SYNTHESIS AND OPTIMIZATION OF SYNCHRONOUS LOGIC CIRCUITS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Maurizio Damiani
May, 1994

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Giovanni De Micheli
(Principal Adviser)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

David L. Dill

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Teresa Meng

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies

Abstract

The design automation of complex digital circuits offers important benefits. It allows the designer to reduce design time and errors, to explore more thoroughly the design space, and to cope effectively with an ever-increasing project complexity.

This dissertation presents new algorithms for the logic optimization of combinational and synchronous digital circuits. These algorithms rely on a common paradigm. Namely, global optimization is achieved by the iterative local optimization of small subcircuits.

The dissertation first explores the combinational case. Chapter 2 presents algorithms for the optimization of subnetworks consisting of a single-output subcircuit. The design space for this subcircuit is described implicitly by a Boolean function, a so-called *don't care* function. Efficient methods for extracting this function are presented.

Chapter 3 is devoted to a novel method for the optimization of multiple-output subcircuits. There, we introduce the notion of *compatible gates*. Compatible gates represent subsets of gates whose optimization is particularly simple.

The other three chapters are devoted to the optimization of synchronous circuits. Following the lines of the combinational case, we attempt the optimization of the gate-level (rather than the state diagram -level) representation. In Chapter 4 we focus on extending combinational techniques to the sequential case. In particular, we present algorithms for finding a synchronous *don't care* function that can be used in the optimization process.

Unlike the combinational case, however, this approach is exact only for pipeline-like circuits. Exact approaches for general, acyclic circuits are presented in Chapter 5. There, we introduce the notion of **synchronous recurrence equation.** Eventually, Chapter 6 presents methods for handling feedback interconnection.

Acknowledgements

This thesis would not have been possible without the perseverance and guidance of my thesis advisor, Professor Giovanni De Micheli. His continuous support, encouragement, supervision and and constructive criticism made him a reliable reference in the most critical moments of my research.

I wish to thank Professor D. Dill for his key suggestions in the many discussions on my work, and the other members of my reading and oral committees, Proff. T. Meng and C. Quate, for their time and patience.

I also need to thank my group mates, Polly Siegel, David Ku, Dave Filo, Rajesh Gupta, Frederic Mailhot, Thomas Truong, for making my stay at Stanford especially enjoyable.

Special thanks go to Jerry Yang for sharing late night efforts in code and paper writing. He showed plenty of tolerance and self-control towards an impatient writer.

I must also acknowledge the dear friends outside my research group. Among them John and Noriko Wallace, Tony and Lydia Pugliese.

But I am most indebted to my parents for their love, caring, and understanding, and to my wife Elena, for sharing this experience with me, and helping me in making it through the difficult times.

Contents

Abstract **iii**

Acknowledgements **iv**

1 Introduction **1**

1.1 VLSI and logic synthesis 1

1.2 Previous work and contributions of this thesis. 5

1.2.1 Combinational logic optimization. 5

1.2.2 Synchronous logic optimization. 10

2 Combinational networks **17**

2.1 Introduction 17

2.2 Terminology 18

2.2.1 Boolean functions and their representations 18

2.2.2 Combinational circuits and logic networks. 19

2.2.3 Specifications for combinational networks. 21

2.2.4 Optimization of combinational multiple-level circuits 21

2.3 Perturbation analysis of combinational networks. 23

2.3.1 Single-vertex optimization and observability *don't cares* 26

2.4 Multi-vertex optimization and compatible *don't cares* 41

2.5 Approximating observability *don't cares* 53

2.5.1 Experimental results. 59

2.6 Summary 61

3 Multi-vertex optimization with compatible gates **63**
3.1 Related Previous Work . . . 65
3.1.1 Two-level Synthesis . . . 65
3.1.2 Boolean Relations-based Multiple-level Optimization . . . 66
3.2 Compatible Gates . . . 69
3.3 Optimizing Compatible Gates . . . 70
3.3.1 Implicant Extraction . . . 70
3.3.2 Covering Step . . . 71
3.4 Finding Compatible Gates . . . 75
3.5 Unate Optimization . . . 81
3.5.1 Optimizing Unate Subsets . . . 81
3.5.2 Implicant Extraction . . . 81
3.5.3 Covering Step . . . 84
3.6 Implementation and Results . . . 87
3.7 Summary . . . 89

4 Acyclic synchronous networks **91**
4.1 Terminology . . . 92
4.1.1 Synchronous logic networks. . . . 92
4.1.2 Sequences and sequence functions. . . . 93
4.1.3 Pattern expressions and functions. . . . 94
4.1.4 Functional modeling of synchronous circuits. . . . 98
4.2 Sequential *don't cares* . . . 99
4.2.1 Retiming-invariant *don't care* conditions . . . 100
4.2.2 Controllability and observability *don't cares* . . . 101
4.3 Local optimization of acyclic networks . . . 105
4.3.1 Internal observability *don't care* conditions. . . . 106
4.4 Computation of observability *don't cares* in acyclic networks . . . 112
4.5 Experimental results. . . . 114
4.6 Summary . . . 115

5 Recurrence Equations **116**
5.1 Introduction . . . 116
5.2 Synchronous Recurrence Equations . . . 119
5.2.1 Optimization of synchronous circuits by recurrence equations . . 119
5.3 Finding acyclic solutions. . . . 120
5.3.1 Representing feasible solutions . . . 120
5.4 Minimum cost solutions. . . . 125
5.4.1 Extraction of primes . . . 126
5.4.2 Covering Step. . . . 130
5.5 Recurrence equations for sequential optimization. . . . 132
5.5.1 Image of a SRE. . . . 133
5.6 Implementation and experimental results. . . . 134
5.7 Summary. . . . 135

6 Cyclic synchronous networks **137**
6.1 Modeling of cyclic networks. . . . 138
6.1.1 The reset assumption. . . . 139
6.2 Feedback and external controllability *don't cares* . . . 140
6.2.1 *Don't cares* and state-space traversals . . . 144
6.3 Perturbation analysis of cyclic networks. . . . 145
6.3.1 An iterative procedure for external observability *don't cares* . . . 148
6.4 Experimental results. . . . 151
6.5 Summary . . . 152

7 Conclusions **153**

Bibliography **155**

Chapter 1

Introduction

Logic synthesis is the process of transforming a register-transfer level description of a design into an optimal logic-level representation. Traditionally, it has been divided into combinational and sequential synthesis. This chapter first reviews the VLSI design process, describing the role played by logic synthesis, its status, and previous contributions in the field. It then provides an outline of this dissertation, highlighting the contributions of this work.

1.1 VLSI and logic synthesis

Very Large Scale Integration (VLSI) has emerged as a central technology for the realization of complex digital systems. The benefits in terms of performance, reliability, and cost reduction of integrating large systems on a single chip have pushed designs from the tens of thousands of transistors into the millions in just over a decade.

Computer aids play an important role in coping with the complexity of such designs, by partitioning them into a sequence of well-defined steps. Quality and time-to market of the final product are also improved by automating the most tedious, lengthy and error-prone phases of the project.

The design process typically begins with a functional description of the desired functionality by means of a *high-level* description language. Several languages have been developed to this purpose (VHDL, VerilogHDL, HardwareC) [1].

High-level synthesis is the first design step for which CAD tools have been developed. It consists of mapping a functional description of a circuit, along with a set of area and performance constraints, into a *structural* one, in terms of registers and combinational functional units (*"primitives"*), such as ports, adders, multipliers, shifters, comparators. At this stage, the view of a circuit is therefore largely independent from the format of data and control signals [1, 2].

The output of high-level synthesis is a *register-transfer level* (RTL) representation of the circuit. Such representations are typically divided into data path and control portions. The task of the control unit is to activate portions of the data path according to a given schedule, so as to achieve the desired computation. The selection of a schedule requiring a minimal number of computational resources (and possibly satisfying a given timing constraint) is a classical problem of high-level synthesis.

Logic synthesis follows immediately high-level synthesis. Its task is the mapping of RTL descriptions into *gate-level* circuit representations. Logic synthesis therefore introduces constraints on the data representations and on the types of primitives used (ANDs, ORs, D-type flip-flops etc...)

The level of abstraction of high-level synthesis does not allow accurate estimates of the figures of merit of a circuit. Consequently, a straightforward mapping of an RTL design into a logic circuit very seldom meets area, speed, or power requirements. Optimization at the logic level is therefore a necessary step: indeed, its relevance has made it the subject of intense research ever since the inception of electronic computers. The following example highlights the different nature of high-level and logic optimization.

Example 1.

Consider the fragment of code in part (a) of the following figure. The quantity $x + y$ is compared against two constant thresholds $n1$ and $n2$. A typical optimization step at high level consists of transforming this code into the code of part (b), by means of standard software compilation techniques. This optimization leads to the RTL representation shown in Fig. (1.1-a).

```
while(x+y < n1) do {
        :
     while (x+y < n2) do {
              :
              update(x,y)
      }
      update(x,y);
}
```

(a)

```
t = x+y;
while(t < n1) do {
        :
     while (t < n2) do {
              :
              update(x,y)
              t = x+y;
     }
     update(x,y);
     t = x+y
}
```

(b)

At the logic level, this implementation is further simplified, as will be seen later, by merging the adder and comparator, and by regarding the entire block as realizing a combinational logic function. This type of optimization requires the knowledge of the data representation for x and y (*i.e.* 2's complement, etc ...), and it is therefore impossible at the RTL level. □

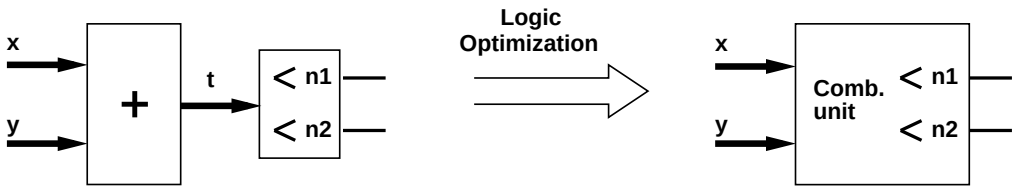


Figure 1.1: A logic optimization step.

Over the years, the breadth of the field has led to its fragmentation into a number of disciplines, most notably into a distinction between *combinational* and *sequential* logic synthesis: while combinational logic circuits have been considered mostly as tools for the realization of *functions*, sequential circuits have mostly been regarded from a state-machine viewpoint and manipulated accordingly. This distinction is less motivated in a

VLSI context, where RTL representations of data path and control are often provided directly in terms of registers and combinational units.

The notion of *degrees of freedom* (or *don't care* conditions) is central to every step of the synthesis process. In high-level synthesis, slacks in the scheduling of operations represent degrees of freedom which can be used advantageously for sharing computational resources. These slacks represent degrees of freedom also on the control unit: some control signals can be delayed or anticipated to simplify the control circuitry at the sequential logic level. Other *don't cares* at the logic level represent our knowledge that some input combinations or input sequences cannot occur, or that the response to some input or input sequence sequence is not sampled.

Don't care conditions that arise directly from the specification are due to the interfacing of the system in a larger environment. Similarly, the embedding of a functional block in a larger circuit results in *don't care* conditions on its functionality. For example, in the circuit of Fig. (1.1), there are degrees of freedom on the adder, and they arise because of the "filtering" effect of the comparator.

Unlike *don't cares* given by a specification, those due to embedding are *implicit*. The mathematical characterization, derivation, and efficient use of such *don't care* conditions are therefore very relevant issues in logic optimization.

This dissertation is concerned with these three topics at the combinational and sequential synthesis level. The mathematical characterization is in terms of *perturbation theory*: *don't care* conditions are interpreted as the set of possible functional perturbations of an original description.

Throughout the thesis, perturbation theory is used in several contexts, in particular for obtaining efficient *don't care* -extraction algorithms and for the classification of *don't care* conditions according to their complexity.

These algorithms have been implemented in a logic optimization tool, *Achilles*, and integrated with novel algorithms for combinational and sequential logic optimization algorithms. A more detailed description of these contributions is presented in the upcoming section.

Achilles is part of Olympus, a CAD system for VLSI synthesis being developed at Stanford. Achilles has been applied successfully to the optimization of several large

combinational and synchronous logic benchmark circuits.

1.2 Previous work and contributions of this thesis.

1.2.1 Combinational logic optimization.

Combinational logic optimization is traditionally divided into two-level and multiple-level logic synthesis.

Two-level synthesis targets the realization of combinational logic functions by a two-layer interconnection of elementary logic gates, such as AND-OR, NAND-NAND, etc.

Early research has led to efficient algorithms for the synthesis of combinational logic circuits in two-level form. Exact algorithms were developed originally in the early 50's by Quine [3] and McCluskey [4], and are practical for the synthesis of functions with at most a dozen inputs.

The popularity of PLA-based synthesis in the early 80's revamped the interest in their approach. The necessity of synthesizing functions with a very large number of inputs and outputs has led to the development of several effective approximate solvers, including MINI [5], and ESPRESSO [6], as well as to the re-visitation of exact approaches [7]. These solvers have been used for the optimization of very large PLAs, with over fifty inputs and outputs and thousands of product terms, and their efficiency makes them the basic engine for most current logic optimization tools.

Degrees of freedom in classical two-level synthesis are represented by a *don't care* function. This function represents input combinations that cannot occur and inputs that generate irrelevant output values.

Somenzi *et al.* considered in [8] *don't care* conditions expressed by a *Boolean relation*. Boolean relations specify the functionality of a combinational circuit by associating with each input combination a *set* of possible outputs. Further research in the area showed, however, that unlike *don't care* functions, this type of degrees of freedom is much more difficult to use, and efficient optimizers for this case are the object of ongoing research [9].

Multiple-level combinational logic synthesis targets the implementation of a logic

function by an arbitrary, acyclic network of logic gates. The interest in multiple-level synthesis is due to the fact that very often multiple-level interconnections of logic gates are much faster and more compact than two-level implementations. Some simple functions, like 32-bit parity, are indeed practically impossible to realize in a two-level form, while having simple multiple-level realizations.

Similarly to two-level synthesis, exact multiple-level logic synthesis algorithms have been known for a long time [10, 11, 12, 13]. All such methods are essentially based on an orderly, exhaustive enumeration of all possible acyclic graphs. For example, Davidson [13] considers NAND networks. His procedure starts by enumerating all sets of possible functions whose NAND can yield the desired function. Once such a set is found, the procedure is repeated recursively, until a simple function (an input or its complement) is met or a cost limit is exceeded. The size and complexity of the search space is such that none of the exact methods could prove itself practical for functions requiring more than a dozen gates, and the difficulty of exact multiple-level synthesis was referenced as one motivation for later work in complexity theory [14].

Nowadays, the optimization of multiple-level logic is carried out almost exclusively by approximate methods developed over the past decade. These methods consist mainly of the iterative refinement of an initial network, until key cost figures (typically area or delay) meet given requirements or no improvement occurs. Refinement is carried out by identifying subnetworks to be optimized and replacing them by simpler, optimized circuits. Iteration is carried over until cost figures no longer improve.

An important observation in this context is that the embedding of a functional block in a larger circuit results in *don't care* conditions on its functionality:

Example 2.

> Consider the adder/comparator structure given in Fig. (1.1). Suppose, for simplicity, that x and y are two two-bit quantities, and that $n1$ and $n2$ are the numbers 3 and 4.
>
> Consider optimizing the circuitry producing the middle bit add_1 of the adder, shaded in Fig. (1.2). The function realized at that output is shown in the

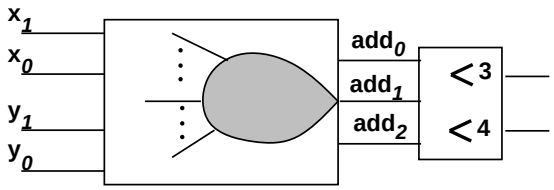


Figure 1.2: A two-bit adder. Shading indicates the circuitry generating the output add_1.

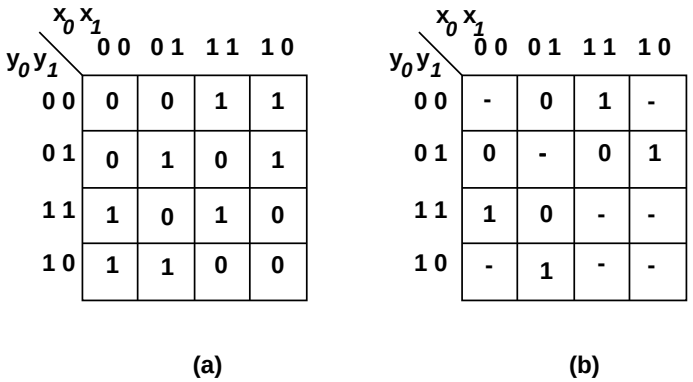


Figure 1.3: a) Karnaugh map of the function realized at add_1. b) *Don't care* conditions at add_1, represented by a symbol '-'.

Karnaugh map of Fig. (1.3). Consider the situations where the input combinations result in a sum larger than, or equal to, 4. The MSB of the adder is 1, and both outputs of the comparator will take value 0, *regardless* of the values taken by add_0 and add_1: the output add_1 has become irrelevant. By similar reasonings, one gets that the value of add_1 is also irrelevant whenever the inputs produce a sum equal to 0 or 2. These *don't care* conditions are shown in table of Fig. (1.3-b). □

Optimization methods can be classified by the size of the subnetworks considered (*e.g*, consisting of a single-output logic gate versus multiple-output subcircuits) and by the complexity of the optimization style. There are two main optimization styles, *algebraic* and *Boolean*, in order of complexity.

Algebraic methods are based on treating logic expressions as ordinary polynomials over a set of logic variables. Common factors in such polynomials are extracted and

a logic network is restructured and simplified accordingly. Algebraic methods form the backbone of the interactive logic optimization system MIS [15], developed at Berkeley by Brayton *et al.*

Example 3.

In the circuit of Fig. (1.4), the primary outputs are expressed by $x = (a+b)d$ and $y = ac+bc$. By recognizing that $y = (a+b)c$ and extracting the common factor $a + b$, the circuit of Fig. (1.4-b) is obtained. □

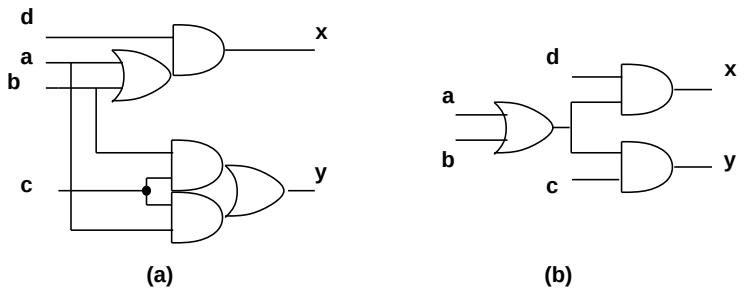


Figure 1.4: a) original circuit. b) Circuit optimized by factorization.

Algebraic methods do not take full advantage of the properties of Boolean algebra. For this reason, they take advantage of *don't care* conditions only to a limited extent. Boolean methods target instead the full use of implicit *don't cares* . In order to construct a *don't care* -based logic optimization system, it is therefore necessary to :

- characterize first mathematically such *don't care* conditions, and
- provide algorithms for their efficient extraction and use.

With regards to the characterization problem in the combinational case, Bartlett *et al.* [16] and Muroga *et al.* [17] have shown that the *don't cares* on single-output subnetworks can be described by an ordinary Boolean function, termed the *don't care function* of the gate. An important consequence of this property is that ordinary two-level synthesis algorithms can be applied. Boolean optimization of single-output subnetworks is a part of the program MIS.

Somenzi *et al.* [8] have showed that the optimization of arbitrary multiple-level logic gates requires instead modeling *don't care* conditions by a Boolean relation:

Example 4.

> Consider optimizing simultaneously the entire adder in the adder/comparator of Fig. (1.1). The functional requirement on the adder are the following: corresponding to each input combination with sum less than three (namely, $x_1x_0y_1y_0 = 0000, 0001, 0010, 0100, 0101$ or 1000) , the adder output can be any pattern drawn from the set $A = \{000, 001, 010\}$, as all such patterns result in the same output at the comparator. Similarly, corresponding to all input combinations with sum 4 or more, the adder output can be any pattern drawn from the set $B = \{100, 101, 110, 111\}$. If the sum is three, then the network output is drawn from the one-element set $C = \{011\}$. □

This specification style cannot be summarized into a set of independent *don't care* conditions on the individual outputs. For the circuit of Example (4), corresponding to the input pattern 0000, the first and second output are both allowed to change, but not simultaneously: choosing $add_1(0000) = 1$, however, implies that $add_2(0000)$ must be 0. This is reflected by complications in the subsequent logic optimization step [8]. Approximations to Boolean relations are represented by *compatible don't cares* , first introduced by Muroga *et al.* [1]. Informally, compatible *don't cares* represent *don't care* functions that allow us to optimize each vertex independently in multiple-vertex optimization. Since compatible *don't cares* represent only a subset of degrees of freedom, the key issue in the extraction of compatible *don't cares* is their *maximality*.

Contributions to combinational synthesis

Chapter 2 of this dissertation is concerned with the problem of extracting *don't care* representations (be it *don't care* functions, Boolean relations, or compatible *don't cares*) in combinational networks in an efficient way. In this respect, the following contributions are presented:

[1] Muroga actually referred to *compatible sets of permissible functions*

- The general problem of characterizing *don't care* conditions is cast uniformly in terms of *perturbation theory*. The modification of one or more logic functions inside a network is regarded as the introduction of local errors, that are modeled by added error signals. The conditions for which such errors are tolerated (*i.e.* they do not affect the primary outputs) represent the degrees of freedom available for optimization.

- Efficient algorithms for the derivation of *don't care* functions are presented. The efficiency of such algorithms is drawn from a *local* paradigm: the *don't care* function of a logic gate is derived from that of adjacent gates by means of local rules. Such algorithms are completed by a suite of methods for approximating such rules in case the explicit representations of *don't care* functions become intractable. The theoretical understanding of the problem provided by perturbation analysis provides a means for evaluating previous approaches to the problem.

- New algorithms are presented for deriving compatible *don't cares* . It is argued that maximal compatible *don't cares* cannot be derived on a local basis. Those presented in this work constitute, however, the best approximations known so far.

In Chapter 3, the problem of multiple-vertex optimization is considered from a different angle. The difficulty of multiple-vertex optimization is due in part to the arbitrariness of the subnetwork selected for optimization. This difficulty is circumvented by introducing the notion of *compatible gates.* A set of compatible gates is a subset of gates for which the problem of solving a Boolean relation is substantially simplified, and in particular ordinary two-level synthesis algorithms can be used for exact optimization. An approach for multiple-vertex optimization based on the search of compatible gates, instead of optimizing arbitrary subnetworks, is presented.

1.2.2 Synchronous logic optimization.

The presence of clocked memory elements (for reference, assumed to be D-type flip-flops) and possibly of feedback distinguishes synchronous circuits from combinational ones.

A common model of such circuits is the *finite-state machine* (FSM) model, shown in Fig. (1.5). Flip-flops and combinational logic elements are grouped into a register and a combinational network, respectively. The content of the register is termed the *state* of the circuit, and the combinational portion implements output and next-state functions. A finite-state machine description of a circuit is typically provided in terms of a *state diagram* (also shown in Fig. (1.5) or *state table*.

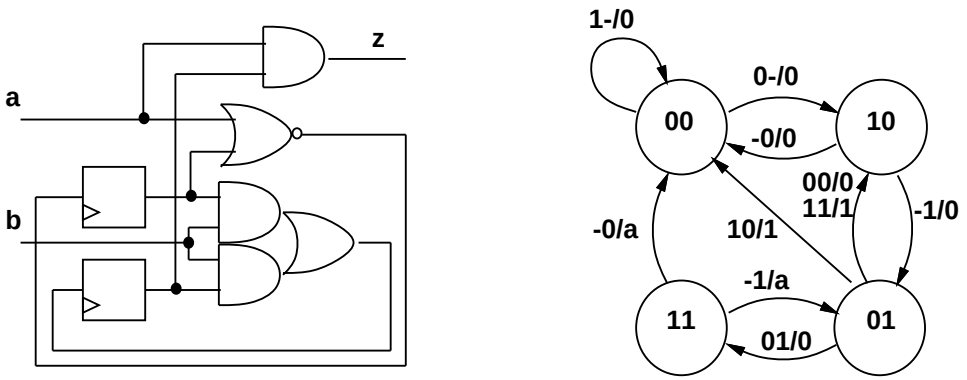


Figure 1.5: A synchronous circuit and its finite-state machine model.

The classical approach towards the optimization of synchronous circuits consists of extracting their FSM description and then resorting to known FSM synthesis algorithms.

FSM synthesis is a classic subject of switching theory. The process is typically divided into three main steps, consisting of state minimization, state assignment, and the synthesis of the combinational portion. State minimization has two main objectives, namely to minimize the number of flip-flops and to increase the number of unused combinations of state variables. Such unused combinations represent in fact *don't care* conditions for the combinational portion.

State assignment is the process of encoding each state in a binary format. It defines to a large extent the functionality of the combinational circuit, and therefore good state assignment algorithms are still object of research. Heuristics targeting two-level [18] and multiple-level [19] implementations of the combinational logic have been considered. Other strategies include the *decomposition* of a FSM into a set of smaller, interconnected machines [20, 21], for which the optimal state assignment problem can be solved more

accurately.

Similarly to the combinational case, an exact synthesis algorithm for finite-state machines is also available, but it reduces essentially to the orderly enumeration of all possible state assignments, and it is impractical for all but very small machines.

Unfortunately, the difficulty of evaluating the effect of state manipulation operations (most notably state assignment) on the final hardware makes it impossible to *drive* this approach towards an actual reduction of the original circuit. It is also worth noting that in modern VLSI technology the cost of flip-flops is actually comparable to that of a few logic gates. The significance of state minimization is in this context greatly reduced, in favor of more general network restructuring approaches.

These difficulties motivate the search of algorithms targeted at the direct optimization of synchronous netlists. The underlying model for this style of optimization is the *synchronous logic network*. Informally, a synchronous logic network is a generalization of the combinational logic network, with vertices representing logic elements and edges representing logic dependencies. Registers are modeled by *delay elements*, and intervene in the description of the logic as delay labels in logic expressions.

One optimization strategy, proposed originally by Malik *et al.* [22] and later refined by Dey *et al.* [23], is *peripheral retiming*. Retiming is a circuit transformation originally developed by Leiserson *et al.* [24] for the optimal placement of delay elements in a circuit so as to minimize the clock period. The basic step of retiming is illustrated in Fig. (1.6).

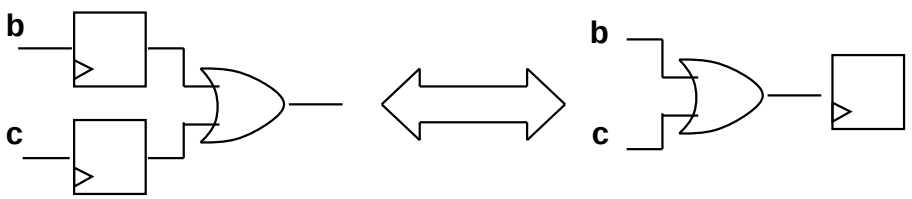


Figure 1.6: An elementary retiming operation.

Peripheral retiming consists of identifying pipeline-like subnetworks [2] and pushing all registers to their periphery by retiming, so as to evidence the underlying combinational

[2]Informally, a pipeline is a synchronous circuit where all paths from each input to each output contain the same number of delay elements.

structure. This portion is then optimized using ordinary combinational techniques, and eventually registers are re-distributed along the pipeline. An example of this transformation is shown in Fig. (1.8).

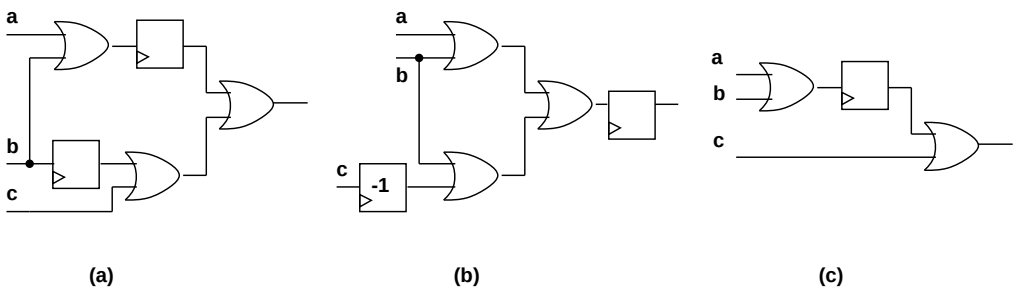


Figure 1.7: (a) A pipeline circuit. (b) A peripherally-retimed version. Notice the temporary introduction of a negative-delay register. (c) Optimized circuit, after the elimination of the negative-delay register.

In practice, in most circuits pipelined subnetworks are too small or have too many outputs, which leaves little room for optimization. A second difficulty occurs when different inputs have different register counts to the primary outputs, as in Fig. (1.7). In this case, peripheral retiming requires the introduction of " negative-delay" registers. After optimization, it may be impossible to eliminate such registers, thereby invalidating the result. One such instance is the circuit of Fig. (1.8), borrowed from [22].

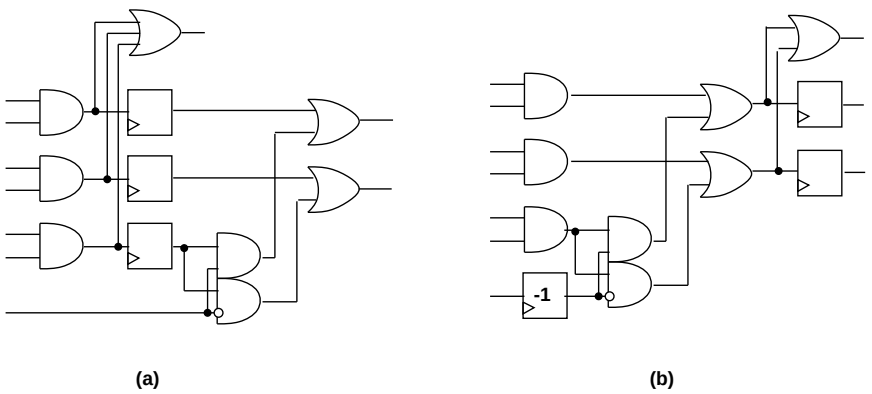


Figure 1.8: (a) Original circuit and (b), an unrealizable optimized version.

The extension of algebraic operations for synchronous logic networks was considered

by De Micheli in [25]. An example of synchronous algebraic transformation is illustrated by the following example.

Example 5.

The functionality of the circuit of Fig. (1.9-(a)) can be expressed by the two relations $x = a_1 + b$ and $y = c(a_2 + b_1)$, where the subscripts indicate the delays associated with a and b. The expression $a_2 + b_1$ is then an algebraic factor of y, and coincides with the delay by 1 of x. Output y can then be expressed as cx_1, leading to the realization shown in Fig. (1.9-(b)). □

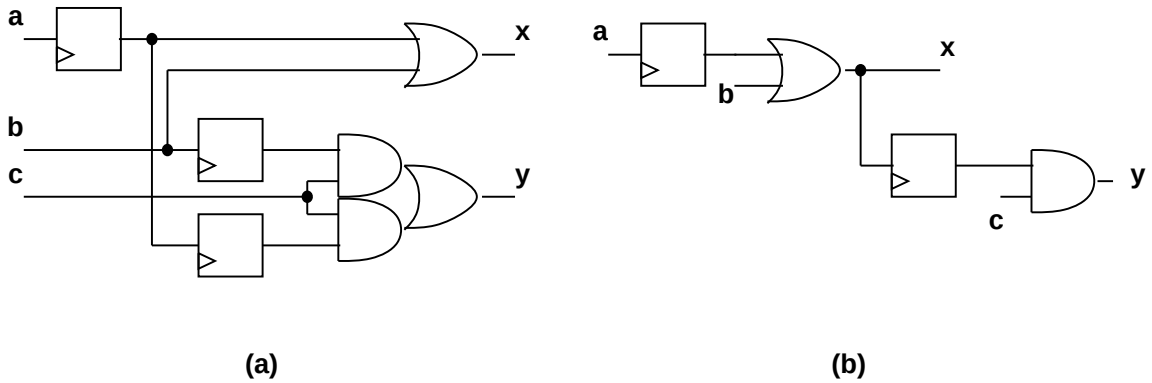


Figure 1.9: A circuit before (a) and after (b) optimization by synchronous factorization.

These optimization methods are typically not powerful enough to capture the optimization space for a synchronous circuit, and Boolean optimization models are required.

