of a designated initial state. In the rest of this thesis,

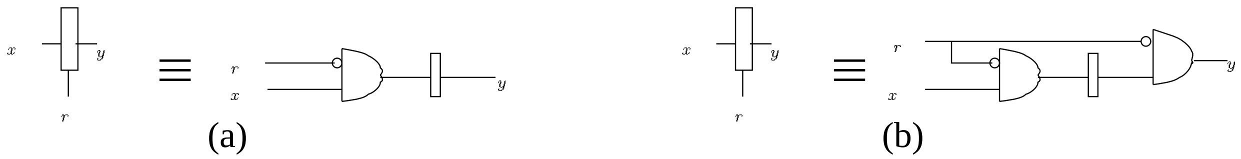

Figure 2.1: Replacing reset-latches with no-reset-latches (a) synchronous reset latch, (b) asynchronous reset latch

unless explicitly mentioned, we will simply use the term "latch" to denote a no-reset latch. Next we show how we can model reset latches with no-reset latches plus some combinational logic.

2.1.1 Modeling Reset Latches

There are two kinds of reset latches: synchronous, where the latch output is reset in the clock cycle after the reset line is activated; asynchronous, where the latch output is immediately reset. We will consider the case when pulling the reset line high (i.e., 1) sets the output of the latch to be 0. The other three combinations of setting/resetting the latch by pulling the reset line high/low can be modeled similarly. Figure 2.1 illustrates the transformation for the case of both synchronous and asynchronous reset latches. It is easily seen that for the synchronous case, pulling the reset line forces the output to be 0 in the next clock cycle; for the asynchronous case, pulling the reset line forces the output to be 0 in the current clock cycle. In the remainder of this thesis we will call the above transformation the **reset transformation**.

Once all reset latches have been modeled using the above transformation, we are ready for address the problem of design equivalence, logic synthesis and verification with the assumption that all latches are no-reset latches. For logic synthesis we just have to be careful that usually we may not want to separate this additional combinational logic we have added in the above transformation from the latch itself. This will enable us to replace the no-reset latch back with a reset latch when we are done with our logic synthesis. On the other hand, in many situations it may even be desirable to separate this extra transformation logic away from the latch, an example of which we shall see in Chapter 8. Thus, modeling reset latches with no-reset latches only gives us extra flexibility for synthesis and does not take away anything from the scope of the designs we can handle.

2.2 Levels of abstraction

The theory of sequential replacement that we present in this thesis is applicable at many levels of abstraction. However, we will restrict ourselves to two different levels of abstraction for representing digital designs: behavioral-level (represented by state transition graphs, or STGs) and gate-level (represented by interconnection of gates and latches, or netlists).

A **state transition graph** (STG) is a directed graph where the vertices represent the states and the edges are transitions between states. The edges are labeled with the input value which excites that transition and the output value that is produced; the destination node of the edge represents the next state for that input value. STGs will represent the behavior of digital designs we will analyze. We will assume that inputs and outputs are encoded and of fixed length. Thus, if the design has, say, m input ports and n output ports, each input value is one of 2^m values and each output value is one of 2^n values. Another term used to describe behavioral-level designs is finite-state machine (FSM). Unless specified, we will assume that a design has a deterministic and complete (under all inputs) FSM. This means that for each state, each input value (in the space of 2^m values) is associated with exactly one transition in the STG. There is the well-known notion of state equivalence for states in an FSM:

Definition 2.1 *Two states s and t are* **equivalent** *if and only if starting from these states, for every input sequence, they produce the same output sequence.*

In this thesis we will treat FSMs modulo state equivalence; in other words, we will consider two FSMs where one can be reduced to other by collapsing equivalent states into a single state as the same FSM. The problem of state minimization is orthogonal to our research; for its application in digital circuits, it has been studied in detail in [32, 65, 40]. For completely specified deterministic machines, the minimum state machine is unique and hence can be thought of as the canonical representation of the FSM.

A **netlist** is an interconnection of elementary circuit elements connected by wires. The basic circuit elements are either latches or combinational gates (an m-input n-output gate maps each of 2^m values to one of 2^n values). An example of a netlist is the circuit in Figure 2.2. The basic circuit elements are connected by wires, which are also allowed to fan out to multiple wires at points called **junctions**. A netlist has some primary input

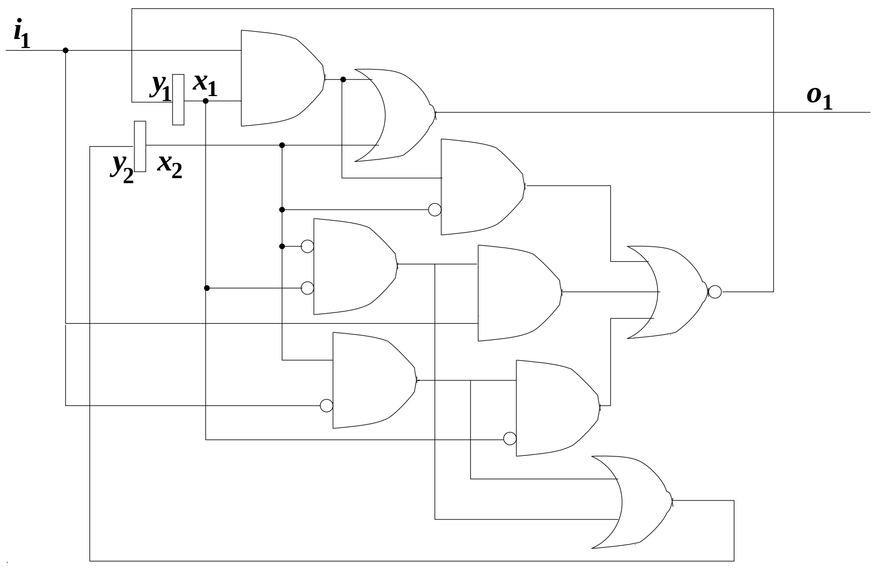

Figure 2.2: Example netlist C

wires and some primary output wires. We require every cycle in a netlist to consist of at least one latch (in other words, the netlist does not have a combinational cycle). Given a netlist consisting of m input wires, n output wires and t latches it corresponds to an STG which has 2^m possible input values, 2^n output values and 2^t states. Because there are no combinational cycles in the netlist, given a state (an assignment of values to the latches) and an input value (an assignment of binary values to the input wires), the output value and the next state are determined. Once we collapse the equivalent states into a single state, we get the associated STG for the netlist. The number of states in the canonical FSM is less than or equal to 2^t. For example, consider the netlist in Figure 2.2. This circuit corresponds to the STG in Figure 2.3(a): state $t2$ corresponds to circuit state 10 (the upper latch is assigned to 1 and the lower latch is assigned to 0), $t3$ corresponds to 00, and $t1$ corresponds to 01 and 11. The original state graph before collapsing equivalent states is in Figure 2.3(b): $s0$ corresponds to 01, $s1$ corresponds to 11, $s2$ corresponds to 10, and $s3$ corresponds to 00.

Given an STG, we have infinitely (and countably) many netlists which correspond to this STG. The problem of obtaining an "optimal" netlist, with respect to some cost function, is the state-encoding and FSM implementation problem. This is usually a provably intractable problem and many heuristics have been suggested to attack many variants of

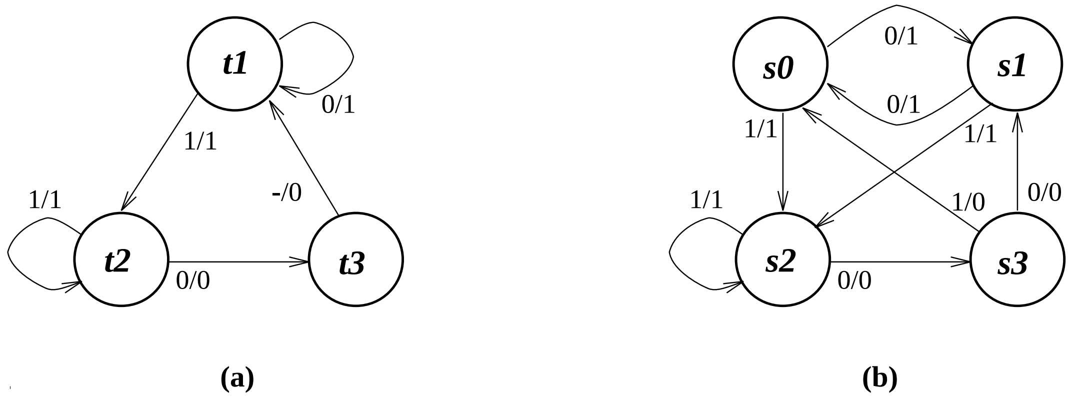

Figure 2.3: (a) The canonical STG for netlist C. (b) the original state graph before minimization.

this problem [25, 26, 74, 67]. This problem is orthogonal to the issues of this thesis.

2.3 Composition of designs

Because we are interested in the problem of replacing designs without affecting the interaction with the environment designs, the issue of composing designs is crucial to this thesis. Therefore, it is important to define what we mean by "design composition" and the subtleties involved in the composition of designs at both the levels of abstraction we are interested in.

Composition of two design entails hiding of some signals– each signal that is hidden is an output of one design and an input of the other. The remaining input or output signals, respectively, become the input or output signals of the composed design. For example, consider Netlist 2 in Figure 2.4. Design D has inputs x and y, and output z; design D' has input z' and outputs w and y'. The composition involves connecting z with z' and y with y'; thus, x is the only input of the composed design and w is the only output.

Composition of netlists

Composition of two netlists simply comprises of placing the two netlists next to each other and connecting the pairs of input-output signals which are required by the composition.

However, such a composition may sometimes create combinational cycles (cyclical paths in the netlist which do not contain any latches in the path). It should be clear that if a netlist has a combinational cycle, it is not easy to associate an STG for such a netlist because the next state and output functions may not be determined from such a circuit. A real implementation (i.e., the hardware realization in silicon) of such a circuit may or many not oscillate for an indeterminate time. Issues regarding behavior of circuits with combinational loops have been dealt elsewhere [85, 77]. As stated in the previous section, in this thesis we assume that we never encounter a combinational loop. This means that we talk about composition of designs only when the composition does not create combinational loops. Few real gate-level designs have combinational cycles. Most of the exceptions have cycles such that the entire cycle can never be excited, i.e. these cycles are false paths and such circuits can effectively be implemented without cycles also. Our work can be extended to this class of circuits also but we avoid this for ease of presentation and for an easier reading.

Composition of STGs

Next we will give an **STG composition procedure** to define the semantics of the composition of two STGs. Then we prove that for two design D and D', if there exist implementations of D and D' such that composing these two implementations does not create a combinational loop, then the STG composition procedure returns a deterministic and complete composed STG. Also, no matter what pair of gate-level implementations we choose, as long as the composed netlist does not have a combinational cycle, the composed netlist has the same behavior as the composed STG defined by the composition procedure.

The composition of two STGs is defined as follows. Suppose, we have two STGs that we want to compose: design D (with states $(s_1, s_2, \ldots, s_p)$, input wires $(i_1, \ldots, i_m)$ and output wires $(o_1, \ldots, o_n)$), and design D' (with states $(s'_1, \ldots, s'_{p'})$, input wires $(i'_1, \ldots, i'_{m'})$ and output wires $(o'_1, \ldots, o'_{n'})$). Say we want to compose these two STGs such that, without loss of generality, the first k input wires of D are respectively connected to the first k output wires of D', and the first k' input wires of D' are respectively connected to the first k' output wires of D. Let the composed design be denoted by $D \otimes D'$, which has pp' states, each of the type $(s_j, s'_{j'})$, where $1 \leq j \leq p$ and $1 \leq j' \leq p'$; the input wires are $(i_{k+1}, i_{k+2}, \ldots, i_m, i'_{k'+1}, \ldots, i'_{m'})$ and the output wires are $(o_{k'+1}, o_{k+2}, \ldots, o_n, o'_{k+1}, \ldots, o'_{n'})$, while the other input and output wires of D and D' are now internal wires of $D \otimes D'$. Given state $(s_j, s'_{j'})$ and an input vector $[(i_{k+1}, \ldots, i_m, i'_{k'+1}, \ldots, i'_{m'}) = (a_{k+1}, \ldots, a_m, a'_{k'+1}, \ldots, a'_{m'})]$, the output vector is $[(o_{k'+1},\ldots,o_n,o'_{k+1},\ldots,o'_{n'}) = (b_{k'+1},\ldots,b_n,b'_{k+1},\ldots,b'_{n'})]$ and the next state is $(t_j,t'_{j'})$ if there exists a vector $(c_1,\ldots,c_k,c'_1,\ldots c'_{k'})$ such that in design D state s_j on input $[(i_1,\ldots,i_m) = (c_1,\ldots,c_k,a_{k+1},\ldots,a_m)$ produces output $[(o_1,\ldots,o_n) = (c'_1,\ldots c'_{k'},b_{k'+1},\ldots,b_n)]$ and goes to state t_j, and in design D' state $s'_{j'}$ on input $[(i'_1,\ldots,i'_{m'}) = (c'_1,\ldots c'_{k'},a'_{k'+1},\ldots,a'_{m'})]$ produces output $[(o'_1,\ldots,o'_{n'}) = (c_1,\ldots,c_k,b'_{k+1},\ldots,b'_{n'})]$. Next we prove the following claim:

Claim: If there exists a netlist implementation for each STG such that the composed implementation does not have a combinational loop, then (a) this composed STG is well-defined, i.e. from each state and for each input vector there is a unique output vector and a next state, and (b) for each netlist implementation of the STGs D and D' such that the composed netlist does not have a combinational cycle, the composed netlist has the same behavior as the composed STG $D \otimes D'$.

Proof. The proof will rely on the fact that each of the two STGs has a netlist implementation such that the composed netlist does not contain a combinational cycle. Let us assign values to the latches of this composed netlist to reflect the state $(s_j,s'_{j'})$, and assign the inputs to $[(i_{k+1},\ldots,i_m,i'_{k'+1},\ldots,i'_{m'}) = (a_{k+1},\ldots,a_m,a'_{k'+1},\ldots,a'_{m'})]$. Since the netlist has no combinational cycle, the values to all the internal wires, specifically the output wires, the wires which feed the latches and the wires which connect the two component netlists representing D and D', are *uniquely* determined from this assignment; let the values of these three sets of wires respectively be $(p_{k'+1},\ldots,p_n,p'_{k+1},\ldots,p'_{n'})$, $(q_1,\ldots,q_t,q'_1,\ldots,q'_{t'})$ and $(d_1,\ldots,d_k,d'_1,\ldots,d'_{k'})$. Since the two netlist respectively implement the two STGs, we know that the the first STG D from state s_j on input $[(i_1,\ldots,i_m) = (d_1,\ldots,d_k,a_{k+1},\ldots,a_m)$ produces output $[(o_1,\ldots,o_n) = (d'_1,\ldots,d'_{k'},p_{k'+1},\ldots,p_n)]$ and goes to state $[t_j = (q_1,\ldots,q_t)]$; a similar condition holds for design D'. It follows that we have obtanied the desired vector $[(c_1,\ldots,c_k,c'_1,\ldots c'_{k'}) = ((d_1,\ldots,d_k,d'_1,\ldots,d'_{k'})]$ to show that the STG composition from this arbitrary composed state and the arbitrary input vector has a unique next state and a unique output vector, thus proving (a). It should also be clear the next state and the output vector for the composed STG is identical to that for the composed netlist.

To prove (b), it is sufficient to note that any implementation of D and D' will produce the same assignment to the wire connections between D and D' as long as there is no combinational loop in the composed implementation. $\Box$

We show an example [56] where the design composition of STGs where one netlist implementation may cause a combinational cycle even though there exists another netlist

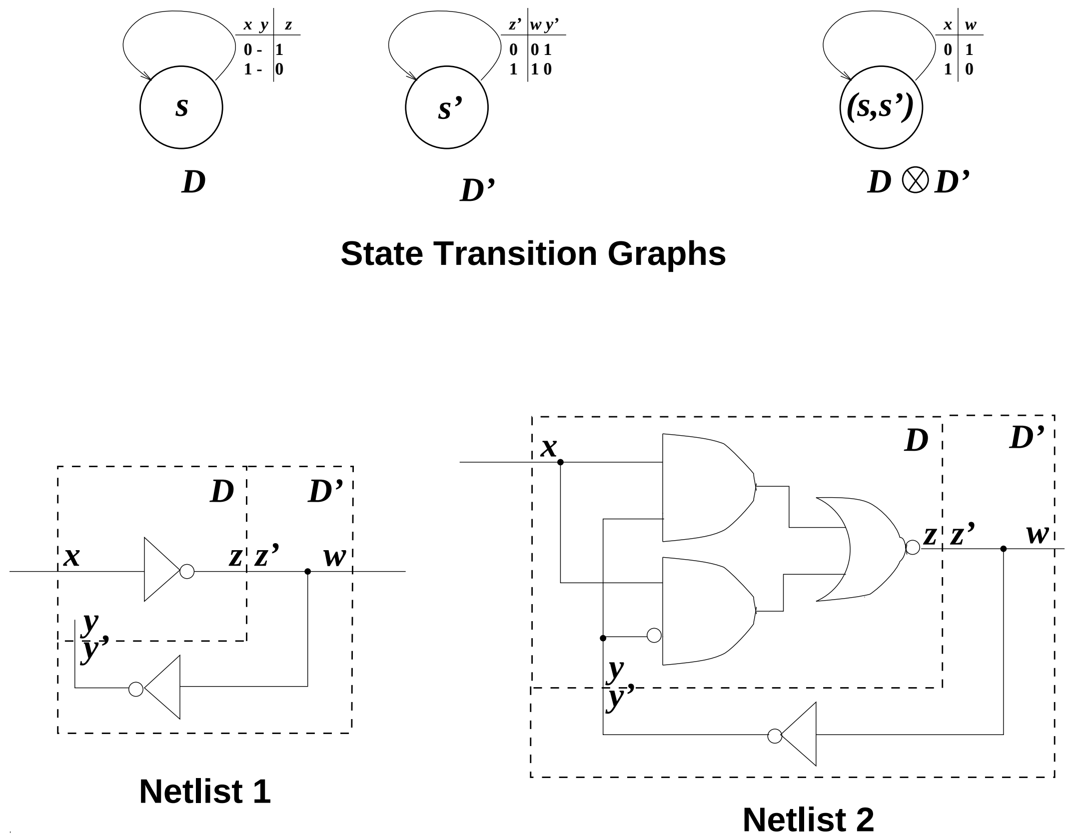

Figure 2.4: Composition of STGs and netlists

implementation where the composition does not yield a cycle. Consider the two STGs D (with inputs x and y and output z) and D' (input z' and outputs w and y') in Figure 2.4. The composition of these STGs is well-defined since there exist netlist implementations of D and D' (Netlist 1 in Figure 2.4) such that the composition does not have a compositional cycle. However, consider Netlist 2 in the figure, which causes a combinational cycle in the composition. We will disallow such compositions in our work since the input-output behavior of such a composed implementation may not be definable. For example, consider the delay assignments on the gates as shown in Figure 2.5. If we simulate this circuit with these delay assignments, assuming a standard event-driven simulator[2], on the input

[2]This is similar to the delay model used in [52] where the output of a t-delay gate changes instantly t time units after its input. It may be argued that a transistor-level implementation of this netlist may not behave exactly this way since gate outputs do not change instantly but change continuously in an analog manner. However, as we argued in Chapter 1, it is extremely important to the design methodology that the input-output behavior of the design is analyzed, *as seen using a simulator*, even though a simulator may be conservative. So, if the output of the simulator, which assumes fixed delays and instantaneous digital

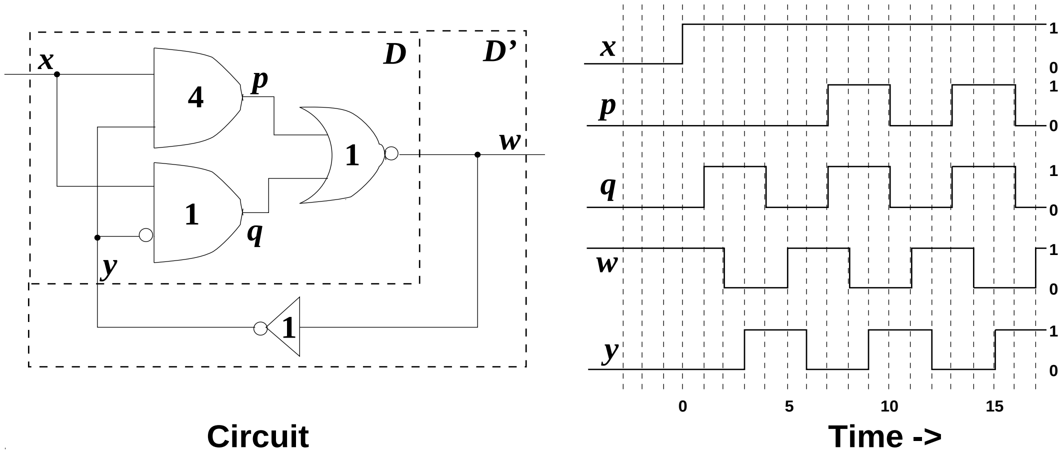

Figure 2.5: Simulation of a non-stabilizing combinational cycle.

sequence shown in the figure, we get the non-stabilizing oscillatory behavior (which never settles) on the output w, as shown in the figure. Thus, the functional behavior of this composed netlist cannot be defined. In the sequel we assume that, whenever we compose netlists and analyze the composed netlist, the composition does not contain a combinational cycle.

We digress to note that for the case of STG composition of two designs, D and D' where every possible gate-level implementation of D and D' creates a combinational loop in the composed netlist, the above-described STG composition procedure may return an STG which is non-deterministic or not complete or both. For example, consider the composition in Figure 2.6. It can be seen that every netlist implementation of D and D' create a combinational loop in the composed netlist. The STG composition procedure return a 1-state STG for $D \otimes D'$, which is non-deterministic (for input $t = 1$, output y can be either 0 or 1) and not complete (for input $t = 0$, there is no value for output z). On the other hand, it may be possible that the composed STG is determistic and complete even though all possible netlist implementations of D and D' create a combinational loop in the composed netlist. Consider the composition in Figure 2.7. However, as argued earlier in this section, in this thesis when we compose STGs we define only those compositions where there exist netlist implementations of D and D' which do not create a combinational loop in the composed netlist.

changes on the wires, shows that the design is unacceptable, it is the design that has to change, rather than the simulator.

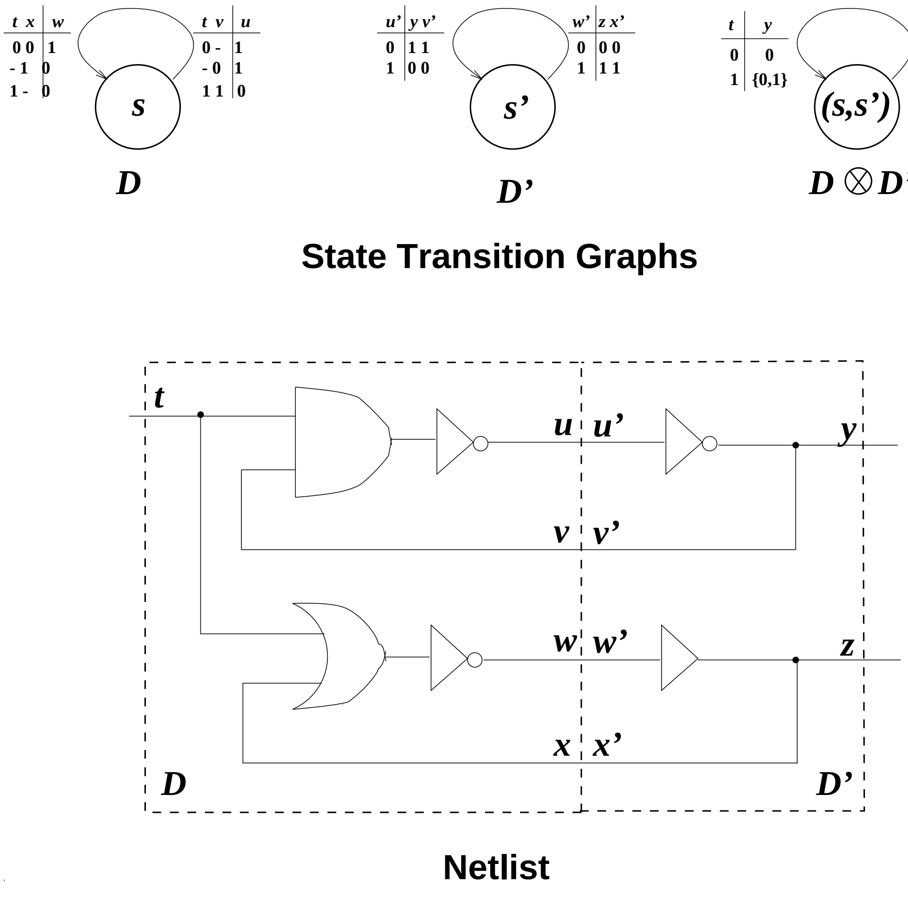

Figure 2.6: Cyclic composition of STGs: first example

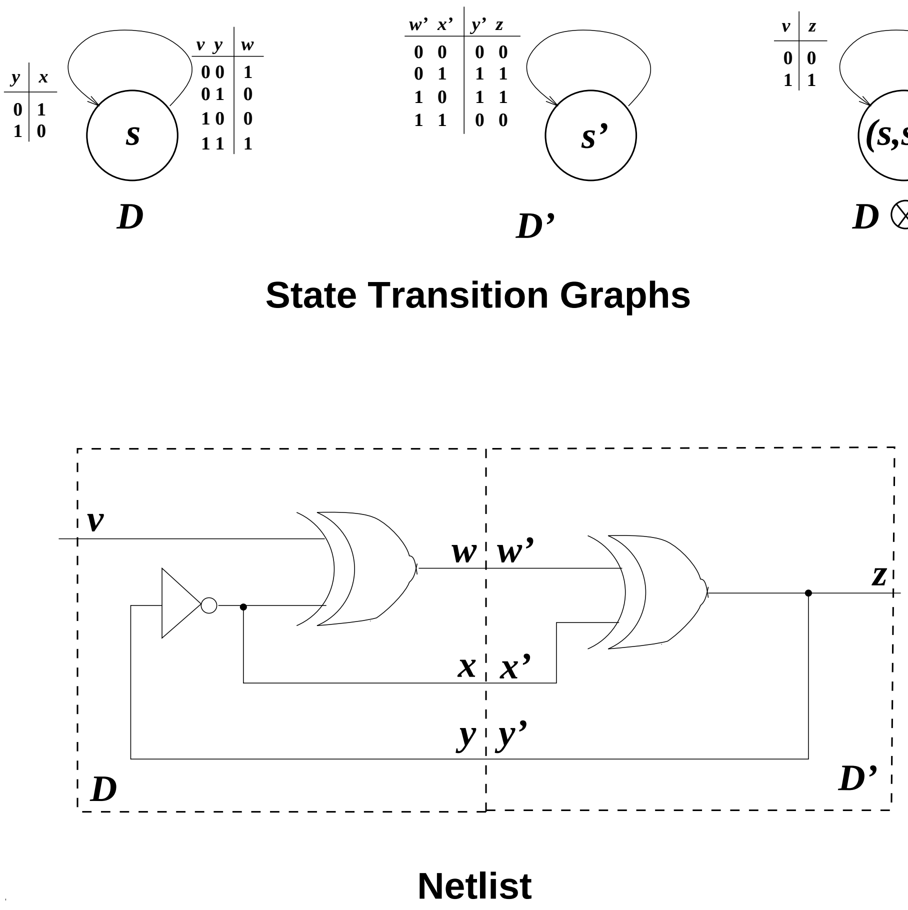

Figure 2.7: Cyclic composition of STGs: second example

2.4 Notation

In this section we describe some notation we will use later.

A **Finite State Machine** (FSM) M represents a quintuple, $(Q, I, O, \lambda, \delta)$, where Q is the set of states, I is the set of input values, O is the set of output values, λ is the output function, and δ is the next state function. The output function λ is a completely-specified function with domain $(Q \times I)$ and range O. The next state function is a completely-specified function with domain $(Q \times I)$ and range Q.

A netlist D consists of a set of interconnected latches and gates. A design with n input wires, m output wires and t latches is naturally associated with an FSM D. We will use D to also denote the set of states of the associated FSM which occupy the state space $\{0,1\}^t$ (it will be clear from the context if D refers to the design or to the set of states). The input space $I_D = \{0,1\}^p$, and output space $O_D = \{0,1\}^m$, the next state function is $\delta_D : D \times I_D \to D$ and output function $\lambda_D : D \times I_D \to O_D$ are defined by the corresponding logic. We are not assuming explicit set or reset pins to any latch; thus when the design powers up it can non-deterministically power up in any one of the 2^t states.

We will also use λ_D and δ_D to denote the output and next state functions on sequences of inputs. So, if $\pi = a_1 \cdot a_2 \cdot a_3 \cdots a_q \in I_D^q$ is a sequence of q inputs, these functions are recursively defined as $\lambda_D(s, \pi) = \lambda_D(s, a_1) \cdot \lambda_D(\delta_D(s, a_1), \pi')$ and $\delta_D(s, \pi) = \delta_D(\delta_D(s, a_1), \pi')$, where $\pi' = a_2 \cdot a_3 \cdots a_p$. Also, for any state $s \in Q_D$: $\delta_D(s, \epsilon) = s$ and $\lambda_D(s, \epsilon) = \epsilon$, where ϵ represents the length-0 sequence.

Two designs are said to be **compatible** with each other if they have the same number of input and output wires. All notions of design replacement described in this thesis are meaningful only for pairs of compatible designs. Henceforth, when talking about two different designs we will implicitly assume compatibility of the two. In this paper we will assume that designs D_0 and D_1 denote the quintuples $(Q_{D_0}, I, O, \lambda_{D_0}, \delta_{D_0})$ and $(Q_{D_1}, I, O, \lambda_{D_1}, \delta_{D_1})$, respectively.

Definition 2.2 *Given two states $s_0 \in Q_{D_0}$ and $s_1 \in Q_{D_1}$, state s_0 is* **equivalent** *to state s_1 ($s_0 \sim s_1$) if for any sequence of inputs $\pi \in I^*$, $\lambda_{D_0}(s_0, \pi) = \lambda_{D_1}(s_1, \pi)$. It can be easily shown that if $s_0 \sim s_1$, then for any input sequence $\pi \in I^*$, $\delta_{D_0}(s_0, \pi) \sim \delta_{D_1}(s_1, \pi)$. We say that π* **distinguishes** *s_0 and s_1 if $\lambda_{D_0}(s_0, \pi) \neq \lambda_{D_1}(s_1, \pi)$.*

Definition 2.3 *A set of states $S \subseteq Q_D$ is a* **strongly connected component** *(SCC) if it* *is a maximal set of states such that for any two states* $s_0, s_1 \in S$*, there exist input sequences* π_0 *and* π_1 *so that* $\delta_D(s_0, \pi_0) = s_1$ *and* $\delta_D(s_1, \pi_1) = s_0$*.*

Definition 2.4 *A set of states* $S \subseteq Q_D$ *is a* **terminal strongly connected component** *(tSCC) if* S *is an SCC and if for any state* $s \in S$ *and any input* a*:* $\delta_D(s, a) \in S$*.*

Chapter 3

Notions of Design Replacement

Before we present our notion of design replacement in the succeeding chapters, in this chapter, we will study other notions of design replacement. A notion of design replacement is a condition which, given an original design and a replacement design, answers the question whether or not the second design is a "valid" replacement of the first. Most such notions are symmetric, reflexive and transitive. Thus the term **design equivalence** is often used to identify such a notion. However, later in this chapter and in the next, the condition for replacing may not satisfy these three attributes and thus may not always yield an equivalence relation; so we do not restrict these notions to be equivalence relations and we will call them notions of design replacement.

We will discuss some techniques for sequential logic optimization in this chapter. Often such techniques are presented without discussing the class of design replacements they cause; when we discuss these sequential optimization methods in this chapter, we will examine the class of design replacements caused by such methods.

In this thesis we will only characterize notions of replacement which compare only the functionality of the two designs and not the timing equivalence of the two designs[1]. So we assume that the clock period of the design is large enough to allow the combinational logic to compute the next-state and output functions within a clock cycle, and that the actual time taken to compute this function is undetectable by the environment of the design.

[1] The problem of characterizing the set of replacements for a combinational designs, where the design specification includes the delay characterization of all gates, so that a replacement design has the same set of timed input-output waveforms as the original design, has been tackled elsewhere [3].

3.1 Combinational Replacement

This notion characterizes combinational designs (i.e. no memory elements) which are valid replacements of existing combinational designs. The notion of combinational replacement used in most logic synthesis tools (such as SIS [72]) is that the Boolean function (from $\{0,1\}^m$ to $\{0,1\}^n$, if there are m input wires and n output wires) associated with the replacement design is the same as the Boolean function associated with the original design.

The only exception to using the above condition for replacement is that the replacement might create a combinational cycle, even through the original design did not have one. For example, refer to design D in Netlists 1 and 2 in Figure 2.4. In both implementations the combinational functionality of the design is identical, namely $z = \overline{x}$. However, Netlist 2 creates a combinational cycle because of the environment D'. In fact, after making a design replacement, most logic synthesis tools [43, 72] either check that the replacement has not caused a combinational cycle or have constraints which prevent a combinational cycle during the synthesis procedure.

3.2 Identical Behavior from Initial State

Most of the previous work on equivalence and replacement of sequential circuits assumes a designated initial state for the circuit. For such a new design, D_1 with designated initial state s_1 is a valid replacement for the original design D_0 (with initial state s_0) if and only if states s_1 and s_0 are equivalent, i.e. for every input sequence π, $\lambda_{D_1}(s_1, \pi) = \lambda_{D_0}(s_0, \pi)$. For such designs, this is the notion of design equivalence used for logic synthesis [71], test pattern generation [31], redundancy removal [19] and sequential verification [21]. Of course, from our perspective, as we have discussed in Chapter 2, such an assumption of a designated initial state is overly restrictive and unreasonable for many real circuits.

3.3 Machine Equivalence

For sequential designs with no designated initial state, there is the classical notion of machine equivalence, for example in [34, page 23]:

Definition 3.1 *Two FSMs M_1 and M_2 are* **equivalent** *($M_1 \equiv M_2$) if for each state s in M_1 there is a state t in M_2 such that $s \sim t$, and for each state t in M_2 there is a state s in*

M_1 such that $s \sim t$.

3.4 Sequential Hardware Equivalence

Because the classical notion of FSM equivalence was thought to be too restrictive in the class of replacement designs it allowed, Pixley introduced the notion of sequential hardware equivalence [59, 60]. When a design with no-reset latches powers up, since the state it powers up in cannot be predicted, the desired input/output behavior is achieved from the design by driving a fixed initializing sequence of input vectors through the design after power-up. The initializing sequence plays a crucial role in Pixley's notion of sequential hardware equivalence.

Definition 3.2 *Given a design D_0, a sequence of inputs $\pi \in I^*$ is called an* **essential reset sequence** *(or an* **initializing sequence***) for the design if for any pair of states $s_0, s_1 \in Q_{D_0}$, $\delta_{D_0}(s_0, \pi) \sim \delta_{D_0}(s_1, \pi)$. A state $s \in Q_{D_0}$ is called an* **essential reset state** *if there exists a state $s_0 \in Q_{D_0}$ and an initializing sequence π such that $\delta(s_0, \pi) \sim s$. A design which has an essential reset state is called* **essentially resetable***.*

Definition 3.3 *Given two designs D_0 and D_1, a state pair $(s_0, s_1) \in Q_{D_0} \times Q_{D_1}$ is* **alignable** *if there is a sequence of inputs $\pi \in I^*$ such that $\delta_{D_0}(\pi, s_0) \sim \delta_{D_1}(\pi, s_1)$. The sequence π is called an* **aligning sequence***.*

The following definition defines the notion of sequential hardware equivalence:

Definition 3.4 *Designs D_0 and D_1 are* **sequentially hardware equivalent** *($D_0 \approx D_1$) if and only if all state pairs are alignable.*

Theorem 3.1 *$D_0 \approx D_1$ if and only if there exists an aligning sequence that aligns any arbitrary state pair in $Q_{D_0} \times Q_{D_1}$.*

From Theorem 3.1, two designs are considered equivalent if and only if there *exists* the same aligning sequence for any arbitrary state pair. This sequence is an initializing sequence for either design. Thus replacing a design by a sequentially hardware equivalent design guarantees that there exists *one* initializing sequence which intializes each of these designs to a state, and these two state are equivalent.

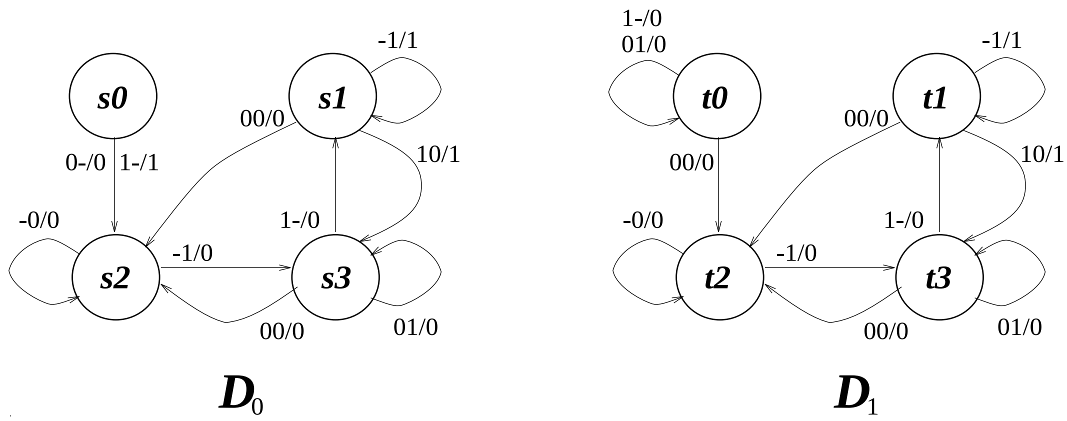

Figure 3.1: Sequential hardware equivalence assumes that environment can produce arbitrary initializing sequence

Now we argue why the notion of sequential hardware equivalence may be undesirable as a notion for arbitrary design replacements.

The condition of sequential hardware equivalence requires the preservation of only one initializing sequence. This may be problematic because it may not be possible for the environment of the design to generate this particular sequence. Consider the designs D_0 and D_1 in Figure 3.1. There is an initializing sequence, viz. the length-1 sequence 00 which initializes the two designs to $s2$ or $t2$, respectively. Thus, the two designs are sequentially hardware equivalent; so by Pixley's criterion it would be fine to replace D_0 by D_1. However, suppose the inputs to D_0 are coming from an environment which can only produce the signals 01, 10 or 11, but can never produce 00. Thus, it will be necessary to use the other initializing sequence, namely the length-3 sequence $11 \cdot 11 \cdot 11$ to initialize D_0 from a random power-up state. However, notice that this sequence cannot initialize D_1, because if D_1 powers up in state $t0$ it cannot leave that state unless it observes the input 00. So, in this sense, it would be wrong to replace D_0 by D_1 if we know nothing about the environment of the design. Replacing D_0 by D_1 may make the larger design (D_1 composed with the environment) non-initializable even though the original design (D_0 composed with the environment) was initializable. So, for a safe replacement we need to preserve <u>all</u> initializing sequences, and not just one. In that case the one used being used by the environment will be preserved, regardless of which that one is.

The notion of sequential hardware equivalence does not place any constraints on the outputs of the designs during the initialization phase. However, we claim that this

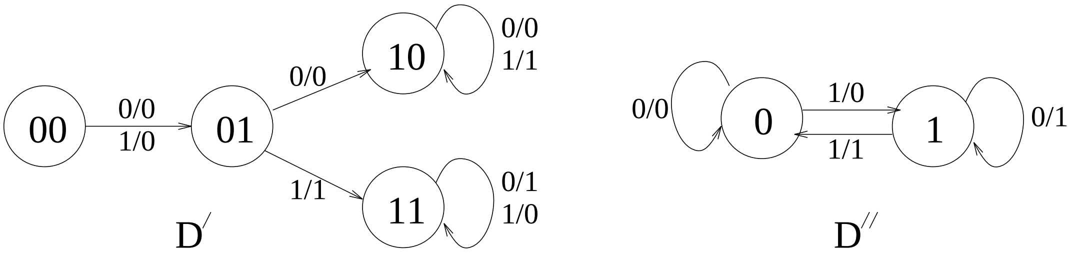

Figure 3.2: Examples of two designs which do not have any synchronizing sequences

condition is too weak for a safe replacement. *A priori*, we cannot assume that the external environment is not sensitive to the outputs during the initialization phase. This is especially important because there may be another interacting design whose initializing sequence may be driven by an output of design D_0. Thus affecting the outputs of D_0 during initialization may destroy that initializing sequence.

Finally, the notion of sequential hardware equivalence does not work for designs which are not essentially resetable. Such a design is not even sequentially hardware equivalent to itself[2] because it does not have an aligning sequence with itself. For example, the design D' in Figure 3.2 is not sequentially hardware to itself because the state pair $(10, 11)$ is not alignable. We think that it is hazardous to assume that such designs do not exist in reality, and to present a theory which fails to replace such designs even by themselves. We can imagine at least two classes of real designs which are not essentially resetable. First, if the environment has some flexibility for the input/output behavior it can accept from the design, the design may have multiple steady-state behaviors (for example, design D' in Figure 3.2). In this example, the environment has a don't care condition such that the design is acceptable to it as long as in its steady-state, the design either 1) always outputs the complement of its input (state 11), or 2) always reproduces the input as the output (state 10). after the synchronization phase. For the second class, consider the design D'' in Figure 3.2. It can be seen that there is no synchronizing sequence for this design, and hence it is not essentially resetable. However, once the design powers up, its state can be determined from its outputs, and based on the outputs the design can be driven to state 0. Thus, the behavior of this design can be controlled. In the next section we will see another example where a sub-circuit of a design may not be resetable but the entire design is still

[2]Thus, sequential hardware equivalence is not an equivalence notion for the class of all designs; only for the class of essentially resetable designs is it an equiavalence relation.

resetable.

3.5 Redundancy Removal for Circuits with no Reset Lines

The physical process of manufacturing a digital gate-level may cause many different kinds of faults in the manufactured circuit. It is important to be able to differentiate such faulty chips from the good ones. Towards this goal, the area of test pattern generation identifies tests for such circuits which, based on the output of the circuits under the test inputs, distinguish good circuits from certain fault ones. Single stuck-at faults comprise one such class of faults for which automatic test pattern generation tools are used. For a given single stuck-at fault (some single wire at the gate-level design is assumed to be stuck at a constant 0 or a constant 1 in the faulty circuit), the task of the test pattern generation tool is to derive a test vector sequence which at some clock cycle produces different Boolean outputs on some output wire in the good and the faulty design.

Test pattern generation is useful for logic optimization (i.e. design replacement) because of the observation that if it can be shown that there exists no test for a single stuck-at-1 (or stuck-at-0) fault then that net could be replaced with a constant 1 (or 0) without affecting the overall functionality of the good circuit. This is indeed true for combinational circuits [2]: a single stuck-at fault can be replaced by a constant (the fault is **redundant**) if the fault is untestable. However, for sequential circuits the notions of untestability and redundancy do not coincide: some faults that are untestable because they prevent initialization are not redundant and cannot be replaced by a constant value [1].

Thus, in order to classify a fault as redundant in a sequential circuit, a notion of design replacement is needed. A fault will be sequentially redundant if and only if the faulty circuit and the good circuit are related by this notion of design replacement. In this section, we present two notions of sequential redundancy (which defines a notion of design replacement)— the first, defined by [15] and subsequently used for logic optimization in [29], and the second, defined by [62] and later used for redundancy removal in [63].

3.5.1 Cheng's notion of sequential redundancy

Definition 3.5 *A fault is* **sequentially redundant** *if for any input sequence, any output line and any state of the faulty circuit D_1, the circuit D_1 produces 1 (0) whenever the original circuit D_0 produces 1 (0) from all states of D_0. If D_0 produces an unknown output*

U for some input sequence (i.e. 1 from at least one power-up state and 0 from at least another) then D_1 is allowed to produce either 0, 1 or U on that input sequence.

The intuition is that if the original design does not produce the same Boolean output from all power-up states, the replacement design can choose to produce either a 0 or a 1. However, we will show that this notion of design replacement is not suitable for use in hierarchical synthesis (optimizing a design which interacts with some environment) because the notion of replacement is not compositional.

Consider the gate-level design shown in Figure 3.3. First consider the left sub-circuit D. If we consider this circuit, the depicted stuck-at-1 fault is redundant by Cheng's definition (Definition 3.5). This is because for any input sequence π there are two power-up states of D such that one produces 0 and the other produces 1 on the output wire u for this input sequence π; so the wire w can be replaced by the constant 1. However, consider the entire circuit $D \otimes D'$. The fault is no longer redundant (even by Definition 3.5); on input sequence $01 \cdot 00$ the good design $D \otimes D'$ produces $0 \cdot 0$ from all power-up states, whereas from the power-up state 01 the faulty design produces $0 \cdot 1$ under the same input sequence. In fact, the length-1 initializing sequence 01 initializes the good design $D \otimes D'$ to 00 or 10 (a pair of equivalent reset states) but 01 is not an initializing sequence for the faulty design.

The intuition why Definition 3.5 does not lead to a design replacement notion which is valid under composition is that it treats the different output wires independently; also, output wires at different clock cycles are treated independently. As a result if the environment of the design is exploiting the correlation between the various output wires or output wires at different clock cycles this replacement notion causes problems. For example, in Figure 3.3, the environment design D' uses the fact whether $u = w$ or not; thus, even though for any input sequence the output u can be either be 0 or 1 it is always the same as w, and replacing u by a constant 1 loses this relationship.

In fact, the circuit in Figure 3.3 also illustrates a problem with the notion of sequential hardware equivalence that we discussed in the previous section. The design D does not have an initializing sequence, and thus the notion of sequential hardware equivalence is not even applicable to this design (i.e., the theory considers such designs "unrealistic". However, in composition with design D' the composed design does have an initializing sequence 01 and is 'well-behaved'.