Unlike the combinational case, *don't care* conditions for synchronous circuits have been far less characterized. Classical works on FSMs considered almost exclusively *incompletely specified* FSMs, *i.e.* FSMs whose next-state functions or output functions contain *don't care* entries. The synthesis path for incompletely specified FSMs follows closely the one for ordinary FSMs, the main difference being the complications added by the incomplete specification to the state minimization step.

This model is however inadequate to interpret the *don't care* conditions that arise in the VLSI context. For example, it often impossible to cast degrees of freedom in the

timing of the output signals of a FSM into *don't care* entries in its next-state or output function:

Example 6.

> Consider the case of a simple fragment of a control unit, whose task is to issue an activation pulse one or two clock periods after receiving an input pulse. Let s denote the state of the control immediately after receiving the control signal. In s it is necessary to choose whether the FSM should issue an output pulse and return to the quiescent start state or should count one more clock period. This choice cannot, however, be represented by a *don't care* condition on the next state entry, or remaining in state s would be included incorrectly among the possible options. □

A second problem is the characterization of the *don't cares* associated with the embedding of a synchronous circuit in a larger one. Only the case of two cascaded FSMs (shown in Fig. (1.10)) has been in practice addressed in the literature. Kim and Newborn [26] showed that the limitations in the sequences that can be asserted by M_1 can be used for the optimization of M_2, even if this information cannot be represented in terms of *don't care* entries on any state of M_2. Their optimization algorithm was rediscovered later by Devadas [27] and by Rho and Somenzi [28]. Heuristics that attempt to capture the filtering effect of M_2 for the optimization of M_1 have also been considered in the two latter works, but they lack a formal setting. Moreover, the scope of these works is limited by the nature of the topologies and optimization steps considered, and by a lack of a general model of the *don't care* conditions that can be associated with a synchronous circuit.

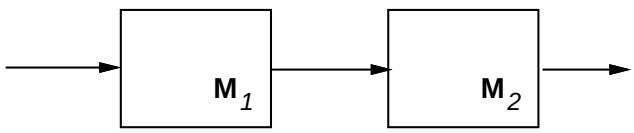


Figure 1.10: Cascaded finite-state machines.

Contributions to sequential logic synthesis

This thesis presents an analysis of *don't care* conditions for synchronous networks that is complete both in terms of theory and engineering techniques. Chapter 4 of this thesis is in particular concerned with the extension of perturbation theory to the synchronous case. With regards to this point, the following contributions are presented:

- the limits to which sequential *don't care* conditions can be represented by a *don't care* function are explored. It is in particular shown that *don't care* functions represent fully the degrees of freedom associated with a vertex only in the special case of pipelines . Methods for the correct handling of these *don't cares* are presented.

- For non-pipelined networks, *don't care* functions represent approximations of the full *don't care* conditions. Algorithms for deriving efficiently *don't care* functions are then considered. Two cases are distinguished, depending on whether feedback is present or not. In the first case, extensions of local algorithms for combinational networks are presented. The presence of feedback is modeled by introducing external *don't cares* that interpret the limited controllability and observability of the feedback wires.

- *Don't care* -extraction algorithms are coupled with generalized two-level optimization procedures that allow a more general restructuring of a logic circuit, by allowing the insertion and removal of delay elements and feedback paths, according to a predefined cost function.

A characterization of *don't care* conditions in synchronous networks is presented in Chapter 5. For acyclic networks, such *don't care* conditions are captured implicitly by a *recurrence equation*. A solution algorithm for recurrence equations is then presented.

Chapter 2

Combinational networks

2.1 Introduction

The present and the next chapters concern mainly Boolean methods for combinational logic optimization. In particular, we introduce in the present chapter *perturbation theory* as a tool for reasoning about local modifications of a Boolean network. We also introduce the main algorithms for extracting *don't care* information from the Boolean network. These algorithms use a local paradigm, that is, they attempt the extraction of the *don't care* information relative to a gate from that of the adjacent gates. The local paradigm presents several key advantages. First, it allows us to construct the observability *don't care* functions without an explicit representation of the circuit's functionality. Second, if the representation of *don't cares* grows too large, it allows us to perform approximations at run time. Third, it allows us to compare quantitatively previous approaches to the problem presented in the literature.

These algorithms are presented in Sections (2.3) and (2.4). Approximation techniques are then presented in Section (2.5).

2.2 Terminology

2.2.1 Boolean functions and their representations

Let $\mathcal{B}$ denote the Boolean set $\{0,1\}$. A k-dimensional Boolean vector $\mathbf{x}= [x_1, \cdots, x_k]^T$ is an element of the set $\mathcal{B}^k$ (boldfacing is hereafter used to denote vector quantities. In particular, the symbol **1** denotes a vector whose components are all 1).

A n_i-input, n_o-output Boolean function **F** is a mapping **F**: $\mathcal{B}^{n_i} \rightarrow \mathcal{B}^{n_o}$.

The *cofactors* (or *residues*) of a function **F** with respect to a variable x_i are the functions
$\mathbf{F}_{x_i} = \mathbf{F}(x_1, \ldots, x_i = 1, \ldots, x_n)$ and $\mathbf{F}_{x_i'} = \mathbf{F}(x_1, \ldots, x_i = 0, \ldots, x_n)$. The *universal quantification* or *consensus* of a function **F** with respect to a variable x_i is the function $\forall_{x_i}\mathbf{F} = \mathbf{F}_{x_i}\mathbf{F}_{x_i'}$. The *existential quantification* or *smoothing* of **F** with respect to x_i is defined as $\exists_{x_i}\mathbf{F} = \mathbf{F}_{x_i} + \mathbf{F}_{x_i'}$. The *Boolean difference* of **F** with respect to x_i is the function $\partial\mathbf{F}/\partial x_i = \mathbf{F}_{x_i} \oplus \mathbf{F}_{x_i'}$. A scalar function F_1 *contains* F_2 (denoted by $F_1 \geq F_2$) if $F_2 = 1$ implies $F_1 = 1$. The containment relation holds for two vector functions if it holds component-wise.

A function **F** is termed *positive unate* in x_i if $\mathbf{F}_{x_i} \geq \mathbf{F}_{x_i'}$, and *negative unate* if $\mathbf{F}_{x_i} \leq \mathbf{F}_{x_i'}$. Otherwise the function is termed *binate* in x_i.

Boolean expressions are a common means for representing Boolean functions. Formally, a Boolean expression is defined as follows:

Definition 2.1 *The symbols* $0, 1$ *are Boolean expressions, and denote the constant functions* $0, 1 : \mathcal{B}^{n_i} \rightarrow \mathcal{B}$*, respectively. Given a set of* n_i *variables* $x, y, \ldots$*, a* **literal** x *(*x'*) is an expression, and denotes a function* $x(x') : \mathcal{B}^{n_i} \rightarrow \mathcal{B}$*, taking the value (the complement of the value) of* x *. Finite sums and finite products of Boolean expressions are Boolean expressions. They denote the functions formed by the logic sums and products of their terms, respectively. Complements of Boolean expressions are Boolean expressions.*

Any given Boolean function can, however, be represented by means of several Boolean expressions. This makes it difficult to check whether two expressions describe the same function. For this reason, it is in practice convenient to represent and manipulate in a computer Boolean functions by means of their associated Binary Decision Diagrams

(BDDs) [29, 30]. BDDs are canonical representation for Boolean functions. We refer to [29] for a detailed description of the use of BDDs for manipulating Boolean functions.

2.2.2 Combinational circuits and logic networks.

The mathematical model of a combinational multiple-level circuit is the **logic network**.

Definition 2.2 *A combinational* **logic network** *is an annotated graph* $N = (V, E)$. *Vertices correspond to primary inputs, single-output logic gates, or primary outputs, while edges correspond to interconnections. For each vertex* $y \in V$

$$\begin{aligned} FI_y &= \{z \in V | (z, y) \in E\} \\ FO_y &= \{z \in V | (y, z) \in E\} \end{aligned} \tag{2.1}$$

denote the vertices corresponding to the inputs of the gate in y *and the vertices where the output of the gate in* y *is used as input, respectively. These sets are termed the* **fanin** *and* **fanout** *of* y. *The* **transitive fanin** *and* **fanout** TFI_y *and* TFO_y *are the sets of vertices reaching and reachable from* y, *respectively.*

Each vertex y *is associated a Boolean variable, also labeled* y, *and a Boolean expression of the variables of* FI_y. *Hereafter, we denote this expression with* e^y. *The variable* y *and local expression* e^y *represent the gate output and the local behavior realized by each gate, in terms of the adjacent variables. Variables associated with logic gates are also termed* **local variables**.

Example 7.

Fig. (2.1) shows an example of a combinational logic network. Variables a, b, c, d, e represent the primary inputs, while u, v, x, y, z and o_1, o_2 denote internal variables and primary outputs, respectively. All variables, except for primary inputs, are given expressions in terms of other network variables. □

The behavior of each vertex can also be described by referring to the function of the primary inputs it ultimately realizes. In more detail, let y denote a vertex. A

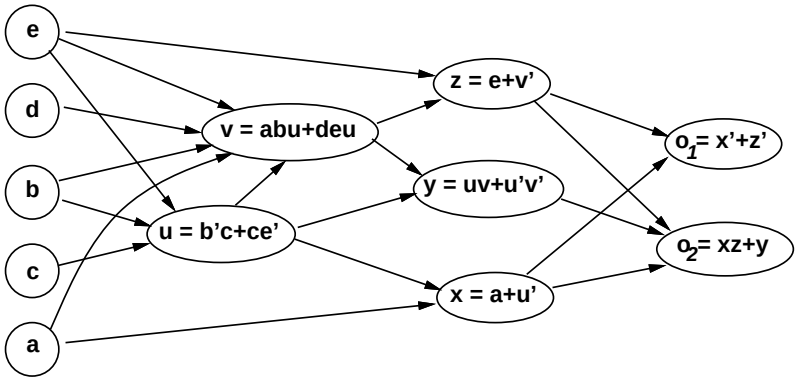


Figure 2.1: A combinational logic network

local function f^y in terms of the primary inputs can be obtained simply by iteratively substituting in e^y each internal variable with its expression, until only primary inputs appear. In particular, the behavior of a logic network is then described by a function $\mathbf{F}$:$\mathcal{B}^{n_i} \rightarrow \mathcal{B}^{n_o}$, where n_i and n_o are the number of primary inputs and outputs, respectively.

Example 8.

In the network of Fig. (2.1), the function realized at vertex u is $f^u = ab+b'c$. By substituting this expression in e^v, the function $f^v = bc+cde+abde$. The functions realized at each vertex are listed below:

$$\begin{aligned} f^u &= b'c + ce' \\ f^v &= abce' + b'cde \\ f^x &= a + be + c' \\ f^y &= ab + be + c' + de \\ f^z &= a' + b' + c' + e \\ f^{o_1} &= a'b'c + bce' \\ f^{o_2} &= a + be + c' + de \end{aligned} \tag{2.2}$$

The behavior of the network is captured by

$$\mathbf{F} = \begin{bmatrix} f^{o_1} \\ f^{o_2} \end{bmatrix} = \begin{bmatrix} a'b'c + bce' \\ a + be + c' + de \end{bmatrix} . \tag{2.3}$$

□

2.2.3 Specifications for combinational networks.

A common style for specifying the desired behavior of a combinational network is by means of two functions **F**(**x**) and **DC**(**x**) respectively, the latter in particular representing the input combinations that are known not to occur or such that the value of some of the network outputs is regarded as irrelevant [16]. Corresponding to these input combinations, some network outputs are therefore left unspecified, which represents a degree of freedom that can be spent during optimization. For this reason, the function **DC** is hereafter termed *don't care* function. This style of specification will be referred to as *don't care -based*.

A formally equivalent specification is in terms of the functions $\mathbf{F}_{min}$ = **F** · **DC**′ and $\mathbf{F}_{max}$ = **F**+**DC**. Specifications are met by a function **G** if

$$\mathbf{F}_{min} \leq \mathbf{G} \leq \mathbf{F}_{max}. \tag{2.4}$$

A more powerful, but also more complex, specification style is by means of a *Boolean relation* [31, 8, 32]. A Boolean relation for the behavior of a n_i-input, n_y-output network, with inputs and outputs labeled **x** and **y**, is a Boolean equation of type

$$\mathbf{F}_{min}(\mathbf{x}) \leq \mathbf{F}(\mathbf{x}, \mathbf{y}) \leq \mathbf{F}_{max}(\mathbf{x}); \tag{2.5}$$

where **F** is a Boolean function **F**: $\mathcal{B}^{n_i+n_y} \rightarrow \mathcal{B}^{n_o}$. A function **G**: $\mathcal{B}^{n_i} \rightarrow \mathcal{B}^{n_y}$ satisfies the specifications if and only if for every input combination **x**$\in \mathcal{B}^{n_i}$,

$$\mathbf{F}_{min}(\mathbf{x}) \leq \mathbf{F}(\mathbf{x}, \mathbf{G}(\mathbf{x})) \leq \mathbf{F}_{max}(\mathbf{x}); \tag{2.6}$$

This second specification style is hereafter referred to as *relational specification*. For simplicity, in the remainder of this chapter, specifications are hereafter assumed to be in *don't care* form.

2.2.4 Optimization of combinational multiple-level circuits

The optimization of a network N, realizing a function **F**, ultimately consists of its replacement by a different network, with better figures of merit in terms of area, delay, or testability. In principle, the new network is allowed to realize a function **G** different from **F**, as long as **G** satisfies the specifications:

$$\mathbf{F} \cdot \mathbf{DC}' \leq \mathbf{G} \leq \mathbf{F} + \mathbf{DC} \ . \tag{2.7}$$

We now use the Boolean identity:

$$a \leq b \Leftrightarrow a' + b = 1 \ . \tag{2.8}$$

By applying this identity to both inequalities of (2.7), we obtain

$$\begin{aligned} \mathbf{F} + \mathbf{G}' + \mathbf{DC} &= \mathbf{1} \\ \mathbf{F}' + \mathbf{G} + \mathbf{DC} &= \mathbf{1} \ . \end{aligned} \tag{2.9}$$

The two equalities (2.9) hold simultaneously if and only their product takes value **1**:

$$\begin{aligned} (\mathbf{F} + \mathbf{G}' + \mathbf{DC})(\mathbf{F}' + \mathbf{G} + \mathbf{DC}) = \\ \mathbf{F}\overline{\oplus}\mathbf{G} + \mathbf{DC} = \mathbf{1} \ . \end{aligned} \tag{2.10}$$

By applying the Boolean identity (2.8) to Eq. (2.10) we eventually obtain

$$\mathbf{F} \oplus \mathbf{G} \leq \mathbf{DC} \ . \tag{2.11}$$

The function $\mathbf{F}\oplus\mathbf{G}$ represents the difference, or "error", in behavior between the original and optimized network. From Eq. (2.11), **DC** then takes the "physical" meaning of a tolerable functional error during optimization.

In practice, due to the complexity of exact optimization, current optimization strategies are based on the local, iterative optimization of small subsets of vertices of N. Neither the network topology nor the behavior of the individual vertices need to be exactly preserved, as long as the outputs of the optimized network satisfy Eq. (2.11): such degrees of freedom thus represent "errors" on the local functions f^y that can be tolerated by the global functional specifications.

A first necessary step of local optimization is therefore the *characterization* of such local errors. Following a *don't care* -based style, the characterization is by upper bounds on the errors tolerated at each vertex, and the means developed in the next section is *perturbation theory*.

2.3 Perturbation analysis of combinational networks.

In this section perturbation theory is introduced as a main tool for the analysis of the degrees of freedom in a logic optimization environment. The modification of each logic function f^y in a network is modeled by introducing a perturbation signal δ. The analysis focuses first on the optimization of a single vertex, described by means of a single perturbation, and is then extended to multiple-vertex optimization, modeled by the introduction of multiple perturbations. The following general definitions are in order.

Definition 2.3 *Given a subset* $\mathbf{y} = \{y_1, \ldots, y_m\} \leq V$ *of variables of a network* N*, we call* **perturbed network** $N^{\mathbf{y}}$ the network obtained form N by replacing each local function e^{y_i} with $e^{y_i,\mathbf{y}} = e^{y_i} \oplus \delta_i$, $y_i \in \mathbf{y}$. The added inputs δ_i are termed **perturbations**.

The functionality of a perturbed network $N^{\mathbf{y}}$ is described by a function $\mathbf{F}^{\mathbf{y}}$, which depends also on $\delta = [\delta_1, \ldots, \delta_m]$: $\mathbf{F}^{\mathbf{y}} = \mathbf{F}^{\mathbf{y}}(\mathbf{x}, \delta)$. In particular,

$$\mathbf{F}^{\mathbf{y}}_{\delta'_1,\ldots,\delta'_m} = \mathbf{F} \tag{2.12}$$

and every internal vertex y realizes a functionality described by $f^{y,\mathbf{y}}(\mathbf{x}, \delta)$.

The functionality of any network N' obtained by replacing each f^{y_i} with an arbitrary function g^{y_i} is described by $\mathbf{F}^{\mathbf{y}}(\mathbf{x}, f^{y_1} \oplus g^{y_1}, \ldots, f^{y_m} \oplus g^{y_m})$.

Example 9.

Fig. (2.2) shows the network of Fig. (2.1), perturbed only corresponding to v. Internal functions are described by

$$\begin{aligned}
f^{u,v} &= b'c + ce' \\
f^{v,v} &= (abce' + b'cde) \oplus \delta \\
f^{x,v} &= a + be + c' \\
f^{y,v} &= (ab + be + c' + de) \oplus \delta \\
f^{z,v} &= e + (abc)\overline{\oplus}\delta \\
f^{o_1,v} &= a'b'c + a'ce' + [(abc) \oplus \delta]e' \\
f^{o_2,v} &= be + \delta'(a + c' + de) + \delta(ae + cd' + c \oplus e)
\end{aligned}$$

For $\delta = 0$, these functions reduce to those of the unperturbed network. Notice also that only the functions of the vertices in TFO_y are affected by δ, the functions $f^{u,v}, f^{x,v}$ being identical to those of the original network. □

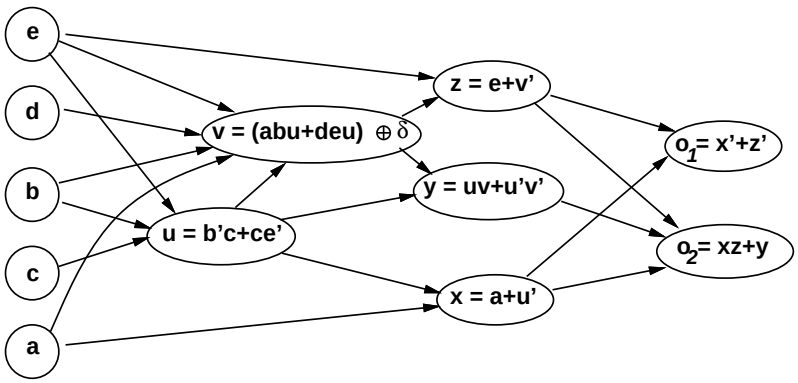


Figure 2.2: Network perturbed in correspondence of variable v

In this work, the vertices of a perturbed network maintain the same labels as in the original network. Boolean expressions in terms of network variables are therefore ambiguous: for instance, the expression $uv + u'v'$ denotes two different functions in the original network N of Fig. (2.1) and in the network N^v of Fig. (2.2). For notational simplicity, however, this ambiguity is dealt with explicitly only when necessary.

The functional errors of the perturbed network with respect to the original one are described by the **error function**

$$\mathbf{E}(\mathbf{x}, \delta) \stackrel{\text{def}}{=} \mathbf{F^y}(\mathbf{x}, \delta) \oplus \mathbf{F^y}(\mathbf{x}, \mathbf{0}) \ . \tag{2.13}$$

From Definition (2.3), functions g^{y_i} can simultaneously replace f^{y_i} if and only if for every $\mathbf{x} \in \mathcal{B}^{n_i}$, $\delta_i = f^{y_i}(\mathbf{x}) \oplus g^{y_i}(\mathbf{x})$,

$$\mathbf{E} \leq \mathbf{DC} \ . \tag{2.14}$$

Eq. (2.14) represents implicitly all the tolerances on the errors δ_i. In this form, however, such degrees of freedom are very difficult to use. The purpose of this section is to present algorithms that efficiently transform Eq. (2.14) into a manageable form, namely a set of individual tolerances on the errors δ_i. To this end, the observability *don't care* functions defined below have a key role:

Definition 2.4 *The* **observability** *don't care of a variable* y_i *in a perturbed network* $N^{\mathbf{y}}$ is the function

$$\mathbf{ODC}^{y_i}(\mathbf{x}, \delta) \stackrel{\text{def}}{=} \mathbf{F}^{\mathbf{y}}_{\delta_i} \overline{\oplus} \mathbf{F}^{\mathbf{y}}_{\delta_i'} \ . \tag{2.15}$$

Corresponding to each combination of inputs and perturbations $(\mathbf{x},\ \delta)$, the quantity $\mathbf{ODC}^{y_i}(\mathbf{x},\ \delta)$ takes value 1 corresponding to those outputs of N^y not affected by a change in δ_i. In particular, the product of all components of $\mathbf{ODC}^{y_i}$ represents the input combinations for which δ_i cannot affect *any* output.

Strictly speaking, $\mathbf{ODC}^{y_i}$ depends on the perturbed network under consideration: $\mathbf{ODC}^{y_i} = \mathbf{ODC}^{y_i,\mathbf{y}}$. The superscript $\mathbf{y}$ is removed for notational simplicity, leaving the task of specifying $N^{\mathbf{y}}$ to the context. Notice also that the complement of $\mathbf{ODC}^{y_i}$ is just the *Boolean difference* $\partial \mathbf{F}^{\mathbf{y}}/\partial \delta_i$ with respect to δ_i, and it represents the combinations of inputs $\mathbf{x}$ and perturbations δ such that δ_i affects the primary outputs. For this reason, it is hereafter denoted by $\mathbf{OC}^{y_i}$.

Example 10.

The functionality $\mathbf{F}^v$ of the network in Fig. (2.2) is given in Example (9). From Eq. (2.13),

$$\mathbf{E} = \begin{bmatrix} \{a'b'c + a'ce' + [(abc) \oplus \delta]e'\} \oplus (a'b'c + bce') \\ \{be + \delta'(a + c' + de) + \delta(ae + cd' + c \oplus e)\} \oplus (a + be + c' + de) \end{bmatrix} =$$

$$\begin{bmatrix} \delta(a + c')e' \\ \delta(a'b'c + a'e' + c'e') \end{bmatrix} .$$

By applying Eq. (2.15), the observability *don't care* of v is

$$\mathbf{ODC}^v = \begin{bmatrix} (a'b'c + a'e' + b'e' + c'e')\overline{\oplus}(a'b'c + bce') \\ \{be + ab + ae + cd' + c \oplus e)\overline{\oplus}(a + be + c' + de) \end{bmatrix} =$$

$$\begin{bmatrix} a'c + e \\ ac + be + c'e \end{bmatrix} .$$

In this particular case only one perturbation signal was considered, and therefore $\mathbf{ODC}^v$ depends only on primary inputs.

Expressions of $\mathbf{ODC}^v$ need not be given necessarily in terms of primary inputs, but network variables can also be used. For instance, another expression of $\mathbf{ODC}^v$ is

$$\mathbf{ODC}^v = \begin{bmatrix} x' + e \\ x(e+u) \end{bmatrix} .$$

Notice that this second expression has a greater flexibility: it can in fact express the observability *don't care* of v not only in N^v, but also in presence of perturbations at other internal vertices, in this case u and/or x. This is possible thanks to the ambiguity left by not relabeling the vertices in a perturbed network. □

We now examine the role played by observability *don't cares* in logic optimization. The simplest approach to the optimization of N consists of optimizing individual vertices, one at a time, thus introducing only one perturbation signal δ. This case is examined first. The case of joint multi-vertex optimization is analyzed in Sect. (2.4).

2.3.1 Single-vertex optimization and observability *don't cares* .

From the standpoint of perturbation analysis, the case of a single perturbation is especially favorable, as constraint (2.14) can be transformed into an array of upper bounds on δ only. The algebra of the derivation is as follows: a Shannon decomposition of Eq. (2.14) results in

$$\delta' \mathbf{E}_{\delta'} + \delta \mathbf{E}_{\delta} \leq \mathbf{DC} . \tag{2.16}$$

On the other hand, from Eq. (2.13), $\mathbf{E}_{\delta'} = \mathbf{0}$ and moreover, by comparing Eq. (2.13) with Definition(2.4), $\mathbf{E}_{\delta} = (\mathbf{ODC}^y)'$. Consequently, Eq. (2.16) can be rewritten as

$$\delta(\mathbf{ODC}^y)' \leq \mathbf{DC} \tag{2.17}$$

which holds if and only if

$$\delta \mathbf{1} \leq \mathbf{DC} + \mathbf{ODC}^y. \tag{2.18}$$

By denoting with DC^y the product of all components of **DC** + $\mathbf{ODC}^y$, Eq. (2.18) eventually reduces to

$$\delta \leq DC^y . \tag{2.19}$$

Result (2.18)-(2.19) was first obtained independently by Bartlett *et al.* [16] and by Muroga *et al.* [33], and it shows that:

- the global tolerance on the network outputs, represented by **DC**, can be *transformed* into a local tolerance on the local error δ;
- this tolerance consists of the sum of a global component (**DC**), plus a local one, represented by the observability *don't care* $\mathbf{ODC}^y$ of y in the network N^y.

Example 11.

The observability *don't care* of v for the network of Fig. (2.2) is given in Example (9). The constraints on δ reduce to

$$\delta \begin{bmatrix} 1 \\ 1 \end{bmatrix} \leq \begin{bmatrix} a'c + e \\ ac + be + c'e \end{bmatrix} .$$

By forming the product of the two bounds,

$$\delta \leq ae + be + c'e = DC^v$$

represents all the functional constraints on δ. □

Although it is in principle possible to compute $\mathbf{ODC}^y$ for any variable y by applying Definition (2.4) in a straightforward manner, the difficulty of representing $\mathbf{F}^y$ explicitly renders this operation very time- and memory-consuming and frequently impossible in practice. In order to make a *don't care* -based logic optimization system practical, it is thus necessary to develop algorithms that extract representations of the observability *don't cares* in a logic network directly from the network topology, thus avoiding explicit representations of $\mathbf{F}^y$. Moreover, as observability *don't cares* may have themselves large representations, effective *don't care* -extraction algorithms must be able to handle *approximations* of *don't cares* . This suite of problems consitutes the object of the rest of this section. In particular, topological methods for extracting the observability *don't cares* in a logic network are presented next, while approximation techniques are dealt with in Sect. (2.5).

Computation of observability *don't cares* by local rules.

As the observability *don't care* of a vertex y describes how an error on its functionality affects the primary outputs, it should be linked by simple local rules to the observability *don't cares* of the vertices in FO_y. Consequently, one should be able to compute all observability *don't cares* by a single sweep of the network from its primary outputs using only local rules. This perception has led to an intense research [34, 35, 33, 36] of rules that can be coupled with one such network traversal algorithm to yield the observability *don't cares* of all vertices.

One such simple rule indeed exists in the particular case of a vertex with a single fanout edge [37]. For a vertex labeled y, with a unique fanout edge (y, z) to a variable z,

$$\mathbf{ODC}^y = \mathbf{ODC}^z + \left(\frac{\partial f^z}{\partial \delta}\right)' \mathbf{1} \tag{2.20}$$

links the observability *don't care* of y to that of z: $\mathbf{ODC}^y$ can be obtained by adding $(\partial f^z/\partial \delta)'$ to all the components of $\mathbf{ODC}^z$. The rationale behind Eq. (2.20) is that an error on f^y will not affect the primary outputs if it does not affect f^z (contribution represented by $(\partial f^z/\partial \delta)'$) or if the error introduced in f^z is then tolerated by the network (contribution represented by $\mathbf{ODC}^z$). A simple expression of $(\partial f^z/\partial \delta)'$ is any local expression of $(\partial e^z/\partial y)'$ [37]. An expression of $\mathbf{ODC}^y$ can then be derived from that of $\mathbf{ODC}^z$ by

$$\mathbf{ODC}^y = \mathbf{ODC}^z + \left(\frac{\partial e^z}{\partial y}\right)' \mathbf{1} \ . \tag{2.21}$$

Eq. (2.21) shows that, ultimately, an expression of $\mathbf{ODC}^y$ can be obtained from that of $\mathbf{ODC}^z$ and that of e^z, thereby avoiding the explicit construction of the function f^z.

Example 12.

In the network of Fig.(2.1), an error on y can affect the primary outputs only through o_2. Consequently, $\mathbf{ODC}^y$ can be derived from

$$\mathbf{ODC}^{o_2} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

and from $\partial o_2/\partial y = (xz+1) \oplus (xz+0) = (xz)'$ by applying Eq. (2.20):

$$\mathbf{ODC}^z = \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \left(\frac{\partial o_2}{\partial z}\right)' \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ xz \end{bmatrix} .$$

□

Example (11) showed that an expression of an observability *don't care* can be correct in more than a perturbed network, thanks to the " ambiguity" left by the choice in vertex labeling. An important attribute of Eq. (2.21) is that the expression of $\mathbf{ODC}^y$ obtained in this way is correct in all networks for which $\mathbf{ODC}^z$ is correct. Local expressions of $\partial e^z/\partial y$ depend " by construction" only on the local expression e^z and not by any property of the rest of the network.

Complementing rule (2.21) gives a rule for expressions of the observability *care* function:

$$\mathbf{OC}^y = \left(\frac{\partial e^z}{\partial y}\right)\mathbf{OC}^z . \tag{2.22}$$

Rule (2.21) is of course insufficient if y has multiple fanout edges. In this case, a naive approach may consist of first finding the observability *don't cares* along each fanout edge. Such *don't cares* represent the tolerance of an error along each edge: their intersection could then represent a tolerance on f^y. The following example shows that, unfortunately, this rule is incorrect.

Example 14.

Consider computing the observability *don't care* of y in the simple network of Fig. (2.3). The observability of x and z can be computed by Eq. (2.20): $ODC^x = z$ and $ODC^z = x$. The observability *don't care* of y, computed according to the previous considerations, would then be

$$ODC^y = \left(ODC^x + \left(\tfrac{\partial x}{\partial y}\right)'\right)\left(ODC^z + \left(\tfrac{\partial z}{\partial y}\right)'\right) =$$
$$(z+a)(x+b) = ab + a'b'(c'+d').$$

In particular, $ODC^y = 1$ for $a = 0, b = 0, c = 0, d = 0$ indicates that a change of y from 0 to 1 would not affect the primary output, trivially incorrect. □

In Example (14), the product rule did not take into account that an error on y, propagating along a path crossing z, contributes positively to the observability of the same error propagating along the path crossing x. More generally, the product rule fails to take correctly into account the interplay of the observability along the various paths.

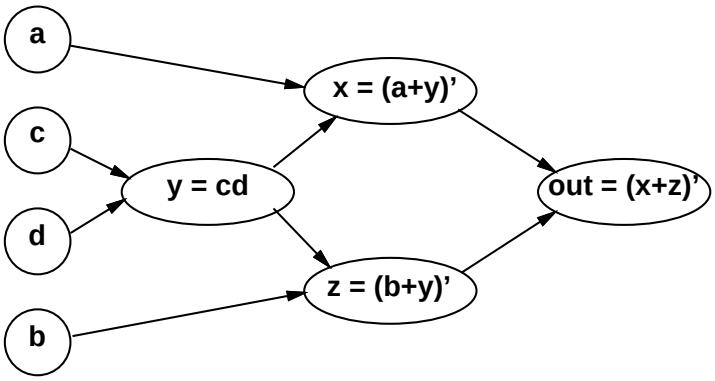


Figure 2.3: Network for counterexample (14).

The general rule for dealing with multiple-fanout vertices is derived here. This rule is best explained by slightly augmenting the topology of the network as follows: first, a vertex labeled y_i , $i = 1, \ldots, |FO_y|$ is added along each edge on the fanout of y. The variables y_i are termed the *fanout variables* of y. The local function e^{y_i} of each added vertex is the identity function $e^{y_i} = y$, so that the network functionality is trivially preserved.

Second, instead of considering directly a network N^y perturbed at y, each new vertex y_i is added a perturbation signal δ_i, so that now $y_i = y \oplus \delta_i$. The network functionality is then described by a function $\mathbf{F^y}(\mathbf{x}, \delta_1, \ldots, \delta_{|FO_y|})$, and the behavior of N^y can be recovered by forcing all perturbations to be identical, *i.e.* $\delta_1 = \delta_2 = \ldots, \delta_{|FO_y|} = \delta$:

$$\mathbf{F}^y(\mathbf{x}, \delta) = \mathbf{F^y}(\mathbf{x}, \delta, \delta, \ldots, \delta). \tag{2.23}$$

Figure (2.4) shows the network of Fig. (2.2), transformed for the calculation of $\mathbf{ODC}^v$.

Consider first the case of only two fanout edges, as in Fig. (2.4), and let y_1, y_2 denote the added variables. From Eq. (2.23) and Definition(2.4), the observability *don't care* of y is

$$\mathbf{ODC}^y(\mathbf{x}) = \mathbf{F}^{y_1 y_2}(\mathbf{x}, 0, 0) \overline{\oplus} \mathbf{F}^{y_1 y_2}(\mathbf{x}, 1, 1) \ . \tag{2.24}$$

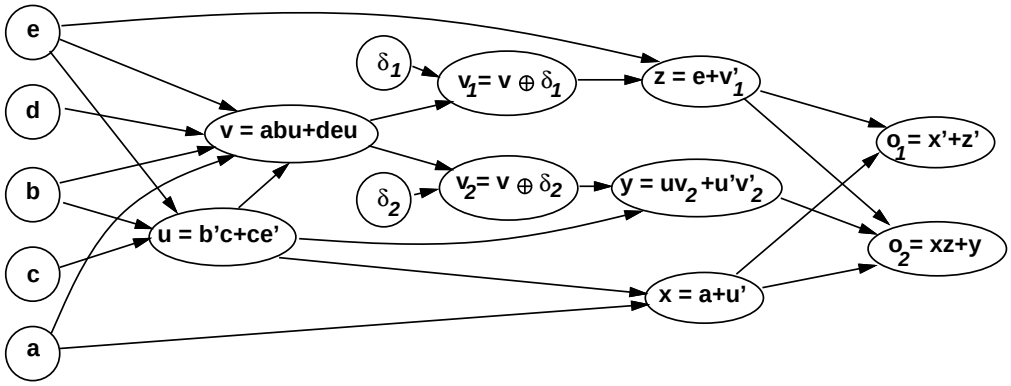


Figure 2.4: Network of Example (14), but perturbed corresponding to the fanout edges of v. Notice the introduction of the two auxiliary variables v_1 and v_2.

By manipulating Eq.(2.24), $\mathbf{ODC}^y$ can be rewritten as

$$\mathbf{ODC}^y(\mathbf{x}) = \Big(\mathbf{F}^{y_1 y_2}(\mathbf{x},0,0)\overline{\oplus}\mathbf{F}^{y_1 y_2}(\mathbf{x},1,0)\Big)\overline{\oplus}\Big(\mathbf{F}^{y_1 y_2}(\mathbf{x},1,0)\overline{\oplus}\mathbf{F}^{y_1 y_2}(\mathbf{x},1,1)\Big) \qquad (2.25)$$

where the term $\mathbf{F}^{y_1 y_2}(\mathbf{x},1,0)$ has been "added and subtracted" in Eq.(2.24). From Definition (2.4), the first term in parentheses is $\mathbf{ODC}^{y_1}_{\delta_2'}$, while the second parentheses describe $\mathbf{ODC}^{y_2}_{\delta_1}$:

$$\mathbf{ODC}^y = \mathbf{ODC}^{y_1}_{\delta_2'}\overline{\oplus}\mathbf{ODC}^{y_2}_{\delta_1}. \qquad (2.26)$$

Eq. (2.26) links the observability *don't care* of y to those of its fanout variables. These *don't cares* , however, are not evaluated in N^{y_1} and N^{y_2}, respectively, but in $N^{y_1 y_2}$. In order to apply Eq. (2.26) it is then necessary to have available expressions of $\mathbf{ODC}^{y_1}, \mathbf{ODC}^{y_2}$ that are correct in presence of multiple perturbations, namely *at least* in presence of δ_2 and δ_1, respectively.

Example 15.

Consider using Eq. (2.26) for computing $\mathbf{ODC}^v$ in the network of Fig. (2.4). Expressions of the observability *don't care* of v_1 and v_2 are given by

$$\mathbf{ODC}^{v_1} = \begin{bmatrix} x' + e \\ x' + y + e \end{bmatrix} \quad \mathbf{ODC}^{v_2} = \begin{bmatrix} 1 \\ xz \end{bmatrix} .$$

It could be verified that these expressions are valid in every perturbed network, therefore in particular in $N^{v_1 v_2}$. In order to cofactor $\mathbf{ODC}^{v_1}$ and $\mathbf{ODC}^{v_2}$ with respect to δ_2' and δ_1, respectively, it is necessary make explicit their dependencies from those perturbations:

$$\mathbf{ODC}^{v_1} = \begin{bmatrix} x' + e \\ x' + e + u\overline{\oplus}v \oplus \delta_2 \end{bmatrix} ; \mathbf{ODC}^{v_2} = \begin{bmatrix} 1 \\ x(e + v\overline{\oplus}\delta_1) \end{bmatrix} .$$

Eq. (2.26) then takes the form: [1]

$$\mathbf{ODC}^{v} = \mathbf{ODC}^{v_1}_{\delta_2'}\overline{\oplus}\mathbf{ODC}^{v_2}_{\delta_1} = \begin{bmatrix} x' + e \\ x' + e + u\overline{\oplus}v \end{bmatrix} \overline{\oplus} \begin{bmatrix} 1 \\ x(e + v) \end{bmatrix} = \begin{bmatrix} x' + e \\ x(e + u) \end{bmatrix} .$$

The expression of $\mathbf{ODC}^{v}$ in terms of the primary inputs is

$$\begin{bmatrix} a'c + e \\ ac + be + c'e \end{bmatrix} .$$

Assuming **DC = 0**, $DC^{v} = (a'c + e)(ac + be + c'e) = ae + be + c'e$, the same results as in the direct method of Example (11). The optimization of v with this *don't care* produces the network of Fig. (2.5). □

Some of the substitutions carried out in Example (15) can actually be avoided. Since $\mathbf{ODC}^{y_1}_{\delta_2'}$ assumes $\delta_2 = 0$, it coincides with the observability *don't care* of y_1 in absence of a perturbation on y_2. It is thus possible to drop the subscript δ_2' and use *directly* the expression of $\mathbf{ODC}^{y_1}$:

$$\mathbf{ODC}^{y} = \mathbf{ODC}^{y_1}\overline{\oplus}\mathbf{ODC}^{y_2}_{\delta_1} . \tag{2.27}$$

Example 16.

[1] These second expressions of $\mathbf{ODC}^{v_1}$ and $\mathbf{ODC}^{v_2}$ do not have the same validity as the previous ones. Since y and z have been replaced by their unperturbed expressions, the validity is now limited to those networks with no perturbation on y or z. This is also the validity of the expression of $\mathbf{ODC}^{v}$ so obtained. More general expressions could be obtained by taking into account perturbations of y and z.

Consider using Eq. (2.27) instead of Eq. (2.26) for computing $\mathbf{ODC}^{v}$. Only $\mathbf{ODC}^{v_2}$ needs to be expressed in terms of δ_1; consequently,

$$\mathbf{ODC}^{v} = \begin{bmatrix} x' + e \\ x' + e + y \end{bmatrix} \overline{\oplus} \begin{bmatrix} 1 \\ x(e + v) \end{bmatrix} = \begin{bmatrix} x' + e \\ x(e + y \overline{\oplus} v) \end{bmatrix} .$$

□

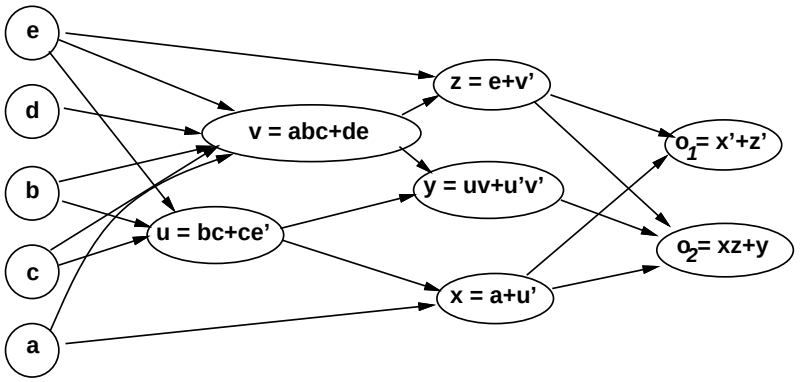


Figure 2.5: An optimized version of the network of Fig. (2.1)

As pointed out by Example (16), Eq. (2.27) does not completely eliminate substitution operations: it is therefore not entirely local. Such operations, however, are carried out on **ODC** functions rather than network functions, and only those variables belonging to TFO_{y_2} *and* actually appearing in $\mathbf{ODC}^{y_2}$ need be substituted by their expressions. Notice, for instance, that in Example (16) one substitution of z in $\mathbf{ODC}^{v_2}$ was sufficient. A straightforward application of Definition (2.4) would instead require an expression of $\mathbf{F}^{y}$: in this case, *both* z and y should be substituted in o_1 *and* o_2.

A second general expression of $\mathbf{ODC}^{y}$ can be obtained by adding twice $\mathbf{F}^{y_1 y_2}(\mathbf{x}, 0, 1)$ in Eq. (2.24) [2]

[2] Another method consisted of the so-called *chain rule* [37], which links the observability *don't care* of y to those of y_1, y_2 by the equation [37]

$$\mathbf{ODC}^{y} = \mathbf{ODC}^{y_1} \overline{\oplus} \mathbf{ODC}^{y_2} \overline{\oplus} \left(\frac{\partial^2 \mathbf{F}^{y_1 y_2}}{\partial y_1 \partial y_2} \right)'$$

Its complexity has reduced its applicability.

$$\mathbf{ODC}^{y} = \mathbf{ODC}^{y_1}_{\delta_2} \overline{\oplus} \mathbf{ODC}^{y_2} \ . \tag{2.28}$$

It follows in particular that the right hand sides of Eq.(2.27) and Eq.(2.28) must be identical:

$$\mathbf{ODC}^{y_1}_{\delta_2} \overline{\oplus} \mathbf{ODC}^{y_2} = \mathbf{ODC}^{y_1} \overline{\oplus} \mathbf{ODC}^{y_2}_{\delta_1} \ . \tag{2.29}$$

This identity will be used extensively in Sect. (2.5), when considering approximations to *don't cares* .

Again, complementation of Eq. (2.27) and (2.28) yields the rules for observability *care*:

$$\mathbf{OC}^{y} = \mathbf{OC}^{y_1} \oplus \mathbf{OC}^{y_2}_{\delta_1} = \mathbf{OC}^{y_1}_{\delta_2} \oplus \mathbf{OC}^{y_2}. \tag{2.30}$$

The expansion of Eq. (2.26) into a sum-of-products

$$\mathbf{ODC}^{y} = \mathbf{ODC}^{y_1}_{\delta_2'} \mathbf{ODC}^{y_2}_{\delta_1} + \mathbf{OC}^{y_1}_{\delta_2'} \mathbf{OC}^{y_2}_{\delta_1} \tag{2.31}$$

evidences its relationship with concepts from the testing literature. Eq. (2.31) shows that there are two contributions to $\mathbf{ODC}^{y}$. The first, $\mathbf{ODC}^{y_1}_{\delta_2'} \mathbf{ODC}^{y_2}_{\delta_1}$, indicates that the presence of a second fanout variable y_2 *can restrict* the observability *don't care* of y with respect to the single-fanout case: *i.e.* helps the observability of errors along y_1. This fact is known as "self-evidencing" of errors in testing literature . The second contribution, $\mathbf{OC}^{y_1}_{\delta_2} \mathbf{OC}^{y_2}_{\delta_1'}$, indicates that an error on y is not observable if an error on y_1 alone *would be* observed, but it is *compensated* by the error along y_2. This is known as "self-masking" .

The extension of Eq.(2.27) to the general case of $|FO_y| > 2$ fanout variables is provided by the following theorem.

Theorem 2.1 *Let y and $y_1, \ldots, y_{|FO_y|}$ denote the variables associated with the fanout of y; then:*

$$\mathbf{ODC}^{y} = \overline{\bigoplus}_{i=1}^{|FO_y|} \mathbf{ODC}^{y_i}_{\delta_{i+1},\ldots,\delta_{|FO_y|}}; \tag{2.32}$$

$$\mathbf{OC}^{y} = \bigoplus_{i=1}^{|FO_y|} \mathbf{OC}^{y_i}_{\delta_{i+1},\ldots,\delta_{|FO_y|}}; \tag{2.33}$$

where each $\mathbf{ODC}^{y_i}_{\delta_{i+1},\ldots,\delta_{|FO_y|}}$ ($\mathbf{OC}^{y_i}_{\delta_{i+1},\ldots,\delta_{|FO_y|}}$) is the observability don't care (care) of y_i, assuming that only $y_j, j > i$ are perturbed.

Proof.

The following identity can be verified by taking into account Eq. (2.15):

$$\begin{aligned}\mathbf{ODC}^{y} = \mathbf{F}^{\mathbf{y}}_{\delta_1,\ldots,\delta_{|FO_y|}} \overline{\oplus} \mathbf{F}^{\mathbf{y}}_{\delta'_1,\ldots,\delta'_{|FO_y|}} = \\ \left(\mathbf{F}^{\mathbf{y}}_{\delta_1,\delta_2,\ldots,\delta_{|FO_y|}} \overline{\oplus} \mathbf{F}^{\mathbf{y}}_{\delta'_1,\delta_2,\ldots,\delta_{|FO_y|}}\right) \overline{\oplus} \left(\mathbf{F}^{\mathbf{y}}_{\delta'_1,\delta_2,\ldots,\delta_{|FO_y|}} \overline{\oplus} \mathbf{F}^{\mathbf{y}}_{\delta'_1,\delta'_2,\ldots,\delta_{|FO_y|}}\right) \overline{\oplus} \\ \ldots \overline{\oplus} \left(\mathbf{F}^{\mathbf{y}}_{\delta'_1,\delta'_2,\ldots,\delta_{|FO_y|}} \overline{\oplus} \mathbf{F}^{\mathbf{y}}_{\delta'_1,\delta'_2,\ldots,\delta'_{|FO_y|}}\right) .\end{aligned} \tag{2.34}$$

Eq.(2.34) can be rewritten as:

$$\mathbf{ODC}^{y} = \overline{\bigoplus}_{i=1}^{|FO_y|} \left(\mathbf{F}^{\mathbf{y}}_{\delta'_1,\ldots,\delta'_{i-1},\delta_i,\ldots,\delta_{|FO_y|}} \overline{\oplus} \mathbf{F}^{\mathbf{y}}_{\delta'_1,\ldots,\delta'_i,\delta_{i+1},\ldots,\delta_{|FO_y|}}\right) . \tag{2.35}$$

Eq. (2.32) then follows by observing that each term of the sum in Eq. (2.35) is precisely $\mathbf{ODC}^{y_i}_{\delta_{i+1},\ldots,\delta_{|FO_y|}}$. Eq. (2.33) then follows trivially by complementing Eq. (2.32). □

Similarly to the case of two fanout variables, a permutation of the order in which the variables y_i are considered results in a different expression of $\mathbf{ODC}^{y}$, the same type as (2.32). All $|FO_y|!$ expressions, however, must describe the same function: there are therefore $|FO_y|!(|FO_y|! - 1)/2$ identities of the type of Eq. (2.29).

Algorithms for observability *don't cares* **.**

It is here shown that rules (2.20) and (2.32) permit the derivation of expressions of all observability *don't cares* of a network by a single traversal of N in topological order, from the primary outputs. Algorithm `OBSERVABILITY` below implements this idea. First, the network is sorted topologically in the array `variable[]` (for example, by a depth-first routine [24]), and then augmented by the addition of the fanout variables of each multiple-fanout vertex. The fanout variables of a vertex y_i are inserted right after y in `variable[]`, so that the new array is still topologically sorted. When `variable[i]` is processed, the observability of all vertices in $TFO_{variable[i]}$, kept in `odc[]`, is thus already known. The observability *don't care* of all internal vertices and primary inputs is set to **1**. The observability *don't care* of the i^{th} output vertex is then initialized to a vector containing a zero in the i^{th} component and 1 otherwise (the i^{th} output vertex is of

course perfectly observable at the i^{th} output). The array `variable[]` is then scanned backwards to determine all *don't cares* . For single-fanout vertices, Eq. (2.20) is applied. Eq. (2.32) is applied to multiple-fanout vertices by a `while` loop on their fanout: as each fanout variable y_i is scanned, its observability is considered and made explicit in terms of each $y_j, j > i$. The cofactoring operation $\mathbf{ODC}^{y_i}_{\delta_{i+1},\ldots,y_m}$ is implemented by iteratively substituting those variables appearing in $\mathbf{ODC}^{y_i}$ and in the fanout of $y_{i+1}, \ldots, y_m$ and then realizing that, for $\delta_j = 1, j > i$, it is $y_j = y', j > i$. Each $y_j, j > i$ is thus just directly replaced by y'. These operations are carried out by `substitute()`. Logic optimization (for example, two-level minimization) of a vertex can be executed immediately after after computing its *don't care* .

```
OBSERVABILITY(N);
N = topsort(N);
N = augment(N);
init_odc(N);
for (i = |V|; i >= 0; i--) {
    /* variable is identified by its position ``i'' in the array */
     if (j = single_fanout(variable[i])) {
         /* Apply Eq. (2.20). */
         /* j is the index of the fanout node */
       odc[i] = odc[j] + local_component(variable[i], variable[j]);
     } else {
         /* Apply Eq. (2.32) */
         /* by scanning the fanout list of variable[i] */
         fanout_list = fanout[i];
         while (fanout_list != NULL) {
              j = fanout_list->variable;
              fanout_list = fanout_list->next;
              tmp_odc = substitute(fanout_list, odc[j]);
              odc[i] = odc[i]⊕tmp_odc;
         }
     }
}
```

Theorem 2.2 *Algorithm* `OBSERVABILITY` *computes correctly an expression of each observability don't care of* N.

Proof.

In order to prove the correctness of the algorithm it is necessary to show that, when the i^{th} vertex is considered, rules (2.20)-(2.27) are applied *correctly*, *i.e.* on expressions that are certainly correct in a sufficiently large set of perturbed networks.

In the following proof this is accomplished by showing that, when vertex y_i is considered, the observability *don't cares* of $y_j, j > i$ derived by `OBSERVABILITY` are correct in every network perturbed in at most all vertices in $V - TFO_{y_j}$, and that the correctness of these expressions is sufficient to derive a correct expression of $\mathbf{ODC}^{y_i}$ by rules (2.20)-(2.27).

This assertion is proved inductively on the index i of the vertices y_i of `variable[]`, $i = |V|, \ldots, 0$.

Base case. Since the vertices of N are ordered topologically, the vertex of index $|V|$ is a primary output, and has no fanout: $TFO_{y_{|V|}} = \phi$. Its observability *don't care* is therefore what assigned at initialization time, and it is trivially correct in *every* network perturbed in at most $\{y_1, \ldots, y_{|V|}\} = V - TFO_{y_{|V|}}$ (*i.e.* every perturbed network).

Inductive step. If the i^{th} vertex has a single fanout edge (y_i, y_j) (with $i < j$ by the topological sorting of N), then $TFO_{y_j} = TFO_{y_i} \cup \{y_j\}$.

Eq. (2.20) gives an expression of $\mathbf{ODC}^{y_i}$ of the same correctness as that of $\mathbf{ODC}^{y_j}$. By the inductive hypothesis, $\mathbf{ODC}^{y_j}$ is correct in all networks perturbed in at most $V - TFO_{y_j}$. The expression of $\mathbf{ODC}^{y_i}$ is thus correct in particular in all networks perturbed in at most $V - TFO_{y_j} = (V - TFO_{y_i}) \cup \{y_j\} \geq V - TFO_{y_i}$.

If the i^{th} vertex y has multiple fanout edges (y_i, y_{j_k}), with $k = 1, \ldots, |FO_y|$ and $i < j_k, k = 1, \ldots, |FO_y|$, OBSERVABILITY considers first expressions of each **ODC**$^{y_{j_k}}$. By the inductive hypothesis, each such expression is correct in every network perturbed in at most $V - TFO_{y_{j_k}}$. As no y_{j_h} can be in $TFO_{y_{j_k}}$, the expression is in particular correct in the case of multiple perturbations introduced in all fanout variables of y_i, and Eq. (2.32) is therefore applicable. The substitution of all variables appearing in $TFO_{y_{j_h}}, j_h > j_k$ and the cofactoring (explicit with respect to $\delta_{j_h}, j_h > j_k$ and implicit with respect to $\delta_{j_h}, j_h < j_k$) results in an expression (stored in tmp_odc) which is correct in every network perturbed in at most

$$V - \bigcup_{h=1, h \neq k}^{|FO_y|} TFO_{y_{j_h}} \geq V - TFO_{y_i} \ . \tag{2.36}$$

The eventual expression of **ODC**^{y_i} is therefore correct in every network perturbed in at most $V - TFO_{y_i}$. □

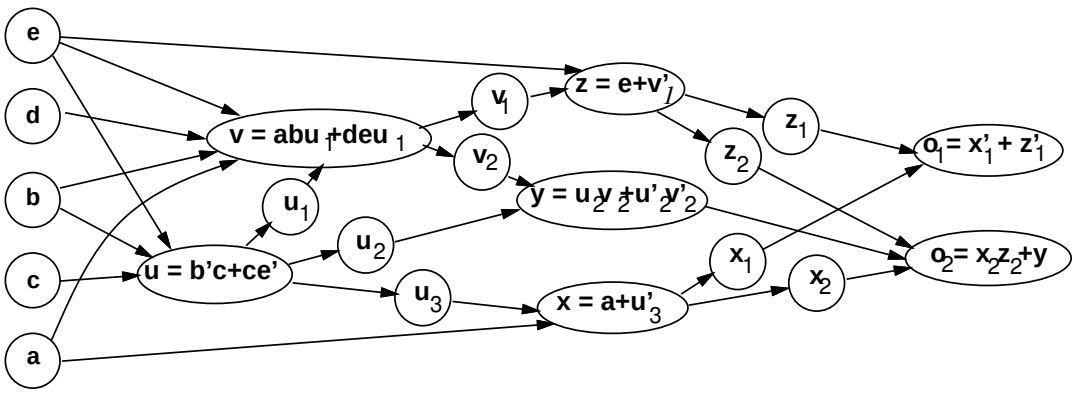


Figure 2.6: Network augmented in all internal edges, for Example (16)

Example 17.

The algorithm OBSERVABILITY is applied on the network of Fig. (2.1). A possible topological sorting of the network is : $a, b, d, e, c, u, v, z, o_1, x, y, o_2$. The augmented network is shown in Fig. (2.6) (for simplicity, only internal vertices have been augmented, and the identity functions are not

indicated). Vertices are eventually stored in `variable[]` in the order: $a, b, d, e, c, u, u_1, u_2, u_3, v, v_1, v_2, z, z_1, z_2, o_1, x, x_1, x_2, y, o_2$. Initially,

$$\mathbf{ODC}^{o_1} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} ; \mathbf{ODC}^{o_2} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} .$$

Single-fanout vertices y, x_2, x_1 are then selected in order, and Eq. (2.20) applied:

$$\mathbf{ODC}^{y} = \mathbf{ODC}^{o_2} + \left(\tfrac{\partial o_2}{\partial y}\right)' \mathbf{1} = \begin{bmatrix} 1 \\ x_2 z_2 \end{bmatrix}$$
$$\mathbf{ODC}^{x_2} = \mathbf{ODC}^{o_2} + \left(\tfrac{\partial o_2}{\partial x_2}\right)' \mathbf{1} = \begin{bmatrix} 1 \\ z_2' + y \end{bmatrix}$$
$$\mathbf{ODC}^{x_1} = \mathbf{ODC}^{o_1} + \left(\tfrac{\partial o_1}{\partial x_1}\right)' \mathbf{1} = \begin{bmatrix} z_1' \\ 1 \end{bmatrix} .$$

Vertex x has multiple fanout, and Eq. (2.32) is applied. As $\mathbf{ODC}^{x_1}$ and $\mathbf{ODC}^{x_2}$ are independent from x_2 and x_1, respectively, no substitutions or cofactors are necessary:

$$\mathbf{ODC}^{x} = \mathbf{ODC}^{x_1} \overline{\oplus} \mathbf{ODC}^{x_2} = \begin{bmatrix} z_1' \\ z_2' + y \end{bmatrix} .$$

It is then possible to compute the observability *don't care* of z_2 and z_1. As they are single-fanout vertices, Eq. (2.20) is applied again, to get:

$$\mathbf{ODC}^{z_2} = \begin{bmatrix} 1 \\ x_2' + y \end{bmatrix} ; \mathbf{ODC}^{z_1} = \begin{bmatrix} x_1' \\ 1 \end{bmatrix} .$$

The observability *don't care* of z, computed by rule (2.32), follows. Again, no substitutions or cofactors are necessary:

$$\mathbf{ODC}^{z} = \mathbf{ODC}^{z_1} \overline{\oplus} \mathbf{ODC}^{z_2} = \begin{bmatrix} x_1' \\ x_2' + y \end{bmatrix} .$$

The observability *don't care* of v_2 and v_1 are then determined by rule (2.20):

$$\mathbf{ODC}^{v_2} = \begin{bmatrix} 1 \\ x_2 z_2 \end{bmatrix} ; \mathbf{ODC}^{v_1} = \begin{bmatrix} x_1' + e \\ x_2' + y + e \end{bmatrix} .$$

Notice that no substitutions have been made so far. The expressions derived up to this point are therefore correct in every perturbed version of the network of Fig. (2.6), and in particular in $N^{v_1 v_2}$. The observability *don't care* of v by rule (2.27) follows:

$$\mathbf{ODC}^{v} = \mathbf{ODC}^{v_1} \overline{\oplus} \mathbf{ODC}^{v_2}_{\delta_1} =$$
$$\begin{bmatrix} x_1' + e \\ x_2' + y + e \end{bmatrix} \overline{\oplus} \begin{bmatrix} 1 \\ x_2(e+v) \end{bmatrix} = \begin{bmatrix} x_1' + e \\ x_2(e + y \overline{\oplus} v) \end{bmatrix} .$$

This calculation has required some substitutions, analyzed in Examples (15)-(16). The derived expression is however correct in every network perturbed in at most $V - TFO_v$. `OBSERVABILITY` determines the *don't cares* of u_3, u_2, u_1 next, using rule (2.20):

$$\mathbf{ODC}^{u_3} = \begin{bmatrix} z_1' + a \\ z_2' + y + a \end{bmatrix} ; \mathbf{ODC}^{u_2} = \begin{bmatrix} 1 \\ x_2 z_2 \end{bmatrix} ;$$
$$\mathbf{ODC}^{u_1} = \begin{bmatrix} a' + b' + x_1' + e \\ (a'+b')(d'+e') + x_2(e + y \overline{\oplus} v) \end{bmatrix} .$$

The observability *don't care* of u is then found by Eq. (2.32). Let $\delta_1, \delta_2, \delta_3$ denote the perturbations associated with u_1, u_2, u_3:

$$\mathbf{ODC}^{u} = \mathbf{ODC}^{u_1}_{\delta_2 \delta_3} \overline{\oplus} \mathbf{ODC}^{u_2}_{\delta_3} \overline{\oplus} \mathbf{ODC}^{u_3} =$$

$$\begin{bmatrix} a' + b' + e \\ (a'+b')(d'+e') + (a+u)(e+u') \end{bmatrix} \overline{\oplus} \begin{bmatrix} 1 \\ (a+u) z_2 \end{bmatrix} \overline{\oplus} \begin{bmatrix} z_1' + a \\ z_2' + a + y \end{bmatrix} .$$

Eventually, the observability *don't cares* of the primary inputs is determined. These can be used as external *don't cares* for the stages of logic controlling the network. □

In practice, fanout variables need not be added, and their *don't cares* need not be considered explicitly when traversing the network: for each multiple-fanout vertex y, the `while` loop can compute the observability *don't care* of each fanout variable by one application of rule (2.20), execute the necessary substitutions and cofactors, and add it to

`tmp_odc`, without resorting to explicit fanout vertices. This has the obvious advantage of not introducing any spurious Boolean variables and maintaining generally simpler expressions.

2.4 Multi-vertex optimization and compatible *don't cares*

The *don't care* -based method considered so far focuses on the optimization of one vertex at a time. A natural extension therefore consists of considering the simultaneous optimization of multiple vertices. Again, this process can be regarded as the introduction of error signals in the network, one for each optimized vertex. Eq. (2.14) again represents the functional constraints on such errors.

Example 18.

Fig. (2.7) shows the introduction of two perturbations. The error function is

$$\mathbf{E} = \begin{bmatrix} \delta_1'\delta_2(a+c')e'+ & \delta_1\delta_2'a'(b\oplus c+c\oplus e)+ \\ & +\delta_1\delta_2(a'c'+e'a+a'be) \\ \delta_1'\delta_2(be)'(a'c+ & c'e)+\delta_1\delta_2'(de)'(a'+bce')+ \\ & +\delta_1\delta_2(ab+ae+b'ce+c'e'+d'e) \end{bmatrix}$$

□

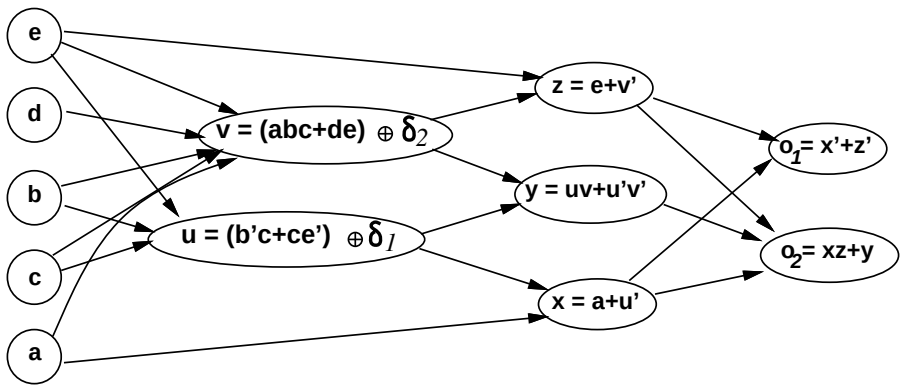


Figure 2.7: Network perturbed in correspondence of variables u and v

A first question concerning multiple-vertex optimization is whether the solution space of Eq. (2.14) can be given a compact explicit representation. A nice extension of the results of single-vertex optimization would be a representation by means of an array of independent tolerances:

$$\delta_i \leq DC^{y_i} \ . \tag{2.37}$$

If this were the case, then each internal function e^{y_i} could be optimized independently, using DC^{y_i} as *don't care* . The following example shows that unfortunately such an extension is not possible:

Example 19.

Consider the optimization of vertices u and v in the network of Fig. (2.7). With the partial assignment of inputs: $a = 0, c = 1, e = 0$, the error function is

$$\mathbf{E} = \begin{bmatrix} \delta_1 \delta_2' \\ \delta_1 \oplus \delta_2 \end{bmatrix}.$$

Assuming $\mathbf{DC} = 0$, Eq. (2.14) reduces to

$$\delta_1 \delta_2' = 0$$
$$\delta_1 \oplus \delta_2 = 0$$

which holds if and only if $\delta_1 = \delta_2$. Clearly, perturbations in this case cannot be independent, as implied by Eq. (2.37). □

The second question is whether multiple-vertex optimization can indeed achieve better results than single-vertex optimization. This is answered by Example (20) below.

Example 20.

In the network of Fig. (2.7), consider choosing $\delta_1 = \delta_2 = b$ if $a = 0, c = 1, e = 0$, and $\delta_1 = \delta_2 = 0$ elsewhere. In other words, $\delta_1 = \delta_2 = a'bce'$. The functions replacing f^u and f^v are now $g^u = f^u \oplus \delta_1 = a + b'c$ and $g^v = f^v \oplus \delta_2 = bc + de$, of lower cost than the original ones, and shown in Fig. (2.8). Notice in particular that g^u and g^v differ from f^u, f^v only for $a = 0, c = 1, e = 0$.