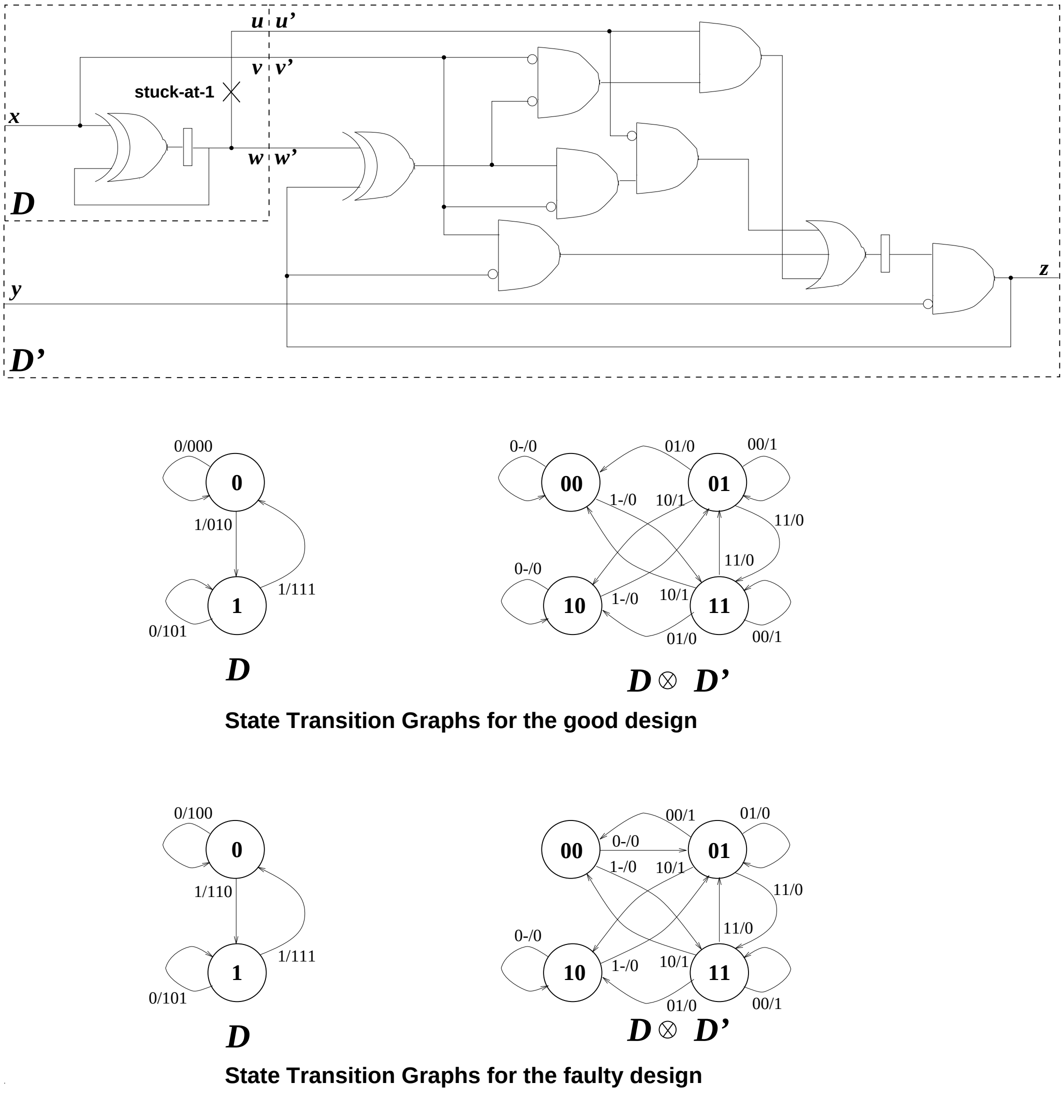

Figure 3.3: An irredundant stuck-at-fault for a circuit is redundant for a sub-circuit

3.5.2 Pomeranz and Reddy's notion of sequential redundancy

In [62] and [63], Pomeranz and Reddy classify faults into three categories. Faults are either **detectable**, **partially detectable** or **redundant**. They further sub-classify partially detectable faults into **Type I** faults and **Type II** faults. Their optimization procedure identifies Type II partially detectable and redundant faults and substitutes these faults by constants. Thus, we need to look at their definition of these two kinds of faults to understand their notion of sequential replacement. We will use D_0 to denote the original fault-free circuit and D_1 to denote the faulty circuit. Also notice that Pomeranz and Reddy restrict their analysis to designs which have single essential reset states; thus, while Pixley's definition of initializing sequence allows the design to arrive to any one of an equivalent set of essential reset states (Definition 3.2), Pomeranz and Reddy require the initializing sequence to drive every state to the same reset state.

Definition 3.6 *A fault is* **partially detectable** *if there exists an input sequence π and some state $t \in D_1$ such that for any state $s \in D_0$, the output behavior from s and t are different, i.e. $\exists \pi, \exists t \in D_1, \forall s \in D_0 : \lambda_{D_0}(s, \pi) \neq \lambda_{D_1}(t, \pi)$. A partially detectable fault is of* **Type II** *if the faulty machine has an initializing sequence.*

Definition 3.7 *A fault is* **redundant** *if there exists no pair (π, t), where π is an input sequence and t is a state in D_1 such that for any state $s \in D_0$, $\lambda_{D_0}(s, \pi) \neq \lambda_{D_1}(t, \pi)$.*

It can be seen that if replace good circuits by faulty circuits, where we allow Type II partially detectable and redundant faults, the notion of replacement is exactly the same as the one suggested by Pixley (the faulty circuit is sequentially hardware equivalent to the fault-free circuit; see Section 3.4), except that Pomeranz and Reddy require exactly one reset state. Thus, the same objections in Section 3.4 apply to this notion also. Our biggest objection, again, is that for arbitrary design replacements in a larger design environment, it may not be possible to control the initializing sequence for any arbitrary sub-design so that we can arbitrarily change such a sequence. Our other objections in Section 3.4 also apply here. We also note that if we replace only redundant faults (as per Definition 3.7), and not the Type II partially detectable faults, the notion of replacement will be the same as the notion of safe replacement, that we present in the next chapter.

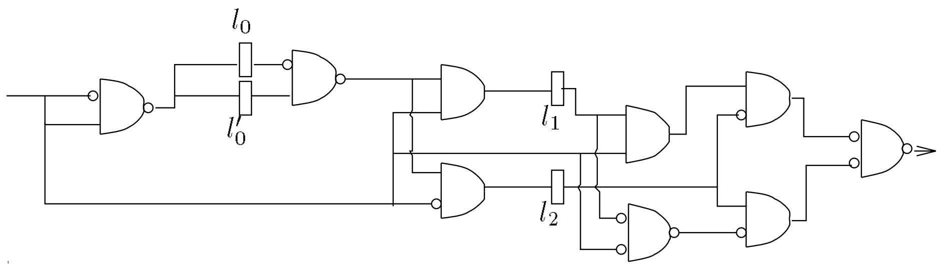

Figure 3.4: Retiming a latch across a fanout

3.6 Retiming and Resynthesis

Retiming and resynthesis [53] can be used to perform sequential optimization by alternating steps of moving of latches across combinational logic (retiming) and performing combinational resynthesis.

Retiming seems to be able to work if latches do not have reset lines. However, consider once again the circuit in Figure 3.3. If we retime this circuit to the circuit in Figure 3.4, we observe that a power-up state ($l_0 = 0, l'_0 = 1, l_1 = 0, l_2 = 0$) produces a 0 on input sequence $1 \cdot 1$, whereas no power-up state in the original design exhibits this behavior. So the reason given in Section 3.5.1, for searching for a new safe replacement condition, applies here as well. We will return to a discussion about the validity of retiming moves in Chapter 7.

3.7 Synchronous Recurrence Equations

Damiani and De Micheli [24] proposed using synchronous recurrence equations (or synchronous relations as used by [79]) to capture don't care information in sequential circuits. A synchronous recurrence equation expresses the flexibility for a sequential circuit as a Boolean relation on finite sequences of inputs and outputs. Any implementation that satisfies this synchronous recurrence equation is defined to be a valid replacement.

The reason why the flexibility provided by synchronous recurrence equations is not satisfactory for general designs (or FSMs) is that because a synchronous recurrence equation can only represent relationships between *finite* sequences of inputs and outputs, it cannot possibly represent designs where it is impossible to represent such flexibility using finite sequences. For example, consider the design D in Figure 3.5. The output at a time

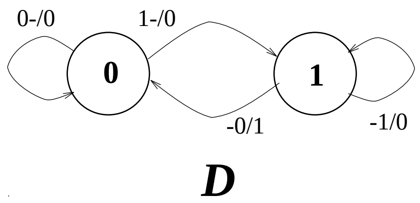

Figure 3.5: A synchronous recurrence equation cannot express the behavior of this design.

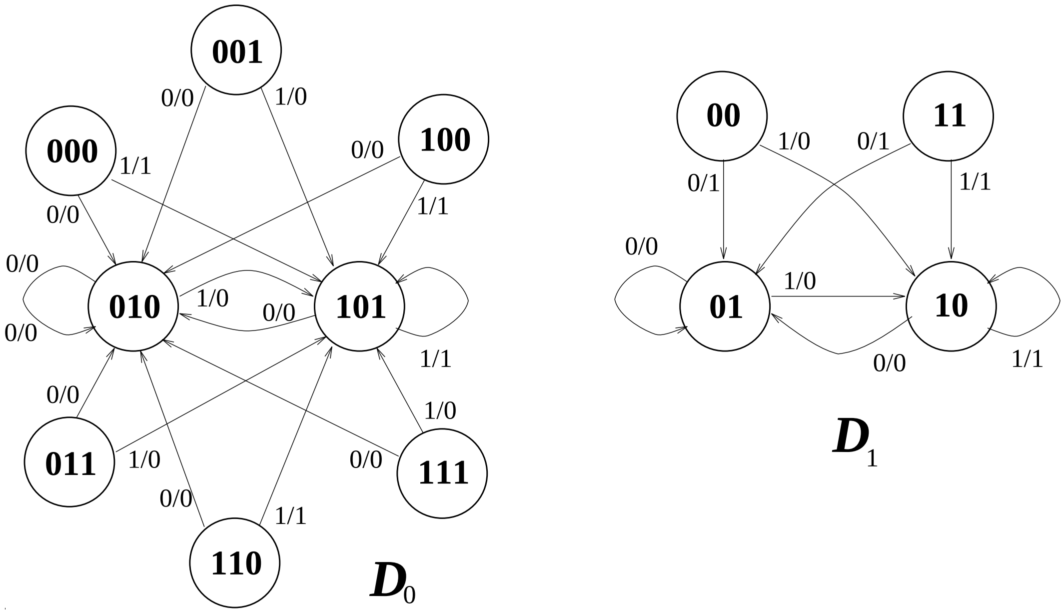

Figure 3.6: A synchronous recurrence equation cannot capture the output behavior on the first clock cycle after power-up.

may not just depend on any finite input sequence seen so far; it also depends on the state the design powered up in. For example, for an arbitrarily long sequence of inputs vectors 01, the output of the design at any cycle is not determined by the history of input and output behavior. It is not possible to write a synchronous recurrence equation for this design. Usually, synchronous recurrence equations are most useful to characterize pipelined (or feedback-free) designs.

One property of synchronous recurrence equations is that if the equation is such that the current output depends on the inputs over the last d clock cycles, it is not possible to represent the output behavior over the first $(d-1)$ clock cycles after power-up. So, consider an implementation of design D_0 in Figure 3.6. If x represents the input and y the output, then the synchronous equation that represents this design is $y_n = x_n \cdot x_{n-1}$, where

w_t represents the signal w at clock cycle t. Design D_1 in Figure 3.6 is another solution to the same recurrence equation. However, all power-up states of design D_0 output 0 on input 0; this is not the case in design D_1. Thus, if the environment of the design depends on the fact that the design outputs 0 on input 0 (for example, for the initialization of the environment) then we have made an "unsafe" replacement. Later, in Chapter 6 we will argue that for many design situations, it is known that for a given value of n, the behavior of the design in the first n clock cycles after power-up is not relevant. Thus, for such situations, a replacement using synchronous recurrence equations is safe.

Chapter 4

Safe Replaceability

In this chapter we will develop a notion of sequential equivalence which can be used to make design replacements on sequential circuits. We will first discuss the motivation and the desirable properties of such a notion.

Because today's digital designs are so large and complex, we would like to take an *arbitrary* piece of sequential logic (call this piece the original design) and be able to replace that piece with another piece (the new design) so that it is not possible to detect this replacement no matter what the surrounding logic is outside the design (see Figure 4.1). We want to make no assumptions about the behavior of the surrounding logic, which we call the environment of the design. Specifically, we do not even want to assume that we know the initializing sequence that the environment may use after power-up.

We saw in the previous chapter, in Section 3.1 that for combinational circuits,

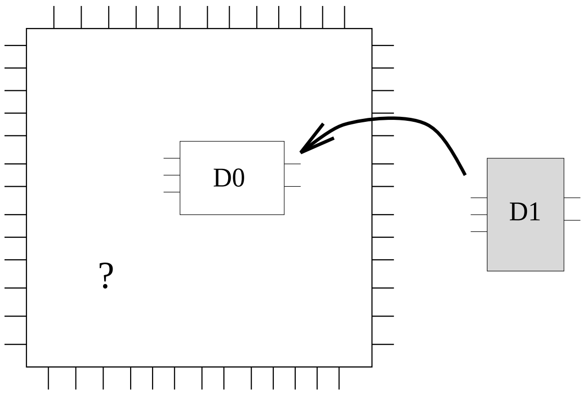

Figure 4.1: Replacement of a sequential design

there is an accepted notion of replacement equivalence which is used in logic synthesis tools such as SIS [72]. A circuit D is a valid replacement for circuit D, if for any input vector in B^m, the output vector (in B^n) produced by D is identical to that produced by C. This is an acceptable notion of replacement because for any reasonable composition with any environment, the composition of the environment with design C is going to behave identically as the composition of the environment with design D. Our motivation in this chapter is to present an analogous notion of replacement for the sequential case— without caring about the environment of the design, we want to present a condition for plug-out plug-in replaceability.

4.1 Notion of Safe Replaceability

Definition 4.1 *Design D_1 is a* **safe replacement** *for design D_0 ($D_1 \preceq D_0$) if given any state $s_1 \in Q_{D_1}$ and any finite input sequence $\pi \in I^*$, there exists <u>some</u> state $s_0 \in Q_{D_0}$ such that the output behavior $\lambda_{D_1}(s_1, \pi) = \lambda_{D_0}(s_0, \pi)$.*

We argue that the condition for safe replacement provides maximum flexibility while guaranteeing that the replacement cannot be detected by the environment. First, if we make the above condition any weaker, then there exists an input sequence π and a state in the new design D_1 so that if the D_1 powers up in this state and sees the sequence π, it will produce a behavior which could not have been seen from any state in D_0. This violates our requirement that no environment should be able to detect the replacement. Secondly, since we have assumed that the power-up state of a design cannot be predicted, if $D_1 \preceq D_0$, then for every input sequence any power-up state of D_1 behaves like some power-up state of D_0. This implies that any behavior from any state of D_1 is acceptable. Thus our condition in Definition 4.1 guarantees that replacing D_0 by D_1 cannot be detected by any environment.

Any design which has the same state transition graph as the original design trivially satisfies the safe replacement condition. As a non-trivial example consider designs D_0 and D_1 in Figure 4.2, where $D_1 \preceq D_0$. States 00, 11 and 01 in D_1 behave like states 000, 011 and 101, respectively, in D_0 for all input sequences. The remaining state 10 in D_1 behaves like state 010 for all input sequences starting with 0, and like state 101 for all input sequences starting with 1. Notice that state 10 in D_1 is not equivalent to any state in D_0; conversely, no state in D_1 is equivalent to state 001 in D_0. Definition 4.1 guarantees that there is no

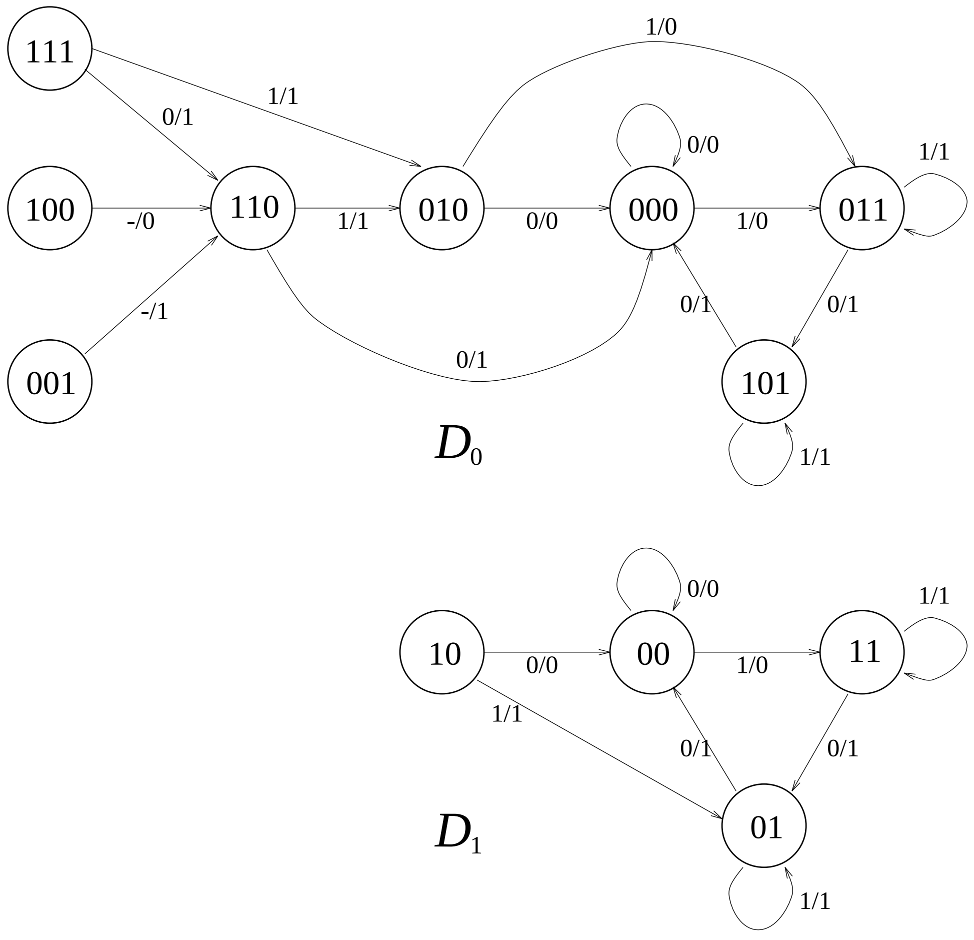

Figure 4.2: Example of a safe replacement ($D_1 \preceq D_0$)

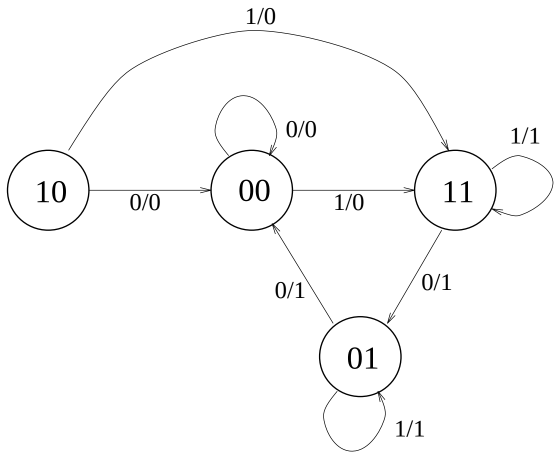

Figure 4.3: Replacement design D_2

input/output behavior in D_1 which is not present in D_0. On the other hand, state 001 in D_0 outputs sequence $1 \cdot 1 \cdot 0$ on the input sequence $1 \cdot 1 \cdot 1$ whereas no state of D_1 can exhibit this behavior. However, we had claimed that no environment can detect if D_0 is replaced by D_1. This apparent paradox can be explained by the observation that since it is not true that every power-up state of D_0 exhibits this behavior, the environment of D_0 could not possibly depend on this behavior ever happening, and hence it cannot always expect the output sequence for $1 \cdot 1 \cdot 0$ on input $1 \cdot 1 \cdot 1$ each time the design powers up.

4.2 Properties

Now, we present some interesting properties of the safe replacement condition:

Remark 1: If $D_1 \preceq D_0$, then there may be states $s_0 \in Q_{D_0}$ and $t_0 \in Q_{D_1}$ such that for all states $s \in Q_{D_0}$ and $t \in Q_{D_1}$, $s_0 \not\sim t$ and $t_0 \not\sim s$. For example, in Figure 4.2, state 111 in D_0 is not equivalent to any state in D_1; also, state 10 in D_1 is not equivalent to any state in D_0. Thus classical machine equivalence, that requires that every state in each design be equivalent to be some state in the other design [34], is not necessary for safe replacement, although it is sufficient.

Remark 2: Although the replacement design has to be compatible with the original design (same number of inputs and outputs), it does not have to have the same number of latches. Consider the design D_2 in Figure 4.3, which has only 4 states (and, thus, 2 latches) and is a safe replacement for the design D_0 in Figure 4.2.

Remark 3: As is obvious from Definition 4.1, the relation $\preceq$ is reflexive and transitive. However, the relation $\preceq$ is not symmetric. In Figure 4.2, although $D_1 \preceq D_0$, it is not true that $D_0 \preceq D_1$ because the state 110 in design D_0 produces an output sequence of $1 \cdot 0$ on the input sequence $1 \cdot 1$ and there is no state in D_1 which exhibits this behavior. However, an equivalence relation can easily be defined from a transitive and reflexive one.

Definition 4.2 *Designs D_0 and D_1 are* **replacement equivalent** *if $D_0 \preceq D_1$ and $D_1 \preceq D_0$.*

Consider the equivalence classes of designs modulo replacement equivalence– a design D belongs to an equivalence class $[D_0]$ if and only if it is replacement equivalent to D_0. We say that $[D_1] \preceq [D_0]$ if and only if $D_1 \preceq D_0$. Now, $\preceq$ is a partial ordering on these design equivalence classes (since it can be easily shown that it is reflexive, transitive and anti-symmetric).

Remark 4: **Compositionality of safe replaceability.** The notion of safe replacement was motivated by the need for making safe compositions with arbitrary environments. Here we show that safe replacements are preserved under arbitrary compositions[1]:

Proposition 4.1 *If $D \preceq C$, then $(R \otimes D) \preceq (R \otimes C)$, for any composed design $R \otimes C$.*

Proof. Consider any input sequence π to the design $R \otimes C$, and any state $(r, d) \in (R \otimes C)$. If we simulate the design $(R \otimes C)$ starting from the state (r, d) and inputing the sequence π, we will observe a sequence ρ which is the input to the sub-design C, and an output sequence σ which is the output of C. Since $D \preceq C$, there must be a state $c \in D$ such that the design D outputs σ on the input sequence ρ. Now, consider the state $(r, c) \in (R \otimes D)$. Clearly, on the input sequence π, the composed design $(R \otimes D)$ will produce the same output sequence from state (r, c) as the design $(R \otimes C)$ would from state (r, d). $\square$

Remark 5: While the theory of sequential hardware equivalence in Section 3.4 cannot be applied to designs which are not essentially resetable, our safe replacement conditions work for any designs. For essentially resetable designs, sequential replaceability is a stronger condition than SHE (sequential replaceability implies SHE, because as the following Theorem 4.2 shows, if $D_1 \preceq D_0$, any initializing sequence for D_0 can align all state pairs in $Q_{D_0} \times Q_{D_1}$.

[1]As stated in Chapter 2, we assume that the composition does not create a combinational loop.

Theorem 4.2 *If $D_1 \preceq D_0$ and ρ is an initializing sequence for D_0 then ρ is also an initializing sequence for D_1 and for any states $s_0 \in Q_{D_0}$ and $s_1 \in Q_{D_1}$, $\delta_{D_0}(s_0, \rho) \sim \delta_{D_1}(s_1, \rho)$.*

Proof. Pick a state $s' \in Q_{D_0}$. For the proof, we just need to show that for any state $t \in Q_{D_1}$, $\delta_{D_1}(t, \rho) \sim \delta_{D_0}(s', \rho)$. Suppose not. Then $\delta_{D_1}(t, \rho) \not\sim \delta_{D_0}(s', \rho)$. There exists a sequence π such that $\lambda_{D_1}(\delta_{D_1}(t, \rho), \pi) \neq \lambda_{D_0}(\delta_{D_0}(s', \rho), \pi)$. However, since $D_1 \preceq D_0$, there exists a state $s \in Q_{D_0}$ such that $\lambda_{D_1}(t, \rho \cdot \pi) = \lambda_{D_0}(s, \rho \cdot \pi)$. This also means that $\lambda_{D_1}(\delta_{D_1}(t, \rho), \pi) = \lambda_{D_0}(\delta_{D_0}(s, \rho), \pi)$. However, since ρ is an initializing sequence for D_0, $\delta_{D_0}(s, \rho) \sim \delta_{D_0}(s', \rho)$. Thus, $\lambda_{D_1}(\delta_{D_1}(t, \rho), \pi) = \lambda_{D_0}(\delta_{D_0}(s, \rho), \pi) = \lambda_{D_0}(\delta_{D_0}(s', \rho), \pi)$, which is a contradiction. $\square$

Remark 6: The idea of safe replacement implicitly uses the fact that the original design D_0 can power up in any state. Power-up states are generally beyond the control of designers for physical reasons. It may be possible that, by design, D_0 cannot power up in some states or that the likelihood of powering up in some states is so remote that D_0 is never observed to do so. The notion of safe replacement still applies with replacing Q_{D_0} and Q_{D_1}, in Definition 4.1, by the power-up states of D_0 and D_1, respectively.

Remark 7: **Relationship to language containment.** It is possible to describe the notion of safe replacement in terms of a containment relation between languages of FSMs. Let $\mathcal{L}_D$ denote the language of design D. Then the alphabet of the language is $\Sigma = I \times O$. $\mathcal{L}_D$ consists of sets of strings of the alphabet. A string $(i_1, o_1)(i_2, o_2) \cdots (i_p, o_p)$ belongs to the language of D if and only if there exists some state $s \in Q_D$ such that $\lambda(s, i_1 \cdot i_2 \cdots i_p) = o_i \cdots o_p$. Notice that, unlike the theory of finite automata (e.g. [35]), we do not need the notion of an initial state or final states to describe the language of a design. Now it is easy to see $D_1 \preceq D_0$ if and only if $\mathcal{L}_{D_1} \subseteq \mathcal{L}_{D_0}$. As an aside we note that the language containment paradigm is also used to specify a relationship between the abstraction and the implementation in the process of formal specification and verification of digital systems [44]— an implementation satisfies the abstraction or the property if the language of the implementation is contained in the language of the abstraction or the property. One problem in this approach is that, although we are able to establish that the implementation satisfies the abstraction or the property, we are unable to guarantee that the implementation does anything useful at all; this is because, for example, the finite automaton with the empty language is trivially contained in all languages and hence satisfies all properties. On the other hand, using language containment on the languages of FSMs does not have this problem. This is because our FSMs, which are always deterministic, are also complete and have an output sequence for every input sequence. Thus, unlike finite automata, it is impossible to have a *design* with the empty language. As the following remark shows, every design that is a safe replacement for an existing design must have the same steady-state behavior as the existing design.

Remark 8: Necessary conditions for a safe replacement. Even though there is some flexibility for the implementation of the replacement design, it cannot have arbitrarily few states; in fact, as the following results show, each tSCC (see Definition 2.4) in the replacement design must be equivalent (Definition 3.1) to some tSCC in the original design.

Lemma 4.3 ((Lemma 2 in [20])) *Suppose that DFSM's M_0 and M_1 have no equivalent states then there is an input sequence π such that for any states s_0 of M_0 and s_1 of M_1, $\lambda_{M_0}(s_0, \pi) \neq \lambda_{M_1}(s_1, \pi)$.*

Lemma 4.4 *If $D_1 \preceq D_0$, and $t \in Q_{D_1}$ lies in a tSCC of D_1, then there exists state $s \in Q_{D_0}$ such that $s \sim t$.*

Proof. (by contradiction). Assume that $D_1 \preceq D_0$. Let M be a tSCC of D_1. Then M is a DFSM. Suppose that no state of M is equivalent to any state of D_0. By Lemma 4.3, there is a sequence π that differentiates every state of M from every state of D_0. *A fortiori*, π differentiates a particular state, say s of M from every state of D_0. Therefore, the assumption that $D_1 \not\preceq D_0$ is false. $\Box$

Theorem 4.5 *If $D_1 \preceq D_0$, and M_1 is a tSCC in design D_1, then there must be a tSCC M_0 in design D_0 such that $M_0 \equiv M_1$.*

Proof. Consider a state t in M_1. From Lemma 4.4, we know that there exists a state $s \in Q_{D_0}$ such that $s \sim t$. We will show that if M_0 is any tSCC reachable from s then M_0 is equivalent to M_1.

Consider any state s' in M_0. Let π be an input sequence such that $\delta_{D_0}(s, \pi) = s'$. Since M_1 is closed under all inputs, $t' = \delta_{D_1}(t, \pi)$ lies in M_1. Also, since $s \sim t$, we have $s' \sim t'$.

Similarly, consider any state t'' in M_1. Let ρ be an input sequence such that $\delta_{D_1}(t', \rho) = t''$. Again, $s'' = \delta_{D_0}(s', \rho)$ lies in M_0, and $s'' \sim t''$.

Thus, DFSM's M_0 and M_1 are equivalent. □

As remarked previously, the equivalence classes modulo replacement equivalence are partially ordered by safe replacement ($\preceq$). Theorem 4.5 shows that each tSCC of design D_0 defines a minimal element that is $\preceq$ to the equivalence class $[D_0]$. Designs with unique minimal predecessors are therefore ones with unique tSCC's, or ones where all tSCC's are equivalent to each other.

Most real designs have a single tSCC [61]. For such designs, any reasonable notion of design replacement will probably preserve the behavior of the tSCC; in fact, all other notions of design replacement that we discussed in Chapter 3 have this property; for example, for the problem of redundant fault removal, Pomeranz and Reddy have proved [62] that single stuck-at faults which are Type II partially detectable (see Section 3.5.2) create designs where the tSCC is equivalent to the tSCC of the original design. Safe replacement differs from the other notions in the behavior of designs before it enters the tSCC; a design may stay outside the tSCC for an arbitrary number of clock cycles. Also, safe replacement works for designs which do have more than one tSCC.

Remark 9: Known Initializing Sequence Set. Sometimes, the designer knows the initializing sequences for the design and does not need a replacement condition as strong as in Definition 4.1. The designer knows that whenever the design powers up, one of sequences from an initializing sequence set Π is applied and the design is reset to some desired behavior. The designer does not care about the outputs while an initializing sequence is applied[2], and knows that the design "works" as long as <u>some</u> sequence from Π is applied after power-up. As an example, consider a design with two input lines a and b. The designer knows that after power-up the input a will be set at 1 for the first 2 clock cycles to initialize the design. For this example, the initializing set $\Pi = \{(10 \cdot 10, 10 \cdot 11, 11 \cdot 10, 11 \cdot 11\}$, where a represents the first input and b the second input. For an initializing sequence set Π we can modify our safe replacement condition from Definition 4.1 to the following:

Definition 4.3 *Design* D_1 *is a* **safe replacement** *for design* D_0 **under the initializing sequence set** Π *(*$D_1 \overset{\Pi}{\preceq} D_0$*) if given any state* $s_1 \in Q_{D_1}$*, an initializing sequence* $\pi_1 \in \Pi$*,*

[2] If the designer does care about the outputs during the initialization phase, it is easy to modify our safe replacement condition for such a case also.

and any finite input sequence $\rho \in I^*$, there exists some state $s_0 \in Q_{D_0}$ and an initializing sequence $\pi_0 \in \Pi$ such that the output behavior $\lambda_{D_1}(\delta_{D_1}(s_1, \pi_1), \rho) = \lambda_{D_0}(\delta_{D_0}(s_0, \pi_0), \rho)$.

We can also derive the following result (the proof is similar to that of Theorem 4.5 and is omitted for brevity).

Theorem 4.6 *If $D_1 \overset{\Pi}{\preceq} D_0$ and M_1 is a tSCC in design D_1, then there must be a tSCC M_0 in design D_0 such that $M_0 \equiv M_1$.*

It should be noted that for a single design a designer may have more than one set of initializing sequences. Each of these set might be used to initialize the design to a different behavior. For such a situation the designer would like to verify that the replacing design is a safe replacement under each initializing sequence set.

Chapter 5

Synthesis and Verification for Safe Replaceability

In the previous chapter we presented the notion of safe replaceability and we argued that this notion of design replacement allows the maximum flexibility for replacement while guaranteeing that the environment of the design does not detect the replacement. In this chapter we will first use the notion of safe replacement to optimize gate-level designs with the goal of reducing the area of such designs. We will then address the question of verifying whether one design is a safe replacement of the other– we will first prove the complexity of this problem, then present a heuristic algorithm which is expected to be efficient for many situations.

5.1 Logic Optimization

We want to exploit the flexibility provided by the safe replacement condition in Definition 4.1 to optimize synchronous sequential circuits. Unfortunately, Definition 4.1 does not directly provide a closed form expression to express all the flexibility for safe replacement.

A sequential gate-level design (like the one in Figure 2.2) can be viewed as a connection between a purely combinational part and a set of latches (Figure 5.1). The inputs to the combinational part are the real primary outputs of the design $\vec{i}$ plus the wires from the latches, or the present state vector, denoted by $\vec{x}$. The outputs of the combinational part are the real primary outputs of the design $\vec{o}$ plus the wires to the latches, or the next

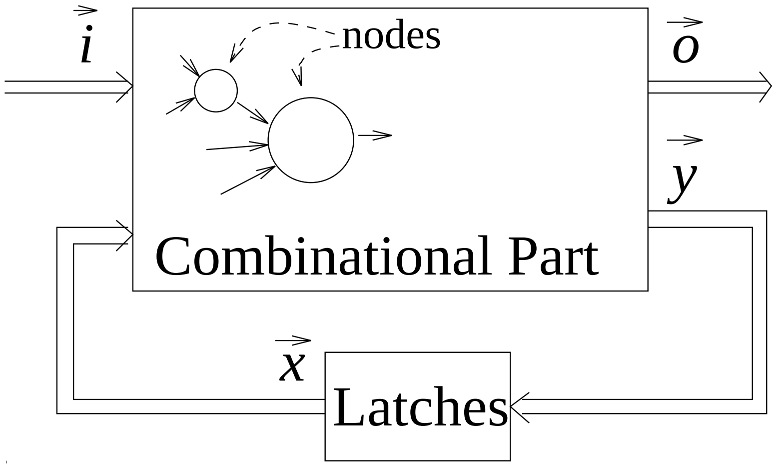

Figure 5.1: Combinational Gates + Latches = Multi-level Sequential Network

state vector, denoted by $\vec{y}$. We want to optimize this combinational part while maintaining the safe replacement condition. If we could express the flexibility in Definition 4.1 by a Boolean relation in $(\vec{i}, \vec{x}, \vec{o}, \vec{y})$, we could use known techniques [70] for minimizing multi-level networks, given a Boolean relation in terms of the inputs and outputs of the network, by treating $(\vec{i}, \vec{x})$ as the inputs and $(\vec{o}, \vec{y})$ as the outputs of the combinational part of the network.

Unfortunately, the flexibility allowed by the safe replacement condition cannot be represented by a Boolean relation between the domain space $(\vec{i}, \vec{x})$ and the range space $(\vec{o}, \vec{y})$. Consider the design D_2 in Figure 5.2 which is a safe replacement of the design D_0 of Figure 4.2 in the previous chapter. The two designs differ on their mappings of the following 6 points in $(\vec{i}, \vec{x})$: $(0, 111), (0, 100), (1, 100, 1), (0, 001), (1, 001), (0, 110)$. One property of Boolean relations is that the flexibility for each point in the domain space is independent of other points [73]. So, if the flexibility for safe replacement could be expressed by a Boolean relation, then every design corresponding to a flexibility choice for each of these 6 domain points would be a valid replacement (there are 2^6 such designs). In particular, design D_3 in Figure 5.3, which behaves like D_2 on point $(0, 110)$ and like D_0 on the other points, would be a safe replacement. However, this is not so because if design D_3 powers up in state 111 and is given the input sequence $0 \cdot 0 \cdot 0$ it produces the output sequence $1 \cdot 1 \cdot 1$, whereas there is no state in D_0 which exhibits this behavior. Thus the flexibility for safe replaceable designs with the same number of latches cannot be expressed as a Boolean

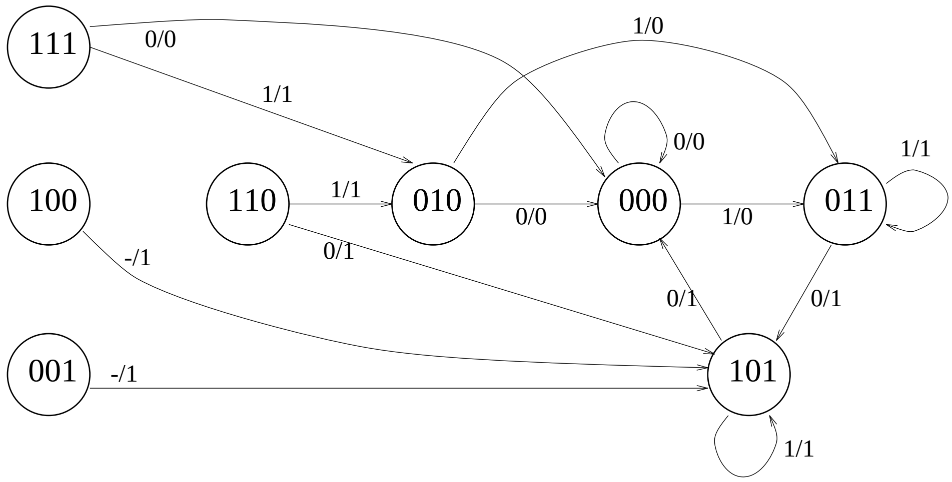

Figure 5.2: Design D_2 (a safe replacement for D_0)

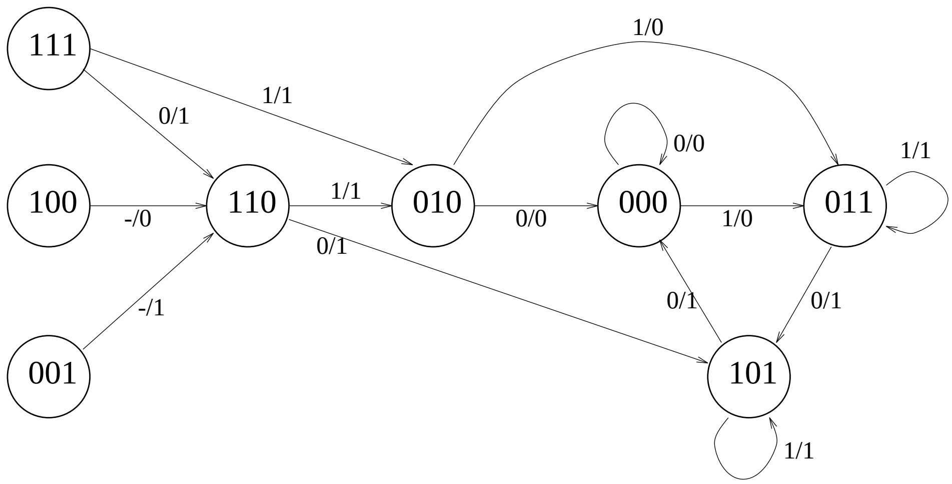

Figure 5.3: Design D_3 (an unsafe replacement for D_0)

relation in $(\vec{i}, \vec{x}) \times (\vec{o}, \vec{y})$. One way to represent such flexibility would be through Multiple Boolean Relations [73], which are arbitrary sets of Boolean relations.

5.1.1 Sufficient Condition for a Safe Replacement

As we have just argued, the complete flexibility for safe replacement can be expressed by a multiple Boolean relation. However, because of the particularly large solution space of multiple Boolean relations, there are no known general techniques to use multiple Boolean relations for logic synthesis. We now provide a sufficient (but not necessary) condition for safe replacement, from which we will obtain a Boolean relation in $(\vec{i}, \vec{x}, \vec{o}, \vec{y})$ to express partial flexibility for safe replacement.

Proposition 5.1 *Given designs D_0 and D_1 such that for every state $s_1 \in Q_{D_1}$ and input $a \in I$, there exists a state $s_0 \in Q_{D_0}$ such that $\lambda_{D_1}(s_1, a) = \lambda_{D_0}(s_0, a)$ and $\delta_{D_1}(s_1, a) \sim \delta_{D_0}(s_0, a)$. Then $D_1 \preceq D_0$.*

Proof. Choose any state $s_1 \in Q_{D_1}$, and any input sequence $\pi = a_0 \cdot a_1 \cdots a_p \in I^*$. Now, there exists a state $s_0 \in Q_{D_0}$ such that $\lambda_{D_1}(s_1, a_0) = \lambda_{D_0}(s_0, a_0)$ and $\delta_{D_1}(s_1, a_0) \sim \delta_{D_0}(s_0, a_0)$. Thus $\lambda_{D_1}(s_1, \pi) = \lambda_{D_0}(s_0, \pi)$, and hence $D_1 \preceq D_0$. $\Box$

Informally, the above condition for safe replacement is a compromise between machine equivalence (Definition 3.1) and the necessary condition for safe replacement: we do not require that each state in the replacement design be equivalent to a state in the old design, but we require that each state in the replacement design which is reachable in one step be equivalent to some state in the old design. We will see in Section 5.1.3 that there is strong evidence that even the above condition, which is not a necessary condition for safe replacement, should provide us enough flexibility for resynthesis.

For an example of safe replacement using the condition in Proposition 5.1, consider designs D_0 and D_1 of Figure 4.2 which satisfy the sufficient condition of Proposition 5.1, and thus $D_1 \preceq D_0$. On the other hand, for designs D_4 and D_5 in Figure 5.4, it can be seen that $D_5 \preceq D_4$. However, in design D_5 state 01, on input 00, goes to itself, and no state in design D_4 is state equivalent to state 01 of design D_5 (as discussed in Remark 4.2 in the previous chapter, there may be states in the replacement design which are not equivalent to any state in the original design).

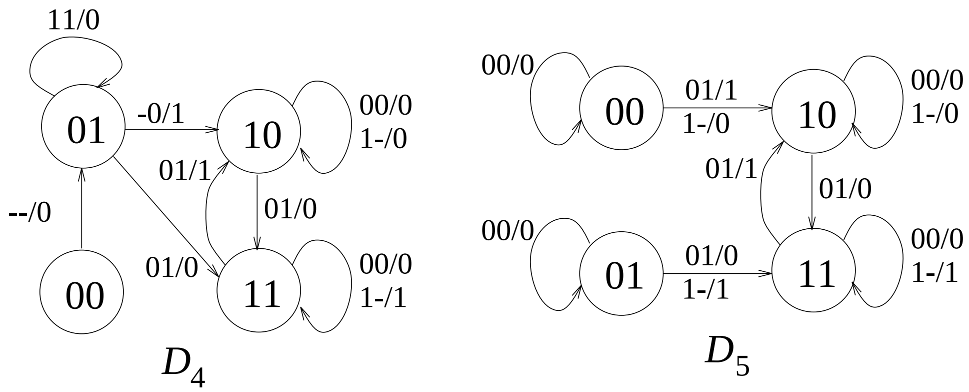

Figure 5.4: Example of a safe replacement

5.1.2 Flexibility for Resynthesis

We will use the sufficient condition in Proposition 5.1 to derive a method to extract and use flexibility for sequential resynthesis.

First, note that a <u>necessary</u> condition for safe replacement is that every tSCC in the new design must be equivalent to a tSCC in the old design (Theorem 4.5)— a tSCC can be thought of as defining a steady state behavior of the design. So at the state-transition level we cannot alter the behavior of the states in <u>some</u> tSCC of the original design. For our synthesis method we will choose to preserve the behavior of a set of states which is closed under all inputs, called the **core**:

Definition 5.1 *Given a design D, a set of states $S \subseteq Q_D$ is called a* **core** *of the design if it is a closed set under all inputs, i.e. for any input a and any state $s \in S$: $\delta_D(s, a) \in S$.*

Note that since the core is closed under all inputs it contains a tSCC of the original design and thus satisfies the necessary condition discussed above.

Proposition 5.1 indicates that each state reachable from some state in D_1 is equivalent to some state in D_0. The set of these states will serve as the core of the old design, which we will reproduce in the new design. We require that each of the remaining states in the new design satisfy the following Boolean relation $\mathcal{P}(\vec{i}, \vec{o}, \vec{y})$. Let $core(\vec{x}) = 1$ if and only if state $\vec{x}$ lies in the chosen core.

$$\mathcal{P}(\vec{i}, \vec{o}, \vec{y}) = \exists \vec{x}_0 [(\lambda_{D_0}(\vec{x}_0, \vec{i}) = \vec{o}) \wedge (\delta_{D_0}(\vec{x}_0, \vec{i}) = \vec{y})] \wedge core(\vec{y})$$

Intuitively, a triple $(\vec{i}, \vec{o}, \vec{y})$ satisfies the relation $\mathcal{P}$ iff there is a state $\vec{x}_0$ in the given design which transitions to state $\vec{y}$ and outputs $\vec{o}$ on input $\vec{i}$, and $\vec{y}$ lies inside the core. It is easy to see that if the core of the old design is preserved and the relation $\mathcal{P}$ holds for the remaining states of the new design, the sufficiency condition of Proposition 5.1 is satisfied. Notice that Proposition 5.1 expresses the flexibility at the STG-level; on the other hand, the above Boolean relation $\mathcal{P}$ uses the encoding of the states as well (in denoting the states by the variables $\vec{x}$ and $\vec{y}$). We are preserving the state encodings of the states in our design, even though this is not required for safe replaceability. This is done to make our synthesis procedure tractable; state encoding is, in general, a hard problem [25, 26, 74, 67] and we do not attempt to solve it in this thesis.

We can now form a Boolean relation $\mathcal{R}(\vec{i}, \vec{x}, \vec{o}, \vec{y})$ which expresses the flexibility for the entire network including states inside the core:

$$\begin{aligned}\mathcal{R}(\vec{i}, \vec{x}, \vec{o}, \vec{y}) \quad &= \quad [\ core(\vec{x}) \wedge (\vec{y} = \delta_{D_0}(\vec{x}, \vec{i})) \wedge (\vec{o} = \lambda_{D_0}(\vec{x}, \vec{i}))] \vee \\ &\quad\quad [\neg(core(\vec{x})) \wedge \mathcal{P}(\vec{i}, \vec{o}, \vec{y})] \end{aligned} \tag{5.1}$$

Intuitively, the relation $\mathcal{R}$ ensures that the states inside the core have the same behavior (next state and output) as the original design, while the states outside the core are free to choose any behavior which satisfies the relation $\mathcal{P}$.