The separate optimization of v can be regarded as a special case of joint optimization, in which δ_1 is set to 0. For $a = 0, c = 1, e = 0$, it must now be $\delta_2 = 0$: g^v no longer belongs to the functions that can replace f^v. □

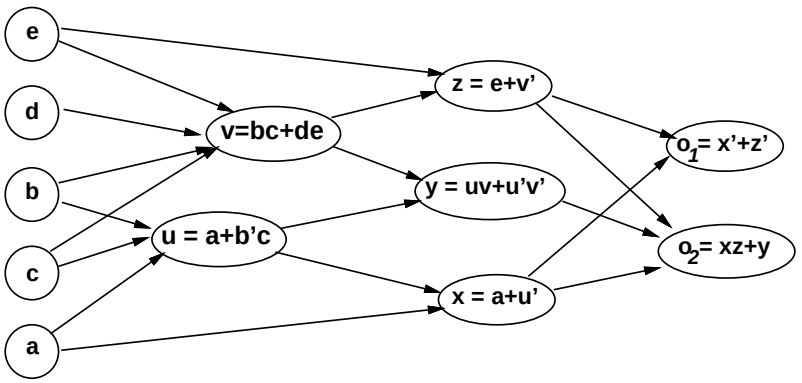


Figure 2.8: Network resulting from the simultaneous optimization of u and v

Example (20) shows that the choices on feasible perturbations for each variable y_i must in general be *correlated*. This correlation is captured in the general case by the following theorem:

Theorem 2.3 *Perturbations $\delta_1, \ldots, \delta_m$ satisfy $\mathbf{E} \leq \mathbf{DC}$ (i.e. Eq. (2.14)) if and only if*

$$\begin{aligned}
\mathbf{DC}'\mathbf{E}_{\delta_1'} &\leq \delta_1\mathbf{1} \leq \mathbf{E}'_{\delta_1} + \mathbf{DC}\ ; \\
\mathbf{DC}'(\forall_{\delta_1}\mathbf{E})_{\delta_2'} &\leq \delta_2\mathbf{1} \leq (\forall_{\delta_1}\mathbf{E})'_{\delta_2} + \mathbf{DC}\ ; \\
&\vdots \\
\mathbf{DC}'(\forall_{\delta_1,\ldots,\delta_{i-1}}\mathbf{E})_{\delta_i} &\leq \delta_i\mathbf{1} \leq (\forall_{\delta_1,\ldots,\delta_{i-1}}\mathbf{E})'_{\delta_i} + \mathbf{DC}\ ;\quad i = 1, \ldots, m
\end{aligned} \tag{2.38}$$

Proof.

The proof generalizes the derivation of *don't cares* for single perturbations. By taking the Shannon expansion of $\mathbf{E}$ with respect to δ_1, Eq. (2.14) is transformed into:

$$\delta_1'\mathbf{E}_{\delta_1'} + \delta_1\mathbf{E}_{\delta_1} \leq \mathbf{DC}. \tag{2.39}$$

Eq. (2.39) holds if and only both terms of its left-hand side are contained in **DC**:

$$\begin{aligned} \delta_1' \mathbf{E}_{\delta_1'} &\leq \mathbf{DC} \\ \delta_1 \mathbf{E}_{\delta_1} &\leq \mathbf{DC} \ . \end{aligned} \tag{2.40}$$

By using the Boolean identity

$$ab \leq c \Leftrightarrow a \leq b' + c \tag{2.41}$$

Eq. (2.40) becomes

$$\mathbf{DC}' \cdot \mathbf{E}_{\delta_1'} \leq \delta_1 \mathbf{1} \leq \mathbf{E}_{\delta_1}' + \mathbf{DC} \ . \tag{2.42}$$

One such δ_1 can exist only if the bounds expressed by Eq. (2.42) are consistent, that is, if and only if

$$\mathbf{E}_{\delta_1'} \cdot \mathbf{DC}' \leq \mathbf{E}_{\delta_1}' + \mathbf{DC} \ . \tag{2.43}$$

The same Boolean property (2.41) can then be used to transform Eq. (2.43) into

$$\mathbf{E}_{\delta_1'} \cdot \mathbf{E}_{\delta_1} = \forall_{\delta_1}(\mathbf{E}) \leq \mathbf{DC} \ . \tag{2.44}$$

Eq. (2.44) can then be expanded with respect to δ_2. By repeating steps (2.39)-(2.44),

$$\begin{aligned} \delta_1' \ (\forall_{\delta_1} \mathbf{E})_{\delta_1'} &\leq \mathbf{DC} \\ \delta_1 \ (\forall_{\delta_1} \mathbf{E})_{\delta_1} &\leq \mathbf{DC} \end{aligned} \tag{2.45}$$

results in

$$\mathbf{DC}' \cdot (\forall_{\delta_1} \mathbf{E})_{\delta_2'} \leq \delta_2 \mathbf{1} \leq (\forall_{\delta_1} \mathbf{E})_{\delta_2}' + \mathbf{DC} \tag{2.46}$$

and in the consistency equation

$$\forall_{\delta_2}(\forall_{\delta_1} \mathbf{E}) = \forall_{\delta_1 \delta_2} \mathbf{E} \leq \mathbf{DC} \ . \tag{2.47}$$

Steps (2.39-2.44) can be repeated to iteratively generate the bounds on δ_i from the consistency equation of the previous step. Theorem (2.3) is then proved completely by showing that the last consistency equation

$$\forall_{\delta_1, \ldots, \delta_m} \mathbf{E} \leq \mathbf{DC} \tag{2.48}$$

holds. But this follows from $\forall_{\delta_1, \ldots, \delta_m} \mathbf{E} \leq \mathbf{E}_{\delta_1', \ldots, \delta_m'} = \mathbf{0}$. □

Theorem (2.3) has two important consequences, that enlighten the difficulties added by dealing with multiple perturbations. First, each individual perturbation may have a *lower* bound to satisfy, in addition to the " regular" upper bound; second, each bound depends not only on the primary inputs, but also on other perturbations. Example (20) presented a case where the lower bound for δ_2 was nonzero and depended on δ_1: introducing a nonzero perturbation in vertex u indeed *forced us to change* f^v. The perturbation on u alone would have introduced an error in the functionality of the network: the lower bound on δ_2 takes then the meaning of an error on v that is *required to compensate* the error introduced by u. These difficulties can be removed by discarding some degrees of freedom and determining conditions simpler than Eq. (2.38), namely in terms of upper bounds only. In this respect, one possibility is to consider the degrees of freedom available for the optimization of each y_i *regardless* of the functions chosen to synthesize $y_j, j \neq i$. This idea is formalized by the concept of *compatible don't cares* [33, 38]. A second possibility consists instead of focusing on the network topology and of selecting suitable subnetworks, based on a "simplified" dependency of **E** on $\delta_1, \ldots, \delta_m$. This approach leads to the concept of *compatible gates*, explored later in Chapter (3).

Definition 2.5 *Don't care functions* $DC^{y_i}; i = 1, \ldots, m$ *associated with* $y_1, \ldots, y_m$ *are termed* **compatible** *if:*

1) none of them depends on any of $\delta_1, \ldots, \delta_m$*; and*

2) $\delta_i \leq DC^{y_i} \quad i = 1, \ldots, m$ *imply* $\mathbf{E} \leq \mathbf{DC}$.

Compatible don't care functions DC^{y_i} *are said to be* **maximal** *if none of them can be increased (i.e. replaced by larger functions* $B^{y_i} > DC^{y_i}$ *without violating* $\mathbf{E} \leq \mathbf{DC}$*.*

For a given array **y** of vertices there are in general several possible choices of maximal compatible *don't cares* . Theorems (2.4)-(2.5) below link one such choice to ordinary observability *don't cares* :

Theorem 2.4 *If perturbations* $\delta_1, \ldots, \delta_m$ *satisfy :*

$$\delta_i \mathbf{1} \leq \mathbf{ODC}^{y_i}_{\delta'_1, \ldots, \delta'_{i-1}}(\mathbf{x}, \delta_{i+1}, \ldots, \delta_m) + \mathbf{DC} \tag{2.49}$$

then $\mathbf{E} \leq \mathbf{DC}$*.*

Proof.

The first step of the proof consists of proving the implication :

$$\begin{cases} \delta_i \mathbf{1} \leq \mathbf{ODC}^{y_i}_{\delta'_1,\ldots,\delta'_{i-1}} + \mathbf{DC} \\ \mathbf{E}_{\delta'_1,\ldots,\delta'_i} \leq \mathbf{DC} \end{cases} \Rightarrow \begin{cases} \delta_i \mathbf{1} \leq \mathbf{E}'_{\delta'_1,\ldots,\delta'_{i-1},\delta_i} + \mathbf{DC} \\ \mathbf{E}_{\delta'_1,\ldots,\delta'_{i-1}} \leq \mathbf{DC} \end{cases} \tag{2.50}$$

for $i = m, m-1, \ldots, 1$. The algebra of the derivation is as follows:

$$\begin{cases} \delta_i \mathbf{1} \leq \mathbf{ODC}^{y_i}_{\delta'_1,\ldots,\delta'_{i-1}} + \mathbf{DC} \\ \mathbf{E}_{\delta'_1,\ldots,\delta'_i} \leq \mathbf{DC} \end{cases} \Rightarrow \begin{cases} \delta_i \mathbf{1} \leq \mathbf{ODC}^{y_i}_{\delta'_1,\ldots,\delta'_{i-1}} + \mathbf{DC} \\ \mathbf{E}'_{\delta'_1,\ldots,\delta'_i} + \mathbf{DC} = \mathbf{1} \ . \end{cases} \Rightarrow$$

$$\begin{cases} \delta_i \mathbf{1} \leq (\mathbf{ODC}^{y_i}_{\delta'_1,\ldots,\delta'_{i-1}} + \mathbf{DC}) \overline{\oplus} (\mathbf{E}'_{\delta'_1,\ldots,\delta'_i} + \mathbf{DC}) \\ \mathbf{E}'_{\delta'_1,\ldots,\delta'_i} + \mathbf{DC} = \mathbf{1} \ . \end{cases} \tag{2.51}$$

By expanding $\mathbf{ODC}^{y_i}$ and $\mathbf{E}'$ in terms of $\mathbf{F}^{\mathbf{y}}$,

$$\begin{aligned} &\mathbf{ODC}^{y_i}_{\delta'_1,\ldots,\delta'_{i-1}} \overline{\oplus} \mathbf{E}'_{\delta'_1,\ldots,\delta'_i} = \\ &(\mathbf{F}^{\mathbf{y}}_{\delta'_1,\ldots,\delta'_{i-1},\delta_i} \overline{\oplus} \mathbf{F}^{\mathbf{y}}_{\delta'_1,\ldots,\delta'_{i-1},\delta'_i}) \overline{\oplus} (\mathbf{F}^{\mathbf{y}}_{\delta'_1,\ldots,\delta'_{i-1},\delta'_i} \overline{\oplus} \mathbf{F}^{\mathbf{y}}_{\delta'_1,\ldots,\delta'_{i-1},\delta'_m}) = \\ &\mathbf{F}^{\mathbf{y}}_{\delta'_1,\ldots,\delta'_{i-1},\delta_i} \overline{\oplus} \mathbf{F}^{\mathbf{y}}_{\delta'_1,\ldots,\delta'_{i-1},\delta'_m} = \mathbf{E}'_{\delta'_1,\ldots,\delta'_{i-1},\delta_i} \ . \end{aligned} \tag{2.52}$$

Using this equality in Eq. (2.51) yields

$$\begin{cases} \delta_i \mathbf{1} \leq \mathbf{ODC}^{y_i}_{\delta'_1,\ldots,\delta'_{i-1}} + \mathbf{DC} \\ \mathbf{E}_{\delta'_1,\ldots,\delta'_i} \leq \mathbf{DC} \end{cases} \Rightarrow \begin{cases} \delta_i \mathbf{1} \leq \mathbf{E}'_{\delta'_1,\ldots,\delta'_{i-1},\delta_i} + \mathbf{DC} \\ \mathbf{E}_{\delta'_1,\ldots,\delta'_i} \leq \mathbf{DC} \ . \end{cases} \tag{2.53}$$

To complete the proof, notice that

$$\begin{cases} \delta_i \mathbf{1} \leq \mathbf{E}'_{\delta'_1,\ldots,\delta'_{i-1},\delta_i} + \mathbf{DC} \\ \mathbf{E}_{\delta'_1,\ldots,\delta'_i} \leq \mathbf{DC} \ . \end{cases} \Rightarrow \begin{cases} \delta_i \mathbf{E}_{\delta'_1,\ldots,\delta'_{i-1},\delta_i} \leq \mathbf{DC} \\ \delta'_i \mathbf{E}_{\delta'_1,\ldots,\delta'_{i-1}} \leq \mathbf{DC} \end{cases} \Rightarrow$$

$$\begin{cases} \delta_i \mathbf{1} \leq \mathbf{E}'_{\delta'_1,\ldots,\delta'_{i-1},\delta_i} + \mathbf{DC} \\ \delta'_i \mathbf{E}_{\delta'_1,\ldots,\delta'_{i-1}} + \delta_i \mathbf{E}_{\delta'_1,\ldots,\delta'_{i-1},\delta_i} \leq \mathbf{DC} \end{cases} \Rightarrow \begin{cases} \delta_i \mathbf{1} \leq \mathbf{E}'_{\delta'_1,\ldots,\delta'_{i-1},\delta_i} \leq \mathbf{DC} \\ \mathbf{E}_{\delta'_1,\ldots,\delta'_{i-1}} \leq \mathbf{DC} \ . \end{cases} \tag{2.54}$$

The last implication is in particular verified by observing that the left-hand side of Eq. (2.54) is the Shannon expansion of the right-hand side.

So far, it has been shown that Eq. (2.49), along with the initial assumption $\mathbf{E}_{\delta'_1,\ldots,\delta'_m} \leq \mathbf{DC}$, implies in particular

$$\mathbf{E}_{\delta'_1,\ldots,\delta'_i} \leq \mathbf{DC}\ ; i = m-1, m-2, \cdots, 0\ . \tag{2.55}$$

On the other hand, since $\mathbf{E}_{\delta'_1,\ldots,\delta'_m} = \mathbf{0}$, the initial assumption is always verified. In order to prove the theorem it is thus sufficient to observe that Eq. (2.14) is just Eq. (2.55) for $i = 0$. □

The bounds expressed by Theorem (2.4) still depend on other perturbation signals. Compatible *don't cares* could be obtained from them, however, in a straightforward manner by *consensus*: *don't cares* DC^{y_i} such that

$$DC^{y_i}\mathbf{1} \leq \forall_{\delta_{i+1},\ldots,\delta_m}\left(\mathbf{ODC}^{y_i}_{\delta'_1,\ldots,\delta'_{i-1}} + \mathbf{DC}\right) \tag{2.56}$$

are indeed compatible, as they are independent from any perturbation and clearly $\delta_i \leq DC^{y_i}$ implies Eq. (2.49). Theorem (2.4) below refines the *consensus* operation to obtain a set of maximal compatible *don't cares* . The idea behind the result is that when an upper bound DC^{y_i} for δ_i is derived, perturbations $\delta_{i+1}, \ldots, \delta_m$ are already bounded by $DC^{y_{i+1}}, \ldots, DC^{y_m}$. This information is equivalent to saying that combinations of inputs and perturbations violating these bounds are forbidden, and can be interpreted as external *don't cares* . Such combinations are given precisely by the terms $\delta_k(DC^{y_k})'$.

Example (20) below compares the compatible *don't cares* obtained from Eq. (2.56) with those of Eq. (2.57), in particular proving the usefulness of the terms $\delta_k(DC^{y_k})'$.

Theorem 2.5 *If functions* $DC^{y_i}; i = 1, \ldots, m$ *satisfy*

$$DC^{y_i}\mathbf{1} \leq \mathbf{DC} + \mathbf{CODC}^{y_i}; i = 1, \ldots, m \tag{2.57}$$

where

$$\begin{aligned}
\mathbf{CODC}^{y_m} &= \quad \mathbf{ODC}^{y_m}{}_{\delta'_1,\ldots,\delta'_{m-1}} \\
&\vdots \qquad\qquad \vdots \\
\mathbf{CODC}^{y_i} &= \forall_{\delta_{i+1},\ldots,\delta_m}\Big(\mathbf{ODC}^{y_i}_{\delta'_1,\ldots,\delta'_{i-1}} + \ (\sum_{k=i+1}^{m} \delta_k(DC^{y_k})')\mathbf{1}\Big); i = 1, \ldots, m\ .
\end{aligned} \tag{2.58}$$

then they represent compatible don't cares . They are maximal if the inequality (2.57) is violated by any function $B > DC^{y_i}$.

Proof.

The upper bounds DC^{y_i} are independent from any $\delta_k, k = 1, \ldots, m$; to prove their compatibility, it is thus sufficient to show that, under the assumptions (2.57),

$$\delta_i \leq DC^{y_i}; i = 1, \ldots, m \tag{2.59}$$

implies every equation of the group (2.49). To this regard, notice that Eq. (2.59) and Eq. (2.57) together imply

$$\delta_i \mathbf{1} \leq \mathbf{DC} + \mathbf{ODC}^{y_i}_{\delta'_1,\ldots,\delta'_{i-1}} + \Big(\sum_{k=i+1}^{m} \delta_k (DC^{y_k})' \Big) \mathbf{1} \tag{2.60}$$

as well as

$$\sum_{k=i+1}^{m} \delta_k (DC^{y_k})' = 0 \tag{2.61}$$

Eq. (2.49) is thus obtained by substituting Eq. (2.61) into (2.60).

The proof of maximality is by contradiction. It is shown in particular that if any upper bounds DC^{y_i} are replaced by larger bounds B^{y_i}, then it is possible to find a combination of inputs and perturbations that, although satisfying the new constraints $\delta_i \leq B^{y_i}$, nevertheless violates $\mathbf{E} \leq \mathbf{DC}$.

To this regard, suppose that there exists at least one index i such that $B^{y_i} > DC^{y_i}$. It is then possible to find an input combination $\mathbf{x}_0$ such that $DC^{y_i}(\mathbf{x}_0) = 0$, but $B^{y_i}(\mathbf{x}_0) = 1$. From $DC^{y_i}(\mathbf{x}_0) = 0$ and Eqs. (2.57)-(2.58) it follows that

$$\mathbf{DC}(\mathbf{x}_0) + \forall_{\delta_{i+1},\ldots,\delta_m} \Big(\mathbf{ODC}^{y_i}_{\delta'_1,\ldots,\delta'_{i-1}} + (\sum_{k=i+1}^{m} \delta_k (DC^{y_k})') \mathbf{1} \Big) \neq \mathbf{1} \; . \tag{2.62}$$

Eq. (2.62) can be rewritten as

$$\forall_{\delta_{i+1},\ldots,\delta_m} \Big(\mathbf{DC}(\mathbf{x}_0) + \mathbf{ODC}^{y_i}_{\delta'_1,\ldots,\delta'_{i-1}} + (\sum_{k=i+1}^{m} \delta_k (DC^{y_k})') \mathbf{1} \Big) \neq \mathbf{1} \tag{2.63}$$

Eq. (2.63) indicates that there must exist a combination δ_0 of the perturbations such that:

$$\mathbf{DC}(\mathbf{x}_0) + \mathbf{ODC}_{\delta'_{1,0},\ldots,\delta'_{i-1,0}}(\mathbf{x}_0, \delta_{i+1,0}, \ldots, \delta_{m,0}) + (\sum_{k=i+1}^{m} \delta_{k,0} (DC^{y_k}(\mathbf{x}_0))') \mathbf{1} \neq \mathbf{1}. \tag{2.64}$$

In particular, δ_0 can always be chosen so as $\delta_{1,0} = \delta_{2,0} = \ldots = \delta_{i-1,0} = 0$, and $\delta_{i,0} = 1$. Eq. (2.64) can be rewritten as a pair

$$\mathbf{DC}(\mathbf{x}_0) + \mathbf{ODC}_{\delta'_{1,0},\ldots,\delta'_{i-1,0}}(\mathbf{x}_0, \delta_{i+1,0}, \ldots, \delta_{m,0}) \neq \mathbf{1} \tag{2.65}$$

and

$$\delta_{k,0} \leq DC^{y_k}(\mathbf{x}_0); k = i+1, \ldots, m. \tag{2.66}$$

Notice that, since $DC^{y_j} \leq B^{y_j}; j = 1, \ldots, m$, the combination δ_0 does not violate the bounds B^{y_i}.

Consider now the identity shown in Theorem (2.4) (specifically, by Eq. (2.52)):

$$\mathbf{E}'_{\delta'_1,\ldots,\delta'_{i-1},\delta_i} + \mathbf{DC} = (\mathbf{ODC}^{y_i}_{\delta'_1,\ldots,\delta'_{i-1}} + \mathbf{DC}) \overline{\oplus} (\mathbf{E}'_{\delta'_1,\ldots,\delta'_i} + \mathbf{DC}). \tag{2.67}$$

It is now shown that, corresponding to $(\mathbf{x}_0, \delta_0)$ the second term of the $\overline{\oplus}$ operator in Eq. (2.67) takes value **1**. To this regard, notice that, from Eq. (2.66), $\delta_k \mathbf{1} \leq DC^{y_k}\mathbf{1} \leq \mathbf{DC} + \mathbf{ODC}^{y_k}_{\delta'_1,\ldots,\delta'_{k-1}}; k = i+1, \ldots, m$. In the proof of Theorem (2.4) this condition was shown to imply Eq. (2.54), that is, the identity to **1** of the term. Eq. (2.65) can then be rewritten as

$$\mathbf{E}'_{\delta'_1,\ldots,\delta'_{i-1},\delta_i}(\mathbf{x}_0) + \mathbf{DC}(\mathbf{x}_0) = \mathbf{ODC}^{y_i}_{\delta'_1,\ldots,\delta'_{i-1}}(\mathbf{x}_0) + \mathbf{DC}(\mathbf{x}_0) \neq \mathbf{1}\ . \tag{2.68}$$

Notice also that, from the choice $\delta_1 = \delta_2 = \ldots = \delta_{i-1} = 0$, $\delta_i = 1$,

$$\mathbf{E}'_{\delta'_1,\ldots,\delta'_{i-1},\delta_i}(\mathbf{x}_0, \delta_0) = \mathbf{E}(\mathbf{x}_0, \delta_0)\ . \tag{2.69}$$

Eq. (2.68) then becomes

$$\mathbf{E}(\mathbf{x}_0, \delta_0) + \mathbf{DC}(\mathbf{x_0}) \neq \mathbf{1} \tag{2.70}$$

indicating that Eq. (2.14) is violated by $\mathbf{x}_0, \delta_0$. Consequently, bounds B^{y_i} cannot be valid. □

Theorem (2.5) proves the intuition that the degrees of freedom associated with y_i (and again expressed by a global plus a local observability *don't care* vector $\mathbf{CODC}^{y_i}$, hereafter termed *compatible observability don't care* of y_i) that are *independent* from the

perturbations of other functions $y_j, j > i$ indeed represent maximal compatible *don't cares* . Independence is actually obtained *explicitly*, by performing the $\forall$ operation in Eq. (2.58).

Example 21.

Consider the extraction of maximal compatible *don't cares* for x and z in the network of Fig. (2.5). Two perturbations are introduced at x and at z, labeled δ_1 and δ_2, respectively. Expressions of $\mathbf{ODC}^x$ and $\mathbf{ODC}^z$ are:

$$\mathbf{ODC}^x = \begin{bmatrix} z' \\ z'+y \end{bmatrix} ; \mathbf{ODC}^z = \begin{bmatrix} x' \\ x'+y \end{bmatrix}$$

and they depend on δ_2 and δ_1 through z and x. Expliciting this dependency results in

$$\mathbf{ODC}^x = \begin{bmatrix} (e+v')\overline{\oplus}\delta_2 \\ (e+v')\overline{\oplus}\delta_2+y \end{bmatrix} ; \mathbf{ODC}^z = \begin{bmatrix} (a+u')\overline{\oplus}\delta_1 \\ (a+u')\overline{\oplus}\delta_1+y \end{bmatrix} .$$

From Theorem (2.5), a maximal compatible *don't care* of x is obtained from

$$\mathbf{CODC}^x = \mathbf{ODC}^x_{\delta_2'} = \begin{bmatrix} e'v \\ e'v+y \end{bmatrix}$$

so that

$$DC^x = e'v = e'v(e'v+y) = abce'$$

while a compatible *don't care* DC^z is obtained from

$$\mathbf{CODC}^z = \forall_{\delta_1}(\mathbf{ODC}^z + \delta_1(e'v)'\mathbf{1}) =$$

$$\begin{bmatrix} a'u \\ a'u+y \end{bmatrix} \begin{bmatrix} a+u'+e+v' \\ a+u'+e+v'+y \end{bmatrix} = \begin{bmatrix} a'u(e+v') \\ a'u(e+v')+y \end{bmatrix}$$

so that eventually

$$DC^z = a'u(e+v') = a'b'c + a'ce'.$$

Notice that if Eq. (2.56) was used instead,

$$\mathbf{CODC}^z = \forall_{\delta_1}(\mathbf{ODC}^z) = \begin{bmatrix} a'u \\ a'u + y \end{bmatrix} \begin{bmatrix} a + u' \\ a + u' + y \end{bmatrix} = \begin{bmatrix} 0 \\ y \end{bmatrix}$$

which would have resulted in $DC^z = 0$. □

As each $\mathbf{CODC}^{y_i}$ is contained in $\mathbf{ODC}^{y_i}_{\delta'_1,\ldots,\delta'_m}$, *don't cares* computed under compatibility constraints are obviously smaller than full observability *don't cares* : some degrees of freedom have been lost. In the context of combinational logic optimization, compatible *don't cares* therefore represent approximations of the full *don't cares* whose relevance lies uniquely in the possibility of changing the optimization strategy: instead of computing each *don't care* and then optimizing each vertex individually, all *don't cares* of **y** can be computed ahead of time, and then vertices optimized jointly. Compatible *don't cares* become instead unavoidable when dealing with sequential circuits, as shown later in Chapter (4).

For the subsequent analysis, it is convenient to introduce also *compatible observability care* vectors

$$\mathbf{COC}^{y_i} = \forall_{\delta_{i+1},\cdots,\delta_m}\left(\mathbf{OC}^{y_i}_{\delta'_1,\cdots,\delta'_{i-1}} + \sum_{k=i+1}^{m} \delta_k (DC^{y_k})'\right) \tag{2.71}$$

These vectors represent the conditions under which a perturbation of y_i is observed at the primary outputs, regardless of other (bounded) perturbations.

Compatible *don't cares* by local rules.

Given an array $\mathbf{y} = [y_1, \ldots, y_m]$ of vertices, their maximal compatible *don't cares* can in principle be computed by first determining their full *don't cares* , expliciting their dependencies on δ, and applying Eq. (2.57).

This complex procedure could be simplified if the network topology provides sufficient information on the dependency of each $\mathbf{ODC}^{y_i}$ on each $\delta_j, j > i$. For arbitrary network topologies and choices of **y** this is obviously not the case. It was however conjectured in [38] that if compatible *don't cares* of *all* vertices are sought (*i.e.* $m = |V|$) and if vertices appear in topological order in **y**, then maximal compatible *don't cares*

could be determined efficiently by resorting only to local rules. The proposed rules, unfortunately, do not yield maximal compatible *don't cares* . They are therefore analyzed together with other approximation methods in Sect. (2.5).

Perturbation analysis is here used to argue that, in presence of vertices with multiple fanout edges, there cannot be local rules based only on maximal compatible *don't cares* . Although no formal proof is given here, we believe that it is not possible to extract the maximal compatible *don't care* of a vertex from those of its fanout, but rather the full *don't cares* of the fanout variables are necessary. This is motivated by the following reasoning.

Consider first the case of a vertex labeled y_i with a single fanout edge (y_i, y_j). The exact *don't care* $\mathbf{ODC}^{y_i}$ is in this case computed by Eq. (2.20), while Theorem (2.5) relates exact *don't cares* to maximal compatible ones. Combining those results, $\mathbf{CODC}^{y_i}$ is given by

$$\mathbf{CODC}^{y_i} = \forall_{\delta_{i+1},\ldots,\delta_m}\left(\mathbf{ODC}^{y_j}_{\delta'_1,\ldots,\delta'_{i-1}} + \left(\frac{\partial f^{y_j}}{\partial \delta_i}\right)'_{\delta'_1,\ldots,\delta'_{i-1}} \mathbf{1} + \sum_{k=i+1}^{m} \delta_k (DC^{y_k})'\mathbf{1}\right) . \quad (2.72)$$

We should now transform Eq. (2.72) in a way that includes $\mathbf{CODC}^{y_j}$ instead of the full *don't care* $\mathbf{ODC}^{y_j}$. The *consensus* operation can be split in two operations, regarding variables $y_k, k > j$ and $y_k, k \leq j$:

$$\mathbf{CODC}^{y_i} = \forall_{\delta_{i+1},\ldots,\delta_j}\left(\forall_{\delta_{j+1},\ldots,\delta_m}\left(\mathbf{ODC}^{y_j}_{\delta'_1,\ldots,\delta'_{i-1}} + \left(\frac{\partial f^{y_j}}{\partial \delta_i}\right)'_{\delta'_1,\ldots,\delta'_{i-1}} \mathbf{1} + \sum_{k=i+1}^{m} \delta_k (DC^{y_k})'\mathbf{1}\right)\right) . \quad (2.73)$$

Eq. (2.72) indicates, however, that in order to compute $\mathbf{CODC}^{y_i}$ one has to know the dependency of $\mathbf{ODC}^{y_j}$ from the perturbations $\delta_{i+1}, \ldots, \delta_{j-1}$. Notice that, instead, $\mathbf{CODC}^{y_j}$ as defined by Eq. (2.57) is independent from all perturbations $\delta_k, k < j$.

This difficulty arises when $\mathbf{ODC}^{y_j}$ can actually depend on perturbations $\delta_k, k < j$. Network topologies of the type shown in Fig. (2.9) represent one such case: the observability *don't care* function of y_j can depend on δ_k, where $i < k < j$.

In this case, $\mathbf{CODC}^{y_i}$ can be computed exactly only if the dependency of $\mathbf{ODC}^{y_j}$ and of the local component $\frac{\partial f^{y_j}}{\partial \delta_i}$ on δ_k are known. Keeping track of such dependencies is unfeasible for large circuits.

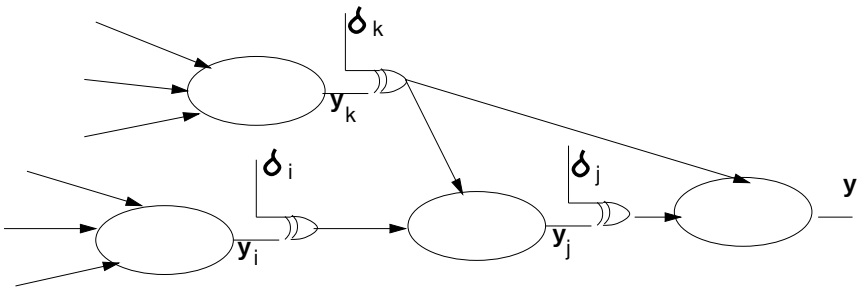


Figure 2.9: Network topology for which local computation of compatible *don't cares* may be impossible.

2.5 Approximating observability *don't cares*

For reasons of representation size and CPU time it is in practice often necessary to approximate observability *don't cares* by simpler (in terms of representation) functions $\widetilde{\mathbf{ODC}}^{y_i} \leq \mathbf{ODC}^{y_i}$. The important question here is whether network traversal rules (2.20)-(2.32) are *robust*, *i.e.* are still correct if observability *don't cares* are replaced by approximations.

Consider first rule (2.20) for single-fanout vertices. Let again y and z denote the variable and its fanout, respectively, and assume that $\mathbf{ODC}^z$ is approximated by $\widetilde{\mathbf{ODC}}^z \leq \mathbf{ODC}^z$. It follows immediately that

$$\widetilde{\mathbf{ODC}}^y = \widetilde{\mathbf{ODC}}^z + \left(\frac{\partial e^z}{\partial y}\right)' \mathbf{1} \leq \mathbf{ODC}^y \ . \tag{2.74}$$

Thus, rule (2.20) is robust with respect to approximations. Moreover, Eq. (2.74) yields the true observability *don't care* of y if $\widetilde{\mathbf{ODC}}^z = \mathbf{ODC}^z$.

The lack of monotonicity of $\overline{\oplus}$ makes instead the local rule (2.27) for multiple-fanout vertices not robust. Consider a vertex labeled y, with two fanout variables y_1, y_2. From approximations $\widetilde{\mathbf{ODC}}^{y_1} \leq \mathbf{ODC}^{y_1}$ and $\widetilde{\mathbf{ODC}}^{y_2} \leq \mathbf{ODC}^{y_2}$ in general

$$\widetilde{\mathbf{ODC}}^{y_1} \overline{\oplus} \widetilde{\mathbf{ODC}}^{y_2}_{\delta_1} = \widetilde{\mathbf{ODC}}^{y_1} \widetilde{\mathbf{ODC}}^{y_2}_{\delta_1} + (\widetilde{\mathbf{ODC}}^{y_1})'(\widetilde{\mathbf{ODC}}^{y_2}_{\delta_1})' \not\leq \mathbf{ODC}^y \tag{2.75}$$

the reason being in particular that, due to complementation, $(\widetilde{\mathbf{ODC}}^{y_i})' \not\leq \mathbf{OC}^{y_i}$. The consequences of Eq. (2.75) are severe: by using Eq. (2.27) directly on approximations,

there is danger of attributing erroneously degrees of freedom to y, thus possibly introducing functional errors during optimization. Rule (2.27) therefore needs to be replaced by a robust one whenever approximations are used.