We recall from Section 5.1 that the total flexibility for the network cannot be expressed by a Boolean relation; we are able to get a relation here because (a) our condition in Proposition 5.1 is only a sufficient condition, and (b) we are preserving the encoding of the states inside the core.

5.1.3 Choice of Core

For a given design D there may be many choices for the core which satisfy Definition 5.1. Some natural choices are:

- The set of all states Q_D.
- Any **onion ring**[1] of the design. Onion rings $A_1, A_2, \ldots$ are defined recursively:

$$\begin{aligned} A_1 &= Q_D, \\ A_{k+1} &= \{y \mid \exists i \in I, x \in A_k : \delta_D(x, i) = y\} \end{aligned} \tag{5.2}$$

[1] This notion was earlier defined in [59]; the term "onion ring" was first used in [37].

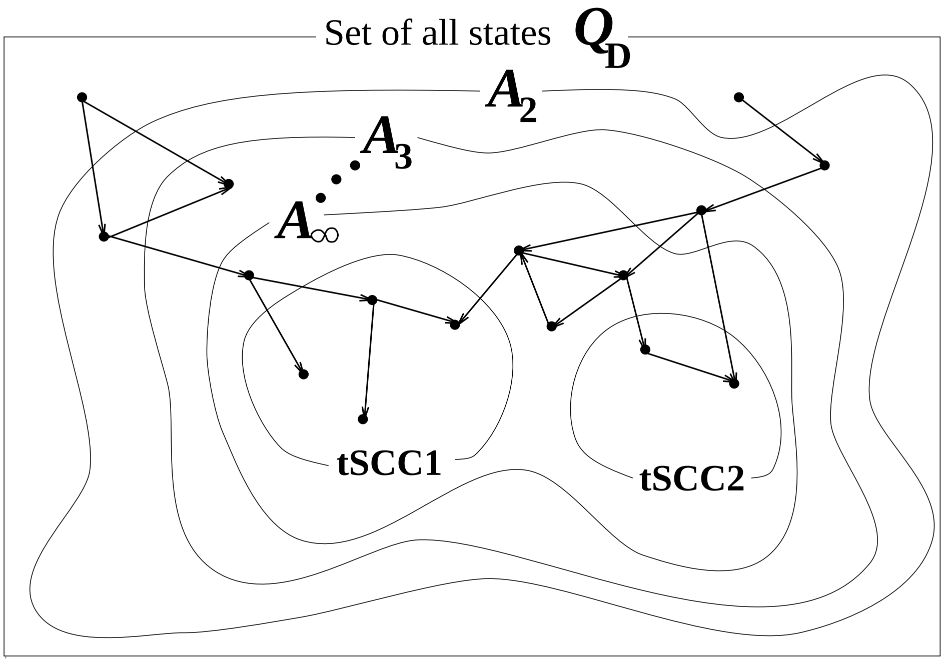

Figure 5.5: The onion ring structure and the tSCCs of a design. Only some states and transitions are shown.

For design D_4 in Figure 5.4, $A_1 = \{00, 01, 10, 11\}$, $A_2 = A_3 = \cdots = A_\infty = \{01, 10, 11\}$. A_∞ is also called the **outer envelope** of a design.

- Any tSCC of the design. Design D_4 has only one tSCC: $\{10, 11\}$.

We depict the various choices for the core in Figure 5.5.

Any of the above choices satisfies Definition 5.1 and can be used in equation 5.1. The ideal choice is a small core to give us a large number of states outside the core because we have flexibility only in modifying the behavior of states outside the core. The smallest tSCC is the smallest set which qualifies as a core.

Now we look at some example circuits to see the flexibility we have in choosing the core. In this thesis, we are using a subset of the ISCAS89 benchmark circuits [12] to demonstrate the synthesis techniques. We use binary decision diagrams (BDDs) and a standard toolset (similar to the ones in [7, 51, 68]) to implement our algorithms. Because of limitations imposed by the use of the BDD package we use the subset of ISCAS89 circuits[2]

[2]We use the benchmark circuits supplied with the SIS package, version 1.2. For some inexplicable reason, at least two of the benchmark circuits, `s208` and `s420` seem to be slightly different from the ones used by others [18, 50, 64].

procedure onion-ring (**input:** BDD for the transition relation $\mathcal{T}(\vec{x}, \vec{i}, \vec{y})$, n) {

$\mathcal{C}(\vec{x}) \leftarrow 1$

loop $(n-1)$ **times** {

$\mathcal{D}(\vec{y}) \leftarrow \exists \vec{x} \exists \vec{i} : [\mathcal{C}(\vec{x}) \cdot \mathcal{T}(\vec{x}, \vec{i}, \vec{y})]$

$\mathcal{D}(\vec{x}) \leftarrow \mathcal{D}(\vec{y})_{(\vec{y} \leftarrow \vec{x})}$

if $(\mathcal{D}(\vec{x}) = \mathcal{C}(\vec{x}))$

return $(\mathcal{C}(\vec{x}))$

$\mathcal{C}(\vec{x}) \leftarrow \mathcal{D}(\vec{x})$

}

return $(\mathcal{C}(\vec{x}))$

}

Figure 5.6: Procedure to obtain the n-th onion-ring A_n.

which have fewer than 30 latches (i.e. upto 2^{30} states). All experiments in this thesis have been conducted on a DEC Alpha server. The algorithm for computing the onion rings is very simple, and appears in Figure 5.6. The outer envelope of the design is the fixed-point of the onion ring computation, and and cabe obtained by computing onion-ring$(\mathcal{T}(\vec{x}, \vec{i}, \vec{y}), \infty)$. A tSCC of a design can be computed using the algorithm in Figure 5.7. Both the algorithms (for the outer-envelope, A_∞, and the tSCC computation) can, in the worst case require an exponential number of iterations in the number of state bits; however, we do not see this blow-up for almost all the circuits. The results for computing the various onion-rings and the tSCC appear in Table 5.1. Each of these designs has only one tSCC. This is in agreement with the thought that all useful designs have exactly one tSCC [61]; on the other hand, other people have argued that if we pick an arbitrary piece of logic from a larger design, this piece may have more than one tSCC even though the larger design has only one tSCC representing the steady-state behavior [66]. In this thesis we are motivated by the problem of replacing arbitrary pieces of logic and thus we allow designs to have multiple tSCCs.

```
procedure tSCC (input: T(x⃗, i⃗, y⃗)) {
    A(x⃗) ← onion-ring(T(x⃗, i⃗, y⃗), ∞)
    forever do {
        Pick an arbitrary state (a minterm) a⃗ ∈ A(x⃗)
        F(x⃗) ← a⃗
        N(x⃗) ← 0
        do {
            F(x⃗) ← F(x⃗) + N(x⃗)
            N(y⃗) ← ∃i⃗∃x⃗ : [T(x⃗, i⃗, y⃗) · F(x⃗)]
            N(x⃗) ← N(y⃗)_(y⃗←x⃗)
        } while (N(x⃗) ⊆ F(x⃗))
        /* F(x⃗) is the set of states reachable from a⃗ */
        B(x⃗) ← a⃗
        N(x⃗) ← 0
        do {
            B(x⃗) ← F(x⃗) + N(x⃗)
            B(y⃗) ← N(x⃗)_(x⃗←y⃗)
            N(x⃗) ← ∃i⃗∃y⃗ : [T(x⃗, i⃗, y⃗) · B(y⃗)]
        } while (N(x⃗) ⊆ B(x⃗))
        /* B(x⃗) is the set of states which can reach a⃗ */
        if (F(x⃗) ⊆ B(x⃗)) return F(x⃗)
        else A(x⃗) ← A(x⃗) · ¬B(x⃗)
    }
}
```

Figure 5.7: Procedure to derive a tSCC

Ckt.	Inputs	Outputs	Latches	Number of states			
				$Q_D = A_1$	A_2	A_∞	tSCC
s27	4	1	3	8	6	6	6
s208	10	1	8	256	256	256	256
s298	3	6	14	16384	5800	218	218
s344	9	11	15	32768	23232	5342	1487
s349	9	11	15	32768	23232	5342	1487
s382	3	6	21	2097152	23740	8864	8864
s386	7	7	6	64	13	13	13
s400	3	6	21	2097152	23740	8864	8864
s420	18	1	16	65536	65536	65536	65536
s444	3	6	21	2097152	23740	8864	8864
s510	19	7	6	64	61	57	47
s526	3	6	21	2097152	401460	8868	8868
s641	35	23	19	524288	6643	6461	1544
s713	35	23	19	524288	6643	6461	1544
s820	18	19	5	32	25	25	25
s832	18	19	5	32	25	25	25
s953	16	23	29	536870912	504	504	504
s1196	14	14	18	262144	2652	2615	2615
s1238	14	14	18	262144	2652	2615	2615
s1488	8	19	6	64	48	48	48
s1494	8	19	6	64	48	48	48

Table 5.1: Number of states in the possible choices for the core.

For the multi-level synthesis method that we describe in the next section (which works incrementally starting from the original design), we are restricted by a requirement that the starting design must itself satisfy the Boolean relation $\mathcal{R}$. The only known method for minimizing multi-level networks under flexibility expressed by a Boolean relation [70] requires this restriction, as we shall see later in Section 5.1.4.

For example, if we choose the tSCC of design D_0 in Figure 4.2 (states 000, 011 and 101) as the core, the starting design D_0 does not satisfy relation $\mathcal{R}$ in expression 5.1 because the state 111 (a non-core state) does not jump to a state inside the core on input 1 and hence does not satisfy $\mathcal{P}$. Choices of core that guarantee that the given design satisfies $\mathcal{R}$ are Q_{D_0} (the set of all states) and the second onion ring A_2 (the set of states reachable in one step from Q_{D_0}). The former does not give any flexibility because all states in the starting design are also present in the new design; so we make the latter choice. While it might seem that we lose much flexibility by having to choose a much larger core than the smallest possible, our experiments in Table 5.1 indicate that choosing A_2 gives us most of the flexibility for most examples; for most circuits, most of the states lie in the difference of the sets A_1 and A_2. For two benchmark circuits, `s208` and `s420`, the number of states in the tSCC is exactly equal to 2^L, where L is the number of latches. Thus for these two circuits we are not going to see any flexibility since we preserve the behavior of all states inside the core; so we have excluded these two circuits from our synthesis experiments in the thesis.

5.1.4 Multi-level Synthesis

As we discussed in the beginning of this chapter, the combinational part of the sequential logic network is an acyclic interconnection of nodes, which computes the output functions $\{o_1, o_2, \ldots, o_n\}$ and next state functions $\{y_1, y_2, \ldots, y_t\}$ for any given input vector $\{i_1, i_2, \ldots, i_m\}$ and present state vector $\{x_1, x_2, \ldots, x_t\}$. Our optimizations are going to be only on the combinational part of the circuit (even though the flexibility derived in the previous section is from the sequential behavior of the network); thus, in this section, we will use the term inputs to refer jointly to $\{i_1, i_2, \ldots, i_m, x_1, x_2, \ldots, x_t\}$ and outputs to refer to $\{o_1, o_2, \ldots, o_n, y_1, y_2, \ldots, y_t\}$.

Intermediate nodes are those internal nodes which do not correspond to inputs or outputs. The network computes a function from the input space B^{m+t} to the output space B^{n+t} derived by composing the functions at the intermediate nodes.

Since hardware designs typically arise by composing small modules, it is very natural for circuits to have a multi-level structure. The nodes of the corresponding Boolean network represent the logical functionality of the modules. It has been observed that the area of the hardware implementation of a design is strongly correlated to the total number of literals in the **factored form** [11] representation of the functions at the logic nodes. Thus minimizing the function (with respect to the literal count) at the node constitutes a powerful synthesis technique.

At any intermediate node of a network there is a local function $f_i : B^r \rightarrow B$, where r is the cardinality of the support. **Node simplification** is the process of optimizing a Boolean network by using don't cares in conjunction with a two level minimizer [10] to optimize the functions at the nodes. These don't cares arise in several ways:

- Because of the structure of the network, only a certain subset of B^r may be generated by assignments to the inputs. This gives rise to **satisfiability don't care** (SDC) points for f_i [11].

- For certain input assignments, the values taken by the primary outputs of $\mathcal{N}$ may be independent of the function computed by a node; these are **observability don't care** points (ODC) for that node [71].
- For certain input assignments the functionality of the node can be changed without destroying safe replaceability; this flexibility, that we have described in the preceding sections, leads to the **replaceability don't cares** points (RDC).

The ODC at a node u can be computed by considering the output α_u of the node to be another input; thus the primary outputs of $\mathcal{N}$ are expressed in terms of the primary inputs and α_u. The ODC is the set of points in B^r where no primary output of $\mathcal{N}$ depends on α_u.

Let $\mathcal{R}(\vec{i}, \vec{x}, \vec{o}, \vec{y})$ be a Boolean relation expressing all the flexibility in the choice of combinational logic for a sequential circuit. Cerny and Marin [14] demonstrate a close relationship between optimizing a Boolean network with respect to a given Boolean relation, and computing observability don't care sets. The starting network $\mathcal{N}$ must satisfy the relation $\mathcal{R}$. The relation can be viewed as a single node with inputs $\vec{i}, \vec{x}, \vec{o}, \vec{y}$; this node is referred to as the **observability node**. Composing this node with the network as shown in Figure 5.8 yields an **observability network** $\mathcal{N}'$. It is shown in [14, 70] that all the don't cares that can be used to optimize the nodes in $\mathcal{N}$ using the relation $\mathcal{R}$ are equal to the ODC of the same node in the network $\mathcal{N}'$.

In our scenario, the relation $\mathcal{R}(\vec{i}, \vec{x}, \vec{o}, \vec{y})$ is given by equation 5.1 in Section 5.1.2, with $core(\vec{x})$ being the set A_2 as defined in equation 5.2 in Section 5.1.3. As discussed in section 5.1.3, for this choice of core, the combinational logic network associated with the initial design is an implementation for $\mathcal{R}(\vec{i}, \vec{x}, \vec{o}, \vec{y})$. Using $\mathcal{R}$ as the observability node guarantees that, for any other internal node, the ODC derived from the observability network contains the RDC for the original network.

We use BDD's to represent the design, flexibility relation, and core. There is a variable associated with each primary input and each primary output; for each latch there is a present state variable and a next state variable. Let u be a node in the design for which the ODC is to computed. We add a new BDD variable α_u corresponding to

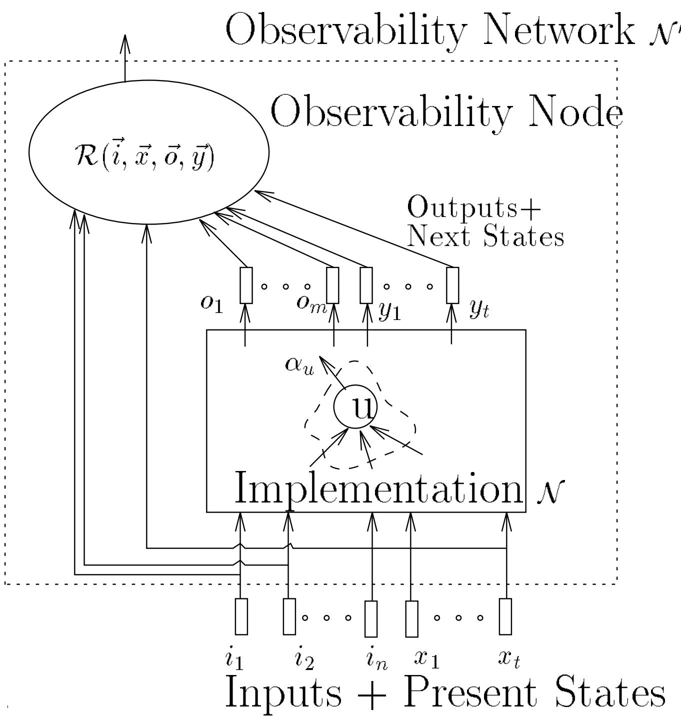

Figure 5.8: ODC's from the Observability Network yield flexibility under Observability Relation for nodes in Implementation Network.

the output of u, and generate BDD's for the next state and output functions in terms of the primary inputs, latch outputs, and α_u. These are composed with the flexibility relation to obtain the BDD for the function computed by the observability network. Let $f(\vec{i}, \vec{x}, \alpha_u)$ be the output of the observability network; then the ODC for node u is given by $f(\vec{i}, \vec{x}, \alpha_u = 1)f(\vec{i}, \vec{x}, \alpha_u = 0) + \bar{f}(\vec{i}, \vec{x}, \alpha_u = 1)\bar{f}(\vec{i}, \vec{x}, \alpha_u = 0)$. This is in terms of primary inputs; we then project this set into the space comprised of the fan-ins of the node (as in [71]). These are used in conjunction with a subset of the satisfiability don't care set to optimize the function at u. The complete procedure to do the multi-level logic optimization is outlined is Figure 5.9. For our experiments we found that computing the BDD for the relation $\mathcal{R}$ required too much memory for our experiments; so instead, we just used the relation $\mathcal{P}(\vec{i}, \vec{o}, \vec{y})$ (described in Section 5.1.2) in place of $\mathcal{R}$, and added the restriction for the states inside the core when we compute the flexibility of each node (see the step of computing $\mathcal{D}(\vec{i}, \vec{x})$ in Figure 5.9). The image projection step in the algorithm in the figure can de done using a variety of algorithms (see [22, 80], for example); we use the algorithm presented in [80].

```
procedure safe-replacement (input: network in terms of (i⃗, x⃗, o⃗, y⃗)) {
    /* let y_j denote the BDD for the j-th next state variable */
    /* let o_k denote the BDD for the k-th output variable */
    for (each node p in the network)
        Compute a BDD S_p(i⃗, x⃗) representing node p in terms of (i⃗, x⃗)
    T(i⃗, x⃗, y⃗) ← ∏_{j=1}^{t} (y_j ≡ S_{y_j}(i⃗, x⃗))
    C(x⃗) ← onion-ring(T(i⃗, x⃗, y⃗), 2)
    C(y⃗) ← C(x⃗)_(x⃗←y⃗)
    P(i⃗, o⃗, y⃗) ← ∃x⃗ : [C(y⃗) · T(i⃗, x⃗, y⃗) · ∏_{k=1}^{n} (o_k ≡ S_{o_k}(i⃗, x⃗))]
    for (each node p in the network) {
        Add BDD variable p to the BDD variable list
        for (each node r in the network) {
            if (r is not in the transitive fanout of p)
                S_r(i⃗, x⃗, p) ← S_r(i⃗, x⃗)
            else
                recompute S_r(i⃗, x⃗, p) representing node r in terms of (i⃗, x⃗, p)
        }
        F(i⃗, x⃗, p) = ∃o⃗, y⃗ : [P(i⃗, o⃗, y⃗) · ∏_{j=1}^{t} (y_j ≡ S_{y_j}(i⃗, x⃗, p)) · ∏_{k=1}^{n} (o_k ≡ S_{o_k}(i⃗, x⃗, p))]
        /* F(i⃗, x⃗, p) represents the observability node */
        G(i⃗, x⃗) = F(i⃗, x⃗, p)_{p=0} · F(i⃗, x⃗, p)_{p=1} + ¬F(i⃗, x⃗, p)_{p=0} · ¬F(i⃗, x⃗, p)_{p=1}
        /* G(i⃗, x⃗) represents input minterms where F(i⃗, x⃗, p) does not depend on p */
        D(i⃗, x⃗) = G(i⃗, x⃗) · ¬C(x⃗)
        /* D(i⃗, x⃗) represents inputs minterms which are don't care for node p */
        Image project ¬D(i⃗, x⃗) to the local input of p
        Use the flexibility to simplify the local functionality of p
        Remove BDD variable p from the BDD variable list
        if (functionality of node p changed)
            for (each fanout node r of node p)
                recompute S_r(i⃗, x⃗) representing node r in terms of (i⃗, x⃗).
    }
}
```

Figure 5.9: Procedure to optimize a multi-level network maintaining safe replacement.

We are computing all inputs under which the outputs are independent of the function computed at the node. Thus we will detect cases where an internal line (which is an input to a node) can be set to a constant while maintaining compatibility with relation $\mathcal{R}$. This means that we automatically remove redundant faults (which satisfy $\mathcal{R}$) from the circuit. Note that we are able to detect these redundancies because we are computing the flexibility of each node just before minimizing it; if we had obtained compatible flexibility (as in [70]) for all nodes before minimizing them simultaneously we would not be able to claim this. The price we pay is in time: we need to simplify nodes on an individual basis, and if the node is simplified, potentially all the BDDs for functions that the node fans out to must be recomputed.

5.1.5 Experiments and Analysis

We have implemented the above method for sequential resynthesis in the SIS sequential synthesis system [72]. We have performed experiments on the same set of ISCAS89 benchmark examples that we chose for the experiments in Table 5.1. We report our experiments in Table 5.2. Before running our algorithm, we executed the SIS commands `sweep; eliminate -1`; this step propagates constants and also collapses those nodes into their fanouts which do reduce the total literal count by this collapsing (for example, nodes which have a single fanout increase the literal count unless they are collapsed into their fanouts). Table 5.2 reports the reduction in the total number of literals of the network. In order to compare the optimizations due to safe replaceability with the known techniques for combinational synthesis, we report the optimizations (in reduction in number of literals) due to satisfiability don't cares (SDC), observability don't cares (ODC) and replaceability don't cares (RDC) separately. We minimized each internal nodes three times: using SDC, using (SDC + ODC), using (SDC + ODC + RDC). The literal reductions under the ODC column are reductions in addition due to both SDC and ODC; those under the RDC column are reductions due to SDC + ODC + RDC. Thus, we get an idea of the additional effectiveness of ODC over SDC and of RDC over (SDC + ODC). We could have reported the amount of savings just exclusively due to the replaceability don't cares, but this would not have been fair since SDC and ODC are existing techniques and they take less CPU time also; we conducted some experiments which showed us that the literal savings due to only RDC captured almost all of the savings due to SDC but since SDC is a cheaper computation step

Circuit	Start size	Final size			Time (in seconds)		
		SDC	ODC	RDC	SDC	ODC	RDC
s27	12	12	12	12	0.02	0.03	0.05
s298	150	130	130	130	0.30	0.56	1.91
s344	156	152	152	152	0.43	0.96	689.75
s349	160	155	154	154	0.43	2.87	114.19
s382	176	164	164	164	0.40	0.74	8.27
s386	204	197	187	173	0.40	0.66	1.16
s400	184	166	162	162	0.43	0.77	8.52
s444	184	167	163	163	0.44	0.77	9.09
s510	280	279	279	279	1.58	2.04	4.65
s526	283	242	240	240	0.88	1.44	6.91
s641	199	timeout after 10000 sec					
s713	204	timeout after 10000 sec					
s820	504	379	361	361	3.48	4.53	14.70
s832	521	389	364	364	3.77	5.00	14.75
s953	489	485	484	484	2.80	6.23	28.10
s1196	618	608	600	600	8.51	28.23	504.04
s1238	690	668	625	625	8.67	33.04	574.69
s1488	813	759	755	755	9.91	10.58	42.11
s1494	819	760	754	742	9.57	10.64	38.32

Table 5.2: Experimental results for safe replacements

it makes sense to preprocess the circuit with SDC reductions before applying our algorithm.

Disappointingly, Table 5.2 illustrates that for most of the circuits all the literal reduction can be obtained just by using SDC and ODC. One hypothesis is that since we are minimizing the network one node at a time, very small nodes are unlikely to yield much optimizations. The node sizes of the benchmark circuits were very small; we executed the SIS commands `sweep; eliminate 10` to partially collapse the network. For some benchmarks, a totally collapsed network had a smaller literal count than a partially collapsed one; for these circuits, we started with the totally collapsed network (using command `sweep; collapse`). The experiments with this preprocessing are reported in Table 5.3. The results, which we analyze next, are a little more promising now, at least for some circuits.

From Table 5.3 we observe that for some examples, ODC gives no literal reductions beyond SDC alone but RDC is able to obtain additional, though modest, reductions. These reductions are of the order of 5% on these circuits. On other circuits RDC gives no

Circuit	Preprocessing command	Start size	Final size			Time (in seconds)		
			SDC	ODC	RDC	SDC	ODC	RDC
s27	`sweep; eliminate 10`	12	12	12	12	0.02	0.02	0.04
s298	`sweep; eliminate 10`	156	137	137	137	0.26	0.49	1.82
s344	`sweep; eliminate 10`	168	160	156	156	0.45	0.84	807.36
s349	`sweep; eliminate 10`	173	160	156	156	0.45	1.85	72.91
s382	`sweep; eliminate 10`	204	168	166	166	0.49	0.80	7.62
s386	`sweep; collapse`	205	153	153	138	0.48	0.61	1.14
s400	`sweep; eliminate 10`	229	173	168	168	0.56	0.90	7.93
s444	`sweep; eliminate 10`	236	172	171	171	0.57	1.03	8.27
s510	`sweep; collapse`	307	256	256	255	1.18	1.38	1.89
s526	`sweep; collapse`	323	244	244	233	3.22	3.40	6.24
s641	`sweep; eliminate 10`	234	timeout after 10000 sec					
s713	`sweep; eliminate 10`	285	timeout after 10000 sec					
s820	`sweep; collapse`	468	361	361	353	1.67	2.41	7.85
s832	`sweep; collapse`	470	362	362	354	1.71	2.48	7.59
s953	`sweep; eliminate 10`	700	615	602	597	5.34	11.41	22.92
s1196	`sweep; eliminate 10`	788	693	626	626	16.90	31.25	265.46
s1238	`sweep; eliminate 10`	882	736	636	636	31.28	61.54	338.30
s1488	`sweep; collapse`	886	576	576	576	13.71	18.27	30.43
s1494	`sweep; collapse`	896	544	543	542	13.79	18.49	30.67

Table 5.3: Experimental results for collapsed nodes in the starting netlist.

additional improvements over SDC and ODC. The CPU time taken by RDC flexibility are, on average, one order of magintude larger than the times taken to utilize the SDC and ODC flexibility. Even though we do get some optimizations beyond SDC and ODC, the additional optimizations are not very large. We observed that flexibility for resynthesis expressed in the Boolean relation $\mathcal{P}(\vec{i}, \vec{o}, \vec{y})$ was too constrained for any significant optimization; in the next chapter, where we use the same optimization techniques but with a greater degree of flexibility we shall observe a more significant success with our experiments.

ISCAS89 benchmarks may not constitute a good source of examples to judge the potential of our approach. This is because for many of the circuits, collapsing the logic to two levels before applying our minimization yields a smaller circuit, thus negating the whole basis of doing multi-level logic minimizations. Furthermore, the average node size is too small to hope for any significant reduction over that given by the SDC. Hence the merit of our approach may be better judged on the basis of real multi-level designs.

Memory explosion is a common problem with BDD's. For our experiments we noted that the ability to finish the examples (and the time taken) was most influenced by the number of input wires to the design. The number of latches and the number of output wires are also a factor, though to a lesser degree. Since large designs routinely arise as set of interacting components, it is natural to decompose the design into smaller components and synthesize them independently. Intuitively, the components being smaller, will be less complex, and hence easier to synthesize. We expect that our methods will be most useful in resynthesizing an arbitrary portion of a design without worrying about any interaction with its environment.

In the next chapter we will introduce a more flexible notion for design replacement which will allow flexibility in the behavior of the replaced design for a small number of clock cycles after power-up. There, we will observe much more impressive optimizations using techniques similar to those described in this section, but exploiting a greater degree of flexibility.

5.2 The Verification Problem

While logic synthesis is an important step in the design flow of digital circuits, design equivalence checking is an equally (and often, more) important step. Design equivalence checking verifies that one design is a valid replacement of another. If we use the safe replacement condition postulated in this chapter as a valid notion of replacement for sequential circuits, then an important question is how does one verify if one design is a safe replacement of another. In this section first we discuss the complexity of the verification problem. Then we will present a heuristic algorithm to verify safe replacements.

5.2.1 Complexity of checking safe replaceability

In this section we show that checking if a design is a safe replacement of another is a PSPACE-complete problem. First, we will need the following lemma to prove the theorem following it.

Lemma 5.2 *Suppose there exists a state d_0 in design D such that for any state $d \in Q_D$, there exists an input sequence ρ such that $\delta_D(d_0, \rho) = d$. If for any input sequence $\pi \in I^*$, there exists some state c in design C such that the output behavior $\lambda_D(d_0, \pi) = \lambda_C(c, \pi)$, then $D \preceq C$.*

Proof. Consider any arbitrary state $d \in Q_D$ and an arbitrary input sequence σ. We will show that there is a state $c \in Q_C$ such that $\lambda_D(d, \sigma) = \lambda_C(c, \sigma)$. From the presupposition, we know that there exists an input sequence ρ such that $\delta_D(d_0, \rho) = d$. Now, consider the input sequence $\rho \cdot \sigma$. We know that there exists a state $c' \in Q_C$ such that $\lambda_D(d_0, \rho \cdot \sigma) = \lambda_C(c', \rho \cdot \sigma)$. Thus it is clear that $\lambda_D(d, \sigma) = \lambda_D(\delta_D(d_0, \rho), \sigma) = \lambda_C(\delta_C(c', \rho), \sigma)$. This simply means that the desired state c can be chosen to be the same as $\delta_C(c', \rho)$. $\Box$

Theorem 5.3 *Given two designs $D = (Q_D, I, O, \lambda_D, \delta_D)$ and $C = (Q_C, I, O, \lambda_C, \delta_C)$, the problem of determining if $D \preceq C$ (the safe replaceability problem) is PSPACE-complete.*

Proof. First we show membership in PSPACE. Consider the complementary problem: is $D \not\preceq C$?. If $D \not\preceq C$, then there must be a witness, a string π and a state $d \in D$ such that for any state $c \in C$: $\lambda_C(c, \pi) \neq \lambda_D(d, \pi)$. We can verify the validity of this witness in polynomial space by simulating the witness on the string π starting from state d and from all states in C, at each step (after each input symbol in π) keeping track of all the reachable states such the output string is the same as seen from d. We have to store at most $(1 + Q_C)$ states at each step. Since (π, d) is a witness, all states in C must eventually die out proving the validity of the witness. Thus the complementary problem is in NPSPACE. By Savitch's theorem [69, 30], the complementary problem is in PSPACE (i.e. a deterministic Turing machine can recognize it). Thus, safe replaceability is also in PSPACE.

The proof of PSPACE-hardness is by transformation from the FINITE STATE AUTOMATA INTERSECTION problem, which is PSPACE-complete [42, 30].

FINITE STATE AUTOMATA INTERSECTION

INSTANCE: Set $D_1, D_2, \ldots, D_n$ of deterministic finite automata or DFAs (see [35] for the definition of a DFA) having the same input alphabet Σ.

QUESTION: Is the intersection of the languages of the DFA empty, i.e. is $\bigcap_{i=1}^{n} \mathcal{L}(D_i) = \emptyset$?

Let DFA $D_i = (Q_i, \Sigma, \delta_i, r_i, F_i)$ where Q_i is the set of states, Σ is the alphabet, δ_i is the deterministic transition function, r_i is the initial state and F_i is the set of final states. We will construct two designs $D = (Q_D, I, O, \lambda_D, \delta_D)$ and $C = (Q_C, I, O, \lambda_C, \delta_C)$, polynomially-large in the size of $\{D_1, \ldots, D_n\}$, such that $D \preceq C$ if and only if $\bigcap_{i=1}^{n} \mathcal{L}(D_i) = \emptyset$. Let $\Sigma' = \Sigma \cup \{a, b\}$, where $a, b \notin \Sigma$. Without loss of generality, assume that the elements in Σ' can be encoded by m binary values, i.e. Σ' has exactly 2^m elements; there m binary values denote the input wires I. We will use the symbol $s \in \Sigma'$ to denote the encoding of that

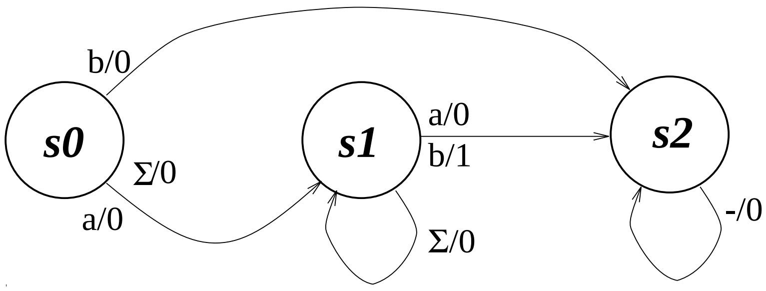

Figure 5.10: Design D

symbol also. O consists of a single output wire; so there are two possible output values, 0 and 1.

Design D is a 3-state design with the next-state and output function as shown in Figure 5.10. Now we describe the state space, output function and next-state function of design C. We introduce $(n+1)$ new states: $t, v_1, v_2, \ldots, v_n$. $Q_C = \{t\} \cup \bigcup_{i=1}^{n} \{v_i\} \cup \bigcup_{i=1}^{n} Q_i$. Now we define the output function λ_C. For any state $q \in Q_C$ and any $c \in (\{a\} \cup \Sigma)$, $\lambda_C(q, c) = 0$. For any state q which is a non-final state of a DFA, i.e. if $q \in \bigcup_{i=1}^{n}(Q_i \setminus F_i)$, $\lambda_C(q, b) = 1$; if q is any other state, i.e. $q \in \{t\} \cup \bigcup_{i=1}^{n}(F_i \cup \{v_i\})$, then $\lambda_C(q, b) = 0$. Now we define the next-state function δ. For any $q \in Q_C$, $\delta_C(q, b) = t$. For any $i \in \{1, \cdots, n\}$ and for any $c \in (\{a\} \cup \Sigma)$, $\delta_C(v_i, c) = r_i$. If $q \in Q_i$, $i \in \{1, \cdots, n\}$ and $c \in \Sigma$, then $\delta_C(q, c) = \delta_i(q, c)$. For any $q \in \bigcup_{i=1}^{n} Q_i$, $\delta_C(q, a) = t$. For any $c \in (\{a\} \cup \Sigma)$, $\delta_C(t, c) = t$.

Now suppose that $D \preceq C$. Suppose we are given an arbitrary string $x \in \Sigma^*$, which is of length p. Consider the power-up state $s0$ in design D: on input sequence $a \cdot x \cdot b$, this state outputs the length-$(p+2)$ output sequence $0^{p+1} \cdot 1$. Since, $D \preceq C$, there must be a power-up state s in design C which has the same input-output behavior. It should be clear that in order to observe an output of 1 on any input sequence starting with a, the design C must power-up in a state v_i, $1 \leq i \leq n$; thus, let $s = v_i$. Since the input sequence $a \cdot x \cdot b$ outputs $0^{p+1} \cdot 1$, $\delta_C(v_i, a \cdot x)$ is a non-final state (i.e. belongs to $Q_i \setminus F_i$). Thus, the string $x \notin \mathcal{L}(D_i)$. Since the choice of x was arbitrary, $\bigcap \mathcal{L}(D_i) = \emptyset$.

Next, suppose that $\bigcap \mathcal{L}(D_i) = \emptyset$. We will prove that $D \preceq C$ by showing that every observable input-output behavior from state $s0$ of D is observable from some state of C (using the result of Lemma 5.2). For any input sequence x of length p, the output sequence from state $s0$ is either 0^p or $0 \cdot 0^q \cdot 1 \cdot 0^r$, where $p = q + r + 2$. If the output sequence is 0^p, this input-output behavior can be observed from state t in design C. If the output sequence is $0 \cdot 0^q \cdot 1 \cdot 0^r$, then the input sequence x must have been $c \cdot y \cdot b \cdot z$, where $c \in \{a\} \cup \Sigma$, length-q sequence $y \in \Sigma^*$ and length-r sequence $x \in (\Sigma \cup \{a, b\})^*$. Since $\bigcap \mathcal{L}(D_i) = \emptyset$, there exists an $i \in \{1, \ldots, n\}$ such that $y \notin \mathcal{L}(D_i)$. Now, consider the behavior of state $v_i \in Q_C$ on the input sequence $c \cdot y \cdot b \cdot z$. Since, $y \notin \mathcal{L}(D_i)$, $\delta_C(v_i, c \cdot y)$ is a non-final state of the DFA D_i. Hence, it is clear that state v_i on input sequence x outputs $0 \cdot 0^q \cdot 1 \cdot 0^r$. Thus, $D \preceq C$.

Thus, we have shown that $\bigcap_{i=1}^{n} \mathcal{L}(D_i) = \emptyset$ if and only if $D \preceq C$. □

5.2.2 A Heuristic Algorithm

In the previous section we proved that verifying safe replaceability is PSPACE-hard. We know from Remark 4.2 (in Chapter 4) that safe replaceability can be reduced to the language containment problem of NDFAs, which is also PSPACE-complete (since language equivalence is PSPACE-hard [41, 30]). Thus we can verify safe replaceability by the standard exponential-time algorithm for checking NDFA containment ($\mathcal{L}(D_1) \subseteq \mathcal{L}(D_2)$?) which determinizes D_2, then complements it and then looks for an accepting string in the product of this automaton with D_1. Since the transformation of a design to an NFA yields an automaton with all states as initial states, we expect the determinization to be exponential for most interesting designs. This motivates the need for an algorithm which is expected to be fast for most interesting design examples.

Earlier we had argued verification of safe replaceability is an important problem for real design situations. So, in this section we present a heuristic algorithm to verify safe replaceability which is exponential in the worst case, but has special cases, described towards the end of this section, which finish faster for many cases. In case the replacement design is *not* a safe replacement, our algorithm will return a power-up state of the replacement design and an input sequence (a counterexample) which will be a witness to the absence of safe replaceability. We will also show an example where the smallest such counterexample is exponential in the size of the two designs, showing that our algorithm is exponential in the worst case.

5.2.2.1 Algorithm - Finding a Discriminating Sequence

The following procedure decides whether (new) design D_1 is a safe replacement for (original) design D_0, i.e, if $D_1 \preceq D_0$. If D_1 is not a safe replacement for D_0 then this algorithm finds a state s_1 of D_1 and an input sequence π that distinguishes s_1 from every state of D_0.

We construct a multiple rooted, acyclic directed graph whose nodes are labeled by pairs of the form (s, A) where s is a state of D_1 and A is a subset of states of D_0. Nodes are either marked or unmarked; the markings may be **FAIL**, **JUMP**(N) or **SUCCEED**, where N is another node. Edges of the graph are labeled by a single input $a \in I$. We presume that the equivalent state pairs of D_0 and D_1 have already been computed, see [59] (this is a procedure which is polynomial in the number of states of the two designs).

The roots of the digraph are pairs of the form (s_0, Q_{D_0}) where s_0 is a state of D_1, Q_{D_0} is the set of all states of D_0 and s_0 is not equivalent to any state in Q_{D_0}. If the set of roots is empty, then clearly $D_1 \preceq D_0$.

loop until some leaf is marked **FAIL** or all leaves are marked **JUMP** or **SUCCEED**:

Choose a node N labeled (s_1, A) of the existing tree that has no **JUMP** or **SUCCEED** mark and choose an input a such that no edge out of N has the label a.

Let $s'_1 = \delta_{D_1}(s_1, a)$ and $A' = \{s' |$ for some s in A, $s' = \delta_{D_0}(s, a)$ and $\lambda_{D_1}(s_1, a) = \lambda_{D_0}(s, a)\}$. If a node labeled (s'_1, A') already exists, say node N', create an edge labeled a from N to N', and goto the beginning of the loop. Else, create a new edge out of N labeled a and pointing to a new node N' labeled (s'_1, A').

Mark the new node N' as follows:

1. If A' is empty, mark the new node N **FAIL** and exit the program.

2. If s'_1 is state equivalent to any state in A', mark N **SUCCEED**, and go to the beginning of the loop.

3. If there exists a node N'' labeled (s'_1, A'') such that $A'' \subset A'$, mark N' as **JUMP**(N''), and go to the beginning to the loop.

4. For each node N'' labeled (s'_1, A'') such that $A'' \supset A'$, mark N'' as **JUMP**(N').

End **loop**

Proof. Termination: Each node must have a distinct label and there can only be finitely many labels. Furthermore each node has an upper bound on the number of edges emanating from it – the number of primary input combinations. Therefore the program must terminate.

If the program terminates because a **FAIL** node is created, any path from a root

(s, Q_{D_0}) to the **FAIL** node gives a sequence that distinguishes s from any state of Q_{D_0}. If there is no **FAIL** node then all leaf nodes are marked **SUCCEED** or **JUMP**.

Claim: If all leaf nodes are marked **SUCCEED** or **JUMP** then no input sequence will distinguish a state of D_1 from all the states of D_0.

Observation: We first observe that there cannot be a loop of **JUMP** nodes— N_1 -**JUMP**$(N_2) \rightarrow N_2$-**JUMP**$(N_3) \rightarrow \cdots \rightarrow N_k$ -**JUMP**$(N_1) \rightarrow N_1$ because each **JUMP** reduces the cardinality of the second coordinate of the label of a node.

The proof of the claim is by contradiction. Suppose there were a state s_1 of D_1 and an input sequence $\pi = a_1 \cdot a_2 \cdots a_k$ that distinguishes s_1 from all states of D_0. Let $N_0 = (s_1, Q_{D_0})$. Notice that node N_0 cannot be marked **SUCCEED**, because then s_1 would be equivalent to a state in D_0, a contradiction. Construct the sequence of nodes $N_0, N_1, \ldots, N_k$, none of which is marked **SUCCEED**, by applying the following procedure recursively. If node N_i is unmarked then node N_{i+1} is the node reached by traversing the edge labeled a_{k+1} from N_i. Otherwise, N_i is marked **JUMP**(N); jump to N and keep jumping nodes until a node N' is reached which is not marked **JUMP** (see the observation above). This node cannot be marked **SUCCEED**. [This is because nodes marked **SUCCEED** have their left hand component equivalent to some state in their right hand component. But then N_i would have the same property and would have been marked **SUCCEED** rather than **JUMP** which is a contradiction.] Now, N_{i+1} is the node reached by traversing the edge labeled a_{k+1} from N'.

Node N_{i+1} labeled (s_{i+1}, A_{i+1}) cannot be marked **SUCCEED**, because s_{i+1} cannot be equivalent to any state in A_{i+1}. [If it was so, by backtracking the edges traversed, we can find a state in Q_{D_0} that cannot be distinguished from s_1.] Since there are no nodes marked **FAIL**, the last node N_k labeled (s_k, A_k) must have a non-empty set A_k. But then by choosing a state in A_k, and backtracking the edges traversed in constructing the sequence of nodes, one can find an element of Q_{D_0} that is not distinguished from s_1 by π. $\square$

We illustrate the working of the above algorithm by a simple example. Consider the two designs D_0 and D_1 shown in Figure 5.11. We would like to verify if it is true that $D_1 \preceq D_0$. So we construct a digraph as shown in Figure 5.12 (for illustration we show all the edges in the graph even though we can terminate as soon as we reach a FAIL node). D_1 is not a safe replacement for D_0 on input sequence $00 \cdot 00$ it produces output sequence $0 \cdot 0$ from state $t2$; no state of D_0 exhibits this input-output behavior. The verification digraph in Figure 5.12 contains two states marked FAIL; the paths to these states provides

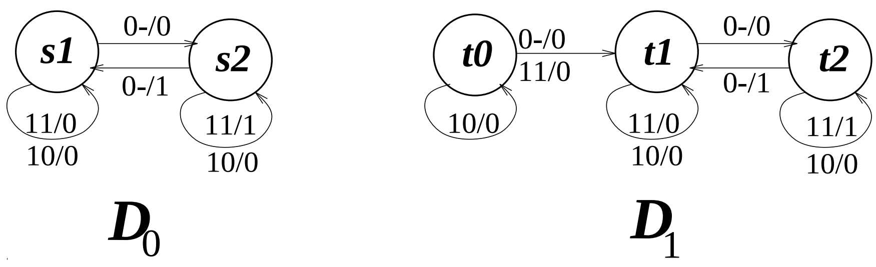

Figure 5.11: Designs D_0 and D_1

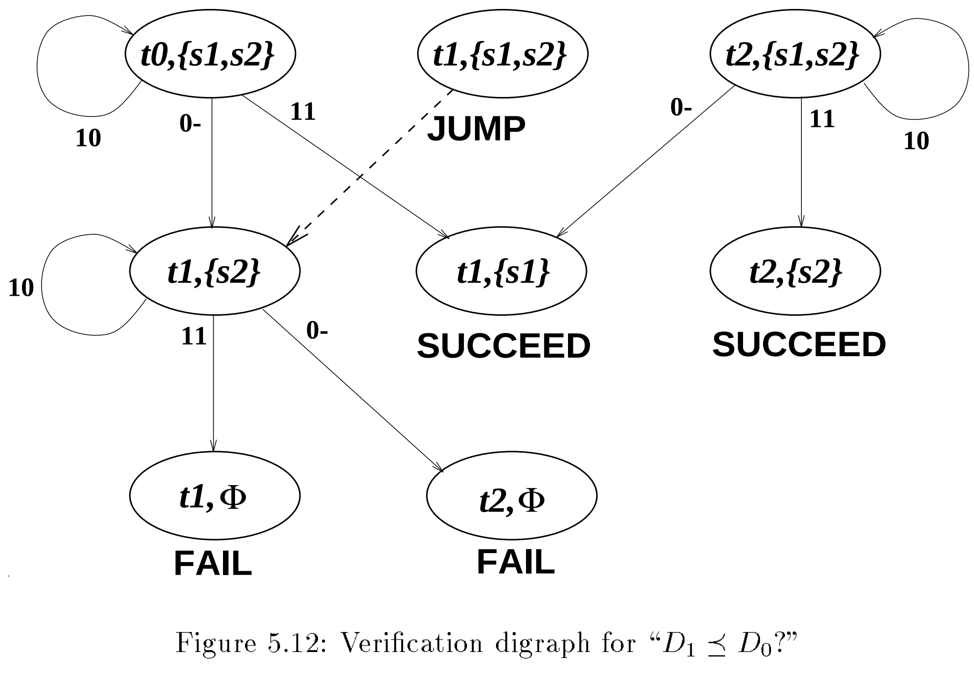

Figure 5.12: Verification digraph for "$D_1 \preceq D_0$?"

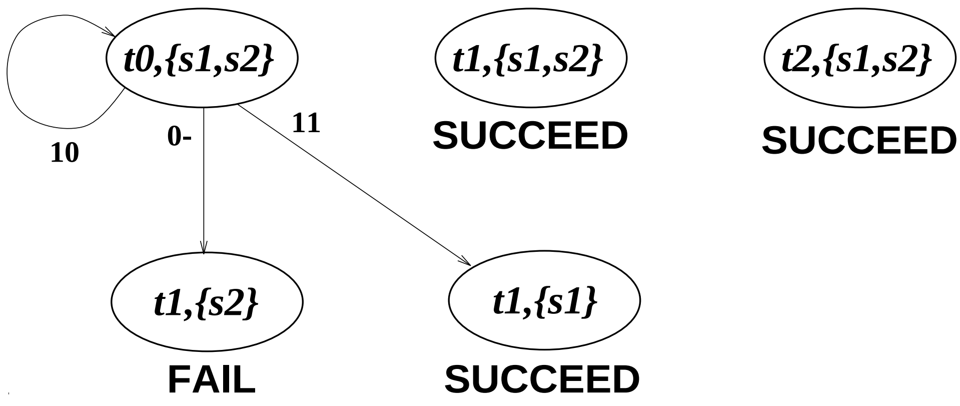

Figure 5.13: The optimized verification digraph

a counter-example to the hypothesis that $D_1 \preceq D_0$— the input sequence $00 \cdot 00$ leads to a FAIL node from state $(t0, \{s1, s2\})$.

5.2.2.2 Optimizations to the algorithm

Before we execute the verification algorithm we just described, we could check to see if each closed SCC of D_1 has a state equivalent to some state of D_0. If not, then by Theorem 4.5, $D_1 \not\preceq D_0$. Otherwise, if there is no state outside the closed SCC's of D_1, then $D_1 \preceq D_0$ and we are done. If there is state outside the closed SCC's of D_1, we use the method outlined above knowing that each root (s_0, Q_{D_0}) of the digraph is such that s_0 is outside all closed SCC's of D_1. Thus, if the number of states outside closed SCC's is small compared to the number of states which lie in closed SCC's, we can probably expect our algorithm to be efficient. For example, if we use the optimization, we get the smaller verification digraph, shown in Figure 5.13 instead of the one in Figure 5.12.

Also, note that the verification algorithm would be correct if we did not mark any states **JUMP** (remove substeps 3 and 4 in the marking step). Marking states as **JUMP** is just a way to prune the search space, and make the algorithm more efficient.

5.2.2.3 Complexity of the algorithm

Suppose we are given designs C and D where we want to check if "$D \preceq C$?" such that the number of states in C and D are m and n respectively. Since each node represents (q, Q) where $q \in Q_D$ and $Q \subseteq Q_C$, we may end up exploring $n \cdot 2^m$ possible nodes before the algorithm terminates. In this section we prove that this exponential bound is possible,

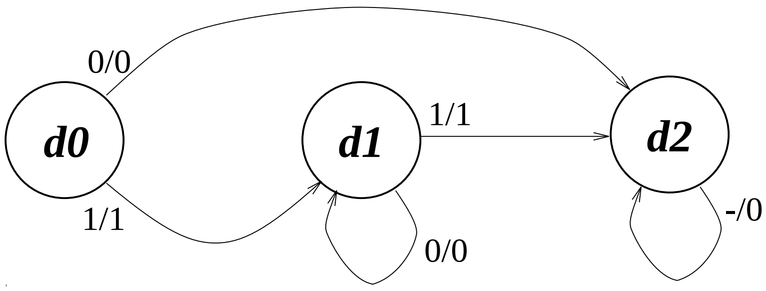

Figure 5.14: Design D

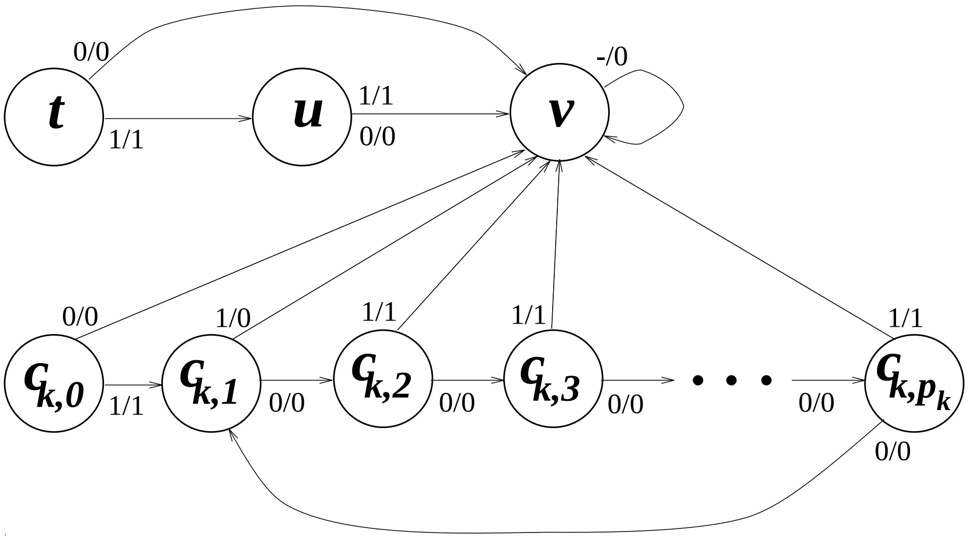

Figure 5.15: Sub-design C_k and the three special states

in the worst case. Given any arbitrary n, we will construct two designs C and D such that C has less that $15n^3$ states and D has 3 states and our verification algorithm will explore greater than 2^n nodes before it can decide that $D \not\preceq C$. This proves that in the worst case, the algorithm may take time exponential in the number of states of C.

Let D be the three-state design shown in Figure 5.14. Let the first n prime numbers be $\{p_1, p_2, \ldots, p_n\} = \{2, 3, 5, \ldots, p_n\}$. The design C consists of n subdesigns $\{C_1, C_2, \ldots, C_n\}$ and three additional special states $\{t, u, v\}$. The transitions from the states in the subdesign C_k as well as from the three special states are also shown in Figure 5.15. Informally, each sub-design C_k counts the number of consecutive 0's starting from state $c_{k,1}$; if it sees a 1 after it has seen $m \cdot p_k$ 0's (for some non-zero integer m) it outputs a 0 (from state $c_{k,1}$), or else if the 1 follows a non-multiple of p_k the sub-design outputs a 1.

The total number of states in C are $N = 3 + \Sigma_{r=1}^{n}(p_r + 1)$. From number theory we know that[3] $p_r < 12r \log r$. From this, we have $N < 3 + 12n + 6n \log n + 6n^2 \log n < 15n^3$. Now, it can be seen that $D \not\preceq C$ and the only witnesses are input strings of the form $1 \cdot 0^q \cdot 1 \cdot y$ and start state d_0 such that the output string is $1 \cdot 0^q \cdot 1 \cdot 1^r$ and the state sequence in design D is $d_0 \cdot d_1^{q+1} \cdot d_2 r + 1$ (where $r = |y|$). Since $\{p_1, p_2, \ldots, p_n\}$ are all primes, the smallest such witness input string is $1 \cdot 0^q \cdot 1$, where $q = \prod_{i=1}^{n} p_i$. Since this is the shortest witness, the path to the FAIL node will have no loops and will be of length $(2 + q)$ which is clearly greater than 2^n.

5.2.2.4 Known initializing sequence set

First, based on Theorem 4.6, we check if each closed SCC of D_1 has a state equivalent to some state of D_0. If this check fails, we know that D_1 is not a safe replacement for D_0 under the initializing sequence set Π. Otherwise, we compute sets $Q_0 = \{s|$ there exists $s_0 \in Q_{D_0}$ and $\pi \in \Pi$ such that $\delta_{D_0}(s_0, \pi) = s\}$, and $Q_1 = \{s|$ there exists $s_1 \in Q_{D_1}$ and $\pi \in \Pi$ such that $\delta_{D_0}(s_1, \pi) = s\}$. Now, we can use the same digraph-construction method described above, except that the roots of the digraph are pairs of the form (s, Q_0) such that $s \in Q_1$ and s is not equivalent to any state in Q_0. The proof of correctness is similar to the one above, and is omitted for brevity.

[3] See [57, Chapter 8] for a proof, originally due to Tschebyschef, that $p_r < dr \log r$ where d is $\frac{8}{\log 2}$.

Chapter 6

Delay Replacements

We presented the notion of safe replaceability in Chapter 4. We had argued that for the condition for safe replacement (Definition 4.1) is the weakest possible condition which allows a replacement to be made such that no environment can detect the replacement based on the input-output behavior of the design. However, this may be too stringent to achieve much optimization during resynthesis; indeed, we saw in the previous chapter that while safe replacement allowed logic optimizations which were beyond the scope of combinational resynthesis methods, the amount of extra optimizations was small. In any realistic use of a design, we can expect that after a design is powered up, some clock cycles will be allowed to run through the design before the design is used by its environment. We can use this additional flexibility in the initial few clock cycles to justify an notion of replacement which is weaker than "safe replacement" because the environment of the design agrees to let the replacement design stabilize for a few extra clock cycles after power-up. In this chapter, we investigate this notion of "delay replacement".

It is already the case that several effective optimization techniques, such as re-timing (as explained in Chapter 7), do not always result in safe replacements, but cause delay replacements. To understand why, consider the state transition graph again. In every design, there is a subset of states into which the design must eventually fall no matter what sequence of inputs is given to the design. For example, suppose there is a state s_1 to which no state (including itself) transitions under any input. This state represents an *ephemeral* state of the machine which cannot be visited beyond one clock cycle. Given that the machine can power up in any state, s_1 might be the initial state of the machine. Any such 1-cycle ephemeral state is irrelevant to the steady state operation of the design and so, by letting the design just "coast" for one cycle, it can be eliminated. To get to the "core" behavior of a design, delete all such 1-cycle ephemeral states. However, notice that a new ephemeral may appear, that is a state, say s_2, to which only a 1-cycle ephemeral states can transition. States such as s_2 cannot re-appear after two clock cycles, no matter what inputs are given during the two cycles. The n-cycle ephemeral state sets form an "onion ring" structure in the STG [37]. The 1-cycle ephemeral states constitute onion ring 1, 2-cycle ephemeral states constitute onion ring 2, and so on. Since designs are finite state machines, for any design, there is a bound on the number of onion rings.

Elimination of all such ephemeral states for all clock cycles leaves a set of states, called the outer-envelope (OE) [60] or D^{∞} (Section 6.1 of this paper). Many well-crafted designs have ephemeral states due to the fact that binary encoding often leaves some states of the implementation without a corresponding state in the specified design. These states should be ephemeral. Also one-hot encodings of machines with n states results in $2^n - n$ states which may also be ephemeral. In addition, as we shall see in Chapter 7, a forward retiming move across a fanout junction will also create ephemeral states However, all retiming moves preserve the outer-envelope.

Recall that the sequential method employed in the previous chapter uses an "aliasing" technique. A core set of states is identified (all states except the 1-cycle ephemeral states). Then any 1-cycle ephemeral state, say s, is allowed to act like an alias. That is, for any input a, the state s is allowed to alias to any state s_0 which transitions to a state *inside the core* on input a, i.e. on input a state s is allowed to have the same next state and output as its alias s_0. One degree of flexibility comes from the fact that s can alias one state s_0 for one input a_0 and to another state s_1 for another input a_1. This technique does not seem to yield dramatic improvement in the design, however.

The method presented here is more flexible. We use the onion ring structure more exactly so that any ephemeral state, say s in onion ring i, can alias to a state which, for input a, transitions to a state inside an onion ring j where $j > i$ or to the OE. But this replacement is only delay-safe (i.e. we have to assume that the environment of the design does not care about the behavior of the design till the i-th clock cycle after power-up).

The notion presented in this chapter relies on the assumption that any realistic design will be used only after some cycles have elapsed after power-up; these cycles are necessary for the voltages and currents to settle immediately after power-up. This initialization slack, often at least a few thousand clock cycles (usually milliseconds), is known in advance and is part of the design specification. Sequential optimization can use this initialization slack, say N cycles, in the following two ways. First, the design can be partitioned into non-overlapping, manageable sized pieces. Then each piece can be optimized using the N initialization slack cycles. Proposition 6.5 will show that this strategy will produce an entire design that is N-cycle delay-safe replacement. Second, designs can be optimized using pieces that do overlap. Proposition 6.6 will show that in the worst case, the delay needed accumulates for overlapping pieces. So the designer must take care not to exceed a total allotment of N cycles. Of course hybrids of the two approaches can be used. It is important to note that for any of the approaches, only the transient behavior of the design is modified. Specifically, the outer-envelope is unchanged.

To summarize, optimization techniques for delay replacements can achieve significant logic optimizations while allowing the designer to specify the flexibility in the power-up delay, N.

In Section 6.1 we present the notion of "delay replacement". In Section 6.1.1 we then explore some properties of successive delay replacements on various pieces of a large design, and then in Section 6.1.2 we study a related notion of replacement, delay-preserving replacements. In Section 6.1.3 we suggest a synthesis methodology which can use our notion of delay replacements to do iterative design by making successive delay replacements on various sub-pieces of the design. In Section 6.2 we present our multi-level synthesis strategy for making delay replacement, and we illustrate that significant sequential logic optimizations can be obtained. We conclude by giving strategies for verifying that one design is a delay replacement for another.

6.1 Delay Replacements

Definition 6.1 *Given a design D, the n-**cycle delayed design** (denoted by D^n) is the restriction of D to the set of states $\{s | \exists \pi \in I^n, s' \in D : \delta_D(s', \pi) = s\}$, i.e. a state s belongs to D^n iff there exist a power-up state s' in D and an input sequence of length n which drives s' to s. Define D^∞ to be the set of states $\{s | \forall n : \exists \pi \in I^n, s' \in D : \delta_D(s', \pi) = s\}$, i.e. a state s belongs to D^∞ iff for each natural number n there exist a power-up state s' in D and an input sequence of length n which drives s' to s.*

D^n is the set of states into which any state must fall when clocked n times with

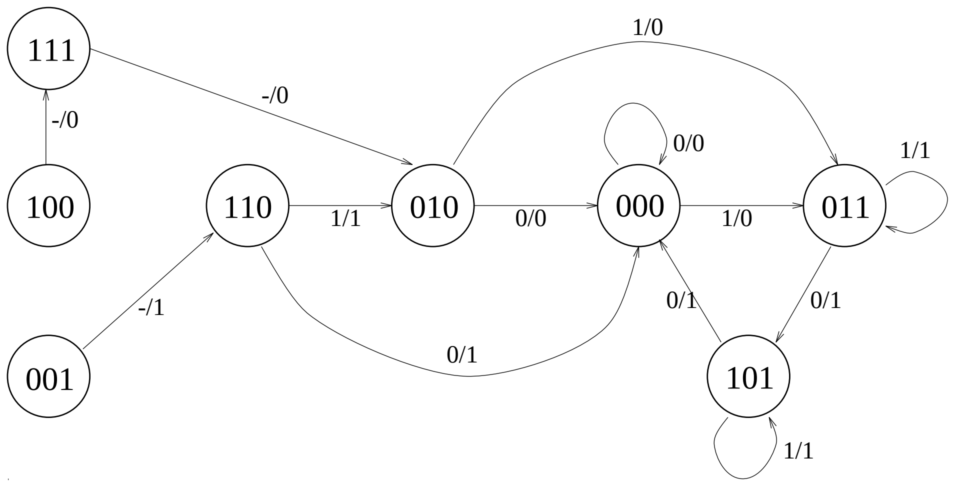

Figure 6.1: Example design R

any sequence of inputs. It is easy to see that if $m > n$, the set of states in D^m is a subset of the states in D^n. The design D^∞ can be obtained by a fixed-point operation starting from D, because $D^\infty = D^n$, where n is the smallest number such that $D^{n-1} = D^n$. Using the terminology in [37], this number n is the number of *onion rings* of the design D. D^∞ is also the same as the outer-envelope (OE) of [60].

We refer to states in D^∞ as the **stable** states of D, and the states in $D \setminus D^\infty$ as the **transient** states of D. After powering up a design, if a sufficiently long sequence of arbitrary inputs is applied to the design, it will enter the stable set.

For example, consider the design shown in Figure 4.2. For this design R, the various n-delayed designs are: $R = \{111, 100, 001, 110, 010, 000, 101, 011\}$, $R^1 = \{111, 110, 010, 000, 101, 011\}$, $R^2 = \{010, 000, 101, 011\}$, $R^3 = R^4 = \cdots = R^\infty = \{000, 101, 011\}$.

We now present our condition for delay replacement. As we noted in the introduction of this chapter, as part of the system specification, an initialization slack of n clock cycles is available. We can use the flexibility afforded by this slack by requiring a design to be an acceptable replacement if it is allowed to clock n extra cycles with arbitrary inputs before it is used:

Definition 6.2 *Given a design D, a new design C is an n-**delay replacement** for D, if $C^n \preceq D$.*

As an example of delay replacement, consider designs D_0 and R in Figures 4.2 and 6.1. It can be seen that $R^1 \preceq D_0$; however, $R \not\preceq D_0$ (the state $100 \in R$ produces the output sequence $0 \cdot 0 \cdot 0$ on input sequence $0 \cdot 1 \cdot 0$; this input-output behavior cannot be seen from any state in D_0).

6.1.1 Properties

The following two properties of delay replacements follow:

Proposition 6.1 *If $C^n \preceq D$, and $m > n$, then $C^m \preceq D$.*

Proof. Clearly each state is C^m is also contained in C^n, and thus $C^m \preceq C^n$. The result follows from the transitivity of $\preceq$. □

Proposition 6.2 *If $C^n \preceq D$ and $B^m \preceq C$, then $B^{m+n} \preceq D$.*

Proof. Consider any state $b \in B^{m+n}$ and an input sequence π. Now, there exist a state $b' \in B^m$ and an input sequence π_0 of length n such that $\delta_{B^m}(b', \pi_0) = b$. Since $B^m \preceq C$, there exists a state $c' \in C$ such that $\lambda_C(c', \pi_0 \cdot \pi) = \lambda_{B^m}(b', \pi_0 \cdot \pi)$. Let $\delta_C(c', \pi_0) = c$. Thus $c \in C^n$ and $\lambda_{C^n}(c, \pi) = \lambda_{B^{m+n}}(b, \pi)$. Since $C^n \preceq D$, there exists a state $d \in D$ such that $\lambda_D(d, \pi) = \lambda_{C^n}(c, \pi) = \lambda_{B^{m+n}}(b, \pi)$. □

Thus, for example, a 2-delay replacement followed by a 3-delay replacement on the same design results in a n-delay replacement for any $n \geq 5$.

Another property of a design that is an n-delay replacement of the original design, is that if an input sequence initializes the original design, then if we wait for n-clock cycles after power-up, the same sequence initializes the replaced design too:

Theorem 6.3 *If $D^n \preceq C$ and ρ is an initializing sequence for C then $\pi \cdot \rho$ is also an initializing sequence for D where π is any arbitrary input sequence of length at least n. Additionally, for any states $s_0 \in Q_C$ and $s_1 \in Q_D$, $\delta_C(s_0, \rho) \sim \delta_D(s_1, \pi \cdot \rho)$.*

Proof. Consider any arbitrary input sequence π of length at least n. From the definition of delayed designs (Definition 6.1), $\delta_D(s_1, \pi)$ belongs to D^n. The proof follows from the result that safe replaceability preserves all initializing sequences (Theorem 4.2). □

Compositionality of Delay Replacements

For the optimization of any design, we can select arbitrary sub-pieces of the design and perform delay replacements on these. In this subsection, we examine the effect of making a delay replacement on the larger design. We will show that making n-delay replacements on non-overlapping sub-pieces of a design will produce an entire design which is n-delay safe. On the other hand, if two consecutive n-delay optimizations are made on sub-pieces which are overlapping, we get an entire design which is $2n$-delay safe (notice that, as a special case of this, if the two sub-pieces are identical, we already know from Proposition 6.2 that the entire design in $2n$-delay safe).

Informally, two given hardware designs can be "wired" together by driving some of the inputs of one by outputs of the other and vice versa. The remaining inputs and outputs will be primary inputs and primary outputs of the composed design. Given designs A and B, we will use $A \otimes B$ to denote their composition. See Chapter 2 for a more precise definition of design composition.

Proposition 6.4 *If $Q^m \preceq R$ and $C^n \preceq D$, then $(Q \otimes C)^p \preceq (R \otimes D)$, where $p = \max(m, n)$.*

Proof. Consider any state (q, c) in $(Q \otimes C)^p$. Thus there exists a sequence of length p which drives some state in Q to q, and so $q \in Q^p$. Similarly for C and c. Thus, (q, c) is also in $(Q^p \otimes C^p)$. This means that $(Q \otimes C)^p \preceq (Q^p \otimes C^p)$. Each state in Q^p is in Q^m and each state in C^p is in C^n; thus, $(Q^p \otimes C^p) \preceq (Q^m \otimes C^n)$. From Proposition 4.1 (in Chapter 4), $(Q^m \otimes C^n) \preceq (R \otimes C^n)$, and similarly $(R \otimes C^n) \preceq (R \otimes D)$. Thus, the required result by the transitivity of $\preceq$. $\square$.

Proposition 6.5 *If $Q^m \preceq R$ and $C^n \preceq D$, then $(Q \otimes C \otimes P)^p \preceq (R \otimes D \otimes P)$, where $p = \max(m, n)$.*

Proof. This is corollary of the above result (Proposition 6.4) after we observe that $\otimes$ is associative and that $P = P^0 \preceq P$. $\square$

This means that delay replacements can be made in different parts of the design, and the resulting overall design is as safe as the weakest individual replacement (the replacement with the greatest slack).

However, if consecutive delay replacements are made on overlapping sub-pieces on a large design, the delays add up. This can be seen as a consequence of Propositions 6.2 and 6.4.

Proposition 6.6 *Let design $D = P \otimes Q$. Let $R^m \preceq P$, and let design $C = R \otimes Q$ also be equal to $S \otimes T$ (another partition of the same design C), and let $U^n \preceq S$. Then, if $B = U \otimes T$, it is true that $B^{m+n} \preceq D$. (See Figure 6.2 for an illustration)*

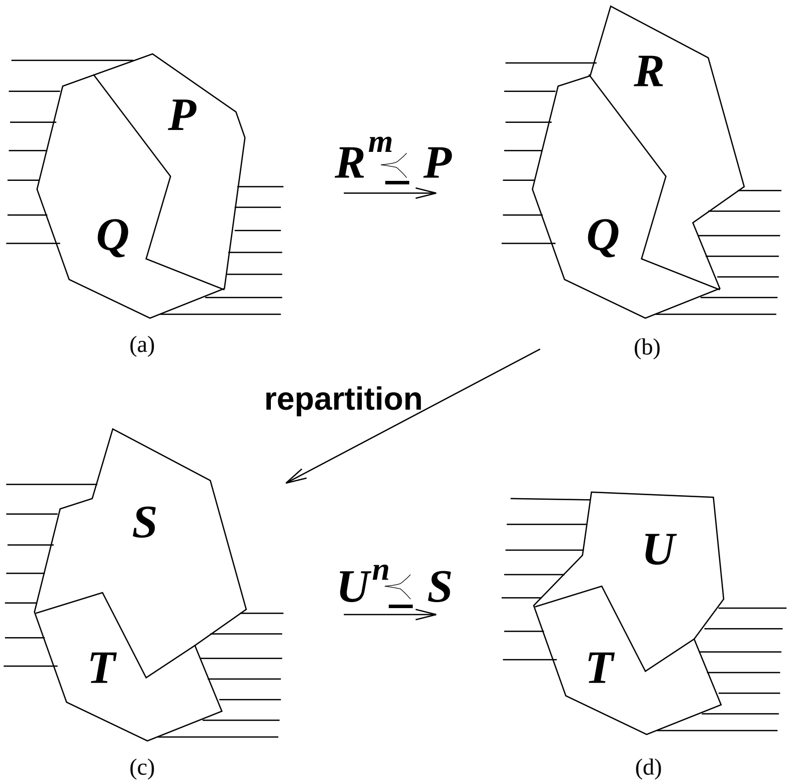

Figure 6.2: Making successive delay replacements on overlapping pieces: $(U \otimes T)^{m+n} \preceq (P \otimes Q)$. (a) denotes design $D = P \otimes Q$; (b) denotes design $C = R \otimes Q$; (c) denotes another partition of the same design $C = S \otimes T$; (d) denotes design $B = U \otimes T$.

Proof. Since $Q^0 \preceq Q$, it follows from Proposition 6.4 that $(R \otimes Q)^m \preceq (P \otimes Q)$, i.e. $C^m \preceq D$. Similarly, $B^n \preceq C$. The result follows from Proposition 6.2. $\Box$

We will use these results to discuss the implications of making delay replacements on various sub-designs of a larger design later in Section 6.1.3.

6.1.2 Delay-Preserving Replacements

This section discusses another interesting replacement notion relating power-up delays to replacements.

We had originally formulated the following conditions for allowing a replacement in the presence of the power-up slack:

Definition 6.3 *Given a design D, a new design C is an n-***delay-preserving** *replacement for D, if $C^n \preceq D^n$.*

As examples of delay-preserving replacements consider designs P and R in Figures 4.2 and 6.1. We already know that R is a 1-delay replacement of P (i.e. $R^1 \preceq P$). However, R is not a 1-delay-preserving replacement of P; instead R is a 2-delay-preserving replacement of P.

We present some properties of delay-preserving replacements. If m is greater than n, then an n-delay-preserving replacement is also an m-delay-preserving replacement:

Proposition 6.7 *If $C^n \preceq D^n$, and $m > n$, then $C^m \preceq D^m$.*

Proof. By contradiction. Suppose $C^n \preceq D^n$, and $m > n$, and $C^m \not\preceq D^m$. Without loss of generality, $m = n + 1$. There exists a state $s \in C^m$ and an input sequence π such that for every state $t \in D^m$: $\lambda(s, \pi) \neq \lambda(t, \pi)$. Since $s \in C^{n+1}$, there must be a state $s' \in C^n$ and an input a such that $\delta(s', a) = s$. Now, consider the state $s' \in C^n$, and the input sequence $a \cdot \pi$. Since, for every state $t' \in D^n$: $\delta(t', a) \in D^m$, it is true that for every state $t' \in D^n$: $\lambda(s', a \cdot \pi) \neq \lambda(t', a \cdot \pi)$. This contradicts the fact that $C^n \preceq D^n$. $\Box$

Delay-preserving replacement is a useful notion because successive n-delay-preserving replacements on the same design still result in an n-delay-preserving replacement (unlike n-delay replacements, which add up, as shown in Proposition 6.2):

Proposition 6.8 (each delay-preserving replacement preserves power-up delay) *If $C^n \preceq D^n$ and $B^m \preceq C^m$, and $p = \max(m, n)$, then $B^p \preceq D^p$.*

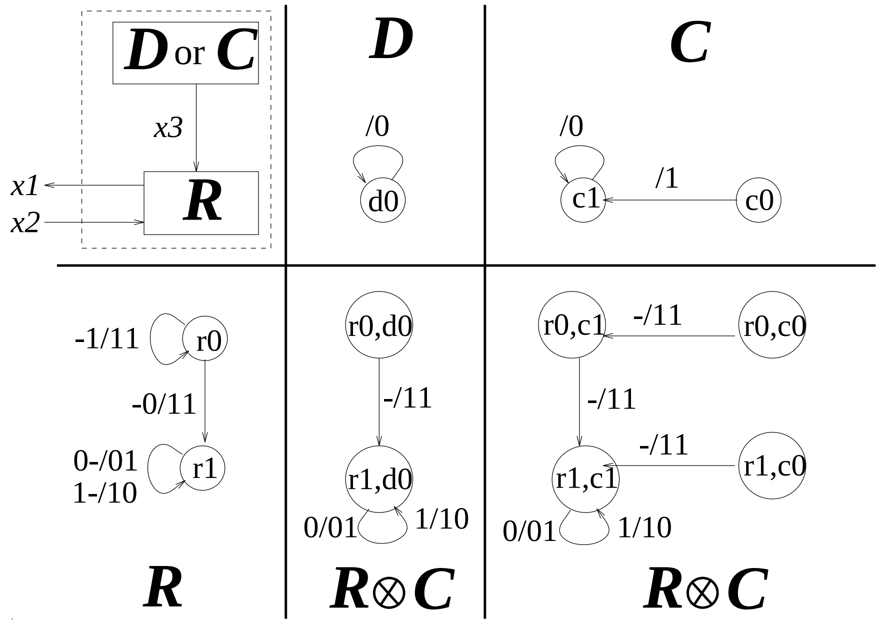

Figure 6.3: Delay-preserving replacement does not preserve compositionality. The input and output wires for the composition are shown at the top left. The labels on R denote x_2x_3/x_1, those on C or D denote $/x_3$, and those on $(R \otimes D)$ or $(R \otimes C)$ denote x_2/x_1. Note that designs D and C do not have any inputs.

Proof. Suppose $m > n$. Then $C^m \preceq D^m$, from Proposition 6.7. Since $\preceq$ is transitive, $B^p \preceq D^p$. The proof is similar if $m \leq n$. $\Box$

Unfortunately, the notion of delay-preserving replacement is not compositional; i.e. it is not always true that if $C^n \preceq D^n$, then for any environment R, $(R \otimes C)^n \preceq (R \otimes D)^n$. An example which illustrates this, for $n = 1$, appears in in Figure 6.3. $(R \otimes C)^1$ consists of states $\{(r_0, c_1), (r_1, c_1)\}$ and $(R \otimes D)^1$ is $\{(r_1, d_0)\}$. Clearly $(R \otimes C)^1 \not\preceq (R \otimes D)^1$. Thus, making a delay-presrving replacement for a design does not give a delay-preserving replacement for the design composed with its environment. This means that the iterative optimization strategy that we proposed in the previous section will not work for delay-preserving replacements.

However, it is easily seen that every n-delay-preserving replacement is also an n-delay replacement:

Proposition 6.9 *If $C^n \preceq D^n$, then $C^n \preceq D$.*

Proof. Since the set of states of D^n is a subset of the set of states in D, it follows that $D^n \preceq D$. The proof follows from the transitivity of $\preceq$. $\Box$

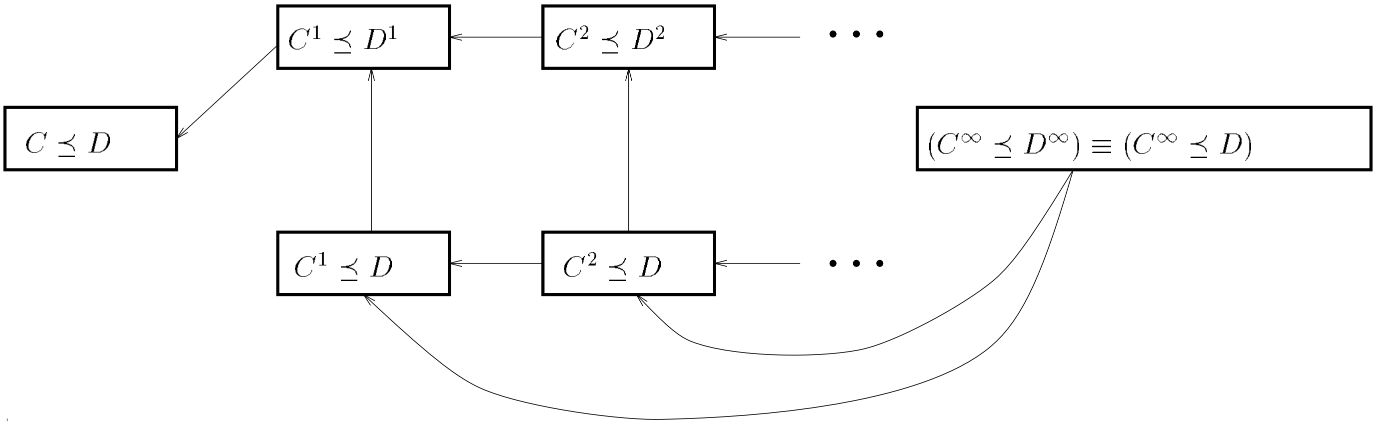

Figure 6.4: The safeness/flexibility tradeoff between the various replaceability notions. The arrows point towards a strictly safer notion which will afford strictly smaller flexibility for resynthesis. The arrows are transitive. The $C^\infty \preceq D^\infty$ notion is the least safe among those shown.

Thus if a synthesis technique that yields an n-delay-preserving replacement, it is automatically an n-delay replacement also. It is also interesting to note that in the limit, as n approaches ∞, the two notions converge (see Figure 6.4 for the complete picture of the safeness/flexibility tradeoffs):

Proposition 6.10 $C^\infty \preceq D^\infty$ *if and only if* $C^\infty \preceq D$.

Proof. (Only if part) Assume that $C^\infty \preceq D^\infty$. Since, D^∞ is a closed subset of states in D, it follows that $D^\infty \preceq D$. From the transitivity of $\preceq$ (Remark 4.2 on page 35), we have $C^\infty \preceq D$.

(If part) Assume that $C^\infty \preceq D$. but $C^\infty \not\preceq D^\infty$. Then there exists a state $s \in C^\infty$ and an input sequence π such that for any state $t \in D^\infty$: $\lambda(s,\pi) \neq \lambda(t,\pi)$ (Claim 1). Suppose n is the number of onion rings of D, (i.e. $D^\infty = D^n$). Then, since $s \in C^\infty$, there exists a state $s' \in C^\infty$ and an input sequence ρ such that $|\rho| > n$ and $\delta(s',\rho) = s$ (Claim 2). Now consider the input string $\rho \cdot \pi$ applied to state $s' \in C^\infty$. For any state $t \in D$, $\delta(t,\rho) \in D^\infty$ since $|\rho| > n$. From Claims 1 and 2, it follows that for any state $t \in D$: $\lambda(s',\rho\cdot\pi) \neq \lambda(t,\rho\cdot\pi)$. Since $s' \in C^\infty$ and t is any state in D this contradicts that $C^\infty \preceq D$. Thus by contradiction, $C^\infty \preceq D^\infty$. $\square$

The above result is useful for verification if we need to answer the question "$C^\infty \preceq D$?", it may be more efficient to check "$C^\infty \preceq D^\infty$?" especially if the number of states in D^∞ is much fewer than in D. The verification algorithm in Section 5.2 for checking $P \preceq Q$

had a worst case complexity of $O(n2^m)$, where m is the number of states in Q and n is the number of states in P. So, checking $C^\infty \preceq D^\infty$ is likely to be much more efficient that checking $C^\infty \preceq D$.

We call such a replacement, a **self-stabilizing replacement**, i.e. if $C^\infty \preceq D^\infty$ then C is a self-stabilizing replacement for D. Intuitively, C can replace D if the designer has sufficient control over the environment so that it can wait for a sufficient number (finite and bounded by a function of the design) of clock cycles after the power-up. So, we can replace design D with design C if we can afford to run enough cycles through the design after the power-up so that the transient behavior of the design disappears.

6.1.3 Resynthesis Methodology

We have introduced two notions (delay replacement and delay-preserving replacement) as two notions which make sense when we know that many clock cycles run through the design after power-up and before the design is used. In this subsection, we elaborate on the need to use *two* different notions, and how these can be used in a resynthesis methodology where successive design replacements are made.

We are interested in choosing arbitrary pieces of sequential logic from a design and making delay replacements. To make successive delay replacements, the designer must start with a number N, which is the allowable slack for his design; in other words, after making all these successive replacements the designer wants the resulting design to be an N-delay replacement of the original design. There are multiple ways in which the results discussed in the preceding subsections can be used to keep track of how much slack has been used up so far. The first is we can *partition* the design into non-overlapping pieces of design. If we do so and make successive delay replacements on non-overlapping designs, the overall effect of such delay replacements is non-accumulative (Proposition 6.5); for example, making the 2-delay replacement and 3-delay replacement in Figure 6.5 on non-overlapping sub-designs results in an overall 3-delay replacement for the entire design. On the other hand, if we have to (or choose to) make successive replacements on overlapping designs, the amount of slack used adds up (Proposition 6.6), and we need of keep track of this; for example, making the 2-delay replacement followed 3-delay replacement in Figure 6.6 on an overlapping sub-design results in an overall 5-delay replacement for the entire design. The utility of the delay-preserving replacement notion is in Proposition 6.8. If we want to make

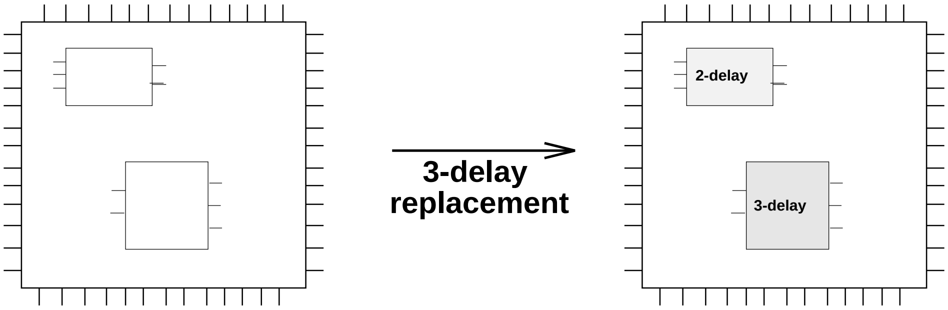

Figure 6.5: Delay replacements on non-overlapping sub-designs

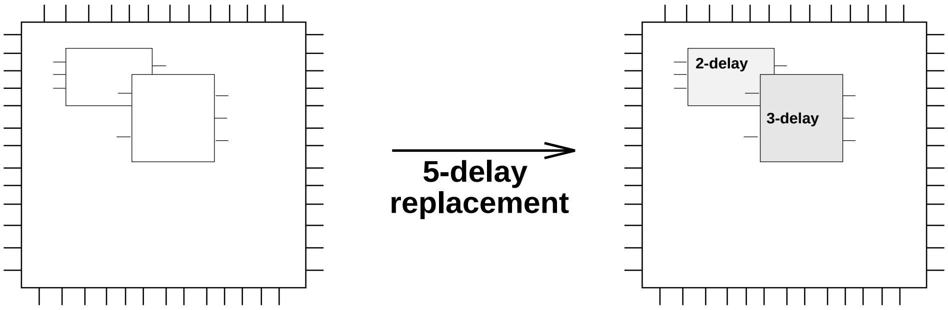

Figure 6.6: Successive delay replacements on overlapping sub-designs

successive design replacements on the *same* design, it is then useful to use the notion of delay-preserving replacement (as opposed to delay replacement); making successive delay-preserving replacements does not add up the delay used in each individual step; rather, the total delay is the maximum delay used in any single step.

In the next section we will discuss how we achieve delay-preserving replacements (as Proposition 6.9 says, each n-delay-preserving replacement is automatically an n-delay replacement).

6.2 Synthesis for Delay Replaceability

We describe two methods for optimizing multi-level circuits such that the optimized circuit is an n-delay replacement of the original circuit, for a given n.

6.2.1 Combinational Resynthesis

In this section we will use the flexibility due to the delay replacements to do optimization on the combinational part of the circuit. We will extend the method described in Section 5.1.4 so that we can optimize the given circuit with respect to delay replacement.

Let the original design D have $(k+1)$ different delayed designs, $D^0, D^1, \ldots, D^k$; for any $j \geq k : D^j = D^k = D^\infty$. These $(k+1)$ delayed designs form an "onion ring" structure (see Section 5.1.3). Clearly, any state reachable from itself under some input sequence belongs to each D^i and to D^∞. We will not alter the behavior of any such state (a "stable" state). All the flexibility for resynthesis comes from the set of "transient" states, i.e. $D \setminus D^\infty$.

Recall that an n-delay replacement allows the new design to have some flexibility over the first n clock cycles after the power-up, and after these clock cycles it is indistinguishable from the original design. As we said before, often it is known in advance that the design will be allowed to settle for at least a fixed number of clock cycles before it is used. We use this slack n for resynthesis— the higher n is, the greater the flexibility allowed for resynthesis; however, the environment of the design cannot rely on the input/output behavior of the design for the first n cycles.

For an n-delay replacement, we will express the flexibility to obtain a new design C such that C^n is exactly the same as D^n. Note, that this is a conservative strategy since only $C^n \preceq D$ is needed.

As described previously in Section 5.1.4, the primary inputs and outputs to the design are denoted by $\vec{i}$ and $\vec{o}$, respectively (Figure 5.1). The outputs of the combinational part which feed the memory elements are denotes by $\vec{y}$, and the lines which feed the output of the memory elements to the combinational part are denoted by $\vec{x}$. To do resynthesis for delay replaceability, we obtain a Boolean relation $\mathcal{V}(\vec{i}, \vec{x}, \vec{o}, \vec{y})$ which describes the flexibility for replacement. We will then use this relation to do multi-level resynthesis on our design.

First, we informally describe the flexibility which will be specified later using a Boolean relation $\mathcal{V}$. We note again that techniques for using a Boolean relation to do multi-level synthesis [70] require the relation to be such that the starting design satisfies the relation. The Boolean relation is such that the behavior of the states in D^n (the "stable" states) is preserved. For $0 \leq i < n$, on any input, the relation allows a state in $(D^{i-1} \setminus D^i)$ to transition to any state in D^i. Clearly, the original design satisfies this flexibility relation.

It can also be seen that for any design C that satisfies the relation it is true that $C^n = D^n$, i.e. after the first n clock cycles every reachable state in C is equivalent to one in D^n. Also note that since we do not care about the outputs during the first n clock cycles, the outputs of the states in $D \setminus D^n$ can be arbitrarily chosen.

Formally, the Boolean relation $\mathcal{V}(\vec{i}, \vec{x}, \vec{o}, \vec{y})$ which characterizes this flexibility is:

$$\begin{aligned}\mathcal{V}(\vec{i}, \vec{x}, \vec{o}, \vec{y}) &= \Sigma_{i=0}^{n-1}[(\vec{x} \in D^i \setminus D^{i+1}) \wedge (\vec{y} \in D^{i+1})] + \\ &\quad [(\vec{x} \in D^n) \wedge (\vec{y} = \delta_D(\vec{x}, \vec{i})) \wedge (\vec{o} = \lambda_D(\vec{x}, \vec{i}))]\end{aligned}$$

The intuition for the above relation $\mathcal{V}$ is that, given n, we choose to preserve the behavior of all states in the set D^n, i.e. states in this set are forced to have the same output and next-state functions as in the original design D. For states outside of D^n, if the state lies in $D^i \setminus D^{i+1}$, we allow the next state of such a state to be any state in D^{i+1}; we do not care about the output from this state. All this ensures that n cycles after power-up, the new design would be in a state in D^n and thus the new design would be an n-delay replacement for the original design (in fact, since $C^n = D^n$, it is also an n-delay-preserving replacement).

Notice that for any integer $m \geq k$, the delayed design D^m is the same as D^∞, i.e. the set of stable states. Thus, the flexibility described by the relation $\mathcal{V}$ for m-delay replacement is the same as that for k-delay replacement. If we compute the number k for a design in advance, we know that we will not get any additional flexibility by allowing a slack greater than k.

Once we have the BDD for the Boolean relation $\mathcal{V}(\vec{i}, \vec{x}, \vec{o}, \vec{y})$ we can use this relation instead of the Boolean relation $\mathcal{R}(\vec{i}, \vec{x}, \vec{o}, \vec{y})$ that we used in Section 5.1.4. Not surprisingly, just like we found that the BDDs for $\mathcal{R}$ were getting too large to build, we encounter the same problem in building the BDDs for $\mathcal{V}$. So, instead we build a relation which is easier to build, but correctly expresses the flexibility for only the states outside D^n:

$$\mathcal{U}(\vec{x}, \vec{y}) = \Sigma_{i=0}^{n-1}[(\vec{x} \in D^i \setminus D^{i+1}) \wedge (\vec{y} \in D^{i+1})] + [(\vec{x} \in D^n)]$$

Notice that the relation $\mathcal{U}(\vec{x}, \vec{y})$ specifies the flexibility of the states inside D^n incorrectly (actually, it says that the states inside D^n can choose their next-state and outputs totally arbitrarily!). But we will take care of this problem by restricting the don't cares to only states outside D^n when we compute the don't cares of internal nodes in the network. See Figure 6.7 for the complete algorithm (notice that it is different from the algorithm in

```
procedure delay-replacement (input: network in terms of (i⃗, x⃗, o⃗, y⃗), parameter n) {
    for (each node p in the network)
        Compute a BDD S_p(i⃗, x⃗) representing node p in terms of (i⃗, x⃗)
    C(x⃗) ← 1
    U(x⃗, y⃗) ← 0
    loop n times {
        C(y⃗) ← ∃x⃗∃i⃗ : [C(x⃗) · ∏_{j=1}^{t} (y_j ≡ S_{y_j}(i⃗, x⃗))]
        C(x⃗) ← C(y⃗)_(y⃗←x⃗)
        U(x⃗, y⃗) ← U(x⃗, y⃗) + ¬C(x⃗) · C(y⃗)
    }
    U(x⃗, y⃗) ← U(x⃗, y⃗) + C(x⃗)
    for (each node p in the network) {
        Add BDD variable p to the BDD variable list
        for (each node r in the network) {
            if (r is not in the transitive fanout of p)
                S_r(i⃗, x⃗, p) ← S_r(i⃗, x⃗)
            else
                recompute S_r(i⃗, x⃗, p) representing node r in terms of (i⃗, x⃗, p)
        }
        F(i⃗, x⃗, p) = ∃o⃗, y⃗ : [U(x⃗, y⃗) · ∏_{j=1}^{t} (y_j ≡ S_{y_j}(i⃗, x⃗, p)) · ∏_{k=1}^{n} (o_k ≡ S_{o_k}(i⃗, x⃗, p))]
        G(i⃗, x⃗) = F(i⃗, x⃗, p)_{p=0} · F(i⃗, x⃗, p)_{p=1} + ¬F(i⃗, x⃗, p)_{p=0} · ¬F(i⃗, x⃗, p)_{p=1}
        D(i⃗, x⃗) = G(i⃗, x⃗) · ¬C(x⃗)
        Image project ¬D(i⃗, x⃗) to the local input of p
        Use the flexibility to simplify the local functionality of p
        Remove BDD variable p from the BDD variable list
        if (functionality of node p changed)
            for (each fanout node r of node p)
                recompute S_r(i⃗, x⃗) representing node r in terms of (i⃗, x⃗) .
    }
}
```

Figure 6.7: Procedure to optimize a multi-level network for n-delay replacement

Ckt.	Initial size	Safe replacement		n-delay replacements							
				$n = 1$		$n = 2$		$n = 5$		$n = \infty$	
		final	time	final	time	final	time	final	time	final	time
s27	12	12	0.05	12	0.04						
s298	150	130	1.91	130	0.95	123	1.01	113	1.00	107	1.56
s344	156	152	689.75	153	1.90	147	3.21	147	5.83	144	10.31
s349	160	154	114.19	154	1.97	148	3.38	148	5.95	145	10.63
s382	176	164	8.27	162	5.87	162	19.85	162	28.41	156	80.62
s386	204	173	1.16	127	1.19						
s400	184	162	8.52	160	6.13	160	21.17	160	29.75	154	83.53
s444	184	163	9.09	161	6.42	161	23.05	161	36.21	156	101.80
s510	280	279	4.65	279	2.94	279	2.95	279	2.96		
s526	283	240	6.91	240	3.68	238	5.78	237	5.98	188	35.89
s641	199	timeout		194	10.64						
s713	204	timeout		195	10.99						
s820	504	371	14.70	359	6.16						
s832	521	364	14.75	353	7.43						
s953	489	484	28.10	484	24.03						
s1196	618	600	504.04	600	95.95	600	103.82				
s1238	690	625	574.69	625	115.57	625	120.16				
s1488	813	755	42.11	697	12.33						
s1494	819	742	38.32	697	12.35						

Table 6.1: Experimental results for delay replacements. The initial size of the circuits and the final size are in the number of literals. Since we know that for any integers $m \geq k$, we would get the same results for both m-delay replacement and k-delay replacement, the redundant experiments (denoted by blanks in the above table) were not performed.