Several substitute rules have been proposed in the past [15, 33, 34, 35]. We use here Eq. (2.27) first for examining their quality, and then for proposing new ones. For simplicity, only the case of two reconvergent fanout branches, labeled y_1, y_2, is considered, the extensions being conceptually straightforward [39].

For the purposes of this analysis, it is convenient to evidence the dependencies of $\mathbf{ODC}^{y_1}$ on δ_2 and of $\mathbf{ODC}^{y_2}$ on δ_1 by a Shannon expansion:

$$\mathbf{ODC}^{y_1} = \delta_2' \mathbf{a}_0 + \delta_2 \mathbf{a}_1 \quad ; \quad \mathbf{ODC}^{y_2} = \delta_1' \mathbf{b}_0 + \delta_1 \mathbf{b}_1. \tag{2.76}$$

where

$$\mathbf{a}_0 = \mathbf{ODC}^{y_1}_{\delta_2'}, \quad \mathbf{a}_1 = \mathbf{ODC}^{y_1}_{\delta_2}, \quad \mathbf{b}_0 = \mathbf{ODC}^{y_2}_{\delta_1'}, \quad \mathbf{b}_1 = \mathbf{ODC}^{y_2}_{\delta_1} \ . \tag{2.77}$$

By substituting these expressions in Eq. (2.27) and (2.32), and recalling that perturbations δ_1 and δ_2 are constrained to be equal ($\delta_1 = \delta_2 = \delta$), the following identity must hold:

$$\mathbf{ODC}^{y} = (\delta' \mathbf{a}_0 + \delta \mathbf{a}_1 \delta) \overline{\oplus} (\delta \mathbf{b}_0 + \delta' \mathbf{b}_1) = (\delta \mathbf{a}_0 + \delta' \mathbf{a}_1) \overline{\oplus} (\delta' \mathbf{b}_0 + \delta \mathbf{b}_1). \tag{2.78}$$

Any combination of $\mathbf{a}_0, \mathbf{a}_1, \mathbf{b}_0, \mathbf{b}_1, \delta$ violating identity (2.78) is therefore impossible. In the scalar case, the Karnaugh map of ODC^y in terms of a_0, a_1, b_0, b_1 and δ is shown in Fig. (2.10). In this map, the symbol "$-$" denotes precisely these impossible combinations.

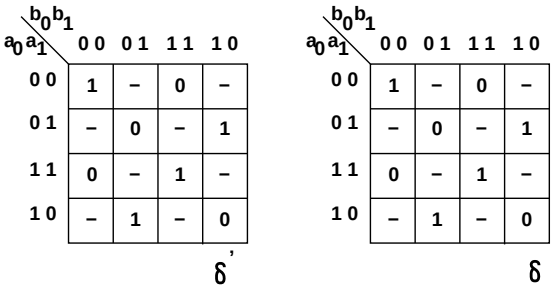


Figure 2.10: Map of ODC^y in terms of the variables a_0, a_1, b_0, b_1.

Any local method must approximate $\mathbf{ODC}^y$ by some function of $\mathbf{a}_0, \mathbf{a}_1, \mathbf{b}_0, \mathbf{b}_1$. One first measure of the quality of any such approximation can therefore be provided by the number of covered 1's in the Karnaugh map.

In [35], the following approximation is proposed. First, $\widetilde{\mathbf{ODC}}^{y_1}$ si replaced by its portion $\forall_{\delta_2}(\widetilde{\mathbf{ODC}}^{y_1})$ independent from δ_2. Clearly, the new approximation $\widetilde{\mathbf{ODC}}^{y_1}$ is now contained in $\mathbf{a}_0\mathbf{a}_1$. Similarly, $\widetilde{\mathbf{ODC}}^{y_2}$ is replaced by its portion independent from y_1. Consequently, now $\widetilde{\mathbf{ODC}}^{y_2} \leq \mathbf{b}_0\mathbf{b}_1$. Eventually, their product is formed:

$$\widetilde{\mathbf{ODC}}^{y} = \widetilde{\mathbf{ODC}}^{y_1}\widetilde{\mathbf{ODC}}^{y_2} \leq \mathbf{a}_0\mathbf{a}_1\mathbf{b}_0\mathbf{b}_1. \tag{2.79}$$

The portion of the map covered by this approximation is shown in Fig. (2.11): at most two 1's out of the 8 possible. Therefore, this rule cannot recover $\mathbf{ODC}^{y}$ from $\widetilde{\mathbf{ODC}}^{y_1} = \mathbf{ODC}^{y_1}$ and $\widetilde{\mathbf{ODC}}^{y_2} = \mathbf{ODC}^{y_1}$.

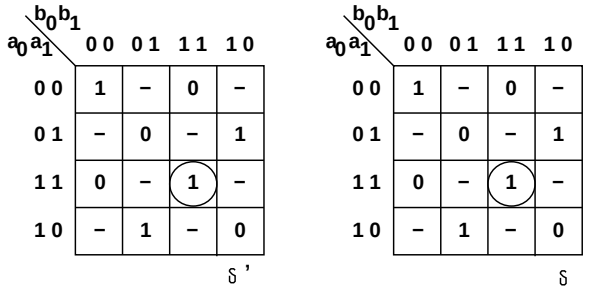


Figure 2.11: Map of $\widetilde{\mathbf{ODC}}^{y}$ as derived by Eq. (2.79)

Muroga proposes in [33] an apparently better approximation. It consists of replacing $\widetilde{\mathbf{ODC}}^{y_2}$ by an approximation that is *compatible* with the perturbation δ_1 (the arrow symbol denotes replacement):

$$\begin{aligned} \widetilde{\mathbf{ODC}}^{y_1} &\leftarrow \widetilde{\mathbf{ODC}}^{y_1}_{\delta_2'} \leq \mathbf{a}_0; \\ \widetilde{\mathbf{ODC}}^{y_2} &\leftarrow \forall_{\delta_1}(\widetilde{\mathbf{ODC}}^{y_2}) \leq \mathbf{b}_0\mathbf{b}_1 \end{aligned} \tag{2.80}$$

and of computing $\widetilde{\mathbf{ODC}}^{y}$ according to

$$\widetilde{\mathbf{ODC}}^{y} = \widetilde{\mathbf{ODC}}^{y_1}\widetilde{\mathbf{ODC}}^{y_2} = \mathbf{a}_0\mathbf{b}_0\mathbf{b}_1. \tag{2.81}$$

The portion of $\mathbf{ODC}^{y}$ that can be covered by Eq. (2.81) is shown in Fig. (2.12). Interestingly, the accuracy of Eq. (2.81) is no greater than that of Eq. (2.79). Note, however, that Eq. (2.81) requires fewer computations: since $\mathbf{ODC}^{y_1}_{\delta_2'}$ is the plain *don't care* of y_2, assuming no other perturbation in the circuit, and only $\mathbf{ODC}^{y_2}$ needs to be made independent from y_1.

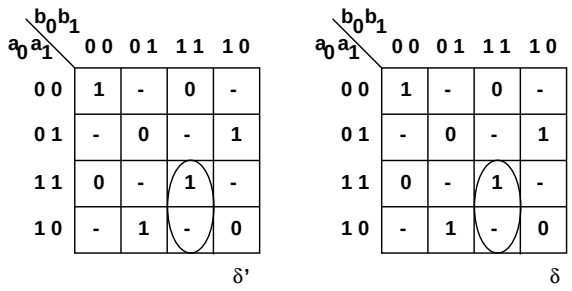


Figure 2.12: Map of ODC^y in terms of the variables a_0, a_1, b_0, b_1. Circles represent the approximation given by Eq. (2.81)

This approach was refined by Savoj *et al.* in [36], and consists essentially of replacing $\widetilde{\mathbf{ODC}}^{y_2}$ by the *maximal don't care* of y_2, compatible with y_1:

$$\widetilde{\mathbf{ODC}}^{y_2} \rightarrow \forall_{\delta_1}(\widetilde{\mathbf{ODC}}^{y_2} + \delta_1(DC^{y_1})'\mathbf{1}). \tag{2.82}$$

Eventually,

$$\widetilde{\mathbf{ODC}}^{y} = \widetilde{\mathbf{ODC}}^{y_1}\widetilde{\mathbf{ODC}}^{y_2}. \tag{2.83}$$

For a single-output network, this approximation yields

$$\begin{aligned} &\widetilde{ODC}^{y_1} = a_0; \\ &\widetilde{ODC}^{y_2} = \forall_{\delta_1}(ODC^{y_2} + \delta_1(ODC^{y_1})') = b_0(b_1 + a_0'); \\ &\widetilde{ODC}^{y} = a_0 b_0(b_1 + a_0') = a_0 b_0 b_1 \end{aligned} \tag{2.84}$$

Although the observability *don't cares* of y_1 and y_2, computed by Eqs. (2.83) or (2.84) are larger than what provided by Eq. (2.81), still when their product is formed the coverage of $\mathbf{ODC}^y$ is not improved over Muroga's method. All methods proposed so far therefore capture essentially the same portion of the observability *don't cares* , although with different degrees of efficiency.

Several more accurate approximation strategies can be derived by expanding Eqs. (2.27)-(2.32) into two-level expressions. Complements of observability *don't cares* can be replaced by approximations $\widetilde{\mathbf{OC}}^{y_i} \leq \mathbf{OC}^{y_i}$ of observability *cares* . For example, by taking into account all terms of the sum-of-products expansions, any approximation based upon:

$$\begin{aligned} \widetilde{\mathbf{ODC}}^{y} \leq \; & \widetilde{\mathbf{ODC}}^{y_1}\widetilde{\mathbf{ODC}}^{y_2}_{\delta_1} + \widetilde{\mathbf{OC}}^{y_1}\widetilde{\mathbf{OC}}^{y_2}_{\delta_1} + \\ & \widetilde{\mathbf{ODC}}^{y_1}_{\delta_2}\widetilde{\mathbf{ODC}}^{y_2} + \widetilde{\mathbf{OC}}^{y_1}_{\delta_2}\widetilde{\mathbf{OC}}^{y_2} \; ; \end{aligned} \tag{2.85}$$

$$\begin{aligned}\widetilde{\mathbf{OC}}^{y} \leq \; & \widetilde{\mathbf{ODC}}^{y_1}\widetilde{\mathbf{OC}}^{y_2}_{\delta_1} + \widetilde{\mathbf{OC}}^{y_1}\widetilde{\mathbf{ODC}}^{y_2}_{\delta_1} + \\ & \widetilde{\mathbf{ODC}}^{y_1}_{\delta_2}\widetilde{\mathbf{OC}}^{y_2} + \widetilde{\mathbf{OC}}^{y_1}_{\delta_2}\widetilde{\mathbf{ODC}}^{y_2}\end{aligned} \tag{2.86}$$

is therefore correct, and can yield $\mathbf{ODC}^{y}, \mathbf{OC}^{y}$ if the observability *don't cares* and *cares* of y_1 and y_2 are exact. The map of Fig. (2.13) shows that by Eq. (2.85)-(2.86) it is indeed possible to fully cover $\mathbf{ODC}^{y}$.

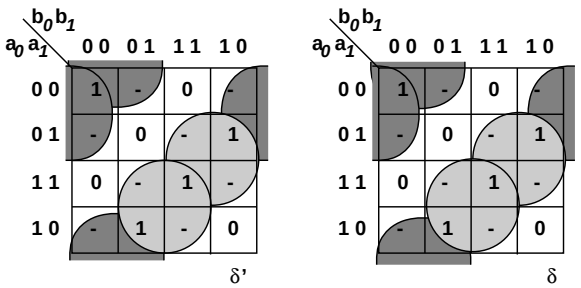


Figure 2.13: Karnaugh map showing the approximations provided by Eq. (2.86)

Approximations of observability *don't cares* can be ranked according to the availability of approximations to $\mathbf{OC}^{y_i}$, that is, upon whether Eq. (2.86) is actually used or it is simply assumed $\widetilde{\mathbf{OC}}^{y_i} = \mathbf{0}$. Solid lines represent the coverage attainable without resorting to approximations of $\mathbf{OC}^{y_i}$. It constitutes 75% of the entire map. Dotted lines represent the contribution by those terms containing $\widetilde{\mathbf{ODC}}^{y_i}$.

Approximating compatible *don't cares* **.**

Since the exact rules for compatible *don't cares* are potentially computationally more complex than those of full *don't cares* , approximation methods yielding approximations $\widetilde{\mathbf{CODC}}^{y_i}$ based on possibly simplified local rules are in this case especially important.

The starting point for single-fanout rules is the exact expression (2.73).

We report again in Fig. (2.14) the network topology causing problems. Recall that the difficulty in making the rule local was the possibility for $\mathbf{ODC}^{y_j}$ to depend on δ_k, $k < j$. This dependency becomes irrelevant if δ_k is suitably constrained, that is, if $\mathbf{ODC}^{y_k}$ is approximated by a suitable smaller function.

Unfortunately, in order to accomplish this, both rules for single- and multiple- fanout vertices must be modified **jointly**.

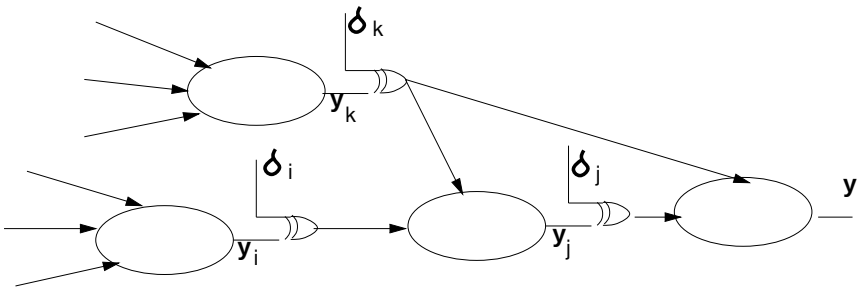


Figure 2.14: Example network for local approximation of compatible *don't cares*

First, vertices are sorted topologically, and multiple-output vertices are augmented by adding their fanout vertices and variables. Consider a single-output vertex y_i, with fanout edge (y_i, y_j). Other inputs to y_j are represented by variables $y_k, k > i$ and $y_h, h < i$. Because of the topological order, all observabilities $\mathbf{CODC}^{y_k}$ are already known when computing $\mathbf{CODC}^{y_i}$. Hence, the range of functions that can replace each f^{y_k} is known as well. The observability *don't care* of y_i is thus computed so as not to change this range. This is accomplished by using the formula

$$\mathbf{CODC}^{y_i} = \forall_{\delta_{i+1},\ldots,\delta_{j-1}} \Big(\mathbf{CODC}^{y_j} + \Big(\frac{\partial f^{y_j}}{\partial y_i} \Big)'_{\delta'_1,\ldots,\delta'_{i-1}} \mathbf{1} + \sum_{k=i+1}^{j-1} \delta_k (DC^{y_k})' \mathbf{1} \Big). \qquad (2.87)$$

Notice that now the *consensus* operation is carried out only on the local component of $\mathbf{CODC}^{y_i}$, which results in a faster implementation. The penalty paid by this approach is that perturbations on the fanout variables of a multiple-fanout vertex are now regarded as independent. Given a multiple-fanout vertex y, a function g^y can replace f^y now only if the introduced error satisfies the tolerances set *independently* on each fanout variable. In [33] the companion local rule for multiple-fanout vertices reduces to computing the intersection of the compatible *don't cares* of each fanout variable:

$$\widetilde{\mathbf{CODC}}^{y_i} = \mathbf{CODC}^{y_j} \mathbf{CODC}^{y_k} \qquad (2.88)$$

where y_j and y_k are the (two, for simplicity) fanout variables of y_i. Example (21) below shows that this rule is only an approximation to $\mathbf{CODC}^{y_i}$.

Example 22.

Consider computing compatible *don't cares* of vertices x, y and z in the network of Fig. (2.15). First, the compatible observability *don't cares* of z and y are determined. Since in particular their full *don't cares* are identically zero, $CODC^z = CODC^y = 0$. To compute $CODC^x$, first compatible *don't cares* of the two fanout variables (not shown in Fig. (2.15) are determined, using rule (2.87), and they are both zero. Consequently, when rule (2.88) is used, $C\widetilde{ODC}^x = 0$. It can however be verified that the full observability *don't care* of x is identically 1, regardless of the perturbations introduced at vertices y and z. Consequently, the maximal compatible *don't care* of x, as given by Eq. (2.57), is $CODC^x = 1$. □

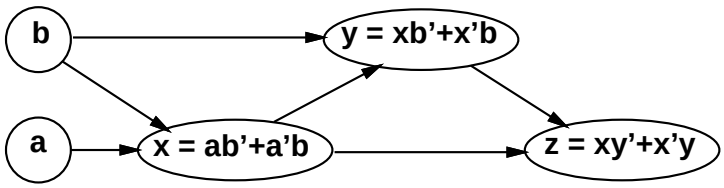


Figure 2.15: Circuit for Example (2.22)

2.5.1 Experimental results.

The algorithms for extracting observability *don't cares* have been written in C and tested against a series of benchmark combinational logic circuits. Their statistics are summarized in Table (2.1).

Each circuit was converted into a network of NOR gates only.

All logic functions are represented by their BDDs [29, 30], and manipulated accordingly, using a "home made" BDD package. Variables are ordered according to Malik's criterion.

It is worth noting that the direct application of Eq. (2.15) requires the knowledge of the function $\mathbf{F}^y$ realized by the perturbed circuit. For some of the benchmarks considered in this work, this is a well-known difficulty. Moreover, the resulting expression is in terms of primary input variables.

Circuit	Inputs	Outputs	NOR gates	Interconnections
f51m	8	7	127	262
9symml	9	1	153	375
alu2	10	6	263	924
alu4	14	8	522	1682
apex6	135	99	746	1411
apex7	49	37	222	508
k2	45	45	297	3129
i9	88	63	408	1475
pair	173	137	1919	3740
x3	135	99	1167	2631
C432	36	7	243	455
C499	41	32	531	945
C880	60	26	459	797
C1355	41	32	571	1089
C1908	33	25	490	936
C3540	50	22	1120	2249
C6288	32	32	2462	4018

Table 2.1: Benchmark statistics.

To this regard, `OBSERVABILITY` has two key potential advantages. First, it extracts the observability *don't cares* without an explicit knowledge of the network functionality. Second, the expression of these *don't cares* is in terms of other internal variables.

Table (2.2) shows the memory and CPU requirements (in terms of BDD vertices and seconds on a Sun SparcStation Classic, respectively) of `OBSERVABILITY` versus those of Eq. (2.15).

In all circuits, `OBSERVABILITY` clearly outperforms the other approach, in terms of CPU time. The use of internal variables helps greatly maintaining BDDs simple. As a result, the overall efficiency of each computation is improved.

With regards to memory occupation, `OBSERVABILITY` also outperforms direct computation. The improvement, however, is not as marked as in the case of CPU time. The reason is the following. With the direct computation, during the computation of the observability *don't care* of one gate, only the BDD of the function $\mathbf{F}^y$ needs be stored. `OBSERVABILITY`, instead, requires that the BDDs of the observability *don't cares* of

the gates along a cutset be kept at any time. Although these BDDs may individually be simpler, their size altogether may be of comparable complexity as that of $\mathbf{F}^y$. This occurs, for example, in the case of `alu2` and `alu4`. It is well known that the BDD of adder-type circuits is generally simple. The computation of the observability *don't care* vector for the internal gates is also made simple by the circuit structure. The presence of plenty of reconvergent fanout, instead, causes a lot of substitution operations during the execution of `OBSERVABILITY`. The latter method remains, however, competitive because of the simplicity of the BDDs on which these operations are performed.

We included here two cases for which the direct approach could not complete, namely, the two largest ISCAS benchmarks considered here. The extraction of the BDDs for these two circuits is extremely lengthy and difficult. Therefore, it is practically impossible to extract the BDD of the perturbed function. On the other hand, we were able to approximate the BDDs of the observability function at each vertex. The approximation was obtained by means of rule (2.85), using no observability care set. The BDDs were approximated whenever the number of BDD vertices in memory reached 200000 vertices.

2.6 Summary

We have presented *perturbation theory* as a tool for exploring *don't cares* . This idea allowed us to develop new algorithms for computing and approximating observability *don't cares* . The efficiency of the algorithms stems from the use of local rules. These rules allow us to compute observability *don't cares* of a circuit without an explicit representation of the circuit's functionality, and with the possibility of using internal variables. Moreover, the local rules we obtained could be easily simplified to yield efficient approximations (with arbitrary trade-offs between accuracy and CPU-time requirements) for the largest circuits.

	BDD nodes		CPU time	
Circuit	Eq. (2.15)	OBSERV	Eq. (2.15)	OBSERV
f51m	912	120	42	14
9symml	8505	1009	120	26
alu2	680	766	126	224
alu4	1482	2808	122	382
apex6	1350	102	821	46
apex7	2280	480	202	53
k2	5602	14550	342	293
i9	15304	16378	53	14
pair	100033	47023	92	48
x3	3506	870	22	26
C432	62505	7311	1194	255
C499	81911	23200	1531	1170
C880	10281	6070	459	79
C1355	41189	32890	571	1089
C1908	78303	2502	399	103
C3540	*	200000[1]	*	2030
C6288	*	200000[1]	*	4450

Table 2.2: Experimental results on OBSERVABILITY.The superscript [1] indicates that approximations have been employed.

Chapter 3

Multi-vertex optimization with compatible gates

Chapter 2 dealt mainly with the optimization of individual vertices of a logic network.

Exact multiple-vertex optimization had been shown to offer potentially better quality networks as compared to single-vertex optimization because of the additional degrees of freedom associated with the re-design of larger blocks of logic. The theory of exact multiple-vertex optimization was laid down by Brayton and Somenzi in [31, 8]. They formulated the problem as that of finding a minimum-cost solution to a Boolean relation, and presented a two-step algorithm for this purpose, conceptually similar to the traditional Quine-McCluskey algorithm.

Unfortunately, exact multiple-vertex optimization suffers from two major disadvantages. First, even if we consider the simultaneous optimization of only very small subsets of vertices, the number of prime implicants that have to be derived can be remarkably large. Second, given the set of prime implicants, it entails the solution of an often complex *binate covering problem*, for which efficient algorithms are still the subject of investigation. As a result, the overall efficiency of the method is limited.

Heuristic approximations to multiple-gate optimization include the use of *compatible don't cares* [33], already analyzed in Sect. (2.4). *Don't care* based optimization is extended to multiple functions by suitably restricting the individual *don't care* sets associated with each function. Although such methods are applicable to large networks, the

restriction placed on *don't care* sets reduces the degrees of freedom and hence possibly the quality of the results.

In this chapter, we show that it is possible to perform **exact** multiple gate optimization with an efficiency comparable with ordinary two-level synthesis. We show that the difficulties of ordinary exact multiple-gate optimization are due essentially from the arbitrariness of the subnetwork selected for optimization. The careful selection of the subnetwork to optimize can improve the performance of multiple-gate optimization, without sacrificing exactness. To this regard, first we introduce the notion of **compatible set of gates** as a subset of gates whose optimization can be solved *exactly* by classical two-level synthesis algorithms. We show that the simultaneous optimization of compatible gates allows us to reach optimal solutions not achievable by conventional *don't care* methods. We then leverage upon these results and present an algorithm for the optimization of more general subnetworks in an internally unate network. The algorithms have been implemented and tested on several benchmark circuits, and the results in terms of literal savings as well as CPU time are very promising.

3.1 Related Previous Work

Most Boolean methods for multiple-level logic synthesis rely upon two-level synthesis engines. For this reason and in order to establish some essential terminology, we first review some basic concepts of two-level synthesis.

3.1.1 Two-level Synthesis

Consider the synthesis of a (single-output) network whose output y is to satisfy Eq. (2.4), imposing a realization of y as a sum of cubes c_k:

$$F_{min} \leq y = \sum_{k=1}^{N} c_k \leq F_{max} \tag{3.1}$$

The upper bound in Eq. (3.1) holds *if and only if* each cube c_k satisfies the inequality

$$c_k \leq F_{max} \tag{3.2}$$

Any such cube is termed an **implicant**. An implicant is termed **prime** if no literal can be removed from it without violating the inequality (3.2). For the purpose of logic optimization, only prime implicants need be considered [40, 41]. Each implicant c_k has an associated **cost** w_k, which depends on the technology under consideration. For example, in PLA minimization all implicants take the same area, and therefore have identical cost; in a multiple-level context, the number of literals can be taken as cost measure [15]. The cost of a sum of implicants is usually taken as the sum of the individual costs.

Once the list of primes has been built, a minimum-cost cover of $\mathbf{F}_{min}$ is determined by solving:

$$\textbf{minimize}: \sum_{k=1}^{N} \alpha_k w_k; \ \textbf{subject to:} \ F_{min} \leq \sum_{k=1}^{N} \alpha_k c_k \tag{3.3}$$

where the Boolean variables α_k are used in this context to **parameterize** the search space: they are set to 1 if c_k appears in the cover, and to 0 otherwise. The approach is extended easily to the synthesis of multiple-output circuits by defining **multiple-output primes** [40, 41]. A multiple-output prime is a prime of the product of some components of $\mathbf{F}_{max}$. These components are termed the **influence set** of the prime.

Branch-and-bound methods can be used to solve exactly the covering problem. Engineering solutions have been thoroughly analyzed, for example, in [41], and have made two-level synthesis feasible for very large problems.

The constraint part of Eq. (3.3) can be rewritten as

$$\forall_{x_1,\ldots,x_n}\left(\sum_{k=1}^{N}\alpha_k c_k(\mathbf{x}) + F'_{min}(\mathbf{x})\right) = 1 \tag{3.4}$$

The left-hand side of Eq. (3.4) represents a Boolean function F_α of the parameters α_i only; the constraint equation (3.3) is therefore equivalent to

$$F_\alpha = 1 \tag{3.5}$$

The conversion of Eq. (3.3) into Eq. (3.5) is known in the literature as *Petrick's method* [40].

Two properties of two-level synthesis are worth remarking in the present context. First, once the list of primes has been built, we are guaranteed that no solution will violate the upper bound in Eq. (2.4), so that only the lower bound needs to be considered (as expressed by Eq. (3.3)). Similarly, only the upper bound needs to be considered during the extraction of primes. Second, the effect of adding/removing a cube from a partial cover of F_{min} is always predictable: that partial cover is increased/decreased. This property eases the problem of sifting the primes during the covering step, and it is reflected by the unateness of F_α: intuitively, by switching any parameter α_i from 0 to 1, we cannot decrease our chances of satisfying Eq. (3.5). These are important attributes of the problem that need to be preserved in its generalizations.

3.1.2 Boolean relations-based multiple-level optimization

Don't care -based methods allow us to optimize only one single-output subnetwork at a time. It has been shown in [8] that this strategy may potentially produce lower-quality results with respect to a more general approach attempting the simultaneous optimization of multiple-output subnetworks.

Figure (3.1) shows an arbitrary logic network, in which some gates have been selected for joint optimization. In the rest of this chapter, given a network output expression $\mathbf{F}(\mathbf{x},\mathbf{y})$, $\mathbf{x}$ is the set of input variables and $\mathbf{y}$ is the set of gate outputs to be optimized.

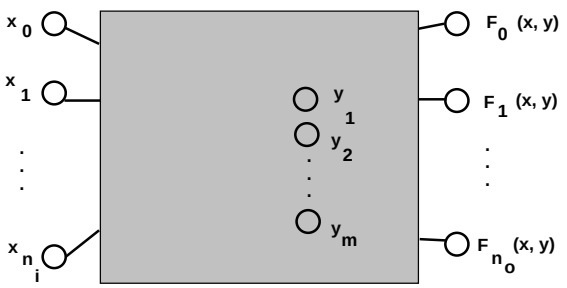


Figure 3.1: Network with Selected Gates

From equation(2.4), the functional constraints on **y** are expressed by

$$\mathbf{F}_{min}(\mathbf{x}) \leq \mathbf{F}(\mathbf{x}, \mathbf{y}) \leq \mathbf{F}_{max}(\mathbf{x}) \tag{3.6}$$

An equation like Eq. (3.6) describes a **Boolean relation**[1]. The synthesis problem consists of finding a minimum-cost realization of $y_1, \ldots, y_m$ such that Eq. (3.6) holds. An exact solution algorithm, targeting two-level realizations, is presented in [8]. It follows the Quine-McCluskey algorithm, and consists of the two steps of prime-finding and covering. The algorithm, however, is potentially very expensive in terms of CPU time. There are two main reasons. The first reason is that, even for very simple problems, a large number of primes can be generated. The second reason is that the branch-and-bound solution of the covering step has more sources of backtracking than the traditional case. We illustrate the causes of backtracking in the following example.

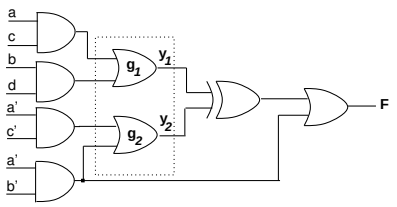


Figure 3.2: Boolean relations optimization example.

Example 23.

[1] An alternative formulation of a Boolean relation is by means of a **characteristic equation**: $R(\mathbf{x}, \mathbf{y}) = 1$, where R is a Boolean function. It could be shown that the two formulations are equivalent.

Consider the optimization of gates g_1 and g_2, with outputs y_1 and y_2, in the circuit of Figure 3.2. Assuming no external *don't care* conditions, $F_{min} = F_{max} = a'b' + (ac + bd) \oplus (a'c' + a'b')$, while $F = y_1 \oplus y_2 + a'b'$. Eq. (3.6) then takes the form:

$$\begin{aligned} a'b' + (ac + bd) \oplus (a'c' + a'b') &\leq y_1 \oplus y_2 + a'b' \\ &\leq a'b' + (ac + bd) \oplus (a'c' + a'b') \end{aligned}$$

By the symmetry of the network with respect to y_1 and y_2, cubes $a'c'$, ac, bd, $a'b'$ would be listed as implicants for both y_1 and y_2. Consider constructing now a cover for y_1 and y_2 from such implicants. An initial partial cover, for example obtained by requiring the cover of the minterm $abcd$ of $\mathbf{F}_{min}$, may consist of the cube ac assigned to y_1. Consider now adding bd to y_2, in order to cover the minterm $abc'd$ of F_{min}. Corresponding to the minterm $abcd$, now $y_1 \oplus y_2 = 0$ while $F_{min} = 1$; that is, the lower bound of Eq. (3.6) is violated. Similarly, with the input assignment $a = 0, b = 1, c = 0, d = 1$, the network output changed from the correct value 0 to 1, while $F_{max} = 0$. Thus, also the upper bound is violated.

Contrary to the case of unate covering problems, where the addition of an implicant to a partial cover can never cause the violation of any functional constraints, here the addition of a single cube has caused the violation of **both** bounds in Eq. (3.6). □

In Sect. (2.4) the difficulties of multiple-vertex optimization were interpreted as being due to the interplay of the various perturbations that makes it impossible to isolate individual bounds for each function.

Another interpretation is the following. When trying to express Eq. (3.6) in a form similar to Eq. (3.1), that is, representing individual bounds on the signals y_i, each bound may depend on other variables y_j. In turn, it could be shown that this results in a **binate** covering step. Fast binate covering solvers are the subject of ongoing research [42]; nevertheless, the binate nature of the problem reflects an intrinsic complexity which is not found in the unate case. In particular, as shown in the previous example, the effect of

adding / removing a prime to a partial solution is no longer trivially predictable, and both bounds in Eq. (3.6) may be violated by the addition of a single cube. As a consequence, branch-and-bound solvers may (and usually do) undergo many more backtracks than with a unate problem of comparable size, resulting in a substantially increased CPU time.

3.2 Compatible Gates

The analysis of Boolean relations points out that binate problems arise because of the generally binate dependence of **F** on the variables y_i. In order to better understand the reasons for this type of dependency, we assume that the vertices of the logic network actually represent individual elementary **gates** (ANDs, NANDs, ORs, NORs, inverters).

We introduce the notion of **compatible gates** in order to perform multiple-vertex optimization while avoiding the binate covering problem.

Definition 3.1 *In a logic network, let* $\mathbf{p}_j = p_j(x_1, \ldots, x_n)$ *and* $\mathbf{q} = q(x_1, \ldots, x_n)$*, where* $j = 1, 2, \ldots m$*, be functions that do not depend on* $y_1, \ldots, y_m$*. A subset of gates* $\mathcal{S} = \{g_1, \ldots, g_m\}$ *with outputs* $y_1 \ldots y_m$ *and functions* $f_1, \cdots, f_m$ *is said to be* **compatible** *if the network input-output behavior* **F** *can be expressed as:*

$$\mathbf{F} = \sum_{j=1}^{m} y_j \mathbf{p}_j + \mathbf{q} \tag{3.7}$$

modulo a polarity change in the variables y_j *or* **F**.

As shown in Sect. (3.3) below, compatible gates can be optimized jointly without solving binate covering problems. Intuitively, compatible gates are selected such that their optimization can only affect the outputs in a monotonic or unate way, and thereby forcing the covering problem to be unate.

Example 24.

Consider the two-output circuit in Figure 3.3. Gates g_1 and g_2 are compatible because F and H can be written as

$$F = (x_1 + x_3 + x_4')y_1 + (x_1 + x_2' + x_3)y_2$$

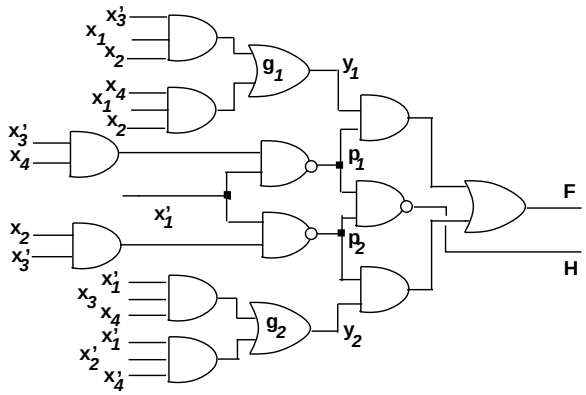


Figure 3.3: Gates g_1 and g_2 are compatible.

$$H = 0y_1 + 0y_2 + ((x_1 + x_3 + x_4')(x_1 + x_2' + x_3))'$$

□

The compatibility of a set $\mathcal{S}$ of gates is a Boolean property. In order to ascertain it, one would have to verify that all network outputs can indeed be expressed as in Definition (3.1). This task is potentially very CPU-intensive. In Section (3.4), we present algorithms for constructing subsets of compatible gates from the network topology only.

3.3 Optimizing Compatible Gates

The functional constraints for a set of compatible gates can be obtained by replacing Eq. (3.7) into Eq. (3.6). From Eq. (3.7) we obtain:

$$\mathbf{F}_{min} \leq \sum_{j=1}^{m} y_j \mathbf{p}_j + \mathbf{q} \leq \mathbf{F}_{max} \tag{3.8}$$

Eq. (3.8) can be solved using steps similar to that of two-level optimization. In particular, the optimization steps consist of *implicant extraction* and *covering*.

3.3.1 Implicant Extraction

Assuming that $\mathbf{q} \leq \mathbf{F}_{max}$, the upper bound of Eq. (3.8) holds *if and only if* for each product $y_j \mathbf{p}_j$ the inequality

$$y_j \mathbf{p}_j \leq \mathbf{F}_{max}$$

is verified, *i.e.* if and only if

$$y_j \mathbf{1} \leq \mathbf{F}_{max} + \mathbf{p}_j'; \quad j = 1, \ldots, m \tag{3.9}$$

or, equivalently,

$$y_j \leq F_{max,j}; \quad j = 1, \ldots, m \tag{3.10}$$

where $F_{max,j}$ is the product of all the components of $\mathbf{F}_{max} + \mathbf{p}_j'$. A cube c can thus appear in a two-level expression of y_j if and only if $c \leq F_{max,j}$. As this constraint is identical to Eq. (3.2), the prime-extraction strategies [40, 41] of ordinary two-level synthesis can be used.

Example 25.

Consider the optimization problem for gates g_1 and g_2 in Fig. (3.3). From Example (24)

$$p_1 = (x_1 + x_3 + x_4')'$$

$$p_2 = (x_1 + x_2' + x_3)'$$

We assume no external *don't care* set. Consequently, $F_{min} = F_{max} = x_1 x_2 x_3' + x_2 x_3 x_4 + x_1' x_2' (x_3 + x_4')$. The Karnaugh maps of F_{min} and F_{max} are shown in Fig. (3.4a), along with those of p_1 and p_2. Fig. (3.4b) shows the maps of $F_{max,1} = F_{max} + p_1'$ and $F_{max,2} = F_{max} + p_2'$, used for the extraction of the primes of y_1 and y_2, respectively. The list of all multiple-output primes is given in Table (3.1). Note that primes 1 through 5 can be used by both y_1 and y_2. □

3.3.2 Covering Step

Let N indicate the number of primes. For example, in the problem of Example (25), $N = 9$. We then impose a sum-of-products representation associated with each variable y_j:

	Primes	Influence sets
c_1	$x_1'x_2'x_3$	y_1, y_2
c_2	$x_1'x_2'x_4'$	y_1, y_2
c_3	$x_1'x_3x_4$	y_1, y_2
c_4	x_2x_4	y_1, y_2
c_5	$x_1x_2x_3'$	y_1, y_2
c_6	x_2x_3'	y_2
c_7	$x_1'x_3'x_4'$	y_2
c_8	$x_1'x_2'$	y_1
c_9	$x_1'x_4$	y_1

Table 3.1: Multiple-output primes for Example (3.25).

$$y_j = \sum_{k=1}^{N} \alpha_{jk} c_k \tag{3.11}$$

with the only restriction that $\alpha_{jk} = 0$ if y_j is not in the influence set of c_k. Since the upper bound of Eq. (3.8) is now satisfied by construction (*i.e.* by implicant computation), the minimization of $y_1, \ldots, y_m$ can be formulated as a minimum-cost covering problem

$$\mathbf{F}_{min} \leq \mathbf{q} + \sum_{j=1}^{m} \sum_{k=1}^{N} \alpha_{jk} c_k \mathbf{p}_j \tag{3.12}$$

whose similarity with Eq. (3.3) is evident, the products $c_k\ \mathbf{p}_j$ now playing the role of the primes of two-level synthesis.

Example 26.

In the optimization problem of Example (25), we are to solve the covering problem

$$F_{min} \leq p_1 y_1 + p_2 y_2$$

Using the set of primes found in Example (25), y_1 and y_2 are expressed by

$$y_1 = \alpha_{1,1}c_1 + \alpha_{1,2}c_2 + \alpha_{1,3}c_3 + \alpha_{1,4}c_4 + \alpha_{1,5}c_5 + \alpha_{1,8}c_8 + \alpha_{1,9}c_9$$

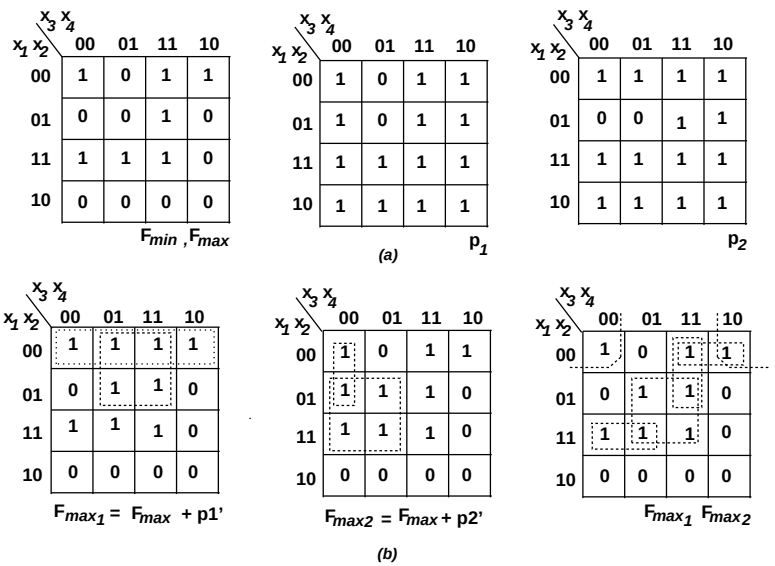


Figure 3.4: (a): Maps of $F_{min}, F_{max}, p_1, p_2$. (b) Maps of $F_{max,1}, F_{max,2}$ and of the product $F_{max,1}F_{max,2}$. Primes of y_1 and y_2 are shown in the maps of $F_{max,1}$ and $F_{max,2}$, respectively. The map of $F_{max,1}F_{max,2}$ shows the primes common to y_1 and y_2.

$$y_2 = \alpha_{2,1}c_1 + \alpha_{2,2}c_2 + \alpha_{2,3}c_3 + \alpha_{2,4}c_4 + \alpha_{2,5}c_5 + \alpha_{2,6}c_6 + \alpha_{2,7}c_7$$

The optimum solution has cost 6 and is given by $y_1 = x_1'x_2' + x_2x_4$; $y_2 = x_2x_3'$, corresponding to the assignments

$$\alpha_{1,1} = \alpha_{1,2} = \alpha_{1,3} = \alpha_{1,5} = \alpha_{1,9} = 0; \quad \alpha_{1,4} = \alpha_{1,8} = 1$$

$$\alpha_{2,1} = \alpha_{2,2} = \alpha_{2,3} = \alpha_{2,4} = \alpha_{2,5} = \alpha_{2,7} = 0; \quad \alpha_{2,6} = 1$$

The initial cost, in terms of literals, was 12. The solution corresponds to the cover shown in Fig. (3.5), and resulting in the circuit of Fig. (3.6). □

It is worth contrasting, in the above example, the role of y_1 and y_2 in covering F_{min}. Before optimization, p_1y_1 covered the minterms $x_1x_2x_3'x_4'$, $x_1x_2x_3'x_4$, $x_1x_2x_3x_4$

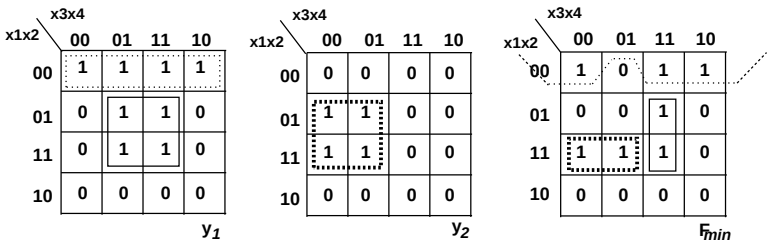


Figure 3.5: A minimum-cost solution for the covering of $\mathbf{F}_{min}$.

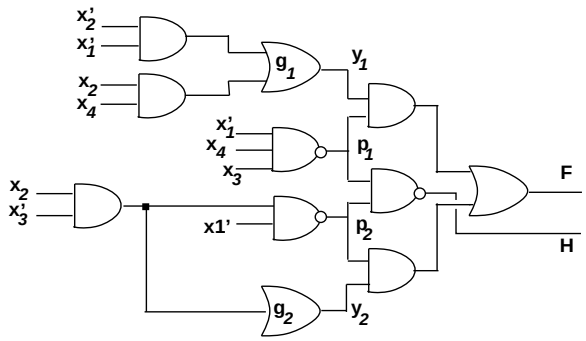


Figure 3.6: Network resulting from the simultaneous optimization of compatible gates g_1 and g_2.

of F_{min}, while $p_2 y_2$ covered $x_1' x_2' x_3' x_4'$, $x_1' x_2' x_3 x_4'$, $x_1' x_2 x_3 x_4$, $x_1' x_2' x_3 x_4$. After optimization, y_1 and y_2 essentially "switched role" in the cover: $p_2 y_2$ is now used for covering $x_1 x_2 x_3' x_4', x_1 x_2 x_3' x_4$, while $p_1 y_1$ covers all other minterms.

In the general case, the possibility for any of $y_1, \ldots, y_m$ to cover a minterm of F_{min} is evident from Eq. (3.8). Standard single-gate optimization methods based on *don't cares* [16] regard the optimization of each gate $g_1, \ldots, g_m$ as separate problems, and therefore this degree of freedom is not used. For example, in the circuit of Fig. (3.3), the optimization of g_1 is distinct from that of g_2. The *don't care* conditions associated with (say) y_1 are those minterms for which either $p_1 = 0$ or such that $p_2 y_2 = 1$, and are shown in the map of Fig. (3.7), along with the initial cover. It can immediately be verified that y_1 can only be optimized into $x_1 x_2 x_3' + x_2 x_4$, saving only one literal.

The *don't cares* for y_2 are also shown in Fig. (3.7). No optimization is possible in this case. Note also that the optimization result is (in this particular example) independent from the order in which g_1 and g_2 are optimized. Unlike the compatible gates case, it is

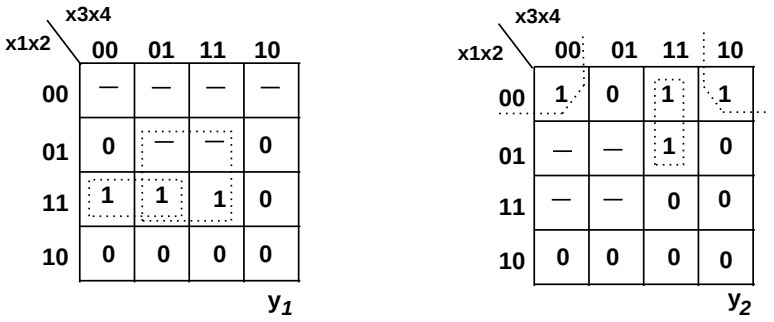


Figure 3.7: *Don't care* conditions associated with y_1 and y_2: only 1 literal can be removed.

impossible for the covers of y_1 and y_2 to exchange their role in covering F_{min}.

3.4 Finding Compatible Gates

In this section, we describe an algorithm for finding compatible gates based on network topology.

Definition 3.2 *A network is termed* **unate** *with respect to a gate* **g** *if all reconvergent paths from* **g** *have the same parity of inversions. A network is* **internally unate** *if it is unate with respect to each of its gates. All paths from* **g** *to a primary output* z_i *in an internally unate network have parity* π_i*, which is defined to be the* **parity of g** *with respect to* z_i*.*

In the subsequent analysis, we make the assumption that the network is first transformed into its equivalent NOR-only form. In this case, the parity of a path is simply the parity of the path length.

In defining Equation (3.7) for compatible gates, it is evident that the dependency of **F** on $y_1, \ldots, y_m$ must be unate. In order to increase the chances of finding sets of compatible gates, it is thus convenient to transform a network into an internally unate one. This is done by duplicating those gates whose fanouts contain reconvergent paths with different inversion parity. The resulting network is therefore at most twice the size of the original one. In practice, the increase is smaller.

Example 27.

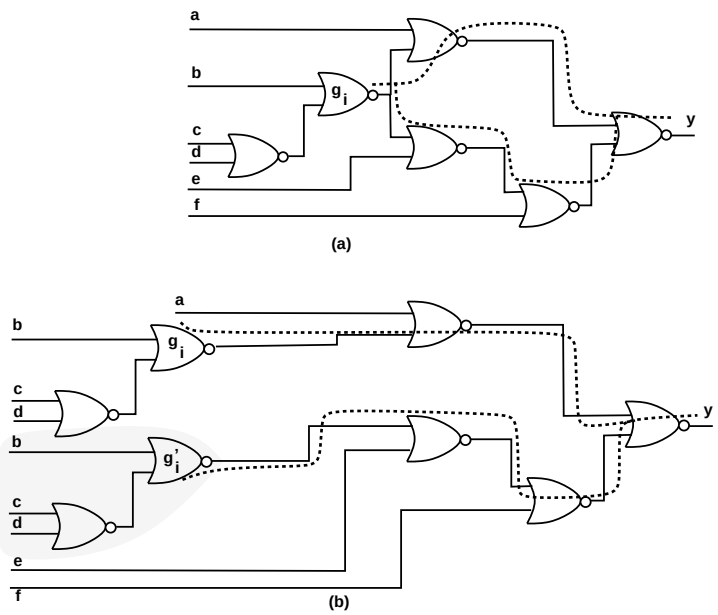


Figure 3.8: Internally Unate Example: (a) Network not internally unate due to gate g_i; (b) Internally unate network after duplication (Duplicated gates are shaded).

> Consider the logic network shown in Figure (3.8.a). The network is not internally unate because the reconvergent paths from gate g_i to the output y do not have the same parity of inversions. We duplicate gate g_i and its fan-in cone into g_i', shown by the shaded gates in Figure (3.8.b). Now gates g_i and g_i' are unate since there are no reconvergent paths from these gates. The network is now internally unate. The increase in size is in the number of gates in the fan-in cone of gate g_i. □

Theorem (3.1) below provides a sufficient conditions for a set $\mathcal{S}$ of gates to be compatible. Without loss of generality, the theorem is stated in terms of networks with one primary output. The following auxiliary definitions are required:

Definition 3.3 *The* **fanout gate set** *and* **fanout edge set** *of a gate g, indicated by $TFO(g)$ and $TFOE(g)$, respectively, are the set of gates and interconnections contained in at least one path from g to the primary outputs. The* **fanout gate set** *and* **fanout edge set** *of a set of gates $\mathcal{S} = \{g_1, \ldots, g_m\}$, indicated by $TFO(\mathcal{S})$ and $TFOE(\mathcal{S})$,*

respectively, are:

$$TFO(\mathcal{S}) = \bigcup_{i=1}^{m} TFO(g_i); \quad TFOE(\mathcal{S}) = \bigcup_{i=1}^{m} TFOE(g_i) \tag{3.13}$$

Theorem 3.1 *In a NOR-only network, let $\mathcal{S} = \{g_1, \ldots, g_m\}$ be a set of gates all with parity π, and not in each others' fanout. Let $y_1, \ldots, y_m$ denote their respective outputs. The following propositions hold:*

1): if each gate in $FO(\mathcal{S})$ with parity π has at most one input interconnection in $FOE(\mathcal{S})$, then the primary outputs can be expressed as in Eq. (3.7) for some suitable functions p_j and q, and consequently $\mathcal{S}$ is a set of compatible gates;

2) if each gate in $FO(\mathcal{S})$ with parity π' has at most one input in $FOE(\mathcal{S})$, then it can be shown that the output can be expressed as in Eq. (3.7), and $\mathcal{S}$ represents a set of compatible gates.

Proof.

We prove only Proposition 1) for the case of gates of **even parity**. The proof of the other cases is symmetric. Moreover, we prove the stronger assertion:

The output of each gate g in the network (and hence the primary outputs) can be expressed by one of the following two rules:

Rule 1: for gates of even parity,

$$f^g = q^g + \sum_{j=1}^{m} p_j^g y_j \tag{3.14}$$

Rule 2: for gates of odd parity,

$$f^g = \left(q^g + \sum_{j=1}^{m} p_j^g y_j \right)' \tag{3.15}$$

Consequently, $\mathcal{S}$ is a set of compatible gates.

Assume the network gates to be sorted topologically, so that each gate precedes its fanout gates in the list. Let $NGATES$ denote the total number of gates. We prove the above proposition by induction, by showing

that if it holds for the first $r-1$ gates, then it must hold for the r^{th} gate, $r = 1, \ldots, NGATES$.

Base step. Consider the first gate, g_1. If $g_1 \in \mathcal{S}$, its output is simply y_1, which can be obtained from Eq. (3.14), by setting $q^{g_1} = 0, p_1^{g_1} = 1, p_j^{g_1} = 0; j = 2, \ldots, m$. If g_1 does not belong to $\mathcal{S}$, by the properties of topological ordering, its inputs can only be among the primary inputs, and consequently its output is still expressed by Eq. (3.14), by setting $p_j^{g_1} = 0$.

Induction step. Consider now the r^{th} gate, g_r. Again, if $g_r \in \mathcal{S}$, the output is expressed by a single variable in $\{y_1, \ldots, y_m\}$, and therefore it satisfies the proposition. If g_r does not belong to $\mathcal{S}$, we note that all its inputs are either primary inputs or gates $g_{r'}, r' < r$, for which the proposition is true by the inductive assumption. We distinguish two cases:

1. g_r is of even parity. Consequently, all its inputs have odd parity. By the assumption of the Theorem, only one of its inputs is in $TFOE(\mathcal{S})$. Hence, only one of them is a function of the internal variables y_i. For simplicity, let g_0 denote the output that (possibly) depends on $y_1, \ldots, y_m$. The output of g_r is then expressed by

$$\left((q^{g_0} + \sum_{j=1}^{m} p_j^{g_0} y_j)' + \sum_{g_i \in FI(g_r)} q^{g_i}\right)' = q^{g_r} + \sum_{j=1}^{m} p_j^{g_r} y_j$$

where

$$q^{g_r} = q^{g_0} \prod_{g_i \in FI(g_r)} (q^{g_i})' \; ; \quad p^{g_r} = p^{g_0} \prod_{g_i \in FI(g_r)} (q^{g_i})'$$

2. g_r is of odd parity, and consequently all its inputs are from gates of even parity and are expressed by Eq. (3.14); therefore the output of g_r is expressed by

$$\left(\sum_{g_i \in FI(g_r)} (q^{g_i} + \sum_{j=1}^{m} p_j^{g_i} y_j)\right)' = \left(q^{g_r} + \sum_{j=1}^{m} p_j^{g_r} y_j\right)'$$

where

$$q^{g_r} = \sum_{g_i \in FI(g_r)} q^{g_i} \; ; \quad p_j^{g_r} = \sum_{g_i \in FI(g_r)} p_j^{g_i}$$

By induction, the output of each gate (in particular, each primary output) is expressed by Eq. (3.14) or (3.15); therefore, the gates in $\mathcal{S}$ are compatible. □

Example 28.

In the internally unate, NOR-only network of Figure (3.9), consider the set $\mathcal{S} = \{g_1, g_2, g_4\}$.

All gates of $\mathcal{S}$ are of **odd** parity and not in each other's fanout.

Moreover, $TFO(\mathcal{S}) = \{g_5,\ g_7, g_8,\ g_9,\ g_{10},\ g_{11},\ g_{12}\}$ and for all gates in $TFO(\mathcal{S})$ of **even** parity (namely, g_8, g_9, g_{10}), there is only one input interconnection that belongs to $TFOE(\mathcal{S})$. $\mathcal{S}$ then represents a compatible set by rule (1) of Theorem (3.1).

Similarly, the set $\mathcal{S} = \{g_3, g_4\}$ is compatible by rule (2), as in this case $TFO(\mathcal{S}) = \{g_6, g_7, g_{10}, g_{12}\}$, and the gates of $TFO(\mathcal{S})$ with even parity (namely, g_6 and g_7) have only one input interconnection in $TFOE(\mathcal{S})$.

Other compatible sets are, for example, $\{g_1, g_{10}\}$ (by rule (1)) and $\{g_5, g_7\}$ (by rule (2)).

It is worth noting that some gates (in this case, g_1 and g_4) can appear in more compatible sets. □

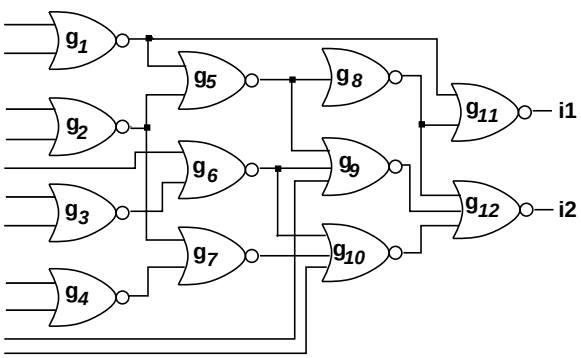


Figure 3.9: Example of Compatible Gates.

Theorem (3.1) also provides a technique for constructing a set of compatible gates directly from the network topology, starting from a "seed" gate g and a parameter ($rule$) that specifies the desired criterion of Theorem (3.1) (either 1 or 2) to be checked during the construction. The algorithm is as follows:

```
COMPATIBLES(g, rule)
label_fanout(g, TFO);
S = {g};
for(i = 0; i ≤ NGATES; i++) {
   if ((is_labeled(g_i) = FALSE) & (parity(g_i) = parity(g))) {
      label_fanout(g_i, TMP);
      compatible = dfs_check(g_i, parity(g), rule)
      if(compatible) {
        label_fanout(g_i, TFO);
        S = S ∪ {g_i};
      }
   }
}
```

$COMPATIBLES$ starts by labeling "TFO" the fanout cone of g, as no gates in that cone can belong to a compatible set containing g. Labeled gates represents elements of the set $TFO(\mathcal{S})$. All gates g_i that are not yet labeled and have the correct parity are then examined for insertion in $\mathcal{S}$. To this purpose, the fanout of g_i that is not already in $TFO(\mathcal{S})$ is temporarily labeled "TMP", and then visited by dfs_check in order to check the satisfaction of $rule$. The procedure dfs_check performs a depth-first traversal on gate g_i. The traversal returns 0 whenever gates already in $TFO(\mathcal{S})$ are reached, or a violation of $rule$ is detected. Otherwise, if the traversal reaches the primary outputs, then 1 is returned indicating that g_i is compatible. If g_i is compatible, it becomes part of $\mathcal{S}$ and its fanout is merged with $TFO(\mathcal{S})$.

Example 29.

Refer again to Figure(3.9) for this example. Consider constructing a set of compatible gates around g_1, using rule (1). Gates $g_5, g_8, g_9, g_{11}, g_{12}$ are labeled first, because they belong to $TFO(g_1)$. The first unlabeled gate is therefore g_2. The depth-first scan of its fanout reaches g_5 first, which has parity opposite to g_1. The check of the fanin of g_5 is therefore not needed.

Gates g_7 and g_{10} are then reached. In particular, since g_{10} has the same parity as g_1, its fanin is checked to verify that there is indeed only one interconnection (in this case, (g_7, g_{10})) to gates in $\mathcal{S}$. dfs_check returns in this case a value $TRUE$ for the compatibility of g_2 to g_1. □

3.5 Unate optimization

In the previous section we showed that in the case of compatible gates, the functional constraints expressed by Eq. (3.6) can be reduced to an upper bound (expressed by Eq. (3.10)) on the individual variables y_i and by a global covering constraint, expressed by Eq. (3.12). These could be solved by a two-step procedure similar to that of two-level optimization. We now generalize this result to the optimization of more general, appropriate subsets $\mathcal{S}$ of gates of an internally unate network:

Definition 3.4 *A subset $\mathcal{S}$ of gates is termed a* **unate subset** *if its elements all have the same parity and are not in each other's fanout.*

3.5.1 Optimizing Unate Subsets

Assume, for the sake of simplicity, that **F** is positive unate with respect to $\{y_1, \ldots, y_m\}$. We can perform optimization on the corresponding subset of gates in a style that is totally analogous to compatible gates by dividing it into *implicant extraction* and *covering* steps.

3.5.2 Implicant Extraction

In this step, for each y_i to be optimized, a set of *maximal functions* is extracted. In particular, the maximal functions of each each y_i can be expressed as Eq. (3.16), which is similar to Eq. (3.10).

$$y_i \leq G_{max,j}; j = 1, \ldots, m \tag{3.16}$$

From Eq. (3.16), prime implicants of y_i can then be extracted.

Intuitively, the maximal functions are the largest functions that can be used while satisfying the bound $\mathbf{F} \leq \mathbf{F}_{max}$. Therefore, they represent the upper bounds on y_i. We introduce the following definition:

Definition 3.5 *A set of local functions*

$$\{G_{max,1}(\mathbf{x}), G_{max,2}(\mathbf{x}), \ldots, G_{max,m}(\mathbf{x})\}$$

is said to be **maximal** *if*

$$\mathbf{F}(\mathbf{x}, G_{max,1}(\mathbf{x}), G_{max,2}(\mathbf{x}), \ldots, G_{max,m}(\mathbf{x})) \leq \mathbf{F}_{max}(\mathbf{x}) \qquad \forall \mathbf{x} \in \mathcal{B}^{n_i} \ . \quad (3.17)$$

and the inequality (3.17) is violated when any $G_{max,j}$ is replaced by a larger function $\tilde{F} > G_{max,j}$.

The idea behind the notion of maximal functions is that by substituting each y_j by any function $\phi_j \leq G_{max,j}$, we are guaranteed that the upper bound

$$\mathbf{F}(\mathbf{x}, \phi_1(\mathbf{x}), \ldots, \phi_m(\mathbf{x})) \leq \mathbf{F}_{max}(\mathbf{x}) \quad (3.18)$$

will not be violated. The conditions

$$y_i \leq G_{max,i}$$

therefore represent *sufficient* conditions for this bound to hold.

The following theorem provides means for finding a set of maximal functions. It also shows that computing such functions has complexity comparable with computing ordinary *don't care* sets.

Theorem 3.2 *Let $\mathcal{S} = \{g_1, \ldots, g_m\}$ be a unate subset of gates. The set of maximal functions, as defined by Eq. (3.17), with respect to the gates in $\mathcal{S}$ can be obtained by:*

$$G_{max,j} = f^{g_j} + DC_j \quad (3.19)$$

where f^{g_j} denotes the output function of g_j in the unoptimized network. DC_j represents the don't care set associated with g_j calculated with the following rule: the output functions for gates $g_1, \ldots, g_{j-1}$ are set to $G_{max,k}, k = 1, \ldots, j-1$, and the output functions for gates $g_j, \ldots, g_m$, are set to $f^{g_k}; k = j, \ldots, m$.

Proof.

The proof is divided into two parts. First, it is shown that the bounds $G_{max,j} = f^{g_j} + DC_j$ satisfy Eq. (3.17). It is then shown, by contradiction, that these bounds are indeed maximal.

To prove the first part, suppose that maximal functions for the gates $g_1, \ldots, g_{j-1}$ have already been computed. They are such that

$$\mathbf{F}(\mathbf{x}, G_{max,1}, \ldots, G_{max,j-1}, f^{g_j}, f^{g_{j+1}}, \ldots, f^{g_m}) \leq \mathbf{F}_{max}$$

The constraint equation on y_j can then be expressed by:

$$\mathbf{F}_{min} \leq \mathbf{F}(\mathbf{x}, G_{max,1}, \ldots, G_{max,j-1}, y_j, f^{g_{j+1}}, \ldots, f^{g_m}) \leq \mathbf{F}_{max}$$

and is satisfied as long as g_j satisfies

$$f^{g_j} \cdot DC'_j \leq g_j \leq f^{g_j} + DC_j$$

where DC_j is the *don't care* set associated with y_j, under the theorem's assumptions. It is then guaranteed that

$$\mathbf{F}(\mathbf{x}, G_{max,1}, \ldots, G_{max,j-1}, G_{max,j}, f^{g_{j+1}}, \ldots, f^{g_m}) \leq \mathbf{F}_{max}$$

for $j = 1, \ldots, m$.

To prove maximality, it is sufficient to show that $G_{max,j}$ cannot be replaced by any function $F > G_{max,j}$. Suppose, by contradiction, that a different bound F can be used, such that for some input combination $\mathbf{x}_0$ we have $G_{max,j}(\mathbf{x}_0) = 0$ but $\tilde{\mathrm{F}}_j(\mathbf{x}_0) = 1$. Notice that $G_{max,j}(\mathbf{x}_0) = 0$ implies that $f^{y_j}(\mathbf{x}_0) = 0$ and $DC(\mathbf{x}_0) = 0$. Corresponding to $\mathbf{x}_0$, it must then be

$$\mathbf{F}(\mathbf{x}_0, G_{max,1}(\mathbf{x}_0), \cdots, G_{max,j-1}(\mathbf{x}_0), 0, G_{max,j+1}(\mathbf{x}_0), \cdots, G_{max,m}(\mathbf{x}_0)) \leq \mathbf{F}_{max}$$

but, because $\mathbf{F}$ is positive unate and because $DC(\mathbf{x}_0) = 0$, it must also be

$$\mathbf{F}(\mathbf{x}_0, G_{max,1}(\mathbf{x}_0), \ldots, G_{max,j-1}(\mathbf{x}_0), 1, G_{max,j+1}(\mathbf{x}_0), \cdots, G_{max,m}(\mathbf{x}_0)) \not\leq \mathbf{F}_{max}\ .$$

Hence, if f^{g_j} is replaced by any other function h^{g_j} such that $h^{y_j}(\mathbf{x}_0) = 1$ (as suggested by the bound F) and all other functions $f^{y_k}, k \neq j$ are replaced by $G_{max,k}$, the upper bound $\mathbf{F} \leq \mathbf{F}_{max}$ is violated corresponding to the input combination $\mathbf{x}_0$. □

Note that the computation of each maximal function corresponds to finding the local *don't care* for the associated vertex. Therefore, the maximal functions computation has the same complexity as computing the *don't care* conditions for each gate.