Figure 5.9 only in the computation of $\mathcal{C}(\vec{x})$ and in the use of $\mathcal{U}(\vec{x}, \vec{y})$ instead of $\mathcal{P}(\vec{i}, \vec{o}, \vec{y})$).

Experiments

In this section we report experimental results using the algorithm described in Section 6.2.1 on ISCAS85 sequential circuits. We used BDDs to manipulate the Boolean relations and sets. We focused on the area reduction for n-delay replacements. We report results for various values of n.

The experimental results are shown in Table 6.1. The starting circuits are the same as in Table 5.2. We show the optimizations obtained by the safe replacement resynthesis method described in the previous chapter. Then, we show the results of the method presented in this chapter for n-delay replacements for $n = 1, 2, 5, \infty$. For the $n = \infty$ column, the value of n was less than 668 for all the examples (because all circuits had fewer than

668 onion rings). The table shows that for many examples, significant additional optimizations are obtained by allowing power-up delay. Even for $n = 1$, we see good results for some examples, e.g. `s386, s1488, s1494`. Also, in most cases the CPU times for n-delay replacements are within an order of magnitude of the CPU time for combinational resynthesis. The much larger CPU times for pure safe replaceability can be attributed to the rigid conditions for safe replacement, which require placing constraints on the outputs from the "transient" states as well as correlating the next states of "transient" states with the inputs. Both these constraints are avoided by the resynthesis method presented in Section 6.2.1. This leads to a much smaller BDD to express the relation $\mathcal{U}(\vec{x}, \vec{y})$ (instead of the relation $\mathcal{P}(\vec{i}, \vec{o}, \vec{y})$ in the previous chapter), and hence, faster multi-level resynthesis.

We suspect that synchronous recurrence equations [23] also lead to delay replacements. However, the experimental results presented there indicate that using synchronous recurrence equations is not very effective; our optimization method (using the SIS commands `sweep; eliminate -1` followed by the optimization procedure described in this paper) produces smaller circuits using less CPU times than those reported in [23]. (Note that the CPU times indicated in Table 6.1 do not include the time used by the SIS preprocessing commands; however, for all the examples, that time is orders of magnitude lower than than the times reported for the sequential optimations in Table 6.1. On the other hand, the CPU times reported in [23] are much greater than the CPU times reported in Table 6.1.)

We performed an experiment on one of the benchmark circuits with a large number of onion rings (`s526`) to explore the tradeoffs between flexibility and the power-up delay allowed. The results are in Table 6.2. The table shows that by allowing more delay n we do get additional flexibility. Also, the CPU times increase with higher values of n, partly because the time taken to compute the Boolean relation $\mathcal{U}$ goes up with higher n.

In the experiments above, the initial nodes of the circuits have been collapsed minimally. We see that safe replaceability gives very marginal improvements over pure combinational reductions, and at a much larger CPU time cost. For this reason, we argued in the previous chapter that we might get better use of the safe replaceability notion by using larger node sizes in the circuits. There we increased the node sizes in these benchmark circuits by using SIS commands `eliminate 10` or `collapse`. We ran our algorithm for delay replacement on these starting points also. Results are reported in Table 6.3. Once again, we see that using n-delay replacements instead of safe replacements allows us greater flexibility for resynthesis and gives better optimizations. Also, the CPU times are once again much

n-delay replacements initial size = 283 literals		
n	reduction	time
1	43	3.68
100	54	11.52
200	58	14.55
300	70	17.33
400	69	20.38
500	69	22.51
600	72	24.37
667	95	36.27

Table 6.2: Power-up delay/flexibility tradeoff for s526; reduction is in number of literals

Ckt.	Initial size	Safe replacement		n-delay replacements							
				$n=1$		$n=2$		$n=5$		$n=\infty$	
		final	time	final	time	final	time	final	time	final	time
s27	12	12	0.04	12	0.03						
s298	156	137	1.82	137	0.93	127	0.96	116	0.96	109	1.54
s344	168	156	807.36	155	1.83	150	3.10	150	5.61	148	9.92
s349	173	156	72.91	156	1.97	151	3.14	151	5.83	149	10.02
s382	204	166	7.62	164	5.59	164	17.55	164	26.53	160	80.32
s386	205	138	1.14	99	0.97						
s400	229	168	7.93	166	5.96	166	19.04	166	27.19	157	83.48
s444	236	171	8.27	169	5.99	169	19.99	169	31.96	166	91.49
s510	307	255	1.89	253	1.52	254	1.50	251	1.54		
s526	323	233	6.24	233	4.73	233	5.56	232	6.20	208	27.13
s641	234	timeout		207	14.11						
s713	285	timeout		202	14.97						
s820	468	353	7.85	331	3.84						
s832	470	354	7.59	334	3.94						
s953	700	597	22.92	590	21.50						
s1196	788	626	265.46	626	109.14	626	112.92				
s1238	882	636	338.30	636	170.87	636	178.53				
s1488	886	576	30.43	541	19.20						
s1494	896	542	30.67	524	20.08						

Table 6.3: Experimental results with collapsed nodes in the starting netlist.

better than those for safe replacements.

6.2.2 Optimization by Removing State Bits

In Section 5.1.4 and in the previous section, we described methods which optimize the combinational part of multi-level circuits while exploiting the sequential flexibility (safe replaceability or delay replaceability). In this section we present a method which makes optimizations on the sequential part of the network (i.e the latches) also, so that the new circuit is a delay replacement of the original circuit.

The method for optimization presented in this section is inspired by sequential optimization techniques for circuits with a designated initial state (i.e. all latches have a designated initial state) and since the circuit is always assumed to start in this designated initial state (we call it the *DIS* assumption in this section), we can arbitrarily change the behavior of the set of states which cannot be reached from this initial state without affecting the functionality of the circuit. As discussed in Chapter 2, we do not make the *DIS* assumption in this thesis. Nevertheless, we shall see how the synthesis technique with the *DIS* assumption will inspire our solution. So we will briefly review this technique in Section 6.2.2.1.

The synthesis methods presented in Sections 5.1.4 and 6.2.1 do not alter the number of latches, and thus, do not change the number of states in the design. However, the replacement criteria (Definitions 6.2 and 4.1) do not require the original design and the replaced design to have identical number of states. In fact, if we just just select the smallest subset of states in the original design which is a tSCC (which is much smaller than the total number of states, as seen from Table 5.1) and reimplement it with the minimum number of latches (an N-state tSCC can be encoded and implemented using $\lceil \log(N) \rceil$ latches), we can minimize the number of latches to obtain a safe replacement. So, a plausible strategy for resynthesis for safe (or delay) replacement can be to extract the tSCC (or the outer envelope) as we do in the previous chapter, throw away the original netlist, and encode this tSCC with a minimal number of latches and re-implement the tSCC obtaining a safe replacement. We decided not to use this strategy for the following reason. There is an analogous strategy for removing latches for circuits under the *DIS* assumption. There we compute the set of reachable states from the initial state, then reencode just this set of states with the minimum number of latches and produce a replacement design (throwing away the original netlist structure in the process). Unfortunately, this strategy has been observed [82] to produce much larger circuits (in the number of literals) than the original circuit reinforcing the belief that the starting multi-level netlists are precious and if our synthesis procedure throws them away it becomes very hard to reconstruct as good a netlist just from the input-output functionality of a design. Thus, for our synthesis methods we decided to start with the original multi-level netlist and make iterative modifications on it rather than extracting the functionality of the netlist and throwing away the original multi-level structure. This leads us back to the strategy of replacing a subset of the original latches with some combinational logic; we describe this in Section 6.2.2.2. First we review the latch removal techniques under the *DIS* assumption.

6.2.2.1 Removing redundant latches under the *DIS* assumption

We briefly review the techniques in [6, 49, 5] for reducing the latches for circuits assuming DIS.

Given the initial state s, we extract the set of states reachable from s (see [48] for example). Let $\mathcal{C}(\vec{x})$ be the characteristic function of the set of reachable states, where $\vec{x} = \{x_1, x_2, \ldots, x_t\}$ represent the t latches in the design. Since the states outside of $\mathcal{C}(\vec{x})$ are not reachable we do not care about their behavior. This allows some latches to be replaced with combinational logic. For example, suppose among all the states in $\mathcal{C}(\vec{x})$, the value of the state bit x_j is a function of $(x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_t)$, Then we can replace latch x_j by that combinational function. This replacement is guaranteed not to affect the behavior of the states in $\mathcal{C}(\vec{x})$. Replacing a latch by combinational logic removes a latch as well as some combinational logic which was only used to drive this latch (see Figure 6.8).

The following two conditions must be satisfied to allow latches $\{x_{t'+1}, \ldots, x_t\}$ to be replaced by a function-set $F = (f_{t'+1}, \ldots, f_t)$ such that for any $i \in \{t'+1, \ldots, t\}$, $f_i : \{0,1\}^t \to \{0,1\}$.

1. Latch redundancy condition. This ensures that for any $i \in \{t'+1, \ldots, t\}$, for every state in $\mathcal{C}(\vec{x})$, the value of latch x_i is uniquely determined by the value of latches $\{x_1, \ldots, x_{t'}\}$:

$$\forall i \in \{t'+1, \ldots, t\} : \forall x_i \exists x_{i+1} \exists x_{i+2} \cdots \exists x_t : [\mathcal{C}(\vec{x})) = 0] \qquad (6.1)$$

2. Function-set selection condition. This ensures that the function-set F is chosen so that

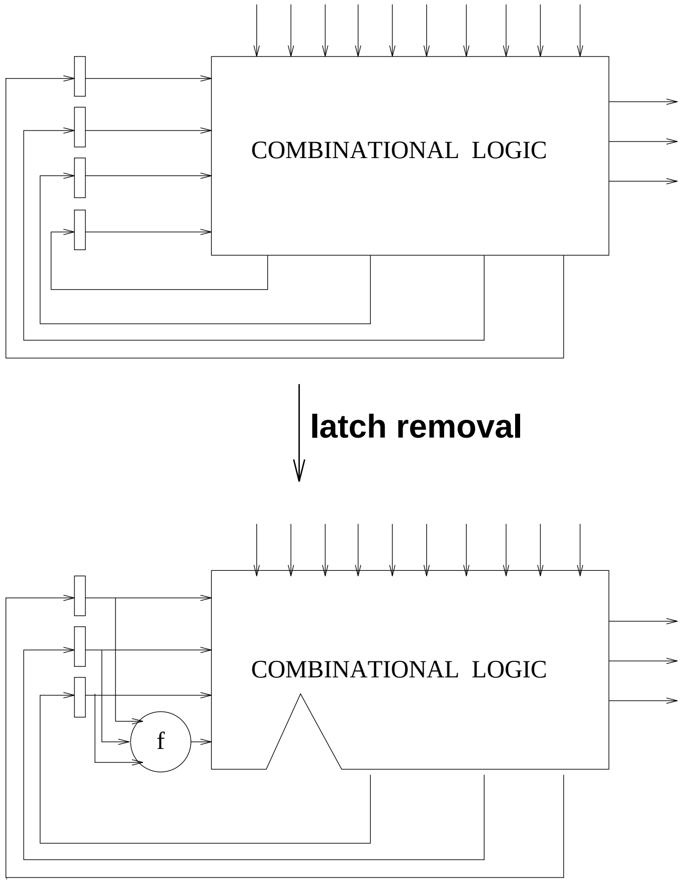

Figure 6.8: Replacing a latch with function f.

for each state in $\mathcal{C}(\vec{x})$, the values of the replaced latches are correctly represented:

$$\text{if } (a_1, \ldots, a_t) \in \mathcal{C}(\vec{x}) \text{ then } \forall i \in \{t'+1, \ldots, t\} : f_i(a_1, \ldots, a_{t'}) = a_i \tag{6.2}$$

For an example, consider a design D whose STG is identical to that of D_0 as the one shown in Figure 4.2 (on page 33). D is implemented by three latches $\{x_1, x_2, x_3\}$ whose encodings are shown in the figure; D also has a designated state 000 (we emphasize that D is a different design than D_0 because it has an extra input line, the reset line, which, when asserted, sends the design to state 000). The set of reachable states from 000 is $\mathcal{C}(\vec{x}) = \{000, 011, 101\}$. Using conditions 6.1 and 6.2 above, we can replace latch x_3 with the function $f_3 = x_1 + x_2$, and we get design C whose STG[1] is identical to that of design A shown in Figure 6.9; the new designated initial state of design C is 00. States 00, 01 and

[1]In the next section we explain in a little more detail, how the selection of x_3 and f_3 leads to this transition

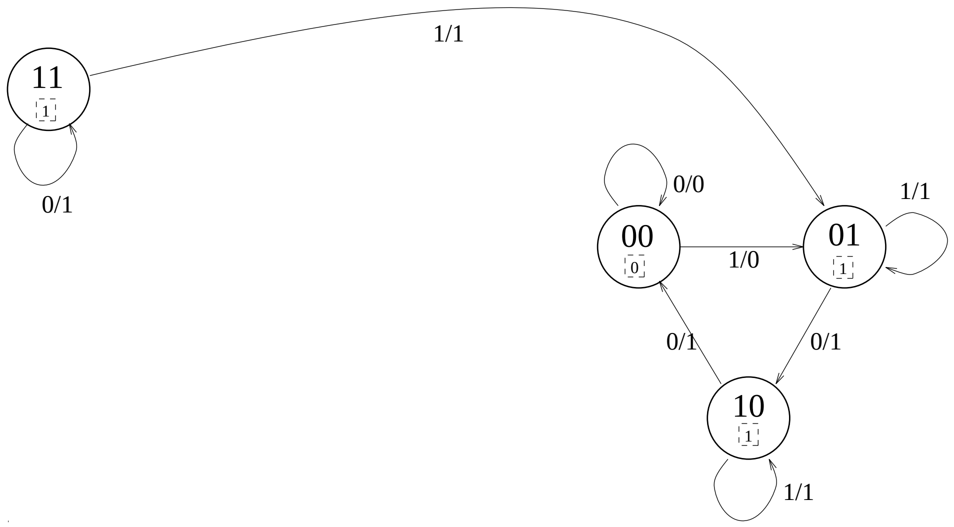

Figure 6.9: Design A. The value of $f_3 = x_1 + x_2$ is shown in the dotted box.

10 are respectively equivalent to states 000, 011 and 101 in D. State 11 is not equivalent to any state in D_0 but its behavior is immaterial because the new design always starts in state 00, and can never reach state 11.

6.2.2.2 Removing redundant latches without the *DIS* assumption

We now look at the naive extension of the algorithm in the previous section, and see why it does not work if the design can power up in any state. Without the *DIS* assumption, the set of states in any core (see Section 5.1.2 for a discussion of choices for a core; a tSCC is an example of a core) represents the desirable steady-state behavior of a design. So a naive strategy for removing redundant latches will be to make any core set of states of a design play the role of $\mathcal{C}(\vec{x})$ in the previous section and then use exactly the same strategy, with conditions 6.1 and 6.2, for redundant latch identification and removal. Thus, we may obtain design A (shown in Figure 6.9) as a safe replacement (a 0-delay replacement) for design D_0 in Figure 4.2. However, A is not an n-delay replacement of D_0 for any n. This is because $A^n = A$ and $A \not\preceq D_0$. Also observe that D_0 can be reset from any state to state 000 under input sequence $0 \cdot 0$, but the same input sequence does to reset A even if we wait for arbitrary number of clock cycles after power-up (because A can remain in state 11 arbitrary many cycles). Thus the behavior of the states outside the core must be controlled

structure.

to make sure that the new design is an n-delay replacement.

Let $\mathcal{C}(\vec{x})$ be a core that we are going to use for delay replacement on the original design D. There are two factors which determine the behavior of the new design A. First, the selection of latches to be replaced, $\hat{x} = (x_{t'+1}, \ldots, x_t)$ partitions the 2^t states in design D into $2^{t'}$ equivalence classes, each with $2^{t-t'}$ states. Two states $\vec{a} = (a_1, \ldots, a_t)$ and $\vec{b} = (b_1, \ldots, b_t)$ lie in the same equivalence class if and only if for all $i \in \{1, \ldots, t'\}$: $a_i = b_i$. We denote the equivalence class of $\vec{a}$ by $\Omega(\vec{a})$. Each state of A represents an equivalence class D. original design, i.e. $\tilde{a} = (a_1, \ldots, a_{t'})$ represents $\Omega(\vec{a})$. Second, the selection of the function-set $F = (f_{t'+1}, \ldots, f_t)$ determines the representative state (of D) in each equivalence class, and the representative state determines the behavior of this class in the new design. Thus, if $f_{t'+1}(\tilde{a}) = c_1$, $f_{t'+2}(\tilde{a}) = c_2$, $f_t(\tilde{a}) = c_{t-t'}$, then $(a_1, \ldots, a_{t'}, c_1, \ldots c_{t-t'})$ is the representative state of the class $\tilde{a}$ (or $\Omega(\vec{a})$). Let $\phi(\tilde{a})$ denote the representative state of the class $\tilde{a}$. Now, for input $\vec{i}$, the next state of $\tilde{a}$ in A is $\delta_A(\tilde{a}, \vec{i}) = \Omega(\delta_D(\phi(\tilde{a}), \vec{i}))$ and the output is $\lambda_A(\tilde{a}, \vec{i}) = \lambda_D(\phi(\tilde{a}), \vec{i})$.

Notice that the set of replaced latches $(x_{t'+1}, \ldots, x_t)$ must satisfy the latch redundancy condition 6.1, where $\mathcal{C}(\vec{x})$ is the core. Also the selection of the function-set F must satisfy the function-set selection condition 6.2. Based on the derivation of the behavior of the new design, discussed above, it can be seen that if we replace latch x_3 in design D_0 of Figure 4.2 with the function-set $F = (f_1)$ such that $f_1 = x_1 + x_2$ we get the new design shown in Figure 6.9 (for example, state 11 goes to state 11 on input 0 because $\delta_A(11, 0) = \Omega(\delta_{D_0}(\phi(11), 0)) = \Omega(\delta_{D_0}(111, 0)) = \Omega(110) = 11$). For this example, the representative elements can be read off by using the number in the dotted box in Figure 6.9; for example, $\phi(\Omega(100)) = \phi(\Omega(101)) = \phi(10) = 101$.

However, as previously discussed, the function replacement $F = (x_1 + x_2)$ is not an n-delay replacement for D_0. Thus we also need to regulate the behavior of equivalence classes which do not contain any state in the core of the original design. Notice that each state in the core is the representative element of its class, and the behavior of this class in the new design is equivalent to that of the core state in the original design.

As in Section 6.2.1 we will satisfy the requirement for n-delay replacement by ensuring, if we wait for n clock cycles with arbitrary input vectors, we reach a state inside the core. The procedure we describe next will guarantee this and thus we will obtain an n-delayed replacement.

procedure compatible-set (**input:** $\mathcal{C}(\vec{x})$, $\{x_{t'+1}, \ldots, x_t\}$, $\mathcal{T}(\vec{x}, \vec{i}, \vec{y})$, n) {

$j \leftarrow 0$

$\mathcal{S}(\vec{x}) \leftarrow \mathcal{C}(\vec{x})$

$\mathcal{P}(\vec{x}) \leftarrow \exists x_{t'+1} \exists x_{t'+2} \cdots \exists x_t : [\mathcal{S}(\vec{x})]$

while $((j < n)$ **and** $(\mathcal{P}(\vec{x}) \neq 1))$ {

$j \leftarrow j + 1$

$\mathcal{P}(\vec{y}) \leftarrow (\mathcal{P}(\vec{x}))_{(\vec{y} \leftarrow \vec{x})}$

$\mathcal{R}(\vec{x}) \leftarrow \forall \vec{i} \exists \vec{y} : [\mathcal{T}(\vec{x}, \vec{i}, \vec{y}) \wedge \mathcal{P}(\vec{y})]$

if $(\mathcal{R}(\vec{x}) \cdot \overline{\mathcal{P}(\vec{x})} = 0)$ **goto** *end*;

$\mathcal{S}(\vec{x}) \leftarrow \mathcal{S}(\vec{x}) + \mathcal{R}(\vec{x}) \cdot \overline{\mathcal{P}(\vec{x})}$

$\mathcal{P}(\vec{x}) \leftarrow \exists x_{t'+1} \exists x_{t'+2} \cdots \exists x_t : [\mathcal{S}(\vec{x})]$

}

end: **if** $(\mathcal{P}(\vec{x}) = 1)$**return**$\mathcal{S}(\vec{x})$

else return FAIL

}

Figure 6.10: Procedure to optimize $\mathcal{S}(\vec{x})$ (input $\mathcal{C}(\vec{x})$ is the core set of states, $\mathcal{T}(\vec{x}, \vec{i}, \vec{y})$ is the transition relation of the design and n is the delay parameter for n-delay replacement)

Choices for the function-set F

Definition 6.4 *Given a set of states S in $\{0,1\}^t$ (i.e. a set of states of the original design), a function-set $F = (f_{t'+1}, \ldots, f_t)$ is* **compatible** *with S if for any vector $\tilde{a} = (a_1, \ldots, a_{t'})$, $(a_1, \ldots, a_{t'}, f_{t'+1}(\tilde{a}), \ldots, f_t(\tilde{a})) \in S$.*

First assume that we have chosen the core set of states and a set of latches $(x_{t'+1}, \ldots, x_n)$ to be replaced (so that condition 6.1 is satisfied). Thus we already have a partition of equivalence classes for states in $\{0,1\}^t$. Now we will select a set of states S in $\{0,1\}^t$, and derive a function-set F compatible with S. We use the procedure in Figure 6.10 to obtain S (which is denoted by its characteristic function $\mathcal{S}(\vec{x})$). States in set S are candidates for representative elements of their respective equivalence classes. Each equivalence class has at least one state in S; each equivalence class whose state is in the core has exactly one state in S.

Theorem 6.11 *If the procedure "compatible-set" returns a set $\mathcal{S}(\vec{x})$, then there exists at least one function-set $F = (f_{t'+1}, \ldots, f_t) : \{0,1\}^{t'} \rightarrow \{0,1\}^{t-t'}$ compatible with $\mathcal{S}$. And if we replace latches $(x_{t'+1}, \ldots, x_t)$ by F, then the new design is an n-delay replacement of the original design.*

Proof. First notice that the set $\mathcal{P}(\vec{x})$ is the set of all equivalence classes which have at least one member in $\mathcal{S}(\vec{x})$. Since the procedure returns a set $\mathcal{S}(\vec{x})$ only when $\mathcal{P}(\vec{x}) = 1$ we know that there exists at least one function-set which is compatible with $\mathcal{S}$.

Now, observe that if a state from an equivalence class is added to $\mathcal{S}(\vec{x})$ in iteration $j = p$, then for any iteration $i > p$, no state from this equivalence class is added to $\mathcal{S}(\vec{x})$. If one or more states from some equivalence classes are added in iteration $j = p$, we say that this equivalence class is covered in the p-th iteration. Suppose we obtain a new design by choosing a function-set which is compatible with $\mathcal{S}(\vec{x})$. Now consider any state in the new design. This state $\tilde{a}$ represents an equivalence class in the old design and derives its behavior from the representative state $\vec{a}$ of that equivalence class. Let this equivalence class be covered in the q-th iteration; thus $\vec{a}$ was added to $\mathcal{S}(\vec{x})$ in iteration $j = q$. Thus $\vec{a} \in \mathcal{R}(\vec{x})$ in iteration $j = q$, and for any input, the next state of $\vec{a}$ lies in an equivalence class which was covered in r-th iteration, for some $r < q$. Thus in the new design, for any input vector, state $\tilde{a}$ goes to some state which represents an equivalence class which was covered in an earlier iteration. Since we have at most $j = n$ iterations, it is clear that for any state $\tilde{a}$ in the new design, if we apply any arbitrary input vectors for n steps, we will reach a state $\tilde{b}$ representing an equivalence class which was covered in iteration $j = 0$. Since iteration $j = 0$ only covers the the equivalence classes for the core states, $\tilde{b}$ is equivalent to a state in the core of the old design. Thus, the new design is an n-delay replacement of the old one. $\square$

As an example, start with design D_0 in Figure 4.2 and choose the set $\{000, 011, 101\}$ as the core and latch x_3 to be replaced. The procedure *compatible-set* returns the set $\mathcal{S}(\vec{x}) = \{000, 011, 101, 110\}$. $F = (f_3)$ where $f_3 = x_1 \oplus x_2$ is the only function-set compatible with this. The resulting design B (shown in Figure 6.11) is a 1-delay replacement of design D_0.

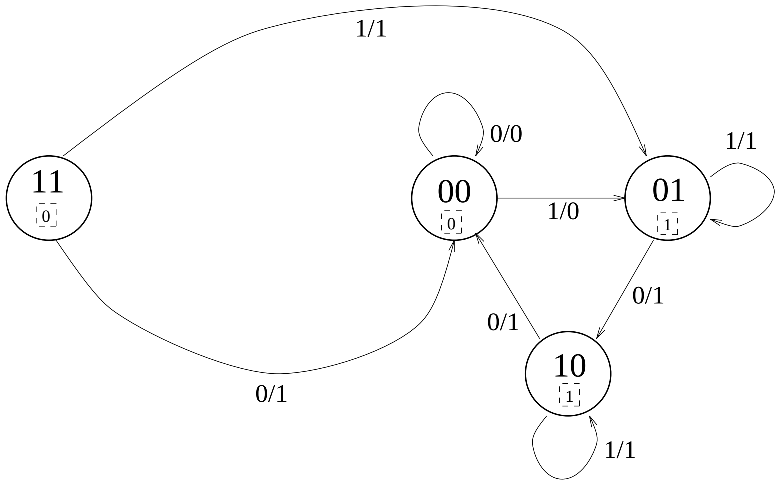

Figure 6.11: Design B. The value of $f_3 = x_1 \oplus x_2$ is shown in the dotted box.

Selecting an Optimal function-set F

If the procedure *compatible-set* returns a set S, any function-set F compatible with S is allowed. However, our overall goal is to optimize the area of the original circuit, so we would like to choose an F which has the smallest area. In fact, if we cannot find F within a certain bound, the savings gained by removing the redundant latches may actually be lost (refer once again to Figure 6.8 for the tradeoff in area saving).

Selecting the function-set F is equivalent to determining a multi-level network with t' inputs and $(t-t')$ outputs so that the network is compatible with $S(\tilde{x}, \hat{x})$ where $\tilde{x} = \{x_1, \ldots, x_{t'}\}$ and $\hat{x} = \{x_{t'+1}, \ldots, x_t\}$. Notice, that this is the problem of obtaining a multi-level network compatible with a Boolean relation in terms of its inputs and outputs. This can be solved in two steps. First, we obtain any arbitrary multi-level network representing some F compatible with S. Then, one approach would be to use the techniques in [70] (like we used in Sections 5.1.4 and 6.2.1) to optimize this network maintaining compatibility with S.

However, our experiments (presented later in this section) show that the size of F is relatively small compared to the given circuit sizes and that the approach described by procedure *greedy-function-set* in Figure 6.12 worked well. We order the latches in an arbitrary order. For each latch to be replaced, we find the on-set and don't care set for the replacement function. Then we use a heuristic to find a function which is compatible with this on-set and don't care set. Since all the Boolean quantities are represented as BDDs, we

procedure greedy-function-set (**input:** $\mathcal{S}(\tilde{x}, \hat{x})$, $\{x_1, \ldots, x_{t'}\}, \{x_{t'+1}, \ldots, x_t\}$) {
  for $j = t'$ **to** t **do** {
    $\mathcal{O}(\tilde{x}) \leftarrow \exists x_{t'+1} \exists x_{t'+2} \cdots \exists x_t : [\mathcal{S}(\tilde{x}, \hat{x}) \cdot (x_i = 1)]$
    /* $\mathcal{O}(\tilde{x})$ is the on set for f_j */
    $\mathcal{D}(\tilde{x}) \leftarrow \forall x_j \exists x_{t'+1} \exists x_{t'+2} \cdots \exists x_{j-1} \exists x_{j+1} \exists x_{j+2} \ldots x_t : [\mathcal{S}(\tilde{x}, \hat{x})]$
    /* $\mathcal{D}(\tilde{x})$ is the dont-care set for f_j */
    $f_j(\tilde{x}) \leftarrow$ bdd-minimize$(\mathcal{O}(\tilde{x}), \mathcal{D}(\tilde{x}))$
    $\mathcal{S}(\tilde{x}, \hat{x}) \leftarrow \mathcal{S}(\tilde{x}, \hat{x}) \cdot (x_j = f_j(\tilde{x}))$
  }
  $F \leftarrow (f_{t'+1}, \ldots, f_t)$
  return F
}

Figure 6.12: Procedure to obtain a function-set

use a BDD minimization procedure, which heuristically finds an implementation which has the smallest support (we use a simpler version of the algorithm in [46]); alternately, we could have used *bdd-generalized-cofactor* (due to [22]) or any of many other BDD minimization algorithms with respect to a don't care set (see [76]). Since we are dealing with very small functions, in our application, the choice of the heuristic does not matter. The minimized BDD is converted to a similar looking network whose nodes are multiplexors controlled by the variable of the corresponding BDD node. Once we find a function replacement for a latch we restrict the compatible set S by this function.

Choosing the core

Before selecting the function-set F, we need to choose a core to identify a set of latches that satisfy condition 6.1 so that we can remove these latches. A core, by definition (Definition 5.1), is closed under all inputs. Obviously, if a set of latches satisfies condition 6.1 for a choice of core, it also satisfies the condition for any subset of the core. As argued earlier most real designs have a single tSCC, and for such designs each choice of the core

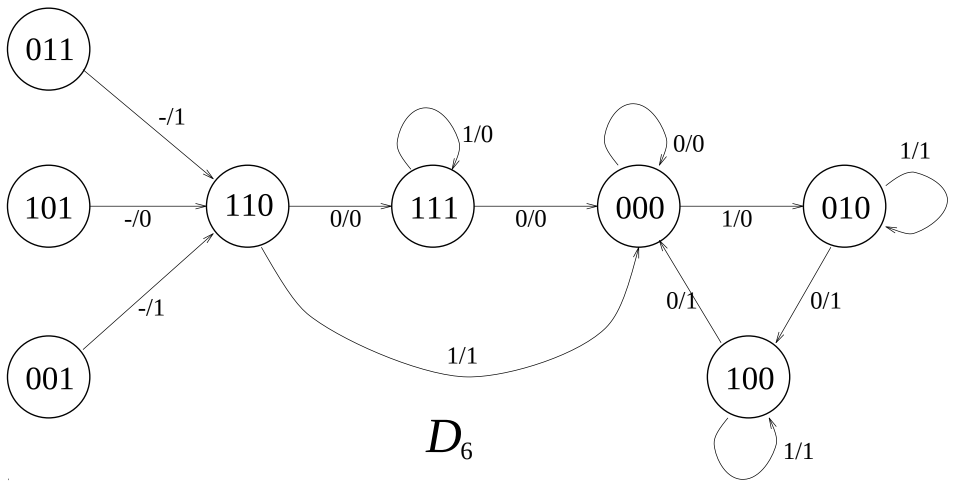

Figure 6.13: Design D_6.

must be a superset of the tSCC. Thus, it should seem that choosing the tSCC as the core should be the obvious choice since it allows us most flexibility in choosing the set of redundant latches. However, this choice may not be the right one for the following reason.

Consider design D_6 in Figure 6.13. Suppose we choose the tSCC $\{000, 010, 100\}$ as the core and the third latch x_3 as redundant. The procedure *compatible-set* identifies candidates to be representatives for their equivalence classes (000, 010 and 100 are already representatives for their respective equivalence classes 00, 01 and 10; we need to find candidates to represent the equivalence class 11). We observe that for all inputs, neither of the states 110 or 111 goes to equivalence class 00, 10 or 01 (the value of $\mathcal{P}(\vec{x})$ at iteration $j = 0$). This means that $(\mathcal{R}(\vec{x}) \cdot \overline{\mathcal{P}(\vec{x})} = 0)$ and the procedure returns FAIL. Thus, we are unable to replace latch x_3. On the other hand if we chose the outer envelope of the design (D_6^∞) $\{000, 010, 100, 111\}$ latch x_3 could be replaced by the function $f_3 = x_1 \cdot x_2$ since the procedure *compatible-set* returns $\mathcal{S}(\vec{x}) = \{000, 010, 100, 111\}$. In fact, we can prove that if we choose the outer-envelope of a design D as the core the procedure *compatible-set* never returns FAIL for any $n \geq m$, where m is such that $D^\infty = D^m$.

Theorem 6.12 *If we set $\mathcal{C}(\vec{x})$ to be the set of states in D^k, the k-cycle delayed design, and set n to be equal to k, then the procedure "compatible-set" does not return FAIL.*

Proof. We will prove the above by showing that after the **while** loop terminates, $\mathcal{P}(\vec{x}) = 1$. We show that by proving the following claim. In the following, we say that state $\vec{a}$ lies in the l-th onion ring if and only if $\vec{a} \in Q_{D^{k-l}} \setminus Q_{D^{k-l+1}}$ for $l \in \{1, 2, \ldots, k\}$; if $\vec{a} \in Q_{D^k}$, we say that $\vec{a}$ lies in the 0-th onion ring. It should be obvious that for any l such that $0 < l \leq k$, for any arbitrary input vector, any state in the l-th onion ring goes to a state which lies in the p-th onion ring for some $p < l$.

Claim: If state $\vec{a}(a_1, \ldots, a_t)$ belongs to the l-th onion ring, then after iteration $j = l$, $\mathcal{P}(\vec{x})$ contains $\tilde{a} = (a_1, \ldots, a_{t'})$.

We prove this claim by induction.

Base case. ($l = 0$): Since $\vec{a}$ belongs to D^k, it belongs to $\mathcal{C}(\vec{x})$ and hence $\tilde{a}$ belongs to $\mathcal{P}(\vec{x})$ when it is initialized at $j = 0$.

Induction step. Assume that the claim is true for any state which lies in the p-th onion-ring for some $p < l$. We will show prove the induction step for a state $\vec{a}$ which belongs to the l-th onion ring. Consider iteration $j = l$. Either, $\mathcal{P}(\vec{x})$ already contains $\tilde{a}$ before this iteration, in which case we are done. Or else, $\mathcal{P}(\vec{x})$ does not contain $\tilde{a}$ before this iteration. However, we know that for any arbitrary input vector, $\vec{a}$ transitions to a state $\vec{b}$ which lies in the p-th onion ring for some $p < l$. By our induction hypothesis, $\tilde{b}$ belongs to $\mathcal{P}(\vec{x})$ from the previous iteration. Thus, $\vec{a}$ must belong to the set $\mathcal{R}(\vec{x})$ computed at $j = l$. Hence, $\tilde{a}$ belongs to $\mathcal{P}(\vec{x})$ after iteration $j = l$. $\square$

This theorem shows that if the core is chosed as the outer-envelope and there is a non-empty set of candidate redundant latches, then those latches can be removed. On the other hand, in the following example, choosing the tSCC as the core lets us to replace a latch, whereas no latch can be removed if the outer-envelope is chosen. Consider the design D_5 in Figure 5.4. If we choose the tSCC $\{10, 11\}$ as the core, it can be seen that we can replace latch x_1 by the function $f_1 = 1$. However, if we choose outer-envelope $\{00, 01, 10, 11\}$, no latch is redundant.

In our experiments, we first chose the tSCC as the core (we use the procedure in Figure 5.7 to obtain the tSCC. If we are not able to replace the latches, choose the outer envelope as the core (we use the algorithm from Figure 5.6 with $n = \infty$). In fact, as we will see in the experimental results, for all except two examples, we were able to successfully use the tSCC as the core. For `s344` and `s349` (see Table 6.4), *compatible-set* returned FAIL in case we used the tSCC as the core; however, when we used the outer-envelope as the core, we were still unable to remove any latches, i.e. no set of latches satisfied Condition 6.1.

Selecting the set of redundant latches

Given a set $\mathcal{C}(\vec{x})$, Lin [49] has given a heuristic of choosing an order of latches to test if a latch is redundant to obtain the set of redundant latches to select the set of latches $\hat{x} = \{x_{t'+1}, \ldots, x_t\}$ which satisfy the latch redundancy condition (Condition 6.1). The heuristic, maximizes the number of latches chosen as redundant by ordering latches by decreasing unateness. The unateness of a variable x_i is the absolute value of the difference between the number of minterms in $\mathcal{C}(\vec{x})_{x_i=0}$ and $\mathcal{C}(\vec{x})_{x_i=1}$. The intuition is that the more unate a variable is, the less it contributes to distinguishing the states in $\mathcal{C}(\vec{x})$. We use this heuristic and then select the set of redundant latches by going through this order and adding a variable to the set if the current set still satisfies Condition 6.1.

Experiments

Our entire algorithm for replacing latches with combinational logic is shown in Figure 6.14. We used ∞ as the value of n to pass to *compatible-set* because in all examples, the **while** loop terminated in less than 1000 iterations. Thus we obtain n-delay replacements, where $n < 1000$. It is usually safe to assume that more than 1000 cycles are allowed before circuit operation starts (for example, for a 20MHz design, 1000 clock cycles amounts to 0.05ms).

We implemented this algorithm in SIS and experimented with the same ISCAS89 circuits that we have used earlier in Sections 5.1.4 and 6.2.1. The results appear in Table 6.4. We preprocessed the circuits with the same commands as before: `sweep; eliminate -1`. We choose the subset of circuits from Table 5.1 where the number of states in the tSCC were at most half the total number of states. We see from table 6.4 that for most the circuits we tried, we are able to remove some latches. Also, the size of the function-set F is usually very small (except for `s953`), and the total final size (of the combinational part of the circuit) is smaller than the original size because the addition of F was more than offset by the removal of the fan-in logic of the latches (Figure 6.8).

So far we have removed latches and replaced them by a combinational function of other latches. It seems very conceivable that such an operation would increase the delay of the paths between the latches. This intuition is based on the observation that a sequential path between two latches in the original design may become a purely combinational path due to the replacement of the latch on the path; thus, there might be combinational paths

```
procedure latch-replacement (input: network in terms of (i⃗, x⃗, o⃗, y⃗)) {
    /* let y_j denote the BDD for the j-th next state variable */
    /* let o_k denote the BDD for the k-th output variable */
    for (each node p in the network)
        Compute a BDD S_p(i⃗, x⃗) representing node p in terms of (i⃗, x⃗)
    T(i⃗, x⃗, y⃗) ← (∏_{j=1}^{t} (y_j ≡ S_{y_j}(i⃗, x⃗))) · (∏_{k=1}^{n} (o_k ≡ S_{o_k}(i⃗, x⃗)))
    C(x⃗) ← tSCC(T(i⃗, x⃗, y⃗))
    x̂ ← ∅; B(x⃗) ← C(x⃗)
    foreach (latch x) do
        if (∀x : [B(x⃗)] = 0) {
            x̂ ← x̂ ∪ {x}
            B(x⃗) ← ∃x : [B(x⃗)]
        }
    x̃ ← x⃗ \ x̂
    A(x⃗) ← compatible-set(C(x⃗), x̂, T(i⃗, x⃗, y⃗), ∞)
    Let n ← j, if the loop in compatible-set() terminates in the j-th iteration
    F ← greedy-function-set(A(x⃗), x̃, x̂)
    Replace latches x̂ by an implementation of F
    Recursively sweep away dead logic which fans out to nowhere
    return (optimized network, n)
}
```

Figure 6.14: Procedure to replaces latches with logic

Ckt.	Orig. size	Final circuit				
		Latches removed	Delay n	Size of F	Total size	Time
s298	150	2	7	4	130	1.26
s344	156	0	–	–	156	5.20
s349	160	0	–	–	160	5.33
s382	176	3	101	6	160	35.08
s386	204	0	–	–	204	0.21
s400	184	3	101	6	166	33.32
s444	184	3	101	6	166	33.09
s526	283	2	667	4	263	44.63
s641	199	5	2	9	172	3.81
s713	204	5	2	9	175	4.33
s953	489	9	1	96	522	12.77
s1196	618	0	–	–	618	11.80
s1238	690	0	–	–	690	9.14

Table 6.4: Experimental results for latch replacement. For `s344` and `s349`, *compatible-set* returned FAIL when passed the tSCC.

in the new designs which are longer than all combinational paths in the original design. To test this hypothesis that our latch replacement algorithm achieves area optimization only at the expense of performance, we performed the following experiment. We ran the SIS script `script.rugged`[2] (to do technology independent optimization) followed by a technology mapping step using the MCNC91 library. The results appear in Table 6.5.

We see that, for this subset of circuits, we get an average of 10.4% (and a maximum of 18.0%) improvement in mapped area and even an average of 3.7% (and a maximum of 23.6%) improvement in longest static delay, which goes against the intuition that latch replacement will increase the delay of the circuit. The only circuit where latch replacement seems to have caused longer delay in `s526`. Note that the CPU time with latch replacement includes the time taken to remove redundant latches. It is worth noting that the total CPU time with latch removal is actually lower than the total CPU time without this option! This is probably because logic optimization becomes simpler once we have fewer latches (and thus fewer state variables). In particular, reachability computations become much faster.

[2] In the last step of `script.rugged`, we used the sequential optimization procedure described in Section 6.2.1 instead of `full_simplify` because we wanted to exploit the sequential flexibility in addition to the combinational flexibility.

Ckt.	With latch replacement			No latch replacement			Ratios	
	Mapped area	Delay	CPU time	Mapped area	Delay	CPU time	Area	Delay
s298	278	17.6	8.49	312	19.1	8.83	0.891	0.921
s382	428	21.6	55.62	472	21.6	168.30	0.907	1.000
s400	419	27.0	53.76	460	26.9	162.71	0.911	1.004
s444	419	27.0	57.10	458	27.0	178.67	0.915	1.000
s526	456	22.4	108.22	460	18.0	73.24	0.991	1.245
s641	378	28.1	30.29	460	36.5	34.91	0.822	0.770
s713	377	27.9	31.10	460	36.5	42.04	0.820	0.764
s953	940	22.8	70.68	1012	20.8	75.85	0.929	1.096

Table 6.5: Latch replacement results after technology mapping.

Overall, for this set of circuits, we have demonstrated that replacing redundant latches reduces number of latches, the size of mapped circuits and even the delay for these circuits.

6.2.3 Final Results

Here, we analyze the combined effect of the algorithms in the previous two sections. We wish to compare technology mapped circuits after using our optimizations versus mapped circuits obtained using existing combinational optimization techniques. For the experiment we first remove redundant latches by using the algorithm in Section 6.2.2. Then we run the SIS script `script.rugged` on this circuit followed by sequential optimization described in Section 6.2.1. At this point we map the circuit to the MCNC91 library of gates and latches. We compared this mapped circuit against the circuit we would get by rjust running `script.rugged` followed by the technology mapping. The results appear in Table 6.6.

These experiments show that we get an average of 7.0% (and a maximum of 19.2%) area optimization over all circuits. More suprisingly, we also see an an average improvement of 4.5% (and a maximum of 23.6%) in the static delay of the the circuits, even though this was not our targeted objective. The CPU times are reasonable compared to those for the existing combinational optimization routines in SIS, and we can expect that the times can be reduced further with a careful tuning of the code.

Ckt.	With delay replacement			Combinational optimization			Ratios	
	Mapped area	Delay	CPU time	Mapped area	Delay	CPU time	Area	Delay
s27	48	10.9	1.3	48	10.9	1.1	1.000	1.000
s298	278	17.6	8.5	344	8.2	18.8	0.808	0.936
s344	355	28.8	31.2	373	33.4	9.2	0.952	0.862
s349	357	28.8	31.6	377	33.4	9.4	0.947	0.862
s382	428	21.6	55.9	472	20.8	10.9	0.907	1.038
s386	262	27.2	9.5	283	27.5	8.1	0.926	0.989
s400	419	27.0	53.8	461	26.9	10.4	0.909	1.004
s444	419	27.0	57.1	462	28.5	10.5	0.907	0.947
s510	513	28.5	42.1	493	30.7	23.2	1.041	0.928
s526	456	22.4	108.2	514	18.6	15.0	0.887	1.204
s641	378	28.1	30.3	460	36.5	12.9	0.822	0.770
s713	377	27.9	31.1	460	36.5	13.3	0.820	0.764
s820	581	19.9	79.7	573	20.8	38.5	1.014	0.957
s832	535	21.7	80.0	544	21.4	32.7	0.983	1.014
s953	940	22.8	70.7	1030	22.0	43.7	0.913	1.036
s1196	1152	29.3	403.7	1172	29.4	195.0	0.982	0.997
s1238	1131	29.4	342.6	1140	32.6	102.3	0.992	0.902
s1488	1173	18.9	299.1	1208	19.0	94.2	0.971	0.995
s1494	1139	20.1	209.4	1216	19.5	112.8	0.937	1.031

Table 6.6: Delay replacement results after technology mapping.

6.3 Verification for Delayed Replacements

To verify than a design D is an n-delay replacement of design C, we need to verify $D^n \preceq C$. We could compute D^n and use the verification algorithm for safe replacement ($\preceq$) from Section 5.2.2 to decide this. Recall that using the algorithm there gives a complexity of $O(p \cdot 2^q)$, where p is the number of states in D^n and q is the number of states in C. However, for all designs we have encountered so far, D^∞ is identical to D^n for some $n < 1000$, and since we have assumed before that if we require n-delay replacement, n would rarely be this small. Thus, in practice, it should suffice to check $D^\infty \preceq C$ as long as we check that $D^\infty = D^n$ for some small n. However, using Proposition 6.10, we know that this is equivalent to checking $D^\infty \preceq C^\infty$. If the outer-envelopes of the two designs coincide with their respective tSCCs, as is the case with many designs as evident from Table 5.1, we just need to check for safe replacement between the two tSCCs. From Theorem 4.5 in Chapter 4, this safe replacement is true if and only if every state in one tSCC is equivalent to some state in the other tSCC and vice-versa. Thus, we have a verification procedure which is polynomial in the number of states of the two designs, as opposed to an exponential procedure for the problem of verification for safe replacement.

In general, it is easy to see that the problem of verification for delay replacement (even for $n = \infty$) is as hard as the problem for verification for safe replaceability (which we showed was PSPACE-complete in Section 5.2.1). This is easily seen by the fact that if we are given an instance of the safe replaceability problem (is $D \preceq C$?) we automatically convert it to an equivalent ∞-delay replaceability problem (is $D_1^\infty \preceq C_1$?) by a simple modification. We pick any arbitrary output vector a, introduce a new input vector b, and add a self loop for input b with output a for each state of D and C to get D_1 and C_1 respectively. D_1 and C_1 have the same number of states as D and C, respectively. It should be obvious that $D \preceq C$ if and only if $D_1 \preceq C_1$. However, since every state in D_1 has a self loop, D_1^∞ is the same as D_1. Thus, if we could solve the delay replaceability problem, that we would give us an answer to the safe replaceability problem too.

We argued earlier in this section that if D^∞ and C^∞ are tSCCs, we can verify delay replaceability in polynomial time. An open question is the complexity of verification if D^∞ and C^∞ are "almost" tSCCs. In other words, if D^∞ has d_1 states inside its tSCC and d_2 states outside, and C^∞ has c_1 states inside its tSCC and c_2 states outside, is it possible to construct an algorithm to verify the question "is $D^\infty \preceq C^\infty$?" in time polynomial in

$(d_1 + d_2 + 2^{c_1} + c_2)$. Such an algorithm would be very useful for the verification of delay replacement where almost all of the states in the outer-envelope are also inside the tSCC, i.e. there are at least exponentially more states inside the tSCC than those which lie inside the outer envelope but not in the tSCC.

Chapter 7

Validity of Retiming Transformations

Retiming was first formulated by Leiserson and Saxe [45] in the context of systolic systems. When applied to digital circuits, retiming is an optimization step which moves the latches across the logic gates and in doing so changes the number of latches and the longest path delay between the latches. In this manner, the number of latches can be reduced, and/or the cycle time of the circuit can be improved. Recent results by Shenoy and Rudell [75] have improved the efficiency of retiming so that circuits up to 50,000 equivalent gates can be retimed for minimum area under a delay constraint. This has sparked further interest in exploring the application of retiming as a general optimization step during logic synthesis.

Prior literature has assumed that retiming can be directly applied to sequential circuit optimization. However, we had earlier shown in Section 3.6 that retiming does not satisfy the safe replacement condition (Definition 4.1). Note that the theorem of Leiserson and Saxe on the validity of retiming is not in doubt. They simply made the assumption that the environment of the circuit could be modified to wait a fixed number of cycles (dependent upon the retiming) before applying its inputs. It is this requirement which violates safe-replacement and casts a doubt on whether retiming is valid as part of a synthesis methodology for net-list level sequential circuits.

In this chapter, we study effect of retiming transformations with respect to a simulator that a designer might use to analyze the designs. We first show in Section 7.1 how a designer who applies retiming to a circuit could be surprised during simulation when the retimed design replaces the original design. Specifically, it is possible that a logic simulator could produce a different output sequence when simulating the retimed circuit. We also show on the same example that an input sequence which tests a fault in a circuit cannot test the same fault in the retimed circuit. In Section 7.2 we formalize our model of retiming on sequential circuits, which is slightly different from that of Leiserson and Saxe in that we distinguish the case where a latch moves from the output of one gate to the inputs of each of its fanouts. Then in Section 7.3 we establish a tighter bound in in the Leiserson and Saxe result that retiming is a safe operation as long as one can wait long enough after power-up before applying the input sequence. As a consequence of our proof, we identify that the only incorrect retiming transformation is one which moves a latch forward across a multiple-fanout junction; i.e., if we limit the retiming transformations, then retiming satisfies the condition of safe-replacement. For Section 7.4, we define a **conservative three-valued logic simulator** (CLS) as a three-valued simulator using the values '0', '1', and 'X' which performs only local propagation of the X values (i.e., $0 \cdot X = 0$ but $1 \cdot X = X$).[1] Further, the CLS begins operation with all latches in the X state. We show that while a simulator could distinguish a retimed circuit from the original, in fact, a conservative three-valued logic simulator cannot. In other words, retiming retains an invariant on the output sequences produced by conservative three-valued simulation.

As a final comment, note that our model of a synchronous circuit does not require a latch to have a set or reset line and does not require any notion of the initial state of the circuit. Hence, we avoid the problem pursued by Touati and Brayton [81] in *retiming the initial state.*

7.1 Retiming Violates Safe Replacement

Here we show how a simple retiming move might change the behavior of a design with respect to the output of a simulator. Consider the circuit D and the retimed version C in Figure 7.1. The STGs for these circuits are shown in Figure 7.2. Design D has two states and is initialized to state 0 on the length-1 input sequence 0, whereas C is not initialized with this input sequence.

If we simulate the two design D and C with an input sequence, we may get

[1] In contrast, an exact three-valued simulator will output an X only in the case that some assignment of values to X on the input yield differing outputs; for a conservative simulator the local propagation of X's may cause an X on the output even though the output is a 0 (or a 1) for every input.

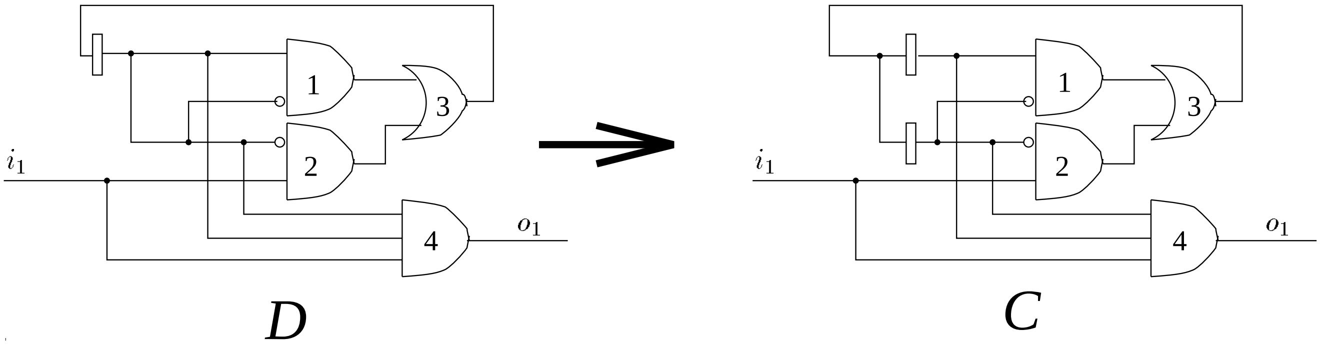

Figure 7.1: Retimed circuit is not initialized with input sequence 0.

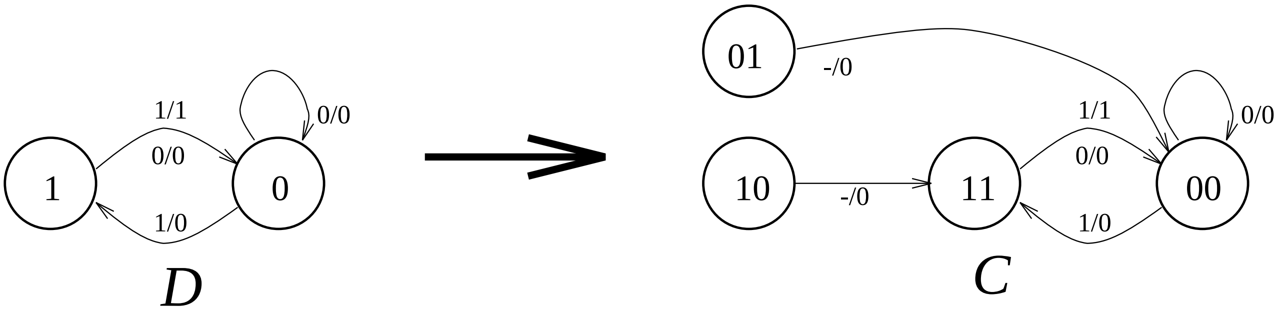

Figure 7.2: Design where retiming breaks down an initializing sequence of length 1.

power-up state of D	output sequence
0	$0 \cdot 0 \cdot 1 \cdot 0$
1	$0 \cdot 0 \cdot 1 \cdot 0$

power-up state of C	output sequence
00	$0 \cdot 0 \cdot 1 \cdot 0$
11	$0 \cdot 0 \cdot 1 \cdot 0$
01	$0 \cdot 0 \cdot 1 \cdot 0$
10	$0 \cdot 1 \cdot 0 \cdot 1$

Table 7.1: Simulation results for D and C on input sequence $0 \cdot 1 \cdot 1 \cdot 1$.

different results. Consider the input sequence $0 \cdot 1 \cdot 1 \cdot 1$. The simulation results for this input sequence from all the power-up states of D and C are shown in Table 7.1. This input sequence produces the same output sequence from every power-up state of D. However, if design C powers up in state 10, it will output $0 \cdot 1 \cdot 0 \cdot 1$, resulting in an input/output behavior which was not present in the original design.

Suppose we had a sufficiently powerful simulator which given any input sequence, outputs

- a 1 at an output at some time step iff all power-up states output 1 at that time step
- a 0 at an output at some time step iff all power-up states output 0 at that time step
- an X otherwise, i.e. if there exist two power-up states, one of which outputs a 0 and the outputs a 1.

For the input sequence $0 \cdot 1 \cdot 1 \cdot 1$ this simulator would output $0 \cdot 0 \cdot 1 \cdot 0$ for design D and output $0 \cdot X \cdot X \cdot X$ for the design C.

Note, however, that if we clock the circuit for one redundant cycle (with arbitrary input) before applying our input sequence, even our imaginary powerful simulator will produce the same output for both circuits. This is the sense of a *delayed circuit* which we use in Section 7.3 and is the notion of equivalence used by Leiserson and Saxe when proving that retiming was a valid transformation on a circuit.

7.1.1 Testing Example

The example in the previous subsection shows that retiming can change the behavior of a design as measured by a simulator. Marchok *et al.* [55] were the first to study the effect of retiming transformations on test sequences for single stuck-at faults. However, we show next that retiming may cause test sets to also change, contradicting their result:

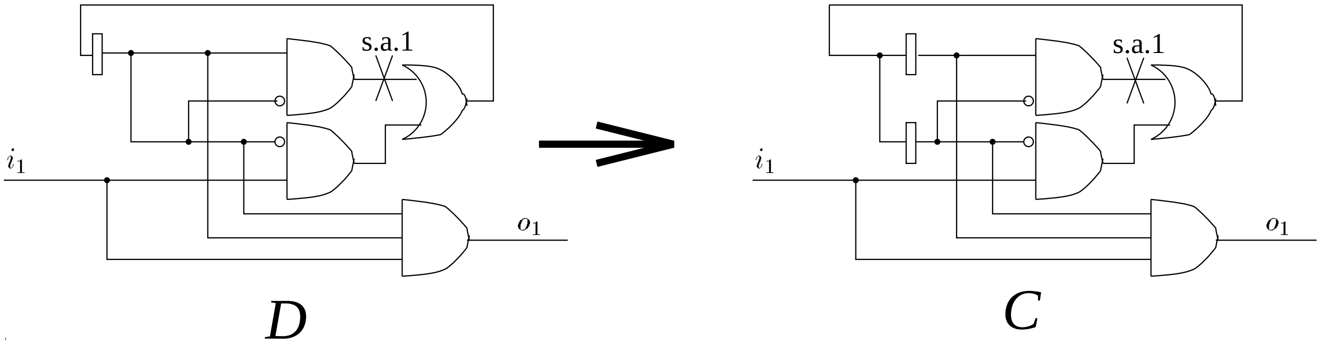

Figure 7.3: Retiming does not preserve test sequence $0 \cdot 1$.

(Theorem 1 in [55]) *The retiming transformation preserves testability with respect to a single stuck-at-fault test set.*

This theorem implies that if a test sequence uncovers a given stuck-at-fault in a circuit, then the same sequence can detect the same faults in a retimed version of the circuit. For a counterexample, consider circuits D and C in Figure 7.3. For the given stuck-at-1 fault shown, the test sequence $0 \cdot 1$ detects the fault[2] in the original circuit D. For the input sequence $0 \cdot 1$, the fault-free version of D produces the output $0 \cdot 0$ from all power-up states whereas the faulty version of D produces the output sequence $0 \cdot 1$. Thus, input sequence $0 \cdot 1$ is a valid test sequence for the stuck-at-fault in D since it distinguishes the faulty design from the fault-free design. However, for the fault-free version of circuit C, the input sequence $0 \cdot 1$ may produce output $0 \cdot 0$ or $0 \cdot 1$ depending on which state C powers up in (see the STG for C in Figure 7.2); the faulty version of C still produces $0 \cdot 1$ from any power-up state. Thus, $0 \cdot 1$ is no longer a test sequence for the retimed design D.

7.2 Background

7.2.1 Leiserson-Saxe Retiming Model

Leiserson and Saxe introduced retiming [45] through a graph-theoretic model. A design is modeled as a finite edge-weighted directed graph $G = (V, E)$. Each vertex in V represents either a gate in the design, a primary input or output, or a special dummy node called the *host*. There is an edge in E from one gate to another if an output of this gate

[2]We should note here that if were using a conservative three-valued fault generator, we would not generate this $0 \cdot 1$ as a test sequence. However, as noted by [16], that is a shortcoming of such a test generator, and in no way disqualifies $0 \cdot 1$ to be a test sequence.

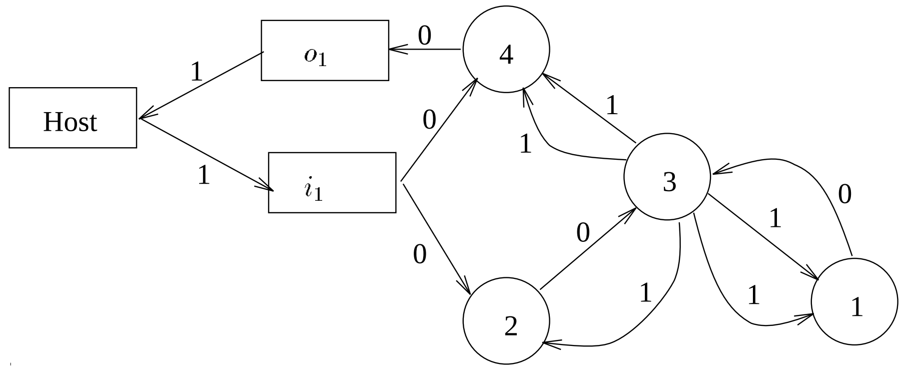

Figure 7.4: The edge-weighted digraph representing the circuits D and C from Figure 1.

fans out to the second gate; there is an edge from the host to each primary input node; and, an edge from each primary output node to the host. The non-negative weight of an edge represents the number of latches on the corresponding path in the design. A retiming of a design is an assignment of each vertex v to an integer $lag(v)$ such that for every edge (u, v) with weight w, the value $w + lag(v) - lag(u)$ is non-negative; additionally, the host and primary input and primary output nodes are required to have a lag of 0. Informally, the lag of a vertex denotes the number of backward retiming moves across this vertex, for example a lag of -2 on a logic element means that 2 forward retiming moves are performed across this element.

This model is not well suited for retiming on gate-level sequential circuits. The problem is that if a single output of an element fans out to more than one element, this model does not distinguish where the latches are placed with respect to the fanout junction. This is best seen with respect to the retiming example discussed in Section 7.1. Both the circuits in Figure 7.1 are represented by the same retiming graph, which is shown in Figure 7.4.

7.2.2 Circuit Model

Our model of a sequential circuit is the traditional net-list level model which consists of elementary cells from a library interconnected with wire connections (for example, the circuits in Figure 7.1). The library cells consist of combinational gates and latches. As discussed in Section 2.1, if some latches are reset latches, we model them with no-reset latches and explicit reset circuitry using the *reset transformation*. Later, in Chapter 8, we show that this tranformation does not restrict the possible set of retiming moves.

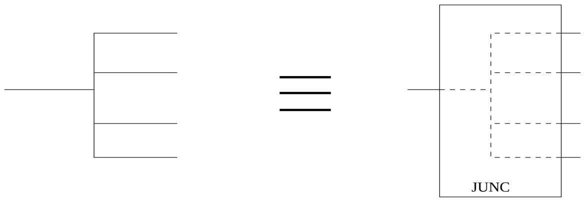

Figure 7.5: A junction can be treated as a multi-output gate.

The reason retiming caused a problem in the example of Section 7.1 is that we retimed a latch forward across a fanout junction and created a power-up state which could not occur in the original circuit. If there are multiple-output gates in the cell library such that there exist output vectors of the cell which cannot be produced by any input vector to the cell, then retiming latches forward across such elements leads to the same problem as retiming latches forward across fanout junctions.

This motivates the definition of justifiable and non-justifiable multiple-output gates. Consider a multi-output gate F with m inputs and n outputs. The n output functions are denoted by $f_1, \ldots, f_n$. F is *justifiable* if and only if for every output $y \in \{0,1\}^n$, there exists an input $x \in \{0,1\}^m$ such that $y = F(x)$; if there exists $y \in \{0,1\}^n$ such that for all $x \in \{0,1\}^m$, $y \neq F(x)$, then F is *non-justifiable*.

A k-way fanout junction ($k > 1$) is a special case of a multi-output gate (which we call JUNC) with 1 input line x and k ($k > 1$) output lines $y_1, y_2, \ldots, y_k$, where $y_1 = \cdots = y_k = x$ (Figure 7.5). The element JUNC is clearly non-justifiable since only two of the 2^k output vectors ($000\ldots0$ and $111\ldots1$) are possible. For the remainder of this paper we assume that all fanout junctions are replaced by JUNC elements. This implies that each output of each gate (latch) fans out to exactly one other gate (latch).

For a gate-level network which replaces junctions with multi-output gates as described above, there are two kinds of atomic retiming moves: *forward* and *backward*. A forward move removes one latch from each of the n inputs and places one latch at each of the m outputs; a reverse move removes one latch from each of the m outputs and places one latch at each of the n inputs (Figure 7.6). We view retiming as starting from an initial circuit and applying a sequence of these atomic moves to result in the retimed circuit.

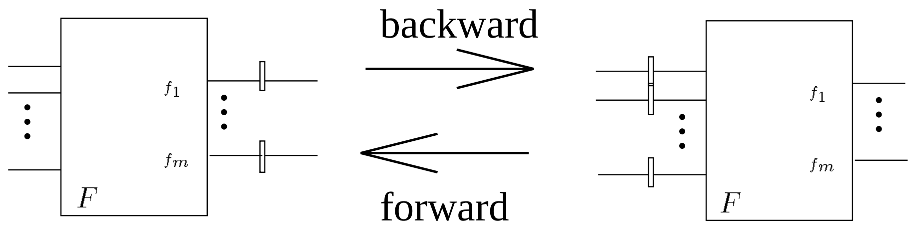

Figure 7.6: Forward and backward retiming moves across a multi-output element.

7.2.3 Notions of Replaceability

Here we discuss the various notions of design replacement that are relevant to study the effect of retiming transformations.

The notion of *safe replaceability* was presented in Chapter 4. A design C is a *safe replacement* for a design D (denoted by $C \preceq D$) if for any state s_1 in design C and any input sequence, there exists a state s_0 in design D such that the output behavior from s_1 is the same as that from s_0 on that input sequence. We showed that the above condition is necessary and sufficient so that the replacement cannot be detected by any environment that can only control and observe only the primary inputs and outputs, respectively, of the design.

A stronger notion of design replacement is the classical notion of state machine implication usually defined in the context of state machine equivalence. A design C *implies* design D (we denote that by $C \subseteq D$) if for any state s_1 in design C there exists a state s_0 in design D such that s_1 is equivalent to s_0 (i.e., on any input sequence, the output behavior from s_1 is the same as that from s_0).

The difference between $\preceq$ and $\subseteq$ lies in the fact that for the former the state s_0 in D depends not only on s_1 but also on the input sequence, whereas for $\subseteq$ s_0 only depends on s_1 and is the same for any input sequence. Recall from Remark 4.2 (on page 34) that if $C \preceq D$, there may be a state in C which is not equivalent to any state in D. Thus, $\subseteq$ is a strictly stronger notion than $\preceq$.

In the following sections we prove results which characterize the conditions under which the relation $\subseteq$ holds between two designs. The following proposition shows that these results automatically imply safe replaceability ($\preceq$) as well.

Proposition 7.1 *If $C \subseteq D$, then $C \preceq D$.*

Proof. Consider any state $s_1 \in C$. Since $C \subseteq D$, there exists a state $s_0 \in D$ which is equivalent to s_1. Thus, given s_1, for any input sequence π, there is a state in D (namely s_0) such that s_1 and s_0 output the same sequence on π. $\square$

7.3 Safety of Retiming Moves

In this section we classify retiming moves into those which are strictly safe for safe-replacement, and those which are safe only under the assumption of a delayed design.

There are four kinds of atomic retiming moves: {backward, forward} across a {justifiable, non-justifiable} element.

Proposition 7.2 *If design C can be obtained from design D by a single retiming move which is either a backward move or across a justifiable element, then $C \subseteq D$.*

Proof. There are two cases for retiming across the element F:

(i) **(backward retiming moves)**. Assume that design D has latches $l_1, l_2, \ldots, l_k$. Let the latches $l_1, l_2, \ldots, l_n$ be retimed to latches $l'_1, l'_2, \ldots, l'_m$. Consider any state in design C, say $s_1 \equiv [(l'_1, l'_2, \ldots, l'_m, l_{n+1}, \ldots, l_k) = (Y'_1, Y'_2, \ldots, Y'_m, Y_{n+1}, \ldots, Y_k)]$. Let $\vec{Y'}$ denote $(Y'_1, Y'_2, \ldots, Y'_m)$. We claim that state $s_0 \equiv [(l_1, l_2, \ldots, l_k) = (f_1(\vec{Y'}), f_2(\vec{Y'}), \ldots, f_m(\vec{Y'}), Y_{n+1}, \ldots, Y_k)]$ in design D has the same input-output behavior as s_1. The proof of this claim is by induction on the length of an input sequence. Outside of the retimed area, the two circuits D and C are identical. On the first clock cycle, the local output of F is $(f_1(\vec{Y'}), f_2(\vec{Y'}), \ldots, f_n(\vec{Y'}))$ in both designs D and C. Suppose the local outputs are identical upto the k-th input vector. Then the k-th input vector $\vec{W}$ reaching the retimed area will be identical for both designs. Now, the local outputs of the retimed area in both designs is $(f_1(\vec{Y'}), f_2(\vec{Y'}), \ldots, f_n(\vec{Y'}))$. Since we have shown that the local outputs are equal in both designs, and outside the retimed area the designs are identical, the two primary outputs of the two designs are also equal. Thus, s_0 and s_1 are equivalent.

(ii) **(forward retiming move across a non-justifiable element)**. Assume that design D has latches $l_1, l_2, \ldots, l_k$. Let the latches $l_1, l_2, \ldots, l_m$ be retimed to latches $l'_1, l'_2, \ldots, l'_n$. Consider an arbitrary state in design C, say $s_1 \equiv [(l'_1, l'_2, \ldots, l'_n, l_{m+1}, \ldots, l_k) = (Y'_1, Y'_2, \ldots, Y'_n, Y_{m+1}, \ldots, Y_k)]$. Since the logic element is justifiable, there must exist an input vector $\vec{Z} = (Z_1, \ldots, Z_m)$ such that for each $i \in \{1, \ldots, n\}$: $f_i(\vec{Z}) = Y'_i$. Now, consider the state $s_0 \equiv [(l_1, l_2, \ldots, l_k) = (Z_1, \ldots, Z_m, Y_{m+1}, \ldots, Y_k)]$ in design D. Using an induction argument, similar to the last case, on the length of the input sequence, we can easily show that the two states s_0 and s_1 are equivalent.

Thus, for both the cases, for every state in C there is a state in D which is equivalent to it. □

Since $\subseteq$ is clearly transitive, it follows that if we disallow forward retiming moves across non-justifiable elements the retimed design is a safe replacement of the original design:

Corollary 7.3 *If C can be obtained from D using an arbitrary sequence of retiming moves, none of which is a forward retiming move across a non-justifiable element, then $C \subseteq D$.*

In the following, we will use the notion of delayed designs (from Definition 6.1) to study the effect of forward retiming moves. The notion of delayed design is similar to the notion of *sufficiently old configuration* used in [45] to show the validity of retiming moves.

Proposition 7.4 *If design C can be obtained from design D by a single forward retiming move across a non-justifiable element, then $C^1 \subseteq D$.*

Proof. Assume that the original design has latches $l_1, l_2, \ldots, l_k$. Let the latches $l_1, l_2, \ldots, l_m$ be retimed to latches $l'_1, l'_2, \ldots, l'_n$. Let the latches $l_1, l_2, \ldots, l_n$ be retimed to latches $l'_1, l'_2, \ldots, l'_n$. Consider any state in design C^1, say $s_1 \equiv [(l'_1, l'_2, \ldots, l'_n, l_{m+1}, \ldots, l_k) = (Y'_1, Y'_2, \ldots, Y'_n, Y_{m+1}, \ldots, Y_k)]$. Since the design C^1 represents the design C after it has been clocked through for 1 cycle, there must exist a vector $\vec{Z} = (Z_1, \ldots, Z_m)$ such that for each $i \in \{1, \ldots, n\}$: $f_i(\vec{Z}) = Y'_i$. The rest of the proof is identical to that of case (ii) in Proposition 7.2. □

The Propositions 7.2 and 7.4 lead to the following corollary which is the primary correctness result proven by Leiserson and Saxe [45]:

Corollary 7.5 (Lemma 1 in [45]) *If C can be obtained from D using an arbitrary sequence of retiming moves, then there exists a non-negative finite integer k such that $C^k \subseteq D$.*

This result requires up to delay using the retimed design for a finite number of cycles after power-up. Notice our short proof of the above lemma; in contrast, the proof given in [45] is about 4 pages long. However, in the proof of Lemma 1 in [45], the integer n was shown to be the equal to $\max_{v \in V}(-lag(v))$, i.e. the maximum number of forward retiming moves across any combinational element in the circuit (since $lag(host)$ is 0, n is well-defined for a given retiming). We can obtain an improvement over that bound by making it independent of the number of forward retiming moves across any *justifiable* elements:

Theorem 7.6 *If C can be obtained from D using a sequence of retiming moves such that there are no more than k forward retiming moves across any non-justifiable element, then $C^k \subseteq D$.*

The above theorem is a simple corollary of:

Lemma 7.7 *If a design D_n can be obtained from design D_0 after n successive atomic retiming moves such that for any non-justifiable element there is at most one forward retiming move, then $D_n^1 \subseteq D_0$.*

Proof. Let D_0 be retimed n times successively to obtain $D_1, \ldots, D_n$ (i.e. the i-th retiming step transforms D_{i-1} to D_i). Of these n moves, the total number of forward retiming moves across any non-justifiable logic element does not exceed 1. Choose an arbitrary state $t_n \in D_n^1$. We will show that for all $i \in \{0, \ldots, n-1\}$, there exists a state $t_i \in D_i$ such that t_n is equivalent to t_i; thus $D_n^1 \subseteq D_i$, for all $i \in \{0, \ldots, n-1\}$.

We will label each net of each D_i (since each junction has been replaced by a JUNC, each net has exactly one fanout) with a 0, 1 or I. The labeling procedure starts by labeling D_n and proceeds inductively labeling D_{i-1} after D_i has been labeled. During the labeling procedure we prove by induction that the following two labeling properties are maintained:

(Labeling Property 1) A net in D_i is labeled I if and only if it lies on a combinational path from a latch output to an input of a non-justifiable logic element, such that of the j-th retiming moves (where $i \leq j \leq n$), one is a forward move across this element, and none are backward. We call such a path *inconsistent* (hence the label I). This property also implies that the input net to any latch will never be labeled I.

(Labeling Property 2) In any design D_i, if the labels on all inputs $\vec{x}$ of any combinational element F are in $\{0, 1\}$, then the labels on the outputs are $\vec{y} = F(\vec{x})$.

Labeling D_n: Let $D_0, \ldots, D_n$ each have l primary inputs. Consider the arbitrarily chosen state $t_n \in D_n^1$. Since D_n^1 consists of states reachable from D in one clock cycle, there exists a state $t_n' \in D_n$ and an input vector $\vec{i}$ in the input space $\{0, 1\}^l$ such that $\vec{i}$ drives

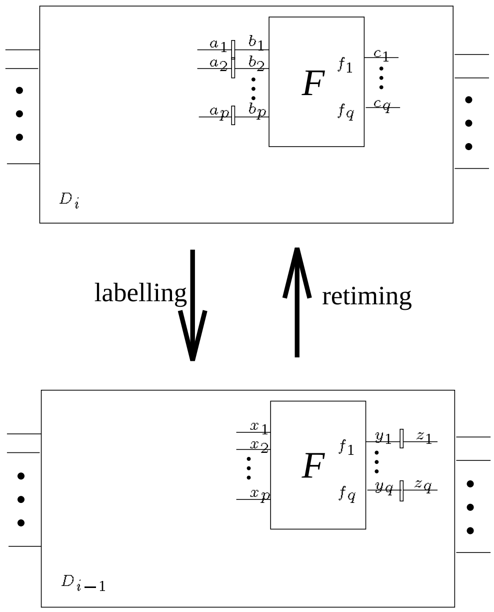

Figure 7.7: Labeling for a backward retiming move.

t'_n to state t_n. Now, set the values of the latches in D_n to correspond to the state t'_n and let the primary inputs of D_n be set to $\vec{i}$. The label of any net in D_n is the value that wire takes during the first clock cycle if $\vec{i}$ is applied to of D_n when it is in state t'_n. No net is labeled I. Clearly Labeling Properties 1 and 2 hold.

Labeling D_{i-1}**:** The labeling procedure starts with D_n and proceeds towards D_0 labeling D_{i-1} relative to D_i. Assume that D_i has been labeled and that the Labeling Properties hold for it. Let the i-th retiming move be across an element with p inputs and q outputs implementing a function $F = (f_1, \ldots, f_q)$. Outside of the retimed area, the nets of D_{i-1} inherit the labels from D_i; in the following we describe how to label the nets affected by the retiming move, based on type of the i-th retiming move:

(a) **Backward retiming across a logic element** (Figure 7.7). Since Labeling Property 1 holds in D_i, then for each j: $a_j \in \{0, 1\}$. For each j, label $x_j = a_j$. For each k, label $y_k = f_k(a_1, a_2, \ldots, a_p)$, and $z_k = c_k$. Thus, Labeling Property 2 holds for D_{i-1}.

For D_{i-1}, none of the y_k's lies on an inconsistent path, and since the i-th retiming move is a backward move across F, none of the x_j's lies of an inconsistent path. Finally, z_k lies on an inconsistent path if and only if c_k lies on one. So, Labeling Property 1 also holds for D_{i-1}.

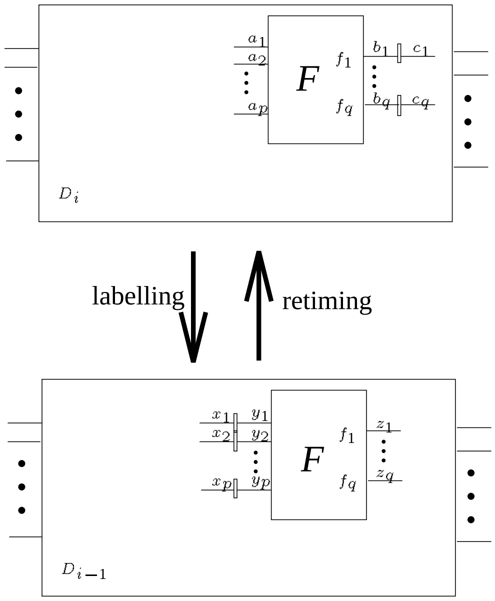

Figure 7.8: Labeling for a forward retiming move.

(b) **Forward retiming across a justifiable logic element** (Figure 7.8). For each j, label $x_j = a_j$, and for each k, label $z_k = c_k$. If any $c_k = I$, then for each j, label $y_j = I$. Otherwise, $c_k \in \{0,1\}$ for all k and since the element is justifiable, there exists $(d_1, \ldots, d_p)$ such that for each k, $f_k(d_1, \ldots, d_p) = c_k$. In this case, for each j, label $y_j = d_j$. Clearly, Labeling Property 2 holds for D_{i-1}.

In D_i, since F is justifiable and each output goes to a latch, F and any of its input/output nets cannot lie on an inconsistent path. Since Labeling Property 1 is true in D_i, then for each j: $a_j \in \{0,1\}$. For D_{i-1}, (a) no $x_j = I$, (b) each $y_j = I$ if and only if some $c_k = I$, (c) and $z_k = I$ if and only if $c_k = I$. Clearly, since Labeling Propery 1 holds in D_i, a net in D_{i-1} is labeled I if and only if it lies on an inconsistent path. So, Labeling Property 1 holds for D_{i-1}.

(c) **Forward retiming across a non-justifiable logic element** (Figure 7.8). For each j, label $x_j = a_j$ and $y_j = I$. For each k, label $z_k = c_k$. Labeling Property 2 holds trivially for D_{i-1}.

Since the i-th retiming move is a forward move across this non-justifiable element, and at most 1 forward move is allowed for each such element, then for each $j > i$, none of the j-th moves is forward across this element. Since Labeling Property 1 holds for

D_i, then no $a_j = I$, i.e. $a_j \in \{0, 1\}$. For D_{i-1}, (a) no $x_j = I$, (b) each $y_j = I$, (c) and $z_k = I$ if and only if $c_k = I$. Clearly, the labeling procedure for D_{i-1} labels a net I if and only if it lies on an inconsistent path. So, Labeling Property 1 holds of D_{i-1}.

Thus, by induction, Labeling Properties 1 and 2 hold for all the designs $D_0, \ldots, D_n$. It may be helpful to note that the labelings on D_i do not imply that nets can ever attain these values during a simulation for this design; rather, the net labelings are just a way of computing a set of equivalent states across $D_0, \ldots, D_n$. Let state $t_i \in D_i$ be the state corresponding to the assignment of each latch in D_i to the label of the input net to the latch (we know from Labeling Property 1 that this label is always 0 or 1). The fact that the net labelings may not be attainable for any simulation just means that state t_i may not be reachable from any state in D_i, <u>but</u> D_i can surely power up in any possible state, including t_i.

Next, analyzing the labeling procedure for D_i, we prove the following:

Claim: State t_{i-1} in D_{i-1} is equivalent to state t_i in D_i.

We prove the claim by showing that any primary input sequence to D_i starting from state t_i produces the same primary output sequence when applied to D_{i-1} from state t_{i-1}; also, the local output sequences (of the retimed area shown in Figures 7.7 and 7.8) in D_i and D_{i-1} are identical. The proof of the claim is by induction on the length of the output sequence. In the following, we abbreviate $(a_1, \ldots, a_p)$ by $\vec{a}$, $(b_1, \ldots, b_q)$ by $\vec{b}$, $(x_1, \ldots, x_p)$ by $\vec{x}$, and $(y_1, \ldots, y_q)$ by $\vec{y}$.

Base case (output vector on first cycle): For case(a) (Figure 7.7), for the subcircuit in D_i, the local output vector from state t_i on the first cycle is $(f_1(\vec{a}), f_2(\vec{a}), \ldots, f_q(\vec{a}))$, and for D_{i-1} the local output vector from state t_{i-1} is $(y_1, \ldots, y_q)$. But we know from the labeling procedure that $y_k = f_k(\vec{a})$. For cases (b) and (c) (Figure 7.8), for the subcircuit in D_i the local output vector from t_i on the first cycle is $(b_1, \ldots, b_q)$, and for D_{i-1} the local output vector from t_{i-1} is $(f_1(\vec{x}), \ldots, f_q(\vec{x}))$. But we know that $\vec{x} = \vec{a}$ and from Labeling Property 2, $b_k = f_k(\vec{a})$. Thus, for all three cases, both the subcircuits in D_i and D_{i-1}, starting from states t_i and t_{i-1} respectively, locally output the same vector on the first clock cycle. Since D_i and D_{i-1} are identical outside the retimed area, the primary output vectors for D_i and D_{i-1} are equal on the first clock cycle.

Induction step: Suppose the local outputs are identical upto the m-th clock cycle. Then the m-th local input vectors reaching the two subcircuits in D_i and D_{i-1} are identical, say $\vec{i} = (i_1, \ldots, i_p)$. For each of the three cases (a), (b) and (c), the local output vector at the $(m+1)$-th cycle is $(f_1(\vec{i}), \ldots, f_q(\vec{i}))$ for the sub-circuits in both D_i and D_{i-1}. Again since the designs are identical outside the retimed area, the primary output vectors for D_i and D_{i-1} are equal on the $(m+1)$-th clock cycle. $\square$

As an example of the labeling procedure in the above proof, consider $D_0, \ldots, D_6$ in Figure 7.9, where D_6 is obtained from D_0 after 6 atomic retiming moves. Starting with a labeling of the nets of D_6 we successively obtain the labelings for each of the other designs. Note that because the 4-th retiming move $D_3 \to D_4$ is a forward move across a junction, each of $D_0, \ldots, D_3$ contain nets labeled I. For $i \in \{0, \ldots, 6\}$, all states t_i's are equivalent to each other where t_i (in design D_i) corresponds to the latch values set to the labels on the input nets of the latches. Note that if we use the bound in the proof of the main theorem in Leiserson and Saxe (the same result as Corollary 7.5), that will show that $D_6^2 \subseteq D_0$ since there are two forward retiming moves across the AND-gate ($D_0 \to D_1$ and $D_4 \to D_5$); on the other hand, Lemma 7.7 gives us $D_6^1 \subseteq D_0$ since there is at most one forward retiming move across any fanout junction.

Tightening the Additional Power-up Delay

The decision to model only atomic retiming moves limits the tightness of the above result. However, we can easily tighten the above result by observing that any collection of interconnected combinational elements can be viewed as a single multi-output gate. It might be possible to combine one or more non-justifiable elements into a single justifiable element. Treating this larger element as a black box we can freely move latches back or forward across (but not inside) this element without causing additional power-up delay.

Theorem 7.6 implies, for example, that if a design has only single-output gates, then any number of retiming moves (forward or backward) across those gates is fine, but we must restrict the number of forward retiming moves across any fanout junction to be at most k for $C^k \subseteq D$ to hold.

Note that the maximum number of forward retiming moves across any gate can be bounded by the maximum number of latches in any simple cycle in the circuit.[3]

[3] Recall that the primary outputs of the circuit are connected to the *host* and the primary inputs are fed by the *host*; hence, cycles may include paths from the primary outputs through the host to the primary inputs.

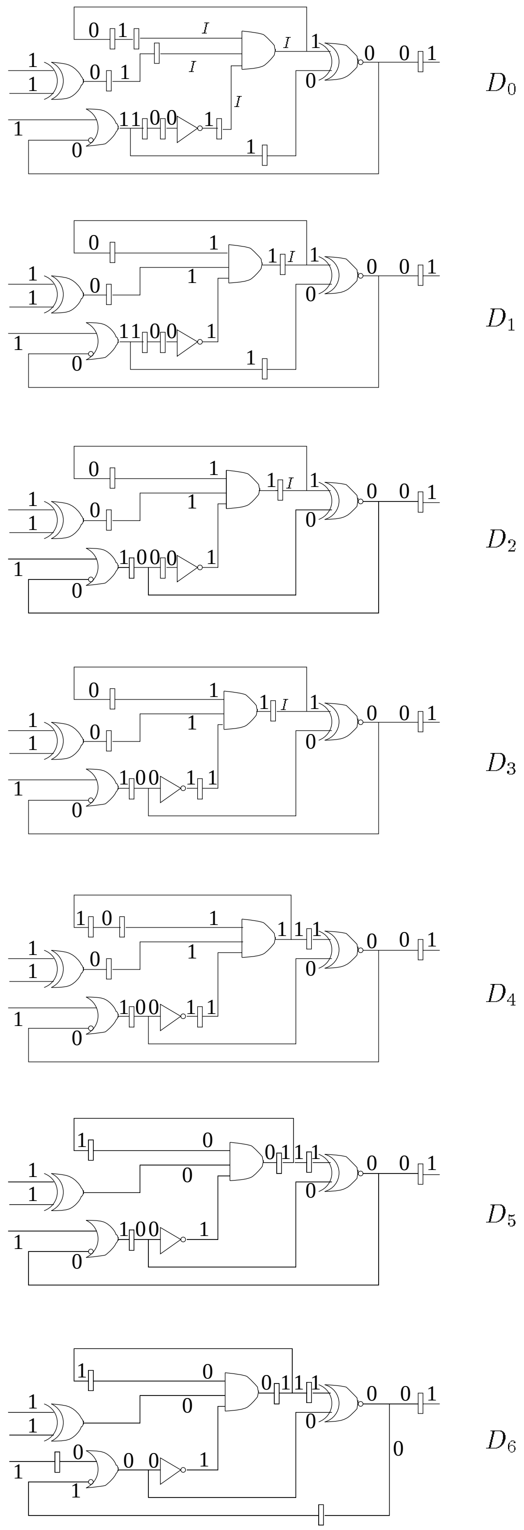

Figure 7.9: Example of the labeling procedure.

Test Set Preservation

The example given in Section 7.1.1 showed that retiming may invalidate a single-stuck-at-fault which was valid before retiming. We can use the results discussed above to show the following result:

Theorem 7.8 *If C is obtained from D using a sequence of retiming moves such that there are no more than k forward retiming moves, then the test set for D is also a test set for C^k.*

Proof. (Sketch) Consider the fault-free circuit G and the faulty circuit F. Create a circuit $T \equiv (G \parallel F)$ which denotes the two circuits next to each other and each pair of outputs of G and F fed to an XNOR gate. Any single stuck-at-fault test will produce a 0 at one of the outputs of T. Consider any single forward retiming transformation which modifies the two parts of T in the same way; the resulting circuit is T'. Now, for any single stuck-at-fault outside the retimed area we can use the techniques in the proof of Theorem 7.6 to prove the desired result. The forward retiming step creates m new nets between the logic element and the m retimed latches. For any single stuck-at-fault on any such input net to a latch, we use the observation that we can use the same test as for the output net of the latch but we may have to delay the circuit by 1 clock cycle to let the stuck-at-value settle inside the latch. □

For the example in Section 7.1.1 this theorem concludes that the test sequence $0 \cdot 1$ is a test sequence for the same fault in design C (Figure 7.3). Thus, either of the two sequences $0 \cdot 0 \cdot 1$ or $1 \cdot 0 \cdot 1$ would serve as a test sequence for C. It can be seen by simulating the design that either of these test sequences produces $X \cdot 0 \cdot 0$ in the fault-free version of C and the sequence $X \cdot 0 \cdot 1$ in the faulty version, thus distinguishing the two versions on the 3rd clock cycle.

7.4 Retiming Preserves Conservative Three-valued Simulation

Consider again circuit D in Figure 7.1. The output of AND gate-1 is 0 whether the latch has value 0 or 1. For this reason, it is easy to deduce that a single cycle with primary input 0 will reset the latch to value 0. However, notice that if the latch is assigned the indeterminate value X, then a conservative three-valued simulator (CLS) will propagate X values to both of the inputs of AND gate-1. Furthermore, the CLS will propagate an X as output of the AND gate because the CLS has lost the information that the X's are complements of each other. Hence a single cycle input of 0 that will *actually* reset design D will not *appear* to reset design D in three-valued simulation. This should not surprise us because the amount of information lost by three-valued simulation is precisely the same information lost by moving a latch forward across an unjustifiable element. We show that this is a general phenomena relating three valued simulation and retiming.

Simulation is an important component of the IC design verification process. The most popular and fastest way of simulating gate-level designs is three-valued simulation [38]. It is assumed that all latches power up as X, meaning that the value is undetermined. Three-valued logic is well-known for gate-level elements [27]. Three-valued simulation results give the output sequences for a given input sequence. However, it is well-known that three-valued simulation is more conservative than reality: if three-valued simulation shows a 0 (or a 1) for an output, that output will be 0 (or 1) for all power-up states; however, the converse is not true because three-valued simulation might show an X for an output even though that output may be determined to be either 0 or 1 from all power-up states. Although three-valued simulation is conservative, in the absence of other fast methods of verification it is popular in the design process. In fact, if a three-valued simulation shows an X where a designer expects a 0 or a 1, the designer often changes the design so that the output of a CLS agrees with the desired output, even though the original design may also have been correct.

This motivates the work presented in this section. Here, we show that if our yardstick for correctness of a design is the output of a CLS, starting from the state in which all latches are initialized to X, then retiming transformations do not change the observed behavior of a design (as seen from the output of the CLS).

Given that a conservative three-valued simulation is used for the design, we can assume that three-valued logic is already defined for all the combinational logic elements in the design, for example, for a 2-input NOR gate, the output is 1 if both inputs are 0, 0 if either input is 1, and X otherwise. We have to show that if two designs D_0 and D_n start off with each latch initialized to X, and D_n is obtained from D_0 by a sequence of retiming moves, then any sequence of three-valued inputs will produce the same outputs from both D_n and D_0. As in the last section, we model junctions as a special single-input multiple-

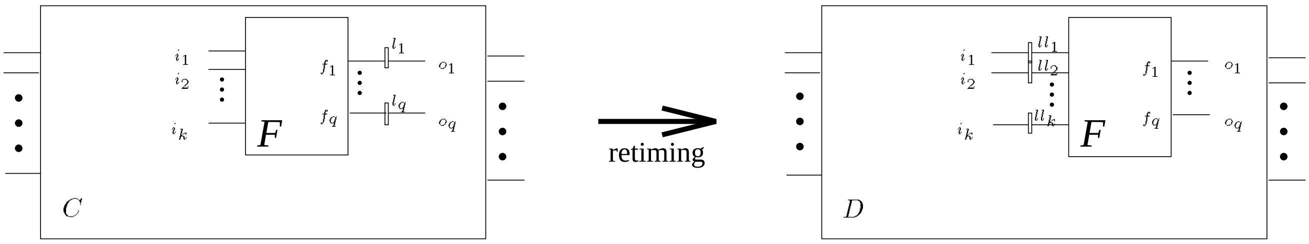

Figure 7.10: Backward retiming move across a multi-output logic element.

output combinational element JUNC. Also, assume that if all inputs of any combinational element are X's, then all outputs are X's. We need this condition to guarantee that when all of the latches are initialized to X's in both an initial and a retimed circuit, they will generate the same outputs. For example, if a gate had a constant value of, say 0, then a forward retiming move across this gate would assign the value of X in the initial state. If the output of the gate were a primary output, they would obviously have different observable behavior.