This theorem states that the maximal function for vertex i depends on the maximal functions already calculated ($j < i$). This means that unlike the case of compatible gates, the maximal function for a given vertex may be not unique.

Example 30.

For the network of Fig. (3.10), assuming no external *don't care* conditions, we find the maximal functions for y_1, y_2, and y_3. The DC_{y_j} terms correspond to the observability *don't care* at y_j, computed using the F_{max} of the previous gates.

$$y_1 = x_1 x_3' x_4; \quad y_2 = x_3'(x_4 + x_2); \quad y_3 = x_3' x_2 + x_1' x_2'$$

Maximal functions derived by Theorem (3.2) are :

$$G_{max,1} = x_1 x_3' x_4 + DC_{y_1} = x_3' x_4 + (x_3' + x_4) x_1' x_2'$$

$$\begin{aligned} G_{max,2} &= x_3'(x_4 + x_2) + DC_{y_2}(y_1 = G_{max,1}) \\ &= x_4 + x_3' x_2' + x_1 x_2' + x_3 x_2 \end{aligned}$$

$$\begin{aligned} G_{max,3} &= x_3' x_2' + x_1' x_2 + DC_{y_3}(y_1 = G_{max,1}, y_2 = G_{max,2}) \\ &= x_3' x_2 + x_1' x_2' + x_4 x_3' \end{aligned}$$

□

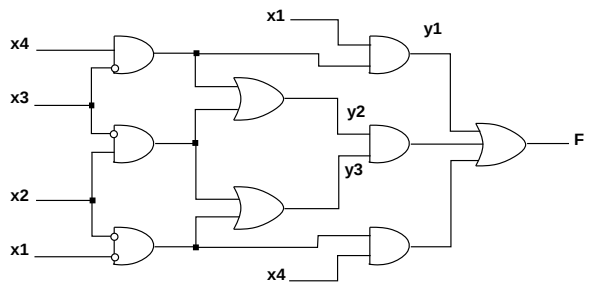


Figure 3.10: Network for Example (8).

3.5.3 Covering Step

Eq. (3.16) allows us to find a set of multiple-output primes for $y_1, \ldots, y_m$. The covering step then consists of finding a minimum-cost sum such that the lower bound of Eq. (3.6) holds.

We now present a reduction for transforming the covering step to the one presented for compatible gates. We first illustrate the reduction by means of an example.

Example 31.

In Fig. (3.10), consider the combination of inputs $\mathbf{x}$ resulting in $\mathbf{F}_{min}(\mathbf{x}) = 1$. To each such combination we can associate the set of values of y_1, y_2, y_3 such that $\mathbf{F}(\mathbf{x}, \mathbf{y}) = 1$. For instance, for the entry $x_1x_2x_3x_4 = 1001$, it must be $F_{x_1x_2'x_3'x_4}(\mathbf{y}) = y_1 + y_2y_3 = 1$. Let us now denote with $G(\mathbf{y})$ the left-hand side of this constraint, *i.e.* $G(\mathbf{y}) = y_1 + y_2y_3$. Notice that $G(\mathbf{y})$ is unate in each y_j and changes depending on the combination of values currently selected for x_1, x_2, x_3, x_4.

Any constraint $G(\mathbf{y}) = 1$ can be represented in a canonical form:

$$G(\mathbf{y}) = (G_{y_1'y_2'y_3'} + y_1 + y_2 + y_3)(G_{y_1'y_2'y_3} + y_1 + y_2) \ldots (G_{y_1y_2y_3'} + y_3)G_{y_1y_2y_3} = 1$$

which, in turn, is equivalent to the 8 constraints

$$\begin{array}{llllll} G_{y_1' y_2' y_3'} + & y_1 + & y_2 + & y_3 & = 1 \\ G_{y_1' y_2' y_3} + & y_1 + & y_2 & & = 1 \\ \ldots & & & & \\ G_{y_1 y_2 y_3'} + & & & y_3 & = 1 \\ G_{y_1 y_2 y_3} & & & & = 1 \end{array} \tag{3.20}$$

By introducing an auxiliary variable z_j for each y_j, we can rewrite Eq. (3.20) as:

$$G(\mathbf{z}) + z_1' y_1 + z_2' y_2 + z_3' y_3 = 1 \quad \forall\ z_1, z_2, z_3$$

or, equivalently,

$$G'(\mathbf{z}) \leq z_1' y_1 + z_2' y_2 + z_3' y_3$$

In this particular example, we get

$$(z_1 + z_2 z_3)' \leq z_1' y_1 + z_2' y_2 + z_3' y_3$$

□

Example (31) shows a transformation that converts the covering problem of an arbitrary unate subset of gates into a form that is similar to optimization of compatible gates, *i.e.* Eq. (3.8).

More generally, corresponding to each combination $\mathbf{x}$ such that $\mathbf{F}_{min}(\mathbf{x}) = 1$, the constraint $\mathbf{F}(\mathbf{x}, \mathbf{y}) = 1$ can be re-expressed as

$$\mathbf{F}(\mathbf{x}, \mathbf{z}) + z_1' y_1 + z_2' y_2 + \ldots + z_m' y_m = 1$$

The resulting covering problem to find the minimum-cost solution is analogous to the compatible gates case. The transformation is formalized in the following theorem:

Theorem 3.3 *Given* **F(x)***, let* **y** *be a unate subset of variables with respect to* **F**. *Let* $\mathbf{z} = [z_1, \ldots, z_m]$ *denote* m *auxiliary Boolean variables. The lower bound of Eq. (3.6) holds if and only if*

$$\mathbf{F}_{min} \leq \mathbf{F}(\mathbf{x}, \mathbf{z}) + \sum_{j=1}^{m} y_j (z_j' \mathbf{1}) \quad \forall \mathbf{z} \tag{3.21}$$

Proof.

We first show by contradiction that

$$\mathbf{F}(\mathbf{x},\mathbf{y}) \leq \mathbf{F}(\mathbf{x},\mathbf{z}) + \sum_{j=1}^{m} y_j(z_j'\mathbf{1}) \tag{3.22}$$

Eq. (3.22) can be violated only by a combination $\mathbf{x}_0, \mathbf{y}_0, \mathbf{z}_0$ such that one component of $\mathbf{F}(\mathbf{x}_0, \mathbf{y}_0)$ takes value 1, the same component of $\mathbf{F}(\mathbf{x}_0, \mathbf{z}_0)$ takes value 0, and the rightmost term of Eq. (3.22) takes value zero. In any such combination, there must be at least one value $y_{i,0} = 1$ and $z_{i,0} = 0$ (or otherwise, by the unateness of $\mathbf{F}$, we would have $\mathbf{F}(\mathbf{x}_0, \mathbf{y}_0) \leq \mathbf{F}(\mathbf{x}_0, \mathbf{z}_0)$).

But if there exists an index i such that $y_{i,0} = 1, z_{i,0} = 0$, then the rightmost term of Eq. (3.22) takes value 1, and the right-hand side of the inequality holds, a contradiction.

Therefore, $\mathbf{F}_{min}(\mathbf{x}) \leq \mathbf{F}(\mathbf{x},\mathbf{y})$ together with Eq. (3.22) implies

$$\mathbf{F}_{min}(\mathbf{x}) \leq \mathbf{F}(\mathbf{x},\mathbf{z}) + \sum_{j=1}^{m} y_j(z_j'\mathbf{1})$$

To complete the proof, it must now be shown that $\mathbf{F}_{min}(\mathbf{x}) \leq \mathbf{F}(\mathbf{x},\mathbf{z}) + \sum_{j=1}^{m} y_j(z_j'\mathbf{1})$, $\forall \mathbf{z}$ implies $\mathbf{F}_{min}(\mathbf{x}) \leq \mathbf{F(x,\ y)}$. Suppose, by contradiction, that this is not true. There exists then a value $\mathbf{x}_0, \mathbf{y}_0$ such that some component of $\mathbf{F}_{min}(\mathbf{x})$ takes value 1, $\mathbf{F}(\mathbf{x}_0, \mathbf{y}_0)$ takes value 0, but $\mathbf{F}_{min}(\mathbf{x}_0) \leq \mathbf{F}(\mathbf{x}_0, \mathbf{z}) + \sum_{j=1}^{m} y_{j,0}(z_j'\mathbf{1}), \quad \forall \mathbf{z}$. In this case, it must be $\mathbf{F}(\mathbf{x}_0, \mathbf{z}) + \sum_{j=1}^{m} y_{j,0}(z_j'\mathbf{1}) = 1$, regardless of $\mathbf{z}$. But this implies that, for $\mathbf{z} = \mathbf{y}_0, \mathbf{F}(\mathbf{x}_0, \mathbf{z}) = 1$, *i.e.* $\mathbf{F}(\mathbf{x}_0, \mathbf{y}_0) = 1$, a contradiction. □

Eq. (3.21) has the same format of Eq. (3.8), with $\mathbf{q}$ and $\mathbf{p}_j$ being replaced by $\mathbf{F}(\mathbf{x},\mathbf{z})$ and $z_j'\mathbf{1}$, respectively. Theorem (3.3) thus allows us to reduce the covering step to the one used for compatible gates. Theorems (3.2) and (3.3) show that the algorithms presented in Sect. (3.2) can be used to optimize arbitrary sets of gates with the same parity, without being restricted to sets of compatible gates only.

3.6 Implementation and Results

We implemented the algorithms of Sects. (3.2) and (3.3) in a general logic optimization framework. The original networks are first transformed into a unate, NOR-only description. All internal functions are represented using BDDs [29]. For each unoptimized gate g_i, the following heuristic is used. First, we try to find a set of compatible gates for g_i, called $\mathcal{S}_c$. In the case where not enough compatible gates can be found, we find a set of gates that is unate with respect to g_i, called $\mathcal{S}_a$.

In the case where $\mathcal{S}_c$ is found, we use Eq. (3.7) to extract the functions $\mathbf{p}_j$ and $\mathbf{q}$. In particular, $\mathbf{q}$ is computed by setting y_j to 0. The functions $\mathbf{p}_j$ are then computed by setting y_j to 1, with $y_i; i \neq j$ stuck-at 0.

In the case of optimizing arbitrary unate subnetworks $\mathcal{S}_a$, Theorem (5.1) is used to determine the maximal functions for each y_j. Note that optimizing $\mathcal{S}_c$ is preferable because for a set of m compatible gates, $m+1$ computations for $\mathbf{p}_j$ and $\mathbf{q}$ are needed to obtain all the required *don't cares* . For $\mathcal{S}_a$, two computations (with y_j stuck-at-0 and stuck-at-1) are required for the extraction of the *don't care* set of each variable y_j, resulting in a total of $2m$ computations.

A set of primes for the gate outputs is then constructed. Because of the possibly large number of primes, we limit our selection to single-literal primes only. These correposnd to wires already existing in the network and that can be used as primes for the function under optimization. The BDD of $\mathbf{F}(\mathbf{x}, \mathbf{z})$ is then built, and the covering problem solved. Networks are then iteratively optimized until no improvement occurs, and eventually folded back to a binate form. The algorithms presented in this chapter were implemented in C program called ACHILLES, and tested against a set of MCNC synthesis benchmarks.

Table (3.2) illustrates initial statistics for the benchmark circuits considered in this experiment. Table (3.3) provides a comparison of ACHILLES with SIS using *script.rugged*. The column *Initial Stat.* lists the network statistics before optimization, where *Int.* is number of internal interconnections and *gates* is the gate count. The column *Interconn.* shows number of interconnections after optimization. The *gates* column compares final gate counts. *Literal* column shows the final literals in factored form. The results in the table show that ACHILLES performs better than SIS for all figures of merit. In particular,

Circuit	Interconnections	Gates
cm85a	108	63
cm162a	113	60
pm1	130	60
9symml	375	152
alu2	924	262
alu4	1682	521
apex6	1141	745
C499	945	530
C880	797	458
C1908	936	489

Table 3.2: Benchmark statistics.

	Interconn.		Lits.(fac)		Gates		*CPU*	
Circuit	ACHILLES	SIS	ACHILLES	SIS	ACHILLES	SIS	ACHILLES	SIS
cm85a	67	77	42	46	31	34	1.5	1.2
cm162a	99	102	47	49	41	52	1.8	1.3
pm1	67	78	47	52	31	36	1.6	1.3
9symml	288	325	163	186	88	101	108.4	64.2
alu2	366	570	303	362	215	231	309.7	403.0
alu4	902	1128	612	703	420	487	1612.6	1718.5
apex6	1009	1315	687	743	589	639	115.1	30.3
C499	913	945	505	552	498	530	202.1	133.6
C880	643	731	355	409	295	342	340.6	30.7
C1908	828	891	518	542	445	482	422.1	138.8

Table 3.3: Optimization results. Runtimes are in seconds on DEC5000/240. ACHILLES does 11% better than SIS in factored literals.

Note that *script.rugged* was chosen because it is the most robust script of the SIS script suite, and it matches closely to our type of optimization. Our objective was to compare optimization results based only on Boolean operations, namely compatible gates versus *don't cares* . The *script.rugged* calls *full_simplify*[38], which computes observability *don't cares* to optimize the network.

The table shows that the ACHILLES runtimes are competitive with that of SIS. In this implementation, we are more interested in the quality of the optimization than the efficiency of the algorithms, therefore an *exact* covering solver is used. We can improve the runtime in the future by substituting a faster heuristic or approximate solvers (such

as used in ESPRESSO [41]).

3.7 Summary

In this chapter we presented a comparative analysis of approaches to multi-level logic optimization, and described new algorithms for simultaneous multiple-gate optimization. The algorithms are based on the notion of **compatible gates** and unate networks. We identify the main advantage of the present approach over previous solutions in its capability of exact minimization of suitable multiple-output networks, by means of traditional two-level optimization algorithms. Experimental results show an improvement of 11% over existing methods.

Chapter 4

Acyclic synchronous networks

Traditional research on the synthesis of synchronous logic has focused very much on the manipulation of state diagram-like representation of sequential functions. Finite-state machine decomposition, state minimization, and encoding are the typical steps leading to the later construction of the combinational network realizing the output and next-state functions. If the result of combinational synthesis is unsatisfactory, these steps need be carried out again. To this regard, it is worth noting that there is little knowledge on the impact of these steps on the definition of the combinational portion. Moreover, it is often the case where a sequential circuit is already given a structural (*i.e.* netlist) description. To date, however, state minimization and assignment algorithms work on explicit representations of the state diagram. For most networks, such an explicit representation is made impossible by the sheer number of internal states and by the network complexity.

The present and the next two chapters attempt the definition of a **structural** approach to synchronous logic optimization. We consider optimization steps such as modifying logic gates so as to reduce their cost, and adding/removing registers. The internal states of the network are therefore changed **implicitly**, *i.e.* as a result of the optimization steps.

To carry out this task, we need to adopt a non-Huffman model of a synchronous logic network, and a description of its functionality in terms of **sequence functions** rather than state diagrams or flow tables. The presence of delays and of feedback paths are distinct causes of difficulties. For this reason, the analysis of networks with feedback is

postponed to Chapter (6).

In the present chapter, we attempt the extension of perturbation analysis and *don't care* -based optimization to the synchronous case. Sections (4.1)-(4.2) are devoted to the modeling of synchronous networks and sequential *don't cares* , respectively. Section (4.3) is then devoted to the extension of perturbation analysis to the case of acyclic networks. There, however, it is also shown that *don't care* sets are an insufficient means for representing the degrees of freedom for sequential optimization. A more accurate analysis leads to the concept of **recurrence equations**, explored later in Chapter (5).

4.1 Terminology

4.1.1 Synchronous logic networks.

The model of synchronous circuits employed here is the **synchronous logic network**.

The graph of a synchronous network differs from that of a combinational network in three respects. First, it has non-negative weights associated with its edges. A weight w on an edge e denotes an interconnection containing w cascaded D-type flip-flops. Second, more edges with distinct weights are allowed to connect the same two vertices (*i.e.* the graph is actually a *multigraph*). Third, the network may contain feedback loops. The restriction of not allowing loops of combinational logic is expressed by constraining all loops to have at least one edge of nonzero weight. A synchronous network is termed *acyclic* if it has no loops. Every network can be decomposed into an acyclic portion, containing in particular all logic and delay elements, and a set of feedback interconnections.

Vertices are labeled by a Boolean variable, and have associated a **pattern function**, formally introduced below by Definition (4.1). Pattern functions are described by **pattern expressions**[1]. They reduce to ordinary Boolean functions and expressions, respectively, in absence of registers: combinational logic networks are thus special cases of synchronous networks, with identically zero edge weights and consequently no feedback loops.

[1]They are also called **synchronous Boolean expressions** in [25, 43]

Example 32.

Fig. (4.1) shows a synchronous circuit and its associated graph. Notice the presence of feedback loops and of edges of different weights, for instance those connecting vertices s and y. Each vertex of the graph encloses the pattern expression associated with it. Subscripts of variables indicate delays. □

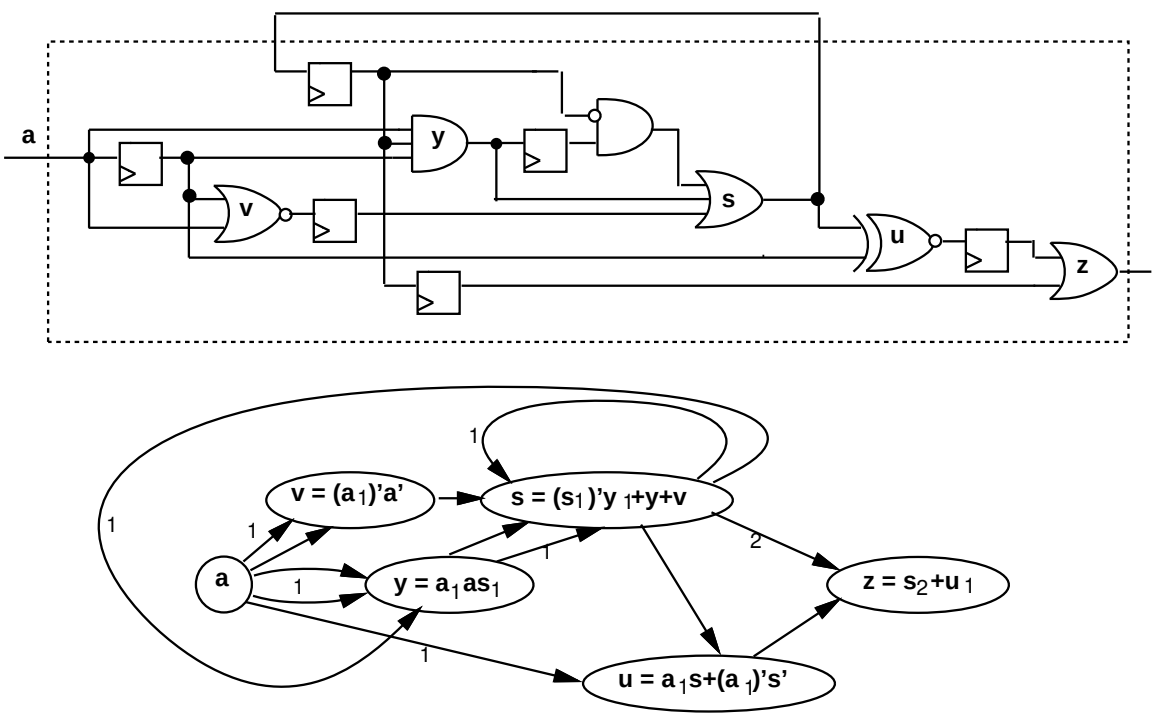


Figure 4.1: A synchronous logic network and its associated graph

4.1.2 Sequences and sequence functions.

The functionality of synchronous circuits is here described by explicitly taking into account the evolution of the network variables over time and reasoning in terms of *sequences* of Boolean values. For this reason it is first necessary to introduce the terminology associated with sequences and sequence mappings.

Time is represented by a set of integer time-points $\mathcal{Z} = \{-\infty, \ldots, -1, 0, 1, \ldots, \infty\}$. A **sequence** s of elements from a finite set $\mathcal{S}$ is a mapping $s : \mathcal{Z} \to \mathcal{S}$. The value of the

sequence at time n is an element of $\mathcal{S}$, and is denoted by $s_{@n}$. The set of all possible sequences of elements in $\mathcal{S}$ is denoted by $\mathcal{S}^\omega$ [44].

A Boolean sequence is a sequence of elements of $\mathcal{B}$. The set of Boolean sequences is hereafter denoted by $\mathcal{B}^\omega$. The sets of possible sequences of input and output vectors of a n_i-input, n_o-output synchronous circuit are denoted by $(\mathcal{B}^{n_i})^\omega$ and $(\mathcal{B}^{n_o})^\omega$, respectively.

Given two finite sets $\mathcal{S}$ and $\mathcal{T}$, a **sequence function** is a mapping **F**: $\mathcal{S}^\omega \rightarrow \mathcal{T}^\omega$. The mapping of a sequence $s \in \mathcal{S}^\omega$ by **F** is thus an element of $\mathcal{T}^\omega$, denoted by $\mathbf{F}(s)$. The n^{th} element of $\mathbf{F}(s)$ is an element of $\mathcal{T}$, denoted by $\mathbf{F}_{@n}(s)$. Two functions **F**, **G** are said to be **equal** if and only if $\mathbf{F}_{@n}(s) = \mathbf{G}_{@n}(s)\ \forall s \in (\mathcal{B}^{n_i})^\omega, \forall n \in \mathcal{Z}$.

Boolean operations can be defined on equi-dimensional sequence functions: for two functions **F**, **G**: $(\mathcal{B}^{n_i})^\omega \rightarrow (\mathcal{B}^{n_o})^\omega$, sum, product and complement are defined as the bitwise sum, product and complement, respectively:

$$\begin{aligned}(\mathbf{F}+\mathbf{G})(s) &= \mathbf{F}(s)+\mathbf{G}(s);\\ (\mathbf{FG})(s) &= \mathbf{F}(s)\mathbf{G}(s);\\ (\mathbf{F}')(s) &= (\mathbf{F}(s))'\end{aligned}$$

Also, $\mathbf{F} \geq \mathbf{G}$ if and only if $\mathbf{F}_{@n} \geq \mathbf{G}_{@n}\ \forall n \geq 0$.

The *retiming* (or time-shift) $\mathbf{F}_k$ of a function **F** by an integer k is defined by

$$(\mathbf{F}_k)_{@n}(s) = \mathbf{F}_{@(n-k)}(s)\ \ \forall n \in \mathcal{Z}\ . \tag{4.1}$$

In other words, $\mathbf{F}_k$ takes the values of **F** with a delay of k time units. The following properties of the retiming operation are self-evident:

$$\begin{aligned}&(\mathbf{F}_k)_h = \mathbf{F}_{h+k};\\ &(\mathbf{F}+\mathbf{G})_k = \mathbf{F}_k+\mathbf{G}_k;\ (\mathbf{FG})_k = \mathbf{F}_k\mathbf{G}_k;\ (\mathbf{F}')_k = (\mathbf{F}_k)'\ .\end{aligned} \tag{4.2}$$

4.1.3 Pattern expressions and functions.

Boolean expressions can be adapted to include integer time labels and represent sequence functions. These expressions are hereafter termed *pattern expressions*, and are defined as follows.

Definition 4.1 *The symbols* $0, 1$ *are pattern expressions, and denote the constant functions* $0, 1 : (\mathcal{B}^{n_i})^\omega \to \mathcal{B}^\omega$*, respectively. Given a set of* n_i *variables* $x, y, \ldots$*, a synchronous literal (or, shortly, a literal)* x *(*x'*) is a synchronous expression, and denotes a function* $x(x') : (\mathcal{B}^{n_i})^\omega \to \mathcal{B}^\omega$*. For every sequence* $s \in (\mathcal{B}^{n_i})^\omega$*, the value of a literal* x *(*x'*) at time* n*,* $x_{@n}(s)$ *(*$x'_{@n}(s)$*) coincides with the value of the variable* x *(the complement of* x*) in* s *at time* n*.*

Finite sums and finite products of pattern expressions are pattern expressions. They denote the function formed by the sum and product of their terms, respectively. The complement and retiming of a pattern expression are pattern expressions, whose value at every time point are the complement and retiming of their argument, respectively. Retiming of literals (e.g. x_k *) are in particular also termed literals.*

Definition 4.2 *A sequence function* **F** *is called a* **pattern function** *if it is expressible by means of a pattern expression.*

Example 33.

The following is a simple pattern expression:

$$a + ((a_1 b')_1 + c_2)_1$$

□

A synchronous expression is said to be *input-retimed* if only literals appear retimed. An expression can be reduced to its input-retimed form by applying repeatedly (4.2), until only literals appear retimed.

Example 34.

By applying (4.2) on the expression of Example (33), the new expression

$$a + a_3 b'_2 + c_3$$

is obtained. This expression contains only retimed literals. □

An input-retimed pattern expression contains literals in a finite interval $[m, M]$. The corresponding function takes value 1 at a time-point n only if its literals take a suitable pattern of values over the time interval $[n - M, n - m]$. Pattern functions can then be used to characterize patterns of events that occur over a finite time span.

Example 35.

For the expression of Example (34), $m = 0, M = 3$. The corresponding function takes value 1 at a time n if and only if a, b, c satisfy one of the patterns:

$$\begin{array}{lcllll} a_{@n}, a_{@n-1}, a_{@n-2}, a_{@n-3} & & 1,-,-,- & -,-,-,1 & -,-,-,- & \\ b_{@n}, b_{@n-1}, b_{@n-2}, b_{@n-3} & = & -,-,-,- \;; & -,-,0,- \;; & -,-,-,- \;; & \\ c_{@n}, c_{@n-1}, c_{@n-2}, c_{@n-3} & & -,-,-,- & -,-,-,- & -,-,-,1 & \end{array}$$

□

Not every sequence function is a pattern function. Consider, for example, a step function H, defined as follows. For each input sequence s, $H_{@n}(s) = 0$ if $n < 0$, and $H_{@n}(s) = 1$ for $n \geq 0$. Clearly, $H(s_k) = H(s)$ for all sequences s, while $H_k(s)$ is a step delayed by k and therefore differs from $H(s_k)$.

Intuitively, pattern functions can only be used to "detect" the occurrence of a finite-length pattern of input values. The position of the pattern along the time axis is not influent. Hence, pattern functions cannot be used to recognize situations that concern, for example, the value of n. Hence, they cannot express functions like the step function.

Representation of pattern functions

An input-retimed pattern expression can be regarded as an ordinary logic expression with literals x_i, x_i'. It is then possible to extend BDD representations of combinational functions to pattern functions in a straightforward manner.

Fig. (4.2) shows the BDD representation of the pattern function described by the expression $a + a_3 b_2' + c_3$.

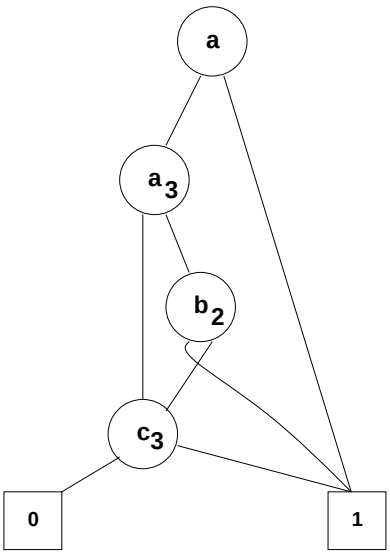


Figure 4.2: BDD representation of a pattern function.

Shannon expansion of pattern functions.

Consider an arbitrary sequence function **F**. The value it takes at some time-point n depends, in general, on the values taken by each input variable at each time-point $n' \neq n$. If we regard these latter values as independent Boolean variables, then $\mathbf{F}_{@n}$ can be regarded as a Boolean function having an infinite number of Boolean variables as support.

One could think of defining a cofactor of $\mathbf{F}_{@n}$ with respect to a variable x at some other time-point n', and then move on to construct a Shannon expansion of $\mathbf{F}_{@n}$ in this context. A similar expansion carried on $\mathbf{F}_{@n+1}$, however, may have no relationship whatsoever with the previous one. The retiming-invariance properties of pattern functions, instead, allow us to define cofactoring and extend Shannon expansion in a more useful way.

Definition 4.3 *The cofactor of a pattern function* **F** *with respect to* x *(*x'*), denoted by* $\mathbf{F}_x$ *(*$\mathbf{F}_{x'}$*), is the function obtained by replacing the values 1 and 0, respectively (0 and 1, respectively) to the literals* x, x' *of an* **input-retimed** *expression of* **F**.

Definitions of consensus, smoothing, and Boolean difference can then be extended to the present case. Eventually, it could then be verified that for a pattern function, the following expansion holds:

$$\mathbf{F} = x'\mathbf{F}_{x'} + x\mathbf{F}_x \tag{4.3}$$

The formal proof of Eq. (4.3) is simple but lengthy, it is therefore omitted in this work.

4.1.4 Functional modeling of synchronous circuits.

In order to extend Boolean optimization models to the synchronous case, it is desirable to first describe their behavior by means of a sequence function.

For an acyclic network N, an expression e^y of the function f^y realized by each vertex y (and, in particular, by the primary outputs) can be derived by iteratively substituting literals appearing in e^y with their corresponding pattern expressions, in a fashion entirely similar to that of combinational networks. The functionality of acyclic networks can then be described by a pattern function $\mathbf{F} : (\mathcal{B}^{n_i})^\omega \rightarrow (\mathcal{B}^{n_o})^\omega$.

The case of cyclic networks, however, is more complex. In particular, their behavior cannot, in general, be expressed by a pattern function in terms of the primary inputs. For this reason, the analysis of the impact of feedback is deferred to Chapter (6).

The initial value problem.

The synchronous network model presented so far implicitly regards D-type registers as unit-delay elements. This is incorrect at power-up, because the register content is essentially random and cannot be related to that of any other network variable. Power-up values, however, are often erased by the preset logic before they can affect the primary outputs. The error becomes irrelevant in this case. If the power-up value of a register is relevant for the circuit behavior, the model can be corrected by introducing fictitious input $Pwr.$ and $Ext.$ as shown in Fig. (4.3). To represent correctly the circuit behavior, input $Pwr.$ takes value 0 after power-up.

4.2 Sequential *don't cares*

In this chapter we consider extending the ideas of *don't care* -based and relational specifications to the case of sequence functions. Therefore, this chapter focuses on *don't care*

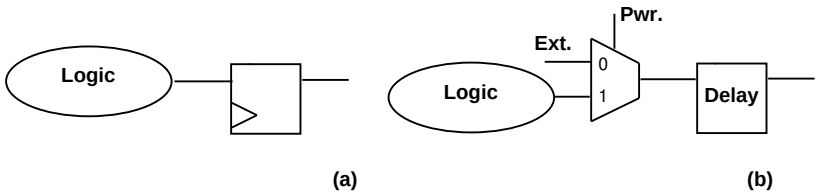


Figure 4.3: (a) A D-type flip-flop driven by logic; (b) equivalent circuit for the insertion of power -up values. $Ext.$ and $Pwr.$ are fictitious.

-based specifications and their representations.

The terminal behavior of a circuit is assumed to be specified in terms of a pair of functions **F**: $(\mathcal{B}^{n_i})^\omega \rightarrow (\mathcal{B}^{n_o})^\omega$ and **DC**: $(\mathcal{B}^{n_i})^\omega \rightarrow (\mathcal{B}^{n_o})^\omega$, or by means of the pair $\mathbf{F}_{min}$ =**F** **DC**′ and $\mathbf{F}_{max}$ =**F**+**DC**. A circuit, realizing a function **G**, satisfies the specifications if and only if

$$\mathbf{F}_{min} \leq \mathbf{G} \leq \mathbf{F}_{max} \ . \tag{4.4}$$

The optimization of a network N ultimately consists of its replacement by another network N' of lower cost, in terms of area or timing performance. The function **DC** takes again the meaning of a *tolerance* on the allowed functional error: N' can replace N if and only if its function **G** satisfies

$$\mathbf{F} \oplus \mathbf{G} \leq \mathbf{DC} \ . \tag{4.5}$$

Of course, in principle N' may differ widely from N, in terms of topology (for example, it may contain feedback) and functionality.

To limit the complexity of this analysis, only algorithms that optimize N and preserve its acyclic nature are considered here. As combinational networks represent a special case of synchronous networks, the algorithms developed should then represent extensions of those presented for combinational networks.

The general framework is again that of optimizing locally subnetworks of N by first identifying their *don't care* conditions and then resorting to known optimization algorithms. Perturbation analysis is then again instrumental in exploring the nature of such *don't care* conditions, and is developed in the next section.