We prove the main result in this section by considering one retiming move at a time, and then using induction on the number of retiming moves. Suppose that design D is obtained from design C by a *single* retiming move (Figure 7.10). We define a relation $\mathcal{R}$ between the states of C and the states of D. We will assume that the retiming move is backward across F. The case for a *forward* retiming move is symmetric — just the roles of C and D are interchanged in the definition of $\mathcal{R}$. Since D results from C by a single retiming move, designs C and D are identical except for the retiming area around the logic element F shown in the figure. Suppose that F implements the functions $f_1, f_2, \ldots, f_q$. For design C, let inputs to F be $(i_1, i_2, \ldots, i_k)$, the latches of C be $(l_1, l_2, \ldots, l_q)$ and the latch outputs be $(o_1, o_2, \ldots, o_q)$. For design D, let the inputs be $(i'_1, i'_2, \ldots, i'_k)$, the latches be $(ll_1, ll_2, \ldots, ll_k)$ and the outputs of F be $(o'_1, o'_2, \ldots, o'_q)$.

Now consider any state s_0 of C and state s_1 of D. Without loss of generality and to simplify the discussion, we assume that the primary inputs of C and D are latched immediately outside the retiming area. We now define what it means for s_0 and s_1 to be related by $\mathcal{R}$. First we require that (1) corresponding latches outside of the retiming area have the same values for states s_0 and s_1. Second we require that (2) corresponding outputs of the retiming area have equal values, that is for each i, $o_i = o'_i$. We observe that conditions (1) and (2) imply that the primary outputs of the two circuits have equal values.

We now show that that if $s_0 \mathcal{R} s_1$ and if both designs are clocked one cycle with the same primary inputs to get states s'_0 and s'_1, respectively, then $s'_0 \mathcal{R} s'_1$. This implies that for any sequence of the same three-valued inputs to the two designs, the primary outputs will also be equal.

Theorem 7.9 *Suppose circuit D is obtained from C by one retiming move. Suppose also that s_0 and s_1 are three-valued states of C and D respectively, and suppose that $s_0 \mathcal{R} s_1$. Let s'_0 and s'_1 be states of C and D respectively after introducing the same three-valued input sequence. Then $s'_0 \mathcal{R} s'_1$.*

Proof. Suppose circuit D is obtained from C by a *backward* retiming move. The forward case is the same with C and D reversed. Suppose that s_0 and s_1 are states of designs C and D respectively, such that $s_0 \mathcal{R} s_1$. We aim to show that next states s'_0 and s'_1 are also related by $\mathcal{R}$. Conditions (1) and (2) of $\mathcal{R}$ and the assumption that primary inputs are the same imply that outside the retiming area, the next state functions of corresponding latches have the same value. Therefore condition (1) of $\mathcal{R}$ is satisfied for next states s'_0 and s'_1.

We observe that conditions (1) and (2) imply that (3) the corresponding inputs to the retiming area are equal, that is, for all j, $i_j = i'_j$. But (3) implies that the inputs (i.e., next state) to latches $l_1, l_2, \ldots, l_q$ equal $F(i'_1, i'_2, \ldots, i'_k)$. That is to say, the inputs to the l latches equal F of the inputs of the ll latches. This in turn implies that, in the next state, corresponding outputs of the retiming area are equal, i.e., condition (2) holds in the next state. We therefore conclude that $s'_0 \mathcal{R} s'_1$.

By induction, we conclude that the relation R is preserved by any sequence of inputs starting from states in which R holds and the theorem is established. □

By induction on the number of retiming moves, we have the following corollary:

Corollary 7.10 *Suppose circuit D_n is obtained from D_0 by a sequence of n retiming moves. Suppose also that s_0 and s_1 are states of D_0 and D_n respectively and that $s_0 \mathcal{R} s_1$. Then for any sequence of three-valued input vectors, the output sequences of D_0 and D_n from the states s_0 and s_1, respectively, are the same.*

When a CLS is used to verify the correctness of designs, all latches are initialized to X's and input vector sequences are supplied to the CLS. If all latches are initialized to X's in both the original design D_0 and the retimed design D_n, then clearly the two initial three-valued states in the two designs are related by the relation $\mathcal{R}$ (it is here that we need our assumption that for any combinational element in the design if all inputs are X's, then all outputs are X's). The above results lead to the following result, which establishes the validity of retiming moves, if three-valued simulation is used as a criterion for correctness of the design:

Corollary 7.11 *Suppose circuit D_n is obtained from D_0 by any sequence of retiming moves. Suppose that s_0 and s_1 are the states of D_0 and D_n respectively obtained by initializing each of the latches to the value X. Then for any sequence π of three-valued inputs, the output sequences from of D_0 and D_n are the same. If π resets D_0 then it also resets D_n and vice-versa.*

Chapter 8

Retiming the Initial State

Even though we have modeled all latches using no-reset latches in this thesis, many latches in real designs do have reset latches. When retiming such latches, one problem that arises is to obtain the new initial values for the retimed latches. This problem has been raised peripherally in [4, 53] and in much more detail in [81]. It has been claimed that if reset latches are modeled using no-reset latches, that restricts the class of possible retimings. In this chapter, we show that, this is not the case and using no-reset latches and reset circuitry provides the most general model. This model even allows us to solve the problem for the case when it is not possible to retime the initial state without altering combinational logic (this solution had earlier been suggested by [54]). We prove the correctness of retiming the initial state using no-reset latches and we suggest an overall retiming strategy for use by a retiming tool.

8.1 Modeling Reset Latches

We will deal with the validity of retiming transformations with the assumption that all latches are no-reset latches. For designs where some latches are reset latches we can transform these latches to no-reset latches using the transformation described in Chapter 2. However, it has been pointed out to us [66] that once we transform an original design to a new design via such transformations we may no longer be able to retime the new design to designs which were retimed versions of the original design. In [54, page54] Malik also claims that making the reset circuitry explicit, as we suggest by our transformation, has two problems: (i) this results in less flexibility for retiming algorithms, and (ii) since latches

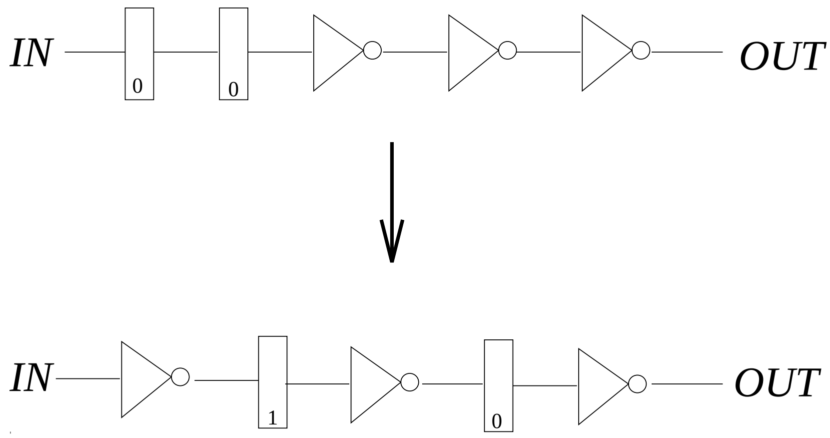

Figure 8.1: Retiming reset latches and the initial state

migrate during retiming, the reset logic may no longer be associated with the latches. To illustrate this "problem", we consider the following example.

Consider the top circuit C in Figure 8.1. The circuit C consists of a chain of 3 inverters. The delay of each of the inverters is 1. The latches are reset latches and when the global reset line is pulled, both latches assume the value 0 (denoted in the figure by the value on the latches). Now, the problem is to retime these latches to achieve a minimum delay for the circuit. We can achieve this by retiming the original circuit to the bottom circuit D in the figure. This circuit has a delay of 1, whereas the original circuit had delay of 3. Also notice that the initial state of the circuit has been retimed so that the initial value for the left latch is 1, and that of the right latch is 0. For a retiming move on a circuit with only reset latches, an equivalent initial state can be obtained by taking the forward (backward) image of the initial values for a forward (backward) retiming move [81]. The forward image is always deterministic but the backward image may be empty or may contain several solutions. If it is empty the retimed initial state cannot be obtained in straightforward say (see Section 8.3).

Now, let us see what happens if we model the reset latches with no-reset latches, as advocated in Section 2.1.1. Using the **reset transformation** (replacing reset latches with no-reset latches plus reset circuitry), the original circuit C transforms to circuit C_1 in Figure 8.2. Notice that this circuit has an extra input, the global reset line. Each of the inverters has delay 1, as before. However, the additional combinational logic added (each of the two AND gates) has delay 0. If we retime this circuit to achieve a minimum delay, it

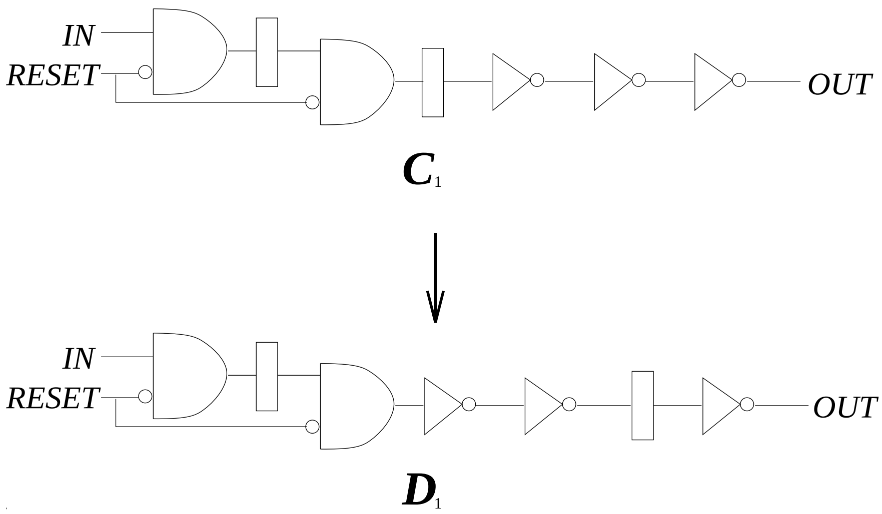

Figure 8.2: Retiming after the reset transformation

seems that we can only retime the front latch forward across the inverters; we can no longer retime the rear latch because the AND gate is in the way. Thus, it seems that we can no longer achieve a minimum delay of 1, as before. So, it seems that the reset transformation may prevent our synthesis tool to reach regions of design space which were reachable before this transformation.

The solution of the above "problem" is to realize that the combinational logic added to model reset latches using no-reset latches is actually part of the latch (like inside of the reset latch). So if a reset latch in retimed in the original circuit, in the new circuit the no-reset latch should be retimed *along* with the combinational logic which was added in the reset transformation. Before giving the general result about how this combinational logic can always be retimed along with the no-reset latch, we show how the problem in the above example can be taken care of.

The solution is shown in Figure 8.3. The transformation logic is retimed along with the latch, and the desired solution (of delay 1) is achieved.

Henceforth, we will use the term "retiming" to also denote retiming the no-reset latches along with their reset circuitry, although in a strict sense, this would classically be called retiming and resynthesis. Throughout this chapter, as we retime the latches, we will try to keep the reset circuitry next to the no-reset latches. Once we are done with retiming, we can efficiently transform such latches back to reset latches, if they are cheaper

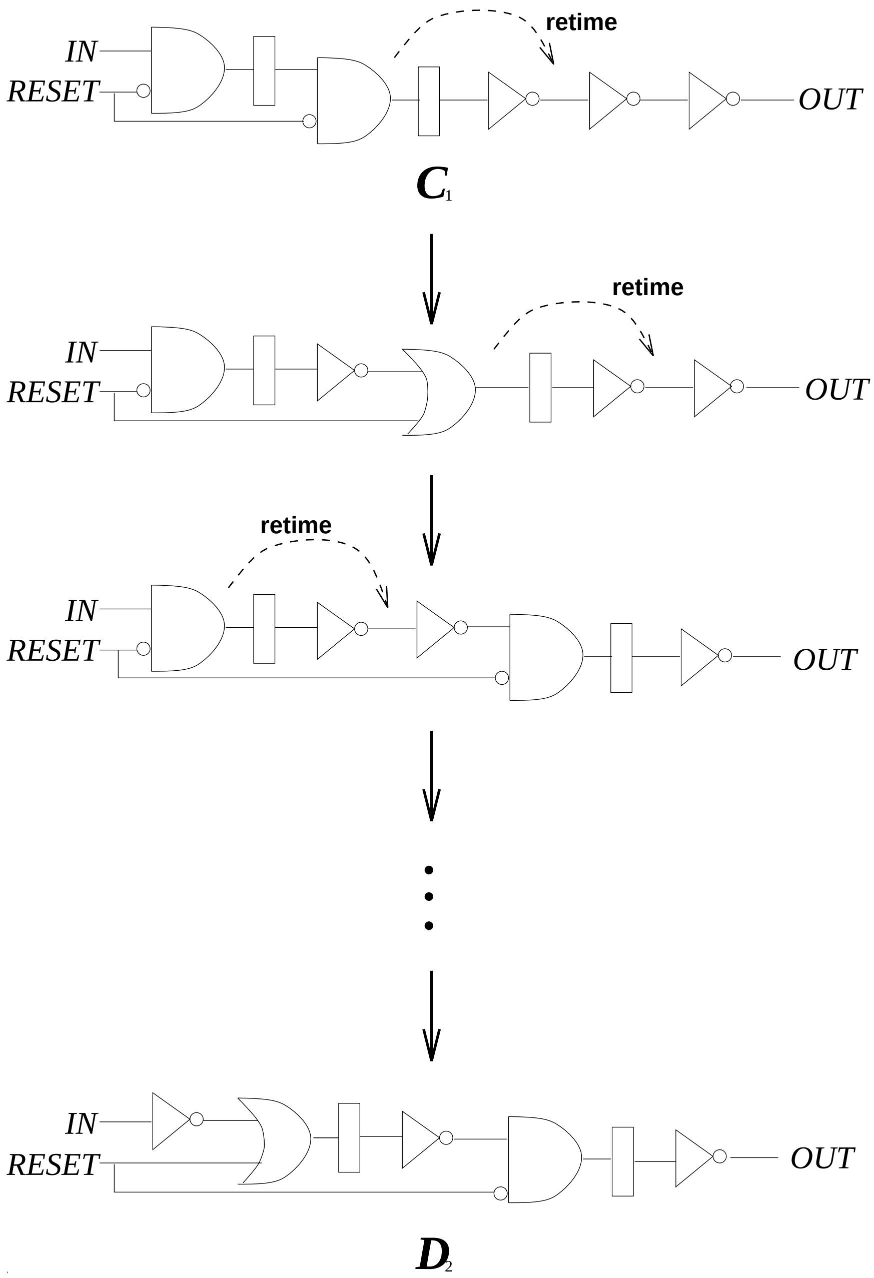

Figure 8.3: Retiming the explicit reset circuitry along with the latches

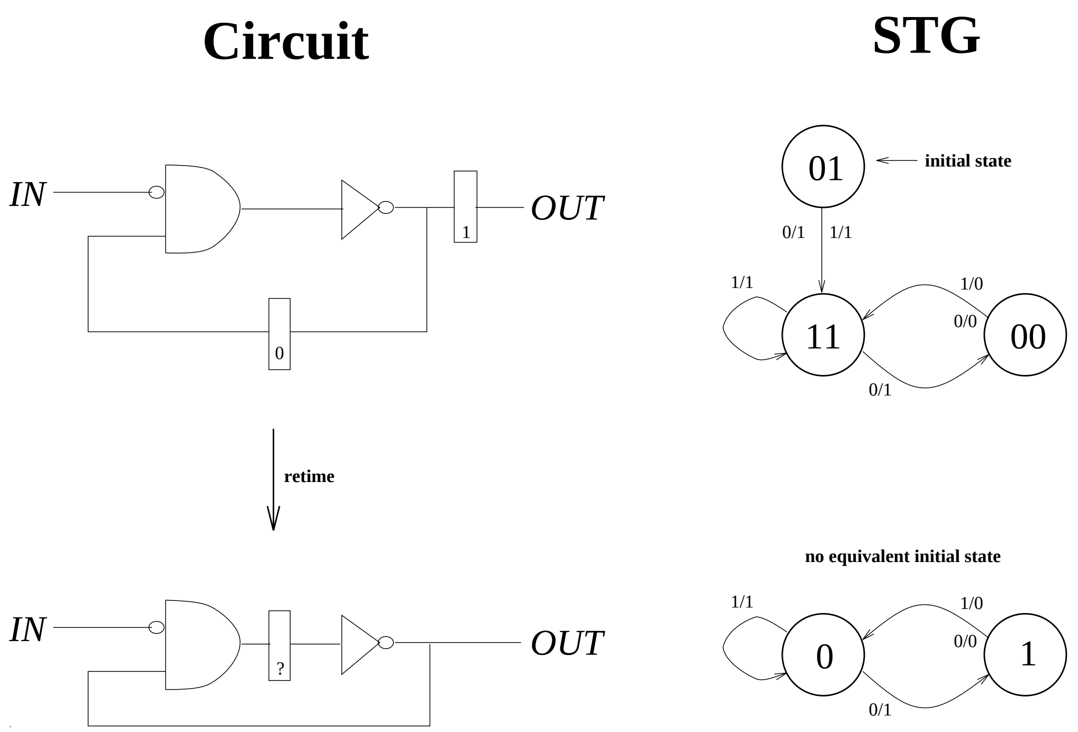

Figure 8.4: Example of impossible retiming without adding logic

or consume less delay, or for any other reason.

8.2 Retiming Initial State Without Adding Logic

First, it should be pointed out that in some cases it is not possible to retime reset latches without *adding* extra combinational and sequential logic to the circuit. This is illustrated by the following example (from [81]):

Consider the upper circuit in Figure 8.4. Each of the reset latches has a hardware initial value which is indicated on the latch. If we want to retime these latches backward across the NOT gate, to obtain the circuit drawn below, we would like to determine the equivalent initial state for this circuit. The goal is to determine a state such that the designated initial state of the lower circuit is equivalent to the initial state of the upper circuit (so that when the implicit reset line is pulled to 1, both circuits have equivalent behavior). However, it is clear from the STG of the lower circuit that no such state exists. The only way to achieve an equivalent design as the above design is by adding extra logic to the circuit; the solution, from [81], is discussed in Section 8.3. There we will show we can simulate retiming for this class of circuits if we transform the design reset latches to a design with only no-reset latches using the *reset transformation*. In this section, we show that if we can retime a circuit with reset latches without adding extra logic, we can achieve the same retiming on the transformed circuit.

We review two strategies for retiming initial states, and show that both can be duplicated even when using the reset transformation. The first strategy is to remap the initial state after each retiming move (as in [4]). We show how this can be achieved in our model in Section 8.2.1. However, this strategy, by virtue of being local, may make local decisions which may hinder a future retiming move down the line. For such situations it may be possible to determine the final initial state by looking at the entire set of retiming moves and analyzing their effect on the transition graph of the design [81]. In Section 8.2.2 we show how our model can simulate this.

8.2.1 Local retiming of the initial state

Suppose we are retiming across a multi-output gate $F = (f_1, \ldots, f_n)$, where each f_i is a function from input space $\{0,1\}^n$ to $\{0,1\}$, as in Section 7.2.2. Recall that we also model junctions as single input multi-output gates. Bartlett *et al.* [4] describe the standard method for obtaining the new initial state if we retime reset latches across the gate F.

Suppose the original design has one reset latch at each of the inputs of the gate, the initial values of the latches are $\vec{a} = (a_1, \ldots, a_m)$, and we want to retime the latches forward across F. The initial values on the reset latches in the retimed design are given by $\vec{b} = (b_1, \ldots, b_n)$, where each $b_i = f_i(\vec{a})$. Now, consider backward retiming across F. Suppose the initial values of the latches on the outputs of F are $\vec{b} = (b_1, \ldots, b_n)$. If there is a set of values $\vec{a} = (a_1, \ldots, a_m)$ such that for each $i \in \{1, \ldots, n\}$: $b_i = f_i(\vec{a})$, we set the initial values on the retimed latches to $\vec{a}$. If there is no such vector, we cannot achieve this retiming; we discuss this scenario in Section 8.3. Notice that there may be multiple choices for selection of initial values in case of backward retiming. Sometimes one choice is worse because it prohibits future retimings while some of the other choices do not; for such cases we may need to branch and backtrack if we make an unwise local choice. We see an example of this at the beginning of Section 8.2.2.

Now, we will give a procedure of retiming no-reset latches along with the reset circuitry so that if circuit D is obtained from C (where all latches are reset latches) after one atomic retiming move, as described by the above remapping of the initial state, then the transformed circuit $C_1 = \Phi(C)$ can be retimed to D_1 such that $D_1 = \Phi(D)$, where Φ denotes the reset transformation (explicit modelling of the reset circuitry). We show that the retimed circuit D_1 (or D) is a delay replacement of the original circuit C_1 (or C).

Proposition 8.1 *For each retiming of a circuit with reset latches, we can achieve the same retiming on the transformed circuit (modeling reset latches with no-reset latches plus reset circuitry) such that the retimed design is a delay replacement of the original design.*

Proof. Suppose we are retiming across a multi-valued combinational logic function $F = (f_1(x_1, \ldots, x_m), \ldots, f_n(x_1, \ldots, x_m))$.

There are two cases:

a) The move from C to D is a forward move. Refer to Figure 8.5. Circuit C_1 represents $\Phi(C)$. Thus if in circuit C the i-th latch has the initial value a_i, then in circuit C_1 the i-th latch is modeled with a no-reset latch and a MUX with inputs x_i and a_i (for a given a_i, the constant a_i can be propagated inside the MUX, to get a simpler gate as in Figure 2.1). The retiming from circuit C_1 consists of two steps: in step 1, just the latches are retimed. In step 2, the reset circuitry is retimed. The values b_i's are such that $b_i = f_i(a_1, \ldots, a_m)$. Step 1 causes 1-delay replacement (Proposition 7.4). To argue that step 2 is a safe replacement, we observe that in this step only the combinational portion has been altered, and thus all we need to prove is that the combinational functionality of the retimed area is identical. If we denote the Boolean value of the reset line by the variable r, then functionality of the i-th output is $r \cdot f_i(a_1, \ldots, a_m) + \overline{r} \cdot f_i(x_1, \ldots, x_m)$ which is the same as $r \cdot b_i + \overline{r} \cdot f_i(x_1, \ldots, x_m)$. Thus, step 2 results in a safe replacement (or a 0-delay replacement), and by the transitivity of delay replacements (Proposition 6.2), D_1 is a 1-delay replacement of C_1. And now we can replace the no-reset latches plus reset circuitry in D_1 by reset latches with the appropriate initial values to get the desired retimed design D.

b) The move from C to D is a backward move. Refer to Figure 8.6. Here, since we know that the retiming can be achieved for reset latches, we can assume that there exist values $(a_1, \ldots, a_m)$ such that for each $i \in \{1, \ldots, n\}$: $f_i(a_1, \ldots, a_m) = b_i$. We achieve the same retiming on the transformed design C_1 in two steps. In step 1, we just retime the reset circuitry backward. By an argument similar to the one for the forward retiming case, this gives us a safe replacement. In Step 2, we retime the no-reset latches backward. From

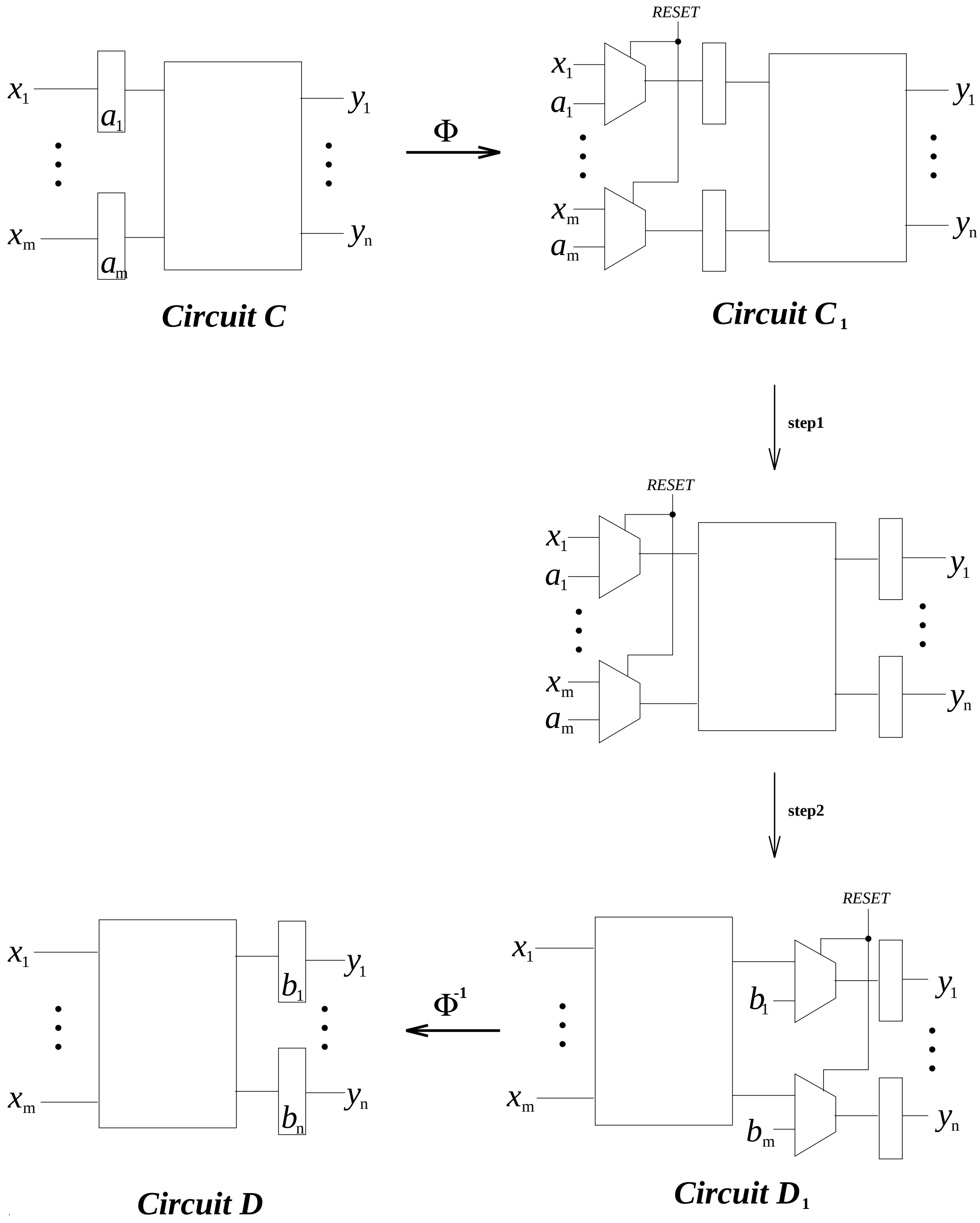

Figure 8.5: Forward retiming of the reset circuitry.

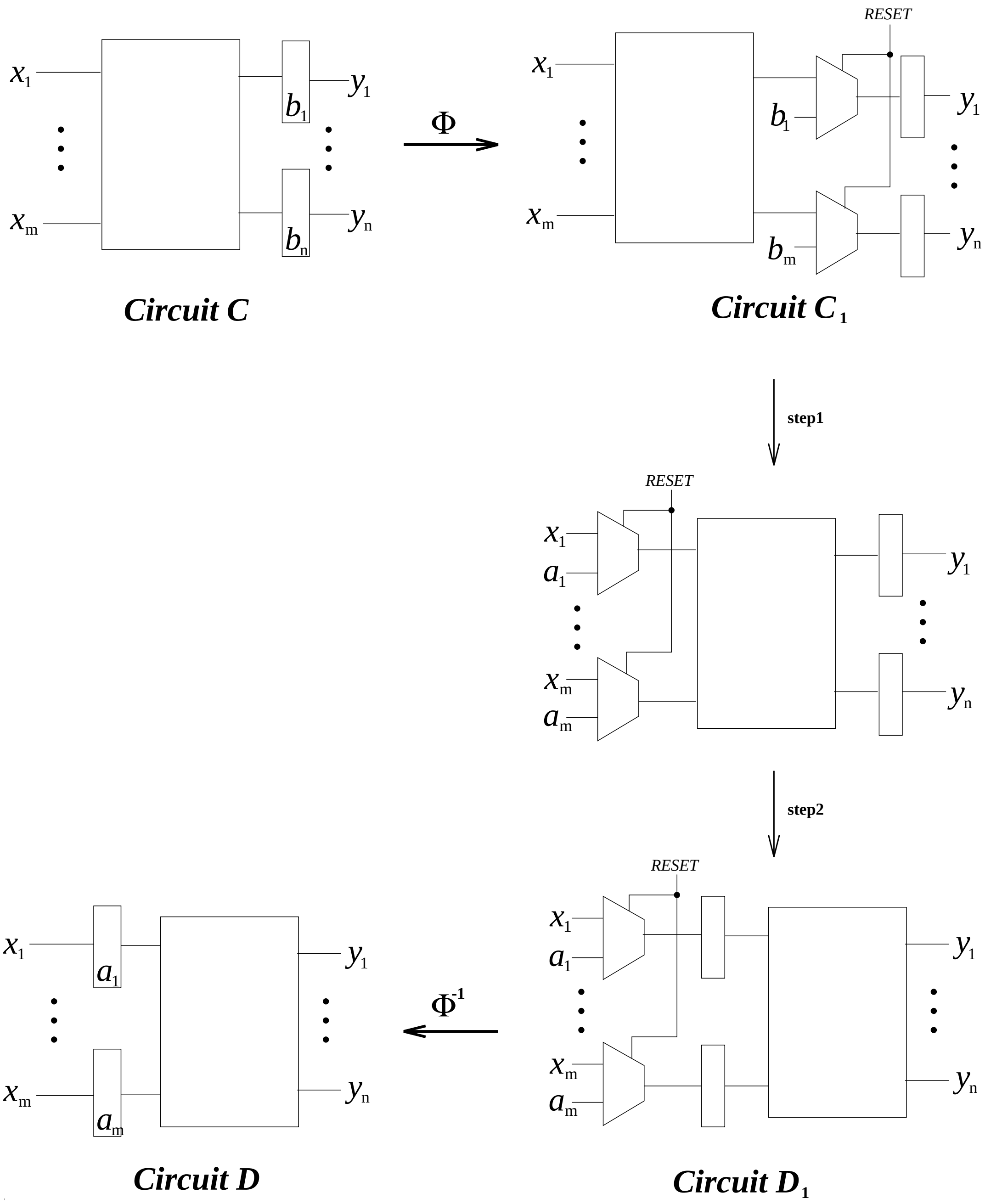

Figure 8.6: Backward retiming of the reset circuitry.

Proposition 7.2, step 2 leads to a safe replacement as well. Thus D_1 is a safe replacement of C_1. Then we can undo the retiming transformation to achieve the desired retimed circuit D. □

Thus, when backward or forward retiming of reset latches across combinational logic elements is possible by remapping the initial values of reset latches, this can always be simulated by no-reset latches by modeling the reset circuitry explicitly.

8.2.2 Global retiming of the initial state

The procedure for obtaining the new initial values for each atomic retiming move that we described at the beginning of the previous section can run into problems because it uses only local knowledge when making decisions about which new initial value to choose for a backward retiming move. This is easily seen by an example. Consider the top circuit in Figure 8.7. We want to retime the latch at the output backwards to obtain one latch at each of the two inputs. After the first backward retiming move across the OR gate (step 1), we choose the new initial values of the two new latches to be (1,1) (the other choices were (1,0) and (0,1)). However, if we continue with this choice we see that after step 2, we are unable to retime the two latches backwards across the junction. On the other hand, if we had chosen (1,0) as the retimed value after the retiming across the OR gate, we would not have run into this problem. This problem arises because of reconvergence in the logic, and we could solve this if our retiming algorithm would have the ability to backtrack. However, a more elegant solution is provided in [81] where they analyze the global behavior of all latches, and thus directly find the value of the initial state for the entire sequence of retiming steps.

First we review the procedure in [81]. Their solution requires forward retiming across the host node (recall, from Section 7.2.1, that the host node is the node in the retiming graph that connects all outputs to all inputs). In the previous section we had only shown the correctness of retiming the no-reset latches plus reset circuitry for retiming across combinational elements. After we review the procedure in [81], we will prove that we can retime the no-reset latches plus reset circuitry across the host node also, so that we can simulate the results of [81]. Later, at the end of this section we will explain the caveats of using this algorithm for obtaining a global retiming of the initial state.

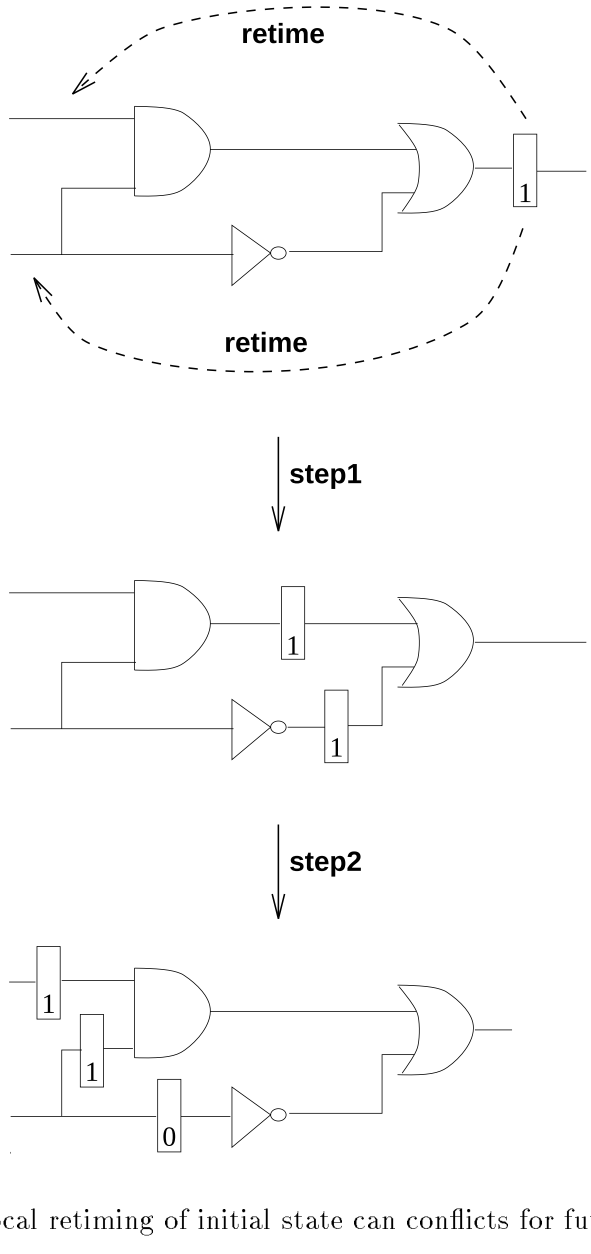

Figure 8.7: Local retiming of initial state can conflicts for future retimings.

Touati-Brayton's algorithm for determining new initial state for a retiming

Here we describe the algorithm in [81] to obtain a new initial state for a given retiming for a circuit where all latches are reset latches.

Recall, from Section 7.2.1, that a retiming graph contains a vertex for every combinational node plus an extra node, the host node. The weight of an edge is the number of latches between the two nodes in the original design. A retiming $\mathcal{R}$ is an integer mapping lag of each vertex such that for every edge (u, v) with weight w, the value $w + lag(v) - lag(u)$, which denotes the number of latches on the path from u to v in the retimed design, is non-negative. It can be easily seen that two different assignments to the lag function lag_1 and lag_2 yield exactly the same number of latches between any two nodes if $lag_1(v) - lag_2(v) = k$ for some constant. Touati and Brayton use this fact to add a smallest constant to each lag so that $lag(v) \leq 0$ for each node v. Thus, they avoid backward retiming moves altogether, which is a good idea, because a single backward retiming move may require choosing between multiple possible retimed initial state values, and the right choice can only be determined by a more global knowledge and computation of all other retiming moves. However, now they have to retime the state for forward retiming moves across the host node also.

They give the following sufficient condition for a circuit D_0 with initial state s_0 when they can obtain a retiming without adding extra logic:

$$\exists s \in Q_{D_0}, \pi : \delta_{D_0}(s, \pi) = s_0 \text{ and } |\pi| = -lag(host) \tag{8.1}$$

The above condition says that there must be a state s and a sequence π of length equal to the number of forward retiming moves across the host such that state s goes to s_0 under this sequence. If this condition is satisfied, they use the following procedure to obtain the new designated initial state of the retimed circuit. We assume that we are given initial design D_0, state s, sequence π and a lag assignment such that for all v: $lag(v) \leq 0$.

1. Change the initial state values in the given design to designate state s. Call this new design D_1.
2. For each vertex v, initialize $r(v) = lag(v)$. If $r(v) < 0$ and each input of the vertex has a latch, insert v in a set S.
3. While set S is not empty do:

(a) Remove any node u from the set S.

(b) Forward retime node u. If node u is a combinational node, get the new initial state by taking the forward image of the input values. If node u is the host, store in the newly created latches on the input lines, the vector π_j, where j is such that this is the j-th retiming of the host node.

(c) For each fanout v of u, if $r(v) < 0$ and each input of v has a latch now, insert v in the set S.

(d) Decrement $r(u)$. If $r(u) < 0$ and each input of v still has a latch, insert u in the set S.

Correctness of the algorithm for no-reset latches

Touati and Brayton proved for the above procedure that the designated initial state obtained for the retimed circuit is equivalent to the designated initial state of the original circuit. However, in our model, even though we can simulate reset latches (with no-reset latches plus reset circuitry) and the retiming of reset latches (by retiming the reset circuitry along with the no-reset latches) across combinational nodes, the reset input line is just like any other input line and has no special status. In particular, we cannot assume that the reset line is going to be pulled on the first cycle of operation. Thus, we still have the obligation of proving the Touati-Brayton procedure can be simulated in our model to obtain delay replacements; or else we would fall short of our claim that in our model we can simulate every retiming that can be achieved in the traditional all-reset-latches model.

Theorem 8.2 *The above procedure for obtaining the new initial state for a retiming can be simulated by using no-reset latches, resulting in a delay replacement.*

Proof. Let the resulting design from the above procedure be D_2. In order to prove that D_2 is a delay replacement of the original design D_0, first we retime D_2 further using another lag assignment lag' to obtain D_3. This new lag assignment is defined as: $lag'(v) = \min\{lag(host) - lag(v), 0\}$, where $lag(v)$ was the lag assignment which was used to derive D_2 from D_1 (recall that D_1 has the same circuit as D_0 but its initial state is s instead of s_0). Since D_1 to D_2 is the given legal retiming, we know that for each edge (u, v), $w' = w + lag(v) - lag(u) \geq 0$; w' denotes the number of latches on the edge in design D_2.

We have to show that lag' produces a legal retiming, i.e. $w'' = w' + lag'(v) - lag'(u) \geq 0$. This can be seen for each of the following four possible cases:

1. Suppose $lag(host) \leq lag(v)$ and $lag(host) \leq lag(u)$.
   Then we have $w'' = w' + lag'(v) - lag'(u) = w' + (lag(host) - lag(v)) - (lag(host) - lag(u)) = w' + lag(u) - lag(v) \ = w \ \geq 0$.

2. Suppose $lag(host) \leq lag(v)$ and $lag(host) > lag(u)$.
   Then we have $w'' = w' + lag'(v) - lag'(u) = w' + (lag(host) - lag(v)) - 0 = w + lag(host) - lag(u) > w \geq 0$.

3. Suppose $lag(host) > lag(v)$ and $lag(host) \leq lag(u)$.
   Then we have $w'' = w' + lag'(v) - lag'(u) = w' + 0 - (lag(host) - lag(u)) \geq w' \geq 0$.

4. Suppose $lag(host) > lag(v)$ and $lag(host) > lag(u)$.
   Then we have $w'' = w' + lag'(v) - lag'(u) = w' + 0 - 0 \geq 0$.

Thus, lag' is indeed a legal retiming. So, we have $D_1 \stackrel{lag}{\rightarrow} D_2 \stackrel{lag'}{\rightarrow} D_3$. Notice that $lag'(host) = 0$; thus, the $D_2 \rightarrow D_3$ retiming consists of only forward retiming moves across combinational nodes. Now, we consider another retiming path from D_1 to D_3: $D_1 \stackrel{lag''}{\rightarrow} D_4 \stackrel{lag'''}{\rightarrow} D_3$. To get this alternate path, we just need to ensure that for each node v: $lag(v) + lag'(v) = lag''(v) + lag'''(v)$, and that lag'' is a legal retiming (that lag''' ia also a legal retiming follows from these two). For each node v, we define the first lag function to be $lag''(v) = lag(host)$. This clearly is a legal retiming. Thus, D_4 has the same lag for each node and it has the same circuit structure as D_1, but a different initial state. Since the Touati-Brayton procedure pumps in π_1 for the first retiming across host, π_2 for the second retiming across the host, and so on, it should be clear that the initial state for design D_4 is $\delta_{D_1}(s, \pi)$, which is s_0. Thus, D_4 is identical to D_0. Now, the lag function for retiming from D_4 (or D_0) to D_3 is defined to be $lag'''(v) = lag(v) + lag'(v) - lag''(v) = lag(v) + \min\{lag(host) - lag(v), 0\} - lag(host) = min\{0, lag(v) - lag(host)\}$. This means that $lag'''(host) = 0$ and $lag'''(v) \leq 0$, that is, lag''' consists of only forward retiming moves across combinational nodes. Thus, D_3 is a delay replacement of D_4 (or D_0), by Proposition 8.1. We also know that D_3 can be obtained from D_2 by just forward retiming moves across only combinational nodes; thus if we reverse this sequence of retiming moves, D_2 can be obtained from D_3 by backward retiming moves across combinational nodes. So, from Proposition 8.1 again, we know that D_2 is a delay replacement of D_3. Thus, from the transitivity of delay replaceability (Proposition 6.2), D_2 is a delay replacement of D_0. □

Caveat

While the global retiming solution has the ability to find a solution as long as condition 8.1 is satisfied, the big drawback of this approach is that it needs to construct the state transition graph of the design (explicitly or implicitly, using BDDs for example). This limits the applicability of this approach to reasonably sized designs, say designs with less than a hundred latches. For larger designs, the local retiming method discussed in Section 8.2.1 is the only possible approach.

8.3 Retiming Initial State by Adding Logic

Sometimes it may be impossible to retime reset latches without adding extra logic in the design. We discussed an example illustrating such a situation in the previous section (Figure 8.4).

Condition 8.1 is a sufficient condition for retiming the initial state, as we saw in Section 8.2.2. The second contribution of [81] is that for designs which do not satisfy this sufficient condition, a method is given for retiming the design; this method adds extra logic to achieve this. We review their method, consider an alternative approach, and then argue why the second approach is more robust and for some designs more efficient than the approach of [81].

8.3.1 Touati-Brayton's approach of modifying STG

Touati and Brayton achieve the desired retiming by adding extra logic to the circuit. First, they modify the circuit by adding a new input line to the circuit such that the modified circuit satisfies Condition 8.1 and then obtain the retiming of the initial state, as in Section 8.2.2. Let the new input line be called $ZERO$ (we call it so because the modified circuit has the same behavior as the original circuit if we guarantee that the input to this line will always be 0). Now, we change the behavior of the circuit for cases when this input is 1 (which we know will never happen), thus guaranteeing that the behavior of the modified circuit has the same behavior as the original circuit. However, the modified circuit satisfies Condition 8.1. We achieve this by choosing states s_1 and s_2 to be such that s_1 is a state reachable from the initial state s_0 and s_2 is a state such that for some input vector a, $\delta_D(s_1, a) = s_2$. We modify the design to D' so that when the input on the $ZERO$ line is 1, the modified design goes from state s_1 to the initial state s_0. This will guarantee that there is a finite input sequence π_0 which takes initial state s_0 back to itself (i.e. $\delta_D(s_0, \pi_0) = s_0$), which in turn ensures that for any arbitrary n (and therefore for $n = -lag(host)$), there is some state s and an input sequence π so that $\delta_D(s, \pi = s_0)$. Thus Condition 8.1 is satisfied for the modified design and we can apply the algorithm shown in previous section. To minimize the added logic, [81] chooses state s_2 in such a way that, among all reachable states from s_0, the binary encoding of s_2 realizes the minimum Hamming distance from the binary encoding of s_0. As as example, borrowed from [81], consider once again the retiming which we were unable to achieve in Section 8.2 (Figure 8.4). With the method just explained, this retiming can be achieved as shown in Figure 8.8. Here, $s0$ is 01, $s1$ is 11 and $s2$ is 10. Notice that the desired retiming was achieved at the cost of one extra OR gate, one extra reset latch and some new wiring associated with the 0 input.

8.3.2 An alternative approach

In this section we will show that the *reset transformation* provides an alternate way of determining this extra logic. This approach is also feasible for large circuits where STG-based analysis is not possible; the approach takes a structural view and is not limited by the size of the state space of the entire design. Also, we will show that it is possible that it some situations it adds less (or no) extra logic to achieve the retiming.

The use of the reset transformation for obtaining a retiming for designs which cannot be retimed using local mapping of initial states was earlier suggested in [54, 53]. However, they thought that this restricts the space of retiming moves and adds unnecessary extra logic. For designs where reset latches can be retimed without adding additional logic, we showed in Section 8.2 that this false if we use the reset transformation. Here we show that even for designs where we the combinational logic must change, like in Figure 8.8, the reset transformation is a more robust and sometimes yields more efficient solutions.

The basic idea is to do the reset transformation, and if we are unable to locally retime the reset circuitry along with the no-reset latch (this happens for a backward retiming move as we saw in Section 8.2.1), leave the reset circuitry in place and just retime the no-

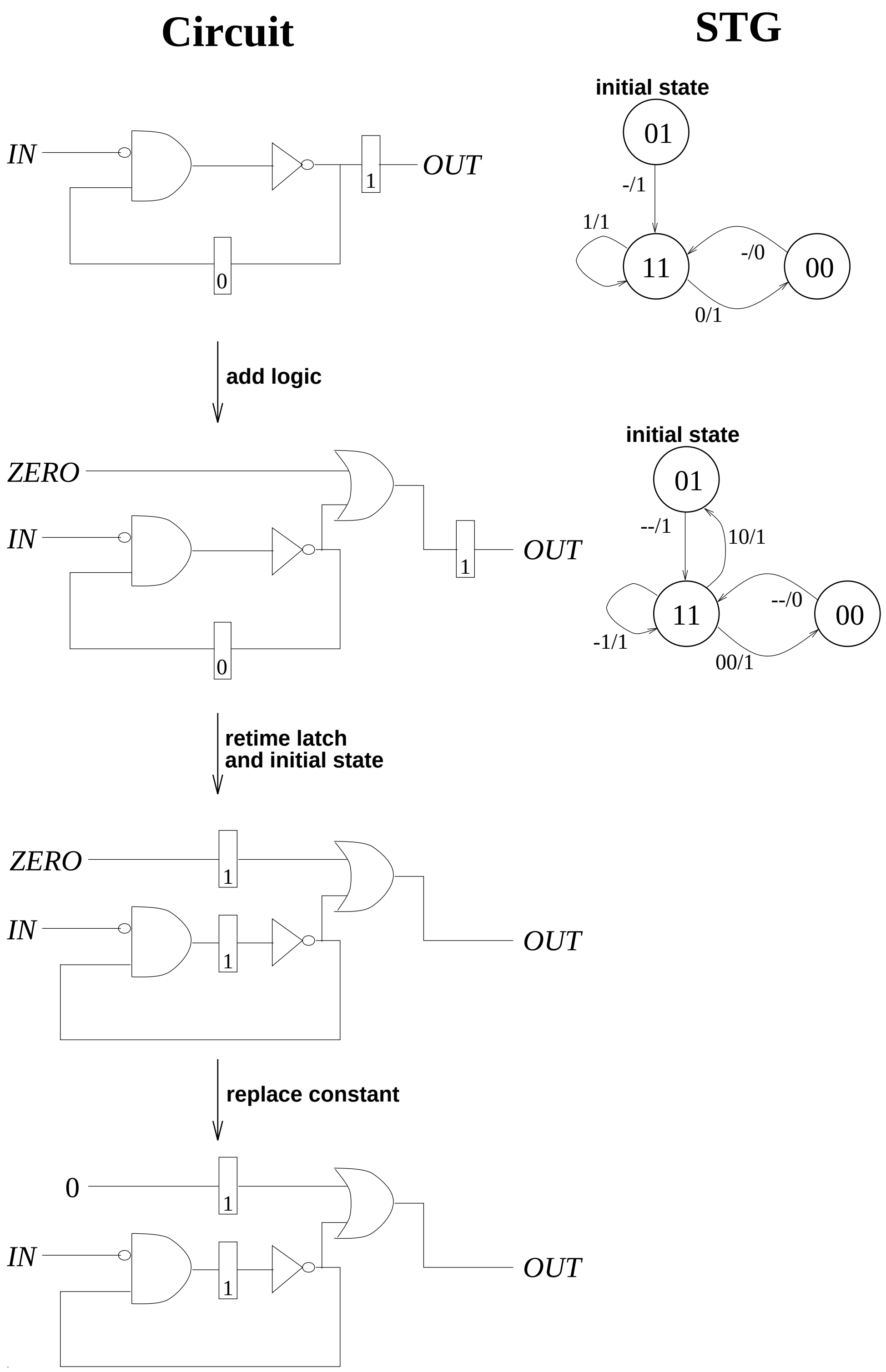

Figure 8.8: Retiming initial state by adding logic

reset latch. So we use the fact that the initial value of the latch is modeled explicitly with the reset circuitry to our advantage. As an example, consider the retiming we wanted to achieve in Figure 8.8. The same retiming can be achieved using the reset transformation, as seen in Figure 8.9. Comparing this solution with the one obtained in Figure 8.8, we have extra AND gate but we have two no-reset latches instead of two reset latches; the wiring from the RESET wire is not extra because that wiring is implicit in the reset latches of Figure 8.8. Thus the two solutions are almost of identical cost. In the next section, in Figure 8.11 we will see that if make an additional assumption about the design environment we can attain a slightly better solution.

There are two advantages of this approach over the approach in [81]:

- The biggest advantage is that this approach is more robust to the size of the designs. Since Touati-Brayton's algorithm requires the construction of the STG of the design, it is limited to the designs where this is possible. The number of states in a design is exponential in the number of latches; so their approach may quickly run out of steam. Using the reset transformation, on the other hand, requires only local retimings. If the reset circuitry can be retimed, we retime it; otherwise we leave it behind. Thus it is not dependent on the size of the entire design and does not require an analysis on the STG of the design.

- In some cases, Touati-Brayton's algorithm may add extra logic while retiming the initial state, even though it may be unnecessary to do so. Recall that they add logic whenever Condition 8.1 is not satisfied. For an example, see Figure 8.10. In this example, we have to retime one latch backwards across the NOT gate. First we need to normalize the lag of each node such that $lag(v) \leq 0$ for each combinational gate v. This gives us $lag(v) = -1$ for each node, including the host node, except the NOT gate. Now, there is no state in the original STG which transitions to the initial state 0000 on an input sequence of length 1; so Condition 8.1 is not satisfied, and we have to add extra logic to get this retiming. Among reachable states from 0000, state 1100 is the state which has the least Hamming distance to the encoding of 0000. So we add the extra input line $ZERO$ and force the first two latches to go to 0 when input $ZERO$ is 1. This gives us the second design in the figure, with its STG shown on the right. Now, we can apply Touati-Brayton's algorithm and we end up with the bottom design in the figure. However, it is easy to see that if we just had to retime

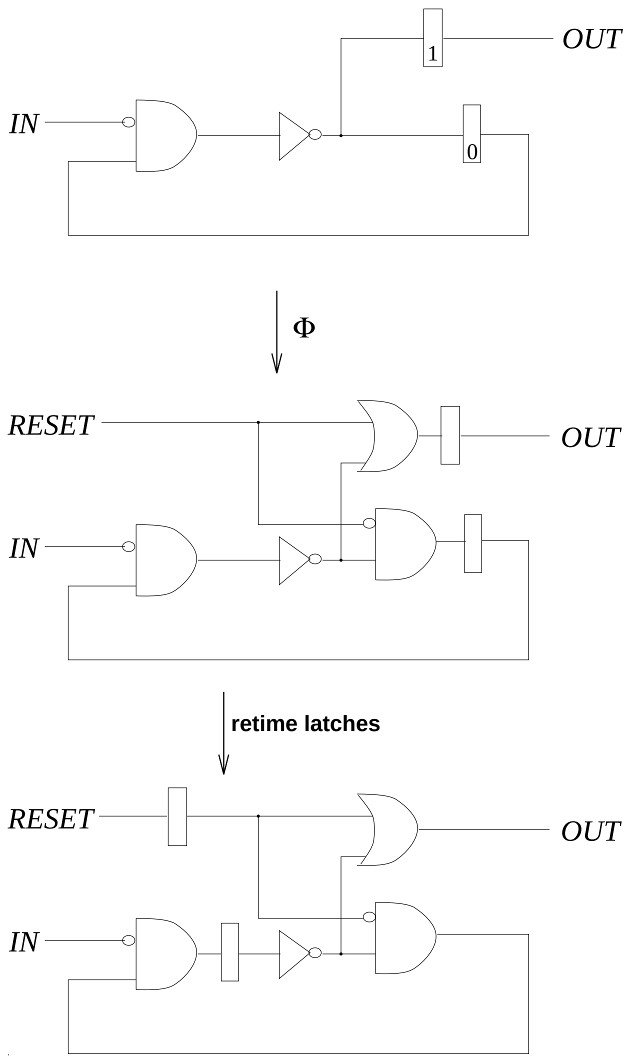

Figure 8.9: Separating the reset circuitry from no-reset latches.

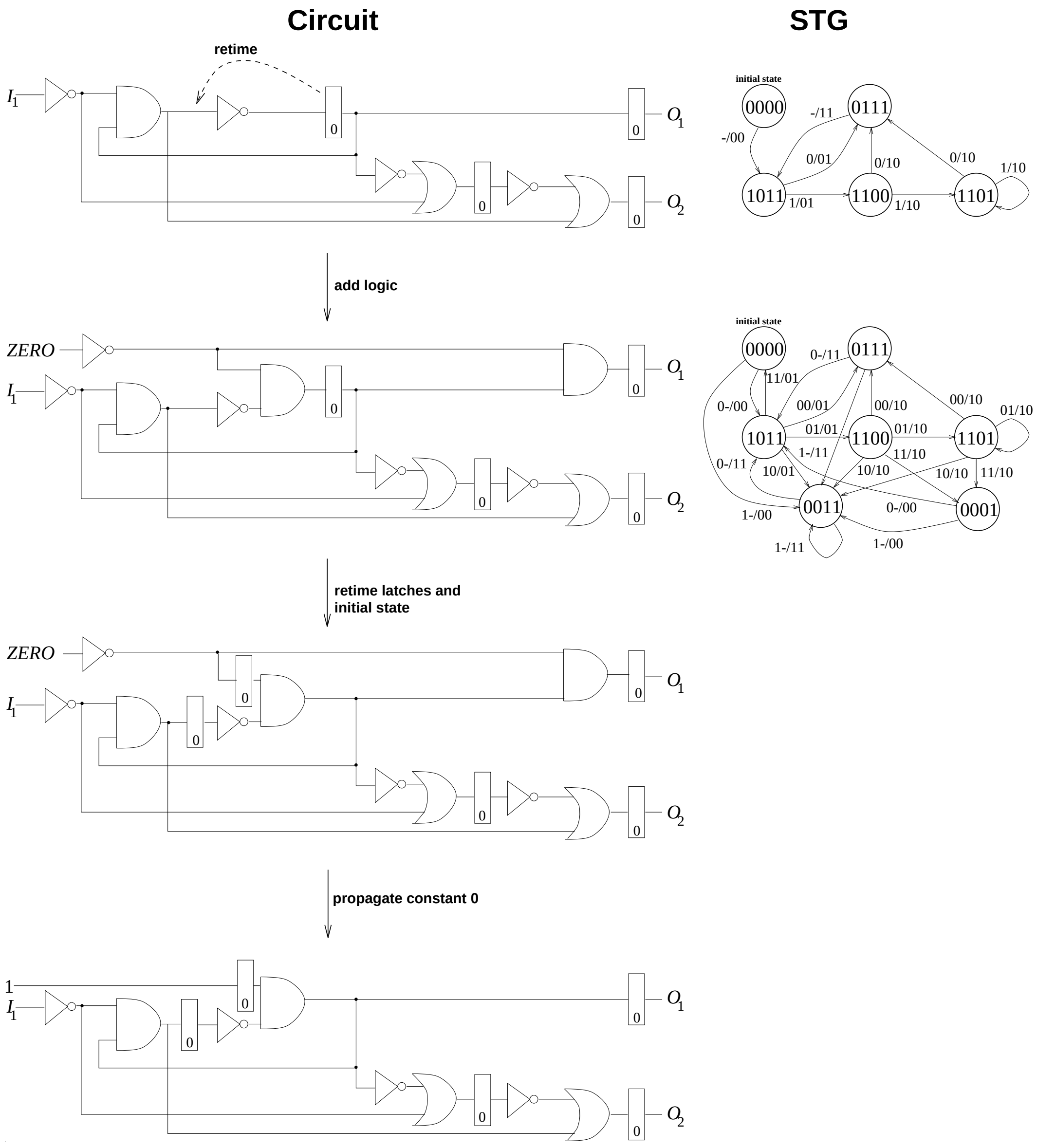

Figure 8.10: Retiming initial state via forward retiming adds unnecessary logic.

the latch backward across the NOT gate, we could simply have retimed it to value 1, and we need not have added this extra logic. Thus, it is possible for backward retiming moves, that the local retiming algorithm may not need to add any additional logic whereas the Touati-Brayton algorithm may add logic because it simulates all backward retiming moves by forward retiming moves across the host node. On the other hand, one can construct examples where, because the Touati-Brayton algorithm computes the global state transition graph, it is able to come up with a retiming of the initial state at a lower cost than the local retiming algorithm of this section.

8.4 Retiming with Both Reset and No-Reset Latches

So far we have seen in some cases we are able to the retime the reset circuitry along with the no-reset latches; in other cases we can retime only the no-reset latch and have to leave the reset circuitry behind. In this section we address the question of how to get the initial values of latches if, before retiming across a combinational gate, some latches were no-reset while others were reset latches (which are modeled with no-reset latches plus reset circuitry). We will extend the remapping algorithm described in Section 8.2.1.

Once again, assume that we are retiming across a multi-output gate $F = (f_1, \ldots, f_n)$, where each f_i is a function from input space $\{0,1\}^n$ to $\{0,1\}$. Suppose the original design has one reset latch at each of the first m' inputs of the gate, one no-reset latch at each of the other inputs, the initial values of the reset latches are $\vec{a} = (a_1, \ldots, a_{m'})$, and we want to retime the latches forward across F. There are two cases: (i) there is a unique vector $\vec{b} = (b_1, \ldots, b_n)$ such that for all combinations of $(x_{m'+1}, \ldots, x_m)$, for any $i \in \{1, \ldots, n\}$: $f_i(a_1, \ldots, a_{m'}, x_{m'+1}, \ldots, x_m) = b_i$, and (ii) there is no such vector $\vec{b}$. In the first case, we can retime the reset circuitry for the reset latches forward such that all the newly retimed latches are reset latches and the new initial values for the retimed latches is $(b_1, \ldots, b_n)$. For example, if the gate was an OR gate with a no-reset latch on one input and a reset latch with initial value 1 on the other input, we could forward retime the latches so that there is one reset latch with initial value 1 on the output of the OR gate. The proof of correctness of this operation is similar to that of Proposition 8.1. For the second case, we are unable to retime the reset circuitry forward; we can retime just the no-reset latches and leave the reset circuitry behind. For example, consider an OR gate with a no-reset latch on one input and a reset latch with initial value 0 on the other input; suppose the two inputs of the latches are denoted by x_1 and x_2. Once we retime the no-reset latches, as in step 1 of Figure 8.5, the functionality of the combinational part becomes $r \cdot x_2 + \overline{r} \cdot (x_1 + x_2) = x_2 + \overline{r} \cdot x_1$. There is no way to get the same functionality if we were to retime the reset circuitry forward, as in step 2. For all backward retiming moves also, if some latches on the output are reset latches and other are no-reset latches, we cannot retime the reset circuitry across the gate; so if we want to retime the latch we have to separate its reset circuitry and leave it behind.

There is one assumption which is true in some design situations where we can retime the reset circuitry more often. Suppose the reset line has a special status and the designer can safely assume that on the clock cycle after the reset line is pulled to 1, the environment of the design can assume that the values in each of the no-reset latches is totally arbitrary (i.e. it is just like the power-up situation where each of the no-reset latches can assume any arbitrary value). This is a reasonable assumption to make for many design scenarios. If this is the case, in the second case of the forward retiming above, we can also retime the reset circuitry forward. The new initial values of the reset latches on the output lines is $\vec{b} = (b_1, \ldots, b_n)$ where $\vec{b}$ is such that there *exists* some vector $(x_{m'+1}, \ldots, x_m)$ such that for any $i \in \{1, \ldots, n\}$: $f_i(a_1, \ldots, a_{m'}, x_{m'+1}, \ldots, x_m) = b_i$. Thus, for the OR gate example, where we have one reset latch with initial value 0 and one non-reset latch we could retime these forward to get a reset latch at the output with value either 0 or 1. Similarly, with this new assumption, we are sometimes able to retime the reset circuitry backward too. Suppose the original design has one reset latch at each of the first n' outputs of the gate, one no-reset latch at each of the other outputs the initial values of the reset latches are $(b_1, \ldots, b_{n'})$, and we want to retime the latches backward across F. If there is a set of values $\vec{a} = (a_1, \ldots, a_m)$ such that for each $i \in \{1, \ldots, n'\}$: $b_i = f_i(\vec{a})$, we set the initial values on the retimed latches to $\vec{a}$. If there is no such vector, we cannot retime the reset circuitry and we have to separate it from the latches.

As an example of retiming both kinds of latches in a design, consider once again the example we used in the previous section (the retiming we want to achieve in Figures 8.8 and 8.9). We have shown this retiming using the remapping algorithm proposed above in Figure 8.11. In retiming step 2, we retime one no-reset latch and one reset latch (with explicit initial value 0) backwards across the junction to get a reset latch with initial value 0. Then we retime this reset latch backwards across the NOT gate to obtain the new initial value 1. It is interesting to note the similarity between this solution and the final one in Figure 8.8.

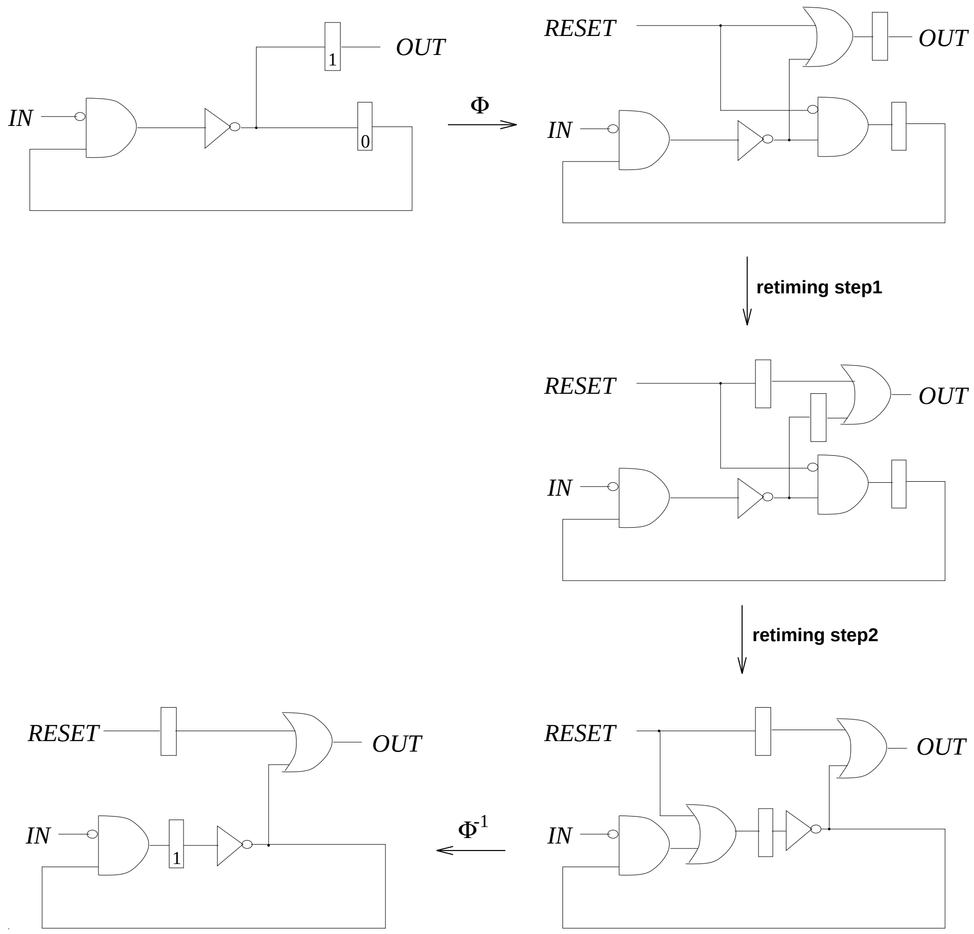

Figure 8.11: Retiming reset and no-reset latches.

For many designs it is not possible to make the above assumption. The reset line may itself be derived from other logic, different latches may have different reset latches and there may be other reasons, like the ones discussed in Section 2.1 that we are unable to designate this special status to the reset line.

8.5 Overall Strategy

In this section, based on our analysis so far in this chapter, we recommend an overall strategy for retiming the initial state for use in a synthesis tool.

We have shown that reset latches can be safely modeled using the reset transformation. Claims that this restricts the space of retiming moves or adds unnecessary extra logic or wiring are unfounded. Whenever possible the no-reset latch plus its associated reset circuitry should be thought of as one unit (and an elementary tool can easily undo the reset transformation to obtain reset latches from no-reset latches plus the reset circuitry). If possible, we retime this whole unit together. We have shown that we can safely retime this unit forward or backward across combinational gates and forward across the host node whenever an equivalent retiming can be done for reset latches. If the design is small enough so that STG analysis can be done on it, we can get the new initial values using the Touati-Brayton algorithm in Section 8.2.2. If not, we can retime the initial values locally, as shown in Section 8.2.1.

In case it is not possible to retime the initial state, the reset circuitry can be left behind and just the no-reset retimed. We have argued in Section 8.3.2 that this approach is more robust and sometimes produces more efficient solutions than given in [81] for this class of retimings.

We have shown how to retime the initial state when only some of the latches are reset latches and others are no-reset latches in Section 8.4. Also, if we can make an assumption about the reset line being special such that whenever the environment pulls the reset line each of the no-reset latches of the design can non-deterministically assume an arbitrary Boolean value, then we can retime the reset circuitry in more cases.

Chapter 9

Conclusions

We have explored the issue of design replacement for sequential circuits. We now point to possible extensions to this work. Then we describe some of the lessons that learned while researching the issues in this thesis. We also point to some other work published concurrently or since our work addressing some of the questions. We mention some other issues which are relevant to our work but not discussed in this thesis because they were not the focus of our work.

We started this dissertation by restricting ourselves to designs which have a well-defined notion of next step. We assumed that the sequential aspect is implemented with memory elements like latches or flip-flops and the combinational gates, implementing the combinational aspect, do not have any cycles. Getting such an abstraction from transistor-level designs to such nicely defined gate-level designs is an important area and a focus of many current researchers [39, 58, 78]. The problem of combinational loops has also received some attention. Some people have identified classes of such circuits where these circuits can be replaced by "equivalent" circuits with no combinational loops [77]; others have endeavored to define the semantics of all circuits, including the ones with combinational loops [13, 85].

We presented our notion of safe replaceability in Chapter 4 and discussed design replacement on any arbitrary design, in the absence of any knowledge of the environment of a design. For redundancy identification in sequential circuits, Pomeranz and Reddy [63] have identified an equivalent notion, but in their algorithm for redundancy identification, they use a notion which allows them to classify more faults as redundant at the expense of making some initialization assumptions about the environment of the design. Recently, Brand *et al.* [8] used the term "safe replaceability" to denote replacement of designs in presence of a given environment (as in [83]); interestingly, they use the term "universal replacement" to mean precisely what we have defined as safe replacement.

In the first half of Chapter 5, we used our notion of safe replaceability to do sequential optimizations on gate-level designs. Many comments are in order here. First, we observe that our optimization algorithm (in Section 5.1.4) works on the state transition graphs (even though it did so implicitly, using BDDs). This limits the size of circuits we can handle; in our experiments, we limited ourselves to designs with less than 30 latches. Certainly, we would not recommend even an optimized implementation of our algorithm on designs with more than several hundred latches. However, the power of our approach is in the condition of safe replacement. Since we make replacement without assuming anything about the environment, we envision a more productive use of our optimization algorithm in scenarios where reasonably sizes designs are cut out of larger designs and safe replacements are made on this. This would work with manual as well as automatic partitioning. Our algorithm combined with a powerful automatic partitioning method and applied to a set of larger designs is a problem worth investigating. It would also be very useful to come up with other optimization algorithms for safe replacement, especially ones which are more structural (do not need state transition graph information) in nature. We have chosen to optimize area of the circuits in our experiments; we would also like to have algorithms for other optimization criteria like minimizing delay, minimizing switching activity (power) or increasing testability.

One more area which needs investigation is state encoding and implementation of designs which are specified at the behavioral level (i.e., as STGs). The traditional method of obtaining such an implementation relies on the fact that the design has a designated initial state and thus the behavior of encoded states which are unreachable from the designated initial state is not important. However, without the designated initial state assumption, all 2^t encodings of an implementation (which has t latches) must satisfy the safe replacement condition with respect to the given behavioral-level specification. This will add additional constraints to the traditional problem of state encoding and implementation (see [67] for the traditional formulation).

In [83], Watanabe and Brayton characterized the set of permissible design replacements for a given design with respect to a given environment. However, they also made the assumption of a designated initial state for both the environment as well as the design.

In [84], they provided a solution to the problem of extracting an optimal implementation from this characterization of the set of permissible behaviors. It whould be interesting to extend these solutions, of characterization of permissible behaviors and of choosing an optimal implementation from this characterization, to our model with no designated initial state assumption (either for the design or for the environment).

In the second part of Chapter 5, we studied the verification problem for safe replaceability. We showed that it is PSPACE-complete. We gave an algorithm, which takes exponential time in worst case but has many heuristics to try to finish faster for many examples. We did not implement this since there are no synthesis algorithms which use safe replaceability as the criterion for verification; the results of our synthesis procedure in Section 5.1 can be verified in polynomial time since we use only a sufficient condition for safe replaceability (see Section 5.1.1).

In Chapter 6, we introduced the notion of delay replacement, motivated by the fact that designers often know that a design is not going to be used for a given number of clock cycles after power-up. We found that this additional knowledge about the environment leads to we can obtain much better optimization results, using two different methods. We also showed that the notion of delay replacement is compositional and can be used in a synthesis methodology using successive design replacements. Finally in Chapter 6, we argued that, in practice, the verification problem for delay replaceability is simpler than the corresponding problem for safe replaceability— if the outer envelopes of designs are tSCCs then we have a polynomial time problem (and algorithm), instead of a PSPACE-hard one. Recently, Iyer *et al.* [36] also used this notion of delay replacements for identifying redundancies in sequential circuits.

In Chapter 7, we saw that retiming results in a delay replacement and not a safe replacement. As we showed in this chapter, and as [45] showed earlier, if C is a retimed circuit obtained from the original circuit D, $C^n \subseteq D$ for some integer n which depends on the sequence of retiming moves. For a given sequence of retiming moves, we found a tighter bound for n than [45]. Designers often use conservative three-valued simulators to validate their designs. We showed that if three-valued simulators are used, retiming transformations do not affect the output of such simulators even though they may not be safe replacements (in the sense of Chapter 4). In [55], Marchok *et al.* studied the effect of retiming transformations on the test vector sequences for single stuck-at faults. We proved in this chapter that new test sequences can be obtained from old ones just by appending an arbitrary prefix sequence of length n, where n is determined by the sequence of retiming moves. Later, El-Maleh *et al.* [28] correct their result from [55] and show that test vector sequences, which have been generated by conservative three-valued simulators, are preserved from the original circuit to the retimed circuit. We note that this result follows from our result in Chapter 7 that retiming transformations do not affect the output of such simulators (Corollary 7.10). The preservation of test vector sequences is an important result because in [28] it is demostrated by experiments that generating test sequences for the retimed circuits may be much more expensive than generating them for the original circuits.

We were interested in proving the main result of Chapter 7 (that retiming does not affect the behavior of conservative three-valued simulators) because designers frequently use simulators to analyze their design and are comfortable with the semantics of the simulator they use. Designers use the rule that if for two designs, their simulator produces identical output sequence for each possible input sequence[1], the two designs can replace each other. This leads to a new paradigm for synthesis— optimize a design such that the optimized design is indistinguishable from the original design using a given simulator, for example, a conservative three-valued simulator. To come up with such optimization algorithms would be a very useful thing to do.

In Chapter 8, we showed our method for modeling memory elements throughout the thesis (namely, using no-reset latches plus reset circuitry to model reset latches) is general enough for arbitrary retiming moves. It allows the exploration of all retiming moves that would otherwise be possible, and it does not add more logic or wiring than necessary. Previously doubts had been raised that such a modeling approach might restrict the space of possible retiming and introduce unnecessary logic and wiring. We have shown that this problem can be avoided by retiming reset circuitry along with latches. In many design settings, latches are often combined with other combinational logic inside one library element [33]. It should be interesting to explore if during retiming, these elements can also be retimed as a whole, similar to how we retime a no-reset latch with its reset circuitry, together as one unit.

In Chapter 8, we also showed that modeling reset circuitry explicitly, suggested earlier [54], provides an alternate strategy for retiming an initial state, for cases where such a retiming could not be achieved without adding additional logic. This provides an alternate

[1]That it may not be possible to verify this by exhaustive simulation in any reasonable amount of time is another issue altogether.

strategy to the algorithm in [81]. We also discussed how to obtain the retimed initial state in an environment with have a mixture of reset and no-reset latches. At the end of the chapter, we presented with an overall strategy for retiming latches for use by a synthesis tool.

We conclude with some final thoughts on the notions of design equivalence. There have been, and will be, various notions of design equivalence and replacement. We have shown that for a methodology where one needs a replacement notion for an arbitrary piece of logic inside a larger design, and one cannot make any assumptions about the environment, the notion of safe replacement is precisely the right one— anything more conservative will allow fewer replacements, and anything more liberal would allow replacements that can be detected by some environment. If we also know that some time is going to elapse after power-up before the circuit is used, we can use this extra information to obtain the concept of delay replacements.

The above paragraph notwithstanding, the designer must know (1) what is the "right" notion of replacement for his/her design process, (2) what is the notion of replacement used by his/her tools, and (3) if these notions indeed match. In many cases, one may know more about the overall design so that more liberal replacement criteria can be used. For example, delay replaceability is more liberal than safe replaceability, and it makes sense to use the former if the design allows for these extra clocks after power-up. Another example was discussed in Section 8.4, the design is robust enough so that whenever a RESET line to some latches is pulled, the values in the no-reset latches can be totally arbitrary; this assumption allows for more flexibility for retiming.

One important constraint in the design process is the importance of simulators used to analyze the behavior of designs. In some of these situations, the semantics of the simulator become so important as to affect the notion of design replacement one wants to use. These semantics may allow a designer to make replacements which are more aggressive (or more conservative in other cases) than replacements based just on the behavior of the design. We have discussed the role of conservative three-valued simulators (CLS) in Chapter 7. We showed that retiming preserves delay replaceability. Thus, if we wait *long enough* (n clock cycles, where n is no more than the number of latches in any cycle in the circuit) then retiming does not affect the circuit operation. When a circuit powers-up there is always some delay before the circuit begins operation; it takes time, potentially many clock cycles, for the voltages to settle, etc. Therefore, the requirement that the circuit settle for a slightly longer number of cycles before it begins computation may not, in actuality, cause a problem. However, modern design methodologies rely heavily on logic simulation to the point that if simulation says the circuit doesn't work, then the designer must assume the circuit doesn't work. During simulation, unlike the real operation of the design, the design is simulated from the first clock cycle. So, retiming, although not changing the resulting design behavior, may change the behavior of the simulator, and this may be unacceptable[2] Fortunately, for the case of retiming transformations, the semantics of the simulator allows some slack because of another property of a CLS. Because of its conservative propagation of X's, a CLS may not be able to distinguish two designs which may actually have different behavior (which may be distinguishable by a more accurate simulator). This allowed us to show that retiming transformation do not affect the output of a conservative three-valued simulator.

Recently another example of the semantics of a simulator affecting the replacement criterion was discussed by Brand *et al.* [8] who gave an example where for some input, a design produces an X. In our situation, this can happen, for example, if there are two power-up states such that for this input one state produces a 1 and the other state produces a 0. For such a design they claim that it may be hazardous to replace this design with another which produces a 0 (or a design which produces a 1) for this input from all power-up states. This may happen because the simulator for the environment of the design may interpret the X's in a non-monotonic way (in [8], they claim this happens with VHDL simulation, where, for example, the simulator evaluates both "X = 0" and "X = 1" to 0, whereas a monotonic simulator evaluates both of these to X). Thus, the simulator semantics force us to be more conservative than we probably need to be since we are unable to replace an X by either 0 or 1.

[2]It should be noted this problem can easily be rectified if the simulator is also allowed to settle down for many clock cycles, i.e. many arbitrary input vectors are applied before applying the test vectors.

Bibliography

[1] M. Abramovici and M. A. Breuer. On Redundancy and Fault Detection in Sequential Circuits. *IEEE Transactions on Computers*, 28(11):864–865, November 1979.

[2] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital System Testing and Testable Design*. IEEE Press, New York, NY, 1990.

[3] A. Aziz, R. K. Brayton, F. Balarin, and V. Singhal. Timing-safe Replaceability for Combinational Designs. In *TAU '95: Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, pages 121–127, Seattle, WA, November 1995.

[4] K. Bartlett, G. Borriello, and S. Raju. Timing Optimization of Multiphase Sequential Logic. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 10(1):51–62, January 1991.

[5] G. Berry and H. J. Touati. Optimized Controller Synthesis Using Esterel. In *Workshop Notes of Intl. Workshop on Logic Synthesis*, Tahoe City, CA, May 1993.

[6] C. Berthet, O. Coudert, and J. C. Madre. New Ideas on Symbolic Manipulation of Finite State Machines. In *Proc. Intl. Conf. on Computer Design*, pages 224–227, Cambridge, MA, October 1990.

[7] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *Proc. of the Design Automation Conf.*, pages 40–45, Orlando, FL, June 1990.

[8] D. Brand, R. A. Bergamaschi, and L. Stok. Be Careful with Don't Cares. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 83–86, San Jose, CA, November 1995.

[9] Daniel Brand. IBM T. J. Watson Research Center, Yorktown Heights, NY, Personal communication, 1995.

[10] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[11] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli. Multilevel Logic Synthesis. *Proceedings of the IEEE*, 78(2):264–300, February 1990.

[12] F. Brglez, D. Bryan, and K. Kozminski. Combinational Profiles of Sequential Benchmark Circuits. In *Proc. Intl. Symposium on Circuits and Systems*, pages 1929–1934, Portland, OR, May 1989.

[13] J. R. Burch, D. Dill, E. Wolf, and G. De Micheli. Modelling Hierarchical Combinational Circuits. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 612–617, Santa Clara, CA, November 1993.

[14] E. Cerny and M. A. Marin. An Approach to Unified Methodology of Combinational Switching Circuits. *IEEE Transactions on Computers*, 27(8), 1977.

[15] K.-T. Cheng. Redundancy Removal for Sequential Circuits Without Reset States. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 12(1):13–24, January 1993.

[16] K.-T. Cheng and H.-K. T. Ma. On the Over-Specification Problem in Sequential ATPG Algorithms. In *Proc. of the Design Automation Conf.*, pages 16–21, Anaheim, CA, June 1992.

[17] H. Cho, G. D. Hachtel, S.-W. Jeong, B. Plessier, E. Schwarz, and F. Somenzi. ATPG Aspects of FSM Verification. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 134–137, Santa Clara, CA, November 1990.

[18] H. Cho, G. D. Hachtel, E. Macii, B. Plessier, and F. Somenzi. Algorithms for Approximate FSM Traversal. In *Proc. of the Design Automation Conf.*, pages 25–30, Dallas, TX, June 1993.

[19] H. Cho, G. D. Hachtel, and F. Somenzi. Redundancy Identification and Removal Based on Implicit State Enumeration. In *Proc. Intl. Conf. on Computer Design*, pages 77–80, Cambridge, MA, October 1991.

[20] H. Cho, S.-W. Jeong, F. Somenzi, and C. Pixley. Synchronizing Sequences and Symbolic Traversal Techniques in Test Generation. *Journal of Electronic Testing: Theory and Applications*, 4(12):19–31, 1993.

[21] O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Based on Symbolic Execution. In J. Sifakis, editor, *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373, Grenoble, France, June 1989.

[22] O. Coudert and J. C. Madre. A Unified Framework for the Formal Verification of Sequential Circuits. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 126–129, Santa Clara, CA, November 1990.

[23] M. Damiani and G. De Micheli. Recurrence Equations and the Optimization of Synchronous Logic Circuits. In *Proc. of the Design Automation Conf.*, pages 556–561, Anaheim, CA, June 1992.

[24] M. Damiani and G. De Micheli. Synthesis and Optimization of Synchronous Logic Circuits from Recurrence Equations. In *Proc. European Conf. on Design Automation*, pages 226–231, Brussels, Belgium, March 1992.

[25] G. De Micheli, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Optimal State Assignment for Finite State Machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 4(7):269–285, July 1985.

[26] S. Devadas and A. R. Newton. Exact Algorithms for Output Encoding, State Assignment and Four-Level Boolean Minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 10(1):13–27, January 1991.

[27] E. B. Eichelberger. Hazard Detection in Combinational and Sequential Circuits. *IBM J. Res. and Devep.*, pages 90–99, March 1965.

[28] A. El-Maleh, T. E. Marchok, J. Rajski, and W. Maly. On Test Set Preservation of Retimed Circuits. In *Proc. of the Design Automation Conf.*, pages 176–182, San Francisco, CA, June 1995.

[29] L. Entrena and K.-T. Cheng. Sequential Logic Optimization by Redundancy Addition and Removal. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 310–315, Santa Clara, CA, November 1993.

[30] M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman and Co., 1979.

[31] A. Ghosh, S. Devadas, and A. R. Newton. Test Generation for Highly Sequential Circuits. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 362–365, Santa Clara, CA, November 1989.

[32] A. Grasselli and F. Luccio. A Method for Minimizing the Number of Internal States in Incompletely Specified Sequential Networks. *IRE Transactions on Electronic Computers*, EC-14(3):536–557, June 1965.

[33] J. Grodstein, E. Lehman, H. Harkness, H. Touati, and B. Grundmann. Optimal Latch Mapping and Retiming within a Tree. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 242–245, San Jose, CA, November 1994.

[34] J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines.* Intl. Series in Applied Mathematics. Prentice-Hall, Englewood Cliffs, N.J., 1966.

[35] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[36] M. A. Iyer, D. E. Long, and M. Abramovici. Identifying Sequential Redundancies Without Search. In *Proc. of the Design Automation Conf.*, Las Vegas, NV, June 1996.

[37] Seh-Woong Jeong. *Binary Decision Diagrams and their Applications to Implicit Enumeration Techniques in Logic Synthesis.* PhD thesis, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO 80309, 1992.

[38] J. S. Jephson, R. P. McQuarrie, and R. E. Vogelsberg. A Three-Value Computer Design Verification System. *IBM J. Res. and Devep.*, pages 178–188, 1969.

[39] T. Kam and P. A. Subramanyam. Comparing Layouts with HDL Models: A Formal Verification Technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 14(4):503–509, April 1995.

[40] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. A Fully Implicit Algorithm for Exact State Minimization. In *Proc. of the Design Automation Conf.*, pages 684–690, San Diego, CA, June 1994.

[41] S. C. Kleene. Representation of Events in Nerve Nets and Finite Automata. In C. E. Shannon and M. McCarthy, editors, *Automata Studies*, number 34 in Annals of Mathematics Studies, pages 3–41. Princeton University Press, Princeton, NJ, 1956.

[42] D. Kozen. Lower Bounds for Natural Proof Systems. In *Proc. Symp. on Foundations of Computer Science*, pages 254–266, Providence, RI, October 1977.

[43] W. Kunz and P. R. Menon. Multi-Level Logic Optimization by Implication Analysis. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 6–13, San Jose, CA, November 1994.

[44] R. P. Kurshan. *Automata-Theoretic Verification of Coordinating Processes*. Princeton University Press, 1995.

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

[46] B. Lin. Efficient Symbolic Support Manipulation. In *Proc. Intl. Conf. on Computer Design*, pages 513–516, Cambridge, MA, October 1993.

[47] B. Lin, G. de Jong, and T. Kolks. Modeling and Optimization of Hierarchical Synchronous Circuits. In *Proc. European Design and Test Conf.*, pages 144–149, Paris, France, March 1995.

[48] B. Lin, H. J. Touati, and A. R. Newton. Don't Care Minimization of Multi-level Sequential Logic Networks. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 414–417, Santa Clara, CA, November 1990.

[49] Bill Lin. *Synthesis of VLSI Design with Symbolic Techniques*. PhD thesis, Electronics Research Laboratory, University of California, Berkeley, CA 94720, November 1991. Memorandum No. UCB/ERL M91/105.

[50] D. E. Long, M. A. Iyer, and M. Abramovici. Identifying Sequentially Untestable Faults Using Illegal States. In *Proc. VLSI Test Symposium*, pages 4–11, Princeton, NJ, April 1995.

[51] David E. Long. BDD Manipulation Library. Public software. Carnegie Mellon University, Pittsburgh, PA, June 1993. `ftp://emc.cs.cmu.edu/pub/bdd/bddlib.tar.Z`.

[52] S. Malik. Analysis of Cyclic Combinational Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(7):950–956, July 1994.

[53] S. Malik, E. M. Sentovich, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Retiming and Resynthesis: Optimization of Sequential Networks with Combinational Techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 10(1):74–84, January 1991.

[54] Sharad Malik. *Combinational Logic Optimization Techniques in Sequential Logic Synthesis*. PhD thesis, Electronics Research Laboratory, University of California, Berkeley, CA 94720, November 1990. Memorandum No. UCB/ERL M90/115.

[55] T. E. Marchok, A. El-Maleh, W. Maly, and J. Rajski. Test Set Preservation under Retiming Transformation. Technical Report CMUCAD-94-23, Carnegie Mellon University, May 1994. Presented at Intl. Test Synthesis Workshop, Santa Barbara, CA, May 1994.

[56] Kenneth L. McMillan. Cadence Berkeley Labs, Berkeley, CA. Personal communication, 1995.

[57] I. M. Niven and H. S. Zuckerman. *An Introduction to the Theory of Numbers*. John Wiley & Sons, New York, NY, 1972.

[58] M. Pandey, A. Jain, R. E. Bryant, D. Beatty, G. York, and S. Jain. Extraction of Finite State Machines from Transistor Netlists by Symbolic Simulation. In *Proc. Intl. Conf. on Computer Design*, pages 596–601, Austin, TX, October 1995.

[59] C. Pixley. A Computational Theory and Implementation of Sequential Hardware Equivalence. In E. M. Clarke and R. P. Kurshan, editors, *Proc. of the Workshop on Computer-Aided Verification*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 293–320. American Mathematical Society, June 1990.

[60] C. Pixley. A Theory and Implementation of Sequential Hardware Equivalence. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 11(12):1469–1494, December 1992.

[61] Carl Pixley. Motorola, Inc., Austin, TX. Personal communication, 1993.

[62] I. Pomeranz and S. M. Reddy. Classification of Faults in Synchronous Sequential Circuits. *IEEE Transactions on Computers*, 42(9):1066–1077, September 1993.

[63] I. Pomeranz and S. M. Reddy. On Removing Redundancies from Synchronous Sequential Circuits with Synchronizing Sequences. Technical Report 12-20-1993, Electrical and Computer Engineering Department, University of Iowa, Iowa City, IA 52242, 1993.

[64] Stefano Quer. Politecnico di Torino, Torino, Italy. Personal communication, 1995.

[65] J.-K. Rho, G. D. Hachtel, F. Somenzi, and R. M. Jacoby. Exact and Heuristic Algorithms for the Minimization of Incompletely Specified State Machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(2):167–177, February 1994.

[66] Richard L. Rudell. Synopsys, Inc., Mountain View, CA. Personal communication, 1995.

[67] A. Saldanha, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Satisfaction of Input and Output Encoding Constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(5):589–602, May 1994.

[68] J. V. Sanghavi, R. K. Ranjan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. High Performance BDD Package Based on Exploiting Memory Hierarchy. In *Proc. of the Design Automation Conf.*, Las Vegas, NV, June 1996.

[69] W. J. Savitch. Relationship between Nondeterministic and Deterministic Tape Complexities. *Journal of Computer and System Sciences*, pages 177–192, 1970.

[70] H. Savoj and R. K. Brayton. Observability Relations and Observability Don't Cares. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 518–521, Santa Clara, CA, November 1991.

[71] H. Savoj, R. K. Brayton, and H. Touati. Extracting Local Don't Cares for Network Optimization. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 514–517, Santa Clara, CA, November 1991.

[72] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *Proc. Intl. Conf. on Computer Design*, pages 328–333, Cambridge, MA, October 1992.

[73] E. M. Sentovich, V. Singhal, and R. K. Brayton. Multiple Boolean Relations. In *Workshop Notes of the Intl. Workshop on Logic Synthesis*, Tahoe City, CA, May 1993.

[74] J.-J. Shen, Z. Hasan, and M. J. Ciesielski. State Assignment for General FSM Networks. In *Proc. Intl. Conf. on Computer Design*, pages 245–249, Cambridge, MA, October 1992.

[75] N. Shenoy and R. Rudell. Efficient Implementation of Retiming. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 226–233, San Jose, CA, November 1994.

[76] T. R. Shiple, R. Hojati, A. L. Sangiovanni-Vincentelli, and R. K. Brayton. Heuristic Minimization of BDDs Using Don't Cares. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 225–231, San Diego, CA, June 1994.

[77] T. R. Shiple, V. Singhal, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Analysis of Combinational Cycles in Sequential Circuits. In *Proc. Intl. Symposium on Circuits and Systems*, Atlanta, GA, May 1996.

[78] K. J. Singh and P. A. Subramanyam. Extracting RTL Models from Transistor Netlists. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 11–17, San Jose, CA, November 1995.

[79] V. Singhal, Y. Watanabe, and R. K. Brayton. Heuristic Minimization of Synchronous Relations. In *Proc. Intl. Conf. on Computer Design*, pages 428–433, Cambridge, MA, October 1993.

[80] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 130–133, Santa Clara, CA, November 1990.

[81] H. J. Touati and R. K. Brayton. Computing the Initial States of Retimed Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 12(1):157–162, January 1993.

[82] Tiziano Villa. University of California at Berkeley, Berkeley, CA. Personal communication, 1995.

[83] Y. Watanabe and R. K. Brayton. The Maximum Set of Permissible Behaviors for FSM Networks. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 316–320, Santa Clara, CA, November 1993.

[84] Y. Watanabe and R. K. Brayton. State Minimization of Pseudo Non-Deterministic FSM's. In *Proc. European Design and Test Conf.*, pages 184–191, Paris, France, 1994.

[85] Elizabeth S. Wolf. *Hierarchical Models of Synchronous Circuits for Formal Verification and Substitution.* PhD thesis, Stanford University, Computer Systems Laboratory, Stanford, CA 94305-4055, 1995. Report No. CSL-TR-95-1557.