Our objective here is to determine *don't care* conditions expressible by a pattern function, and on efficient algorithms for the extraction of such *don't care* sets. There

are two major motivations for this choice. First, pattern functions are strictly related to combinational functions, hence they provide compact, though incomplete, representations of sequential *don't cares* , and combinational optimization algorithms can easily be generalized. Second, Theorem (4.1) below indicates that more complex representations are less likely to be useful for the optimization of acyclic networks.

4.2.1 Retiming-invariant *don't care* conditions

In Eq. (4.5), no assumption is made on the nature of the function **DC**. Not every *don't care* condition, however, is potentially useful for the optimization of an acyclic network.

Intuitively, in order to be useful, a *don't care* condition must express conditions that are valid at every time-point . For example, the knowledge that the outputs are not observed only at a specific time-point is clearly not useful for optimizing a network.

This paragraph identifies the set of useful *don't care* conditions for acyclic networks.

Definition 4.4 *A sequence function* **K** *is termed* **retiming-invariant** *if and only if*

$$\mathbf{K}_n(s) = \mathbf{K}(s_n) \quad \forall n$$

Trivially, all pattern functions are retiming-invariant.

Definition 4.5 *We call* **retiming-invariant portion** $\mathbf{DC}^{ri}$ *of a don't care function* **DC** *the function defined as follows:*

$$\mathbf{DC}^{ri}(s) = \prod_{n=-\infty}^{+\infty} \mathbf{DC}_{-n}(s_n) \tag{4.6}$$

Trivially, $\mathbf{DC}^{ri}$ is a retiming-invariant function. Notice also that $\mathbf{DC}^{ri} \leq \mathbf{DC}$.

The following theorem shows that the retiming-invariant portion of a *don't care* specification is effectively its only "interesting" portion.

Theorem 4.1 *Given two retiming-invariant functions* $\mathbf{F}, \mathbf{G}$*, and a don't care function* **DC***, then*

$$\mathbf{F} \oplus \mathbf{G} \leq \mathbf{DC} \tag{4.7}$$

if and only if

$$\mathbf{F} \oplus \mathbf{G} \leq \mathbf{DC}^{ri} \tag{4.8}$$

Proof.

Trivially, Eq. (4.8) implies Eq. (4.7). Suppose now, by contradiction, that Eq. (4.7) does not imply Eq. (4.8). Then, there exist a sequence s and a time-point n^* such that

$$\mathbf{F}_{@n^*}(s) \oplus \mathbf{G}_{@n^*}(s) \leq \mathbf{DC}_{@n^*}(s); \tag{4.9}$$

and

$$\mathbf{F}_{@n^*}(s) \oplus \mathbf{G}_{@n^*}(s) \not\leq \mathbf{DC}^{ri}_{@n^*}(s) \tag{4.10}$$

From Definition (4.5), there must exist a retiming index n such that

$$\mathbf{F}_{@n^*}(s) \oplus \mathbf{G}_{@n^*}(s) \not\leq \mathbf{DC}_{_n,@n^*}(s_n) \tag{4.11}$$

Retiming by n both members of Eq. (4.11), and using the retiming-invariance properties of **F** and **G**, results in

$$\mathbf{F}_{@n^*}(s_n) \oplus \mathbf{G}_{@n^*}(s_n) \not\leq \mathbf{DC}_{@n^*}(s_n) \tag{4.12}$$

that is, Eq. (4.7) is violated at time-point n^* corresponding to the input sequence s_n. □

4.2.2 Controllability and observability *don't cares*

We borrow from the combinational case also the distinction between external **controllability** and **observability** *don't cares* . The motivation is analogous to the combinational case: in an acyclic interconnection of networks, the driving portion constrains the inputs of the network, while the driven portion limits its observability.

Example 36.

Consider the circuit of Fig.(4.4), representing the cascade interconnection of two simple synchronous networks. The limited controllability of the inputs of N_2 is described by the set of its impossible input sequences. For example, $u_1 v'$ represents a pattern of an impossible input sequence for N_2. For u

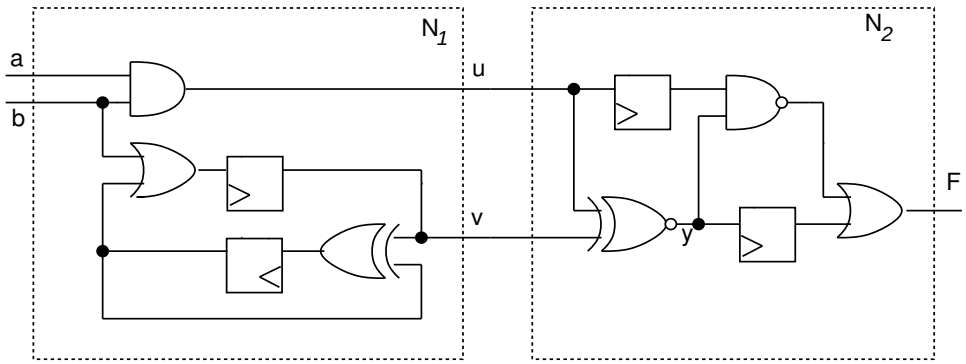


Figure 4.4: Cascaded synchronous circuits.

to be equal to 1 at a time-point $n-1$ it must be $a_{@n-1} = b_{@n-1} = 1$; consequently, $b_{@n-1} = 1$ and $v_{@n} = 1_{@n}$. Hence, for N_2, at no time-point n it can be $u_{@n-1}v'_{@n} = 1$.

A formal derivation of these impossible patterns will be presented when dealing with networks with feedback.

Let us now consider the situation at the outputs of N_1. The interconnection of the two networks limits the observability of the primary outputs of N_1. In particular, the output of N_2 can be expressed in terms of u and v as

$$F = y_1 + (yu_1)' = u_1v_1 + u'_1v'_1 + u'v + uv' + u'_1 = v_1 + u'v + uv' + u'_1 \ .$$

The variable u can affect the primary outputs after no delay or after 1 clock period. The conditions for which it can affect F with no delay can be expressed by the function

$$\left(\frac{\partial F}{\partial u}\right)' = v_1 + u'_1.$$

It cannot affect the outputs after one period when the function :

$$\left(\frac{\partial F}{\partial u_1}\right)'_{-1} = v + u'_{-1}v_{-1} + u_{-1}v'_{-1}$$

takes value 1. The variable u cannot affect the output **at any time** if

$$\left(\frac{\partial F}{\partial u}\right)' \cdot \left(\frac{\partial F}{\partial u_1}\right)'_{-1} = (v_1 + u'_1)(v + u'_{-1}v_{-1} + u_{-1}v'_{-1})$$

takes value 1. A similar reasoning can be carried out for the input v.

The external observability *don't cares* imposed on N_1 by the presence of N_2 are thus expressed in general by a vector of functions. Each function represents the *don't care* conditions on one output. □

The distinction between controllability and observability *don't cares* is of course blurred in presence of global feedback interconnections.

Controllability *don't cares*

In the previous example, impossible input sequences were characterized as having a value $u = 1$ at some time-point $n - 1$ and the value $v = 0$ at n. All such sequences were described by means of a pattern function $u_1 v'$. It is worth recalling that pattern functions are functions from **sequences** to **sequences**, hence $u_1 v'$ is not exactly the characteristic function of a **set** of sequences. The sense in which a pattern function can be used, however, to characterize the set of impossible sequences is defined next.

We restrict our attention to impossible patterns of values in a time span of a predefined, but otherwise arbitrary, length $L \geq 0$. A sequence $s \in (\mathcal{B}^{n_i})^\omega$ is a **controllability** *don't care* **sequence** if it contains one such pattern. We denote the set of these sequences by $\mathcal{C}$, and define a function $C : (\mathcal{B}^{n_i})^\omega \rightarrow \mathcal{B}^\omega$, taking value 1 corresponding to the sequences in $\mathcal{C}$, and value 0 otherwise.

We use a pattern function $CDC : (\mathcal{B}^{n_i})^\omega \rightarrow \mathcal{B}^\omega$ to represent the impossible patterns. The literals of CDC are assumed to have time-stamps in the reference interval $[0, L]$. In particular, $CDC_{@n}(s) = 1_{@n}$ if and only if s contains an impossible pattern in the interval $[n - L, n]$, and $CDC_{@n}(s) = 0_{@n}$ otherwise.

Example 37.

For the network of Fig. (4.4), the input controllability *don't cares* of N_2 can be represented by the functions $CDC = 0$, $CDC = u_1 v'$, or by $CDC = u_1 v' + u_2 v_1'$, depending on the time-span chosen. □

The following Lemma clarifies further the link between the functions CDC and C :

Lemma 4.1

$$C = \sum_{n=-\infty}^{+\infty} CDC_n \tag{4.13}$$

Proof.

If a sequence s belongs to $\mathcal{C}$, then there exists a time-point n such that $CDC_{@n}(s) = 1_{@n}$. Hence, for each time-point n', it is possible to find a retiming index $k = n' - n$ such that $CDC_{k,@n'}(s) = 1_{@n'}$, and the left-hand side of Eq. (4.13) takes value 1.

If s does not belong to $\mathcal{C}$, then $CDC(s) = 0$ and then $CDC_n(s) = 0$, for every n, and Eq. (4.13) holds. □

Notice in particular that

$$C_k = C \quad \forall k \in \mathcal{Z} \tag{4.14}$$

Representing Observability *don't cares*

We represent also observability *don't cares* by means of a pattern function $\mathbf{ODC}^{ext}$: $(\mathcal{B}^{n_i})^\omega \rightarrow (\mathcal{B}^{n_o})^\omega$, with literals in a generic time span $[0, L]$. Corresponding to each input sequence s, the vector $\mathbf{ODC}^{ext}_{@n}(s)$ takes value 1 corresponding to those components of the output that are not observed **at time** n. It is thus assumed that an output is not observed at a given time-point n corresponding to the occurrence of some particular patterns in the interval $[n - L, n]$.

We thus eventually assume that for a network

$$\mathbf{DC} = C\mathbf{1} + \mathbf{ODC}^{ext} \ . \tag{4.15}$$

It is worth remarking that consequently the function CDC is used only to represent C by means of a pattern function.

In the rest of this dissertation, we use the symbols **C** and **CDC** to denote the functions $C\mathbf{1}$ and $CDC\mathbf{1}$, respectively.

4.3 Local optimization of acyclic networks

One major result of multilevel combinational synthesis is that the degrees of freedom for the local optimization of a gate embedded in a network are expressed by a *don't care* function.

The first question is then whether this result can be extended to the sequential case, that is, whether the conditions for replacing a pattern function f^y with a different one g^y can be expressed by an equation like

$$f^y \oplus g^y \leq DC^y \tag{4.16}$$

Example (38) below shows that this is not the case.

Example 38.

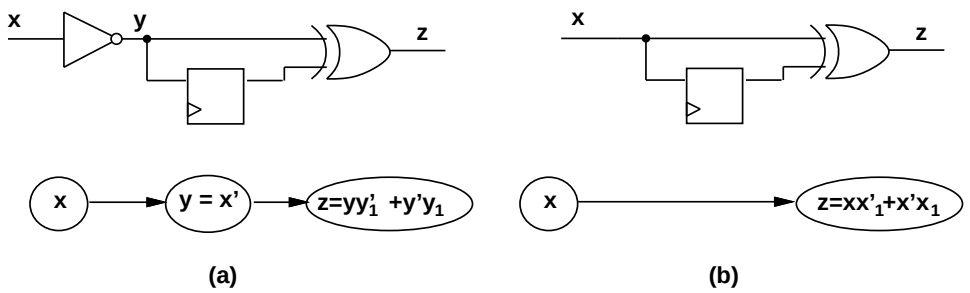


Figure 4.5: The simplification of the inverter in a simple network.

The simple circuit of Fig. (4.5) realizes the function $\mathbf{F} = x' \oplus x'_1$. It can easily be recognized that the inverter can be replaced by a simple connection, *i.e.* $f^y = x'$ can be replaced by $g^y = x$. In this case, $f^y \oplus g^y = 1$: a constant error 1 is therefore introduced at the inverter output. Had an equation like Eq. (4.16) been applicable, then we should conclude that $DC^y = 1$, *i.e.* that the inverter could also be replaced by a constant 0 or 1, which is clearly false . □

Degrees of freedom expressed in the form of Eq. (4.16) are nevertheless the only ones that can currently be handled by logic optimization engines such as ESPRESSO: the

rest of this chapter is therefore devoted to finding *sufficient* bounds in the form of Eq. (4.16).

The modification of a local function f^y in an acyclic network can again be modeled by introducing a perturbation input δ, with $\mathbf{F}^y$ denoting the function realized by the perturbed network N^y.

The function $\mathbf{F}^y$ depends in particular on $\delta, \delta_1, \cdots, \delta_{P_y}$, where P_y denotes the longest path from the vertex y to the primary outputs, in terms of registers.

The error in functionality caused by the presence of δ is expressed by the error function

$$\mathbf{E} \stackrel{\text{def}}{=} \mathbf{F}^y \ \oplus \ \mathbf{F}^y_{\delta', \cdots, \delta'_{P_y}} \ . \tag{4.17}$$

The equation

$$\mathbf{E} \leq \mathbf{DC} \tag{4.18}$$

represents implicitly the tolerance on such errors, and provides the functional constraints on δ.

By construction, $\mathbf{E}$ is a pattern function, expressible in terms of $\delta, \delta_1, \ldots, \delta_{P_y}$ and primary input variables.

Eq. (4.18) represents the functional constraints on δ, for the perturbation signals to be acceptable. In the rest of this section, we try to explicit this constraint into an upper bound on δ.

4.3.1 Internal observability *don't care* conditions.

Consider adding a perturbation input to an acyclic network. The value of the primary outputs at some time-point n may be affected by the values taken by the perturbation at time-point $n, n-1, \cdots, n-P$. The impact of the values $\delta_{@n-P}, \delta_{@n-P+1}, \cdots, \delta_{@n}$ on the primary output at time n is, in this respect, similar to that of a multiple perturbation in a combinational network.

Unlike the case of combinational networks, however, we need to take into account also the fact that the value of $\delta_{@n}$ may affect the network outputs for more clock cycles, namely, at $n+1, \cdots, n+P$ as well.

In order to simplify the analysis, we first consider the case where all paths from y to the primary outputs have the same length, in terms of registers. A perturbation can thus only affect the outputs at one time-point in the future. As intuition suggests, this case is very similar to the combinational case. Later, we tackle the general case.

The methods developed to this purpose hinge upon the definition of observability *don't cares* for acyclic networks:

Definition 4.6 *The observability don't care function of y_k is the function*

$$\mathbf{ODC}^{y_k} = \mathbf{F}^{y}_{\delta'_k} \overline{\oplus} \mathbf{F}^{y}_{\delta_k} \tag{4.19}$$

Intuitively, the function $\mathbf{ODC}^{y_k}$ describes when the output at time n is (not) affected by a perturbation at time-point $n - k$.

Example 39.

Consider the circuit of Fig. (4.6). There are two paths from the multiplexer output to the primary output, of length 0 and 1, respectively.

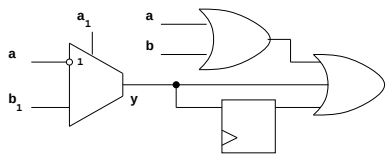


Figure 4.6: A non-pipeline acyclic synchronous network.

The two observability *don't cares* are thus

$$ODC^{y_0} = a + b + y_1 = a + b + (a_2 a'_1 + a'_2 b_2) \oplus \delta_1$$

and

$$ODC^{y_1} = a + b + y = a + b + (a_1 a' + a'_1 b_1) \oplus \delta$$

Notice in particular that each observability *don't care* depends on the perturbation signal δ. □

Let us now consider the special case where all paths from a vertex y to the primary outputs contain the same number P_y of registers. In this case, $\mathbf{F}^y$ and $\mathbf{E}$ depend only on δ_{P_y}, and Eq. (4.18) can still be explicited into a tolerance on δ. The derivation is as follows. A Shannon expansion of $\mathbf{E}$ results in

$$\delta'_{P_y}\mathbf{E}_{\delta'_{P_y}} + \delta_{P_y}\mathbf{E}_{\delta_{P_y}} \leq \mathbf{DC} \ . \tag{4.20}$$

On the other hand, from Eq. (4.17), $\mathbf{E}_{\delta'_{P_y}} = \mathbf{0}$ and therefore Eq. (4.20) reduces to

$$\delta_{P_y}\mathbf{E}_{\delta_{P_y}} \leq \mathbf{DC} \ . \tag{4.21}$$

or, equivalently,

$$\delta_{P_y}\mathbf{1} \leq \mathbf{E}'_{\delta_{P_y}} + \mathbf{DC} \ . \tag{4.22}$$

Eq. (4.22) is very similar to the combinational case, as it explicits the tolerance on δ again as a sum of an external (**DC**) plus an internal ($\mathbf{E}'_{\delta_{P_y}}$) component. By observing that $\mathbf{E}'_{\delta_{P_y}} = \mathbf{ODC}^{y_{P_y}}$, the retiming of Eq. (4.22) by $-P_y$ yields:

$$\delta\mathbf{1} \leq \mathbf{DC}_{-P_y} + \mathbf{ODC}^{y} \ . \tag{4.23}$$

By using the defining Equation (4.15) of **DC** and the property (4.14) of **C**, Eq. (4.23) becomes

$$\delta\mathbf{1} \leq \mathbf{C} + \mathbf{ODC}^{ext}_{-P_y} + \mathbf{ODC}^{y} \tag{4.24}$$

A network is said to be a *pipeline* if for each vertex all paths to a primary output have the same length. Eq. (4.23) shows that *don't care* sets fully describe the degrees of freedom for the optimization of these networks.

Retiming/resynthesis techniques exploit some of these *don't cares* , by identifying pipelined subnetworks and optimizing them. It is worth noting that the identification of pipelines in [22] is **topological.** It might well be the case where a gate, although having paths of different length to a primary output, results in a function $\mathbf{E}$ that depends only on **a single** δ_k. Retiming would miss the opportunity of optimizing this gate by ordinary *don't care* -based methods.

For non-pipelined networks, Example (39) has shown that an equation like Eq. (4.23) cannot fully represent all *don't cares* . This is due to the dependency of $\mathbf{E}$ from multiple, retimed instances of δ. This dependency, however, also suggests the use of the approximation methods developed for multiple perturbations.

Theorems (4.1)- (4.2) below extend the results of Theorems (2.3)-(2.4) on multiple perturbations in combinational networks. Their proofs are very similar to those of Theorems (2.3)-(2.4), and are therefore omitted here.

Theorem 4.2 *A perturbation δ satisfies Eq. (4.18) if and only if*

$$\begin{aligned}
\mathbf{DC}'\mathbf{E}_{\delta'} &\leq \delta\mathbf{1} \leq \mathbf{E}'_{\delta} + \mathbf{DC}\ ; \\
\mathbf{DC}'(\forall_{\delta}\mathbf{E})_{\delta'_1} &\leq \delta_1\mathbf{1} \leq (\forall_{\delta}\mathbf{E})'_{\delta_1} + \mathbf{DC}\ ; \\
&\vdots \\
\mathbf{DC}'(\forall_{\delta,\delta_1,\ldots,\delta_{i-1}}\mathbf{E})_{\delta_i} &\leq \delta_i\mathbf{1} \leq (\forall_{\delta,\delta_1,\ldots,\delta_{i-1}}\mathbf{E})'_{\delta'_i} + \mathbf{DC}\ ;\quad i = 0,\ldots,P_y
\end{aligned} \tag{4.25}$$

Theorem 4.3 *If perturbations $\delta, \delta_1, \ldots, \delta_{P_y}$ satisfy :*

$$\delta_i\mathbf{1} \leq \mathbf{DC} + \mathbf{ODC}^{y_i}_{\delta',\ldots,\delta'_{i-1}} \tag{4.26}$$

then Eq. (4.18) (namely, $\mathbf{E} \leq \mathbf{DC}$) holds.

Example 40.

Consider the circuit of Fig. (4.6). For the optimization of the MUX gate, we determine

$$ODC^{y_0} = a + b + y_1 = a + b + (a_2a'_1 + a'_2b_2) \oplus \delta_1$$

and

$$ODC^{y_1}_{\delta} = a + b + y_{\delta'} = a + b + a_1a' + a'_1b_1$$

From Theorem (4.2), if the modification of the MUX results in a perturbation δ such that

$$\begin{aligned}
\delta &\leq ODC^{y_0} \\
\delta_1 &\leq ODC^{y_1}_{\delta'}
\end{aligned} \tag{4.27}$$

then the modification is acceptable. Later in this section we show how to use this information for the optimization of the MUX gate. □

In Eq. (4.26), the bound on each δ_i depends in general on $\delta_j, j > i$. This is also evident in Example (40). The method for eliminating this dependency consists again of finding, for each δ_i, a portion DC^{y_i} of $\mathbf{ODC}^{y_i}$ that is common to all components of the vector $\mathbf{ODC}^{y_i}$ and independent from $\delta_j, j > i$. When finding DC^{y_i} it is useful to include the information that δ_j is bounded by a *don't care* function:

Theorem 4.4 *If each retimed $\delta_i; i = 0, \cdots, P_y$ is contained in $C + DC^{y_i}$ where each DC^{y_i} satisfies*

$$DC^{y_i}\mathbf{1} \leq \mathbf{CODC}^{y_i}; i = 0, \ldots, P_y \tag{4.28}$$

and

$$\begin{aligned}
\mathbf{CODC}^{y_{P_y}} &= \mathbf{ODC} + \mathbf{ODC}^{y_{P_y}}{}_{\delta', \ldots, \delta'_{P_y - 1}} \\
&\vdots \quad \vdots \quad \vdots \\
\mathbf{CODC}^{y_i} &= \mathbf{ODC} + \forall_{\delta_{i+1}, \ldots, \delta_{P_y}} \left(\mathbf{ODC}^{y_i}_{\delta'_1, \ldots, \delta'_{i-1}} + \left(\sum_{k=i+1}^{P_y} \delta_k (DC^{y_k})' \right) \mathbf{1} \right); i = 0, \ldots, P_y \ .
\end{aligned} \tag{4.29}$$

then $\mathbf{E} \leq \mathbf{DC}$.

Example 41.

Consider extracting the *don't care* for the MUX gate of Fig. (4.6). From Example (40),

$$DC^{y_1} = CODC^{y_1} = ODC^{y_1}_{\delta'} = a + b + a_1 + b_1$$

(the expression of $ODC^{y_1}_{\delta'}$ has been simplified) while

$$\begin{aligned}
&DC^{y_0} = CODC^{y_0} = \forall_{\delta_1} \left(ODC^{y_0} + \delta_1 (DC^{y_1})' \right) = \\
&a + b + a_1 a_2 + a_1 b'_2 + b_1 a'_2 b'_2
\end{aligned} \tag{4.30}$$

□

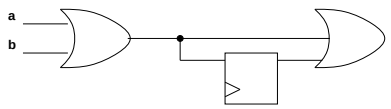


Figure 4.7: Optimized version of the MUX gate circuit.

A unique bound on δ is eventually obtained by retiming and intersecting the individual bounds given on each δ_i:

$$DC^y = C + \prod_{i=0}^{P_y} (DC^{y_i})_{-i} \ . \tag{4.31}$$

Example 42.

For the MUX gate of Fig. (4.6),

$$\begin{aligned} DC^y = DC^{y_0} \cdot (DC^{y_1}_{-1}) = \\ (a + b + a_1 a_2 + a_1 b'_2 + b_1 a'_2 b'_2)(a_{-1} + b_{-1} + a + b) \\ = a + b + (a_{-1} + b_{-1})(a_1 a_2 + a_1 b'_2 + b_1 a'_2 b'_2) \ . \end{aligned} \tag{4.32}$$

The *don't care* just extracted can be used for the optimization of the gate. An optimal expression is $y = a_1 + b_1$.

It is also worth noting that in this case y is just a retimed version of $a + b$. The final, optimized circuit is shown in Fig. (4.7). □

We conclude this section by observing that the controllability *don't care* term C can be replaced in Eq. (4.31) by the expression CDC.

Theorems (4.3) -(4.4) therefore suggest the following procedure for extracting the observability *don't care* of a gate in a synchronous, acyclic network:

- Extract $\mathbf{ODC}^{y_i}_{\delta', \cdots, \delta'_{i-1}}$ for $i = 0, \cdots, P_y$;
- Compute the function DC^y using Eq. (4.31), and using **ODC** as external *don't care* condition;
- Retime and intersect each DC^{y_i}, and add CDC to the result.

Of the three steps, the first one is the most complex. Its description is thus carried next.

4.4 Computation of observability *don't cares* in acyclic networks

Theorem (4.4) relates a *don't care* function for a network variable y to the individual *don't cares* $\mathbf{ODC}^{y_i}$. Local methods for computing such functions by a single sweep of the network can be extended from those of combinational networks as follows.

Consider first the case of a vertex labeled y, with a single fanout edge (y, z) with weight w. If $\mathbf{ODC}^{z_k}$ is known for $k = 0, \ldots, P_z$, then the corresponding expression for y is given by

$$\mathbf{ODC}^{y_{k+w}} = \mathbf{ODC}^{z_k} + \left(\frac{\partial e_k^z}{\partial y_{k+w}}\right)' \mathbf{1}; \quad k = 0, \ldots, P_z \tag{4.33}$$

and $\mathbf{ODC}^{y_0} = \cdots \mathbf{ODC}^{y_{w-1}} = \mathbf{1}$.

Eq. (4.33) is essentially identical to Eq. (2.20) of the combinational case, the only difference consisting in accounting for the delay w by appropriately retiming e^z. The same considerations about the validity of Eq. (2.20) in presence of multiple perturbations hold of course also for Eq. (4.33).

It is similarly possible to extend to the synchronous case Eq. (2.27) for gates with reconvergent fanout, by carrying out the same construction as in Chapter (2). Let v and z denote the fanout variables of a two-fanout vertex y. The function $\mathbf{F}^{v,z}$ describes the function of the perturbed network with two perturbations δ^v, δ^z. It follows that

$$\mathbf{ODC}^{y_k} = \mathbf{F}^{v,z}_{\delta_k^v, \delta_k^z} \overline{\oplus} \mathbf{F}^{v,z}_{\delta_k^{v\prime}, \delta_k^{z\prime}}. \tag{4.34}$$

By adding twice the term $\mathbf{F}^{v,z}_{\delta_k^{v\prime}, \delta_k^z}$, by manipulations similar to those leading to Eq. (2.27) we obtain

$$\mathbf{ODC}^{y_k} = \mathbf{ODC}^{v_k} \overline{\oplus} \mathbf{ODC}^{z_k}_{\delta_k^v} . \tag{4.35}$$

The dual expression is obtained by adding twice $\mathbf{F}^{v,z}_{\delta_k^v, \delta_k^{z\prime}}$:

$$\mathbf{ODC}^{y_k} = \mathbf{ODC}^{v_k}_{\delta_k^z} \overline{\oplus} \mathbf{ODC}^{z_k} . \tag{4.36}$$

The following pseudocode illustrates a one-pass optimization of a network, evidencing the computation of the observability *don't cares* .

```
OBSERVABILITY(G, CDC, ODC);
T = {sink};
S = FI(sink);
while (T < V ) {

   select(v∈V - T such that FO(v)⊆S);
      if (vertex_type(v) == gate) {
        foreach y∈ FI(v) {
          for (j=0, j < P, j++) {
            ODC[y][j] =(∂f^μ_{w_y}/∂y_0)'1 +
                    retime(w(y,v),ODC[v][j-w(y,v)]);
          }
        }
      } else {
        y = FI(v);
        ODC^y = 1;
        for(j=0, j < P, j+) +
          for (z=fanout_var(y); z != NULL; z = z->next_fanout) {
            ODC[y][j] = ODC[y][j] ⊕̄ ODC[z][j]_{u=y';u>z}
          }
        }
      }
      DC = compute_dontcare(y, CDC, ODC);
      optimize(y, DC);
      S = S∪FI(v);
      T = T∪{v};

}
```

The routine `compute_dontcare` performs the steps 2) and 3) of the computation of DC^y. The routine `optimize` does the actual two-level optimization.

Circuit	inputs	outputs	literals	registers	feedback	longest path P
S208	11	2	166	8	8	1
S298	3	6	244	14	14	2
S344	9	11	269	15	15	1
S444	3	6	352	21	15	2
S526	3	6	445	21	21	1
S641	35	24	537	19	11	2
S820	18	19	757	5	5	3
S832	18	19	769	5	5	3
S1196	14	14	1009	18	0	3
S1238	14	14	1041	18	0	3
S1494	8	19	1393	6	6	1
S9234.1	36	39	7900	211	90	6

Table 4.1: Benchmark statistics

4.5 Experimental results.

We report in this section experimental results concerning the extraction of observability *don't care* sets for benchmark synchronous circuits. Table (4.5) shows the statistics of the benchmark circuits. Most of these circuits contain feedback paths. They were removed using the algorithm by Smith and Walford [45] for the minimum feedback vertex set.

Column `feedback` indicates the number of feedback variables that were introduced. P indicates the longest path of the circuit in terms of register counts. These parameters are obviously affected by choice of the feedback variables: for example, for the benchmark s344, a cut based on a depth-first network traversal [46] resulted in $P = 10$.

Table (4.2) reports the results for the computation of observability *don't care* functions, in terms of CPU time and peak BDD nodes.

Circuit	CPU time	BDD nodes
S208	0.9	1280
S298	2.1	889
S344	2.1	2015
S444	4.1	4547
S526	5.6	3289
S641	10.1	2645
S820	19.0	11788
S832	18	6679
S1196	23.4	305622
S1238	17.9	398591
S1494	19.6	12623
S9234.1	151.4	456017

Table 4.2: Computation of the observability *don't care* sets.

4.6 Summary

In this chapter we took a first step towards the construction of algorithms for the structural optimization of synchronous hardware. In particular, Sections (4.1) and (4.2) were concerned with the definition of functional specifications and *don't cares* at for sequential circuits. In Section (4.3) we presented an extension of the perturbation theory applied to combinational circuits. Based on this theory, we developed an algorithm for extracting *don't care* functions that can be used in the optimization of individual gates in a synchronous networks. Unlike the combinational case, however, *don't care* functions express only partially the potential for optimization in a synchronous circuit. Moreover, the algorithms are suited for networks without feedback loops. These two limitations are removed in the next two chapters.

Chapter 5

Recurrence Equations

In the previous chapter we outlined an approximate technique for optimizing synchronous circuits, in particular by optimizing each vertex of the network using *don't care sets*.

The inverter example showed that, unlike the combinational case, even for simple acyclic networks this approach is not sufficient to characterize all the *don't care* conditions that can arise in the synchronous context.

In this chapter we focus on exact methods for acyclic networks. We show that the problem of optimizing a subnetwork can be cast as that of finding the minimum-cost solution to a new type of Boolean equation, called *synchronous recurrence equation*.

Besides acyclic networks, the method is useful in the optimization of the acyclic portion of general synchronous networks containing feedback.

We propose a solution algorithm for recurrence equations. The algorithm relies on the transformation of the equation into a new combinational logic optimization problem. An exact solution algorithm for this latter problem is presented, and experimental results on synchronous benchmark circuits demonstrate the feasibility of the approach.

5.1 Introduction

We introduce the idea of recurrence equations by resorting again to the inverter example, repeated in Fig. (5.1).

Example 43.

The functionality of the original circuit (shown in Fig. (5.1.a)) is expressed by the function $F = x' \oplus x'_1$. Consider another circuit, in which the inverter is replaced by another (yet unknown) logic block, with input x and output y as shown in Fig. (5.1). The functionality of this circuit can be expressed in terms of the internal signal y: $F^y = y \oplus y_1$.

The block is an acceptable replacement of the inverter as long as the global functionality of the circuit is unchanged, that is, as long as the signal y satisfies the equation:

$$y \oplus y_1 = x' \oplus x'_1 \ .$$

This equation can be interpreted as the functional constraint placed on the signal y (and hence on the block replacing the inverter) for the terminal behavior of the global circuit to result unchanged.

It is worth noting that there are several solutions to this equation. Some such solutions are, for instance, $y = x$, $y = x'$ (the inverter), $y = x \oplus x_1 \oplus y_1$. Each solution corresponds to a possible logic block replacing the inverter. These are shown in Fig. (5.1.b). □

Example 44.

As a more complex example, consider the optimization of the subnetwork $N1$ in the network of Fig. (5.2). The desired terminal behavior of the entire network can be described by

$$F = b_2 b_1 (a_1 + b)$$

Its output is expressed in terms of the internal signal y by:

$$F^y = b_1 (b + a_1 + y_1)(y \overline{\oplus} y_1)$$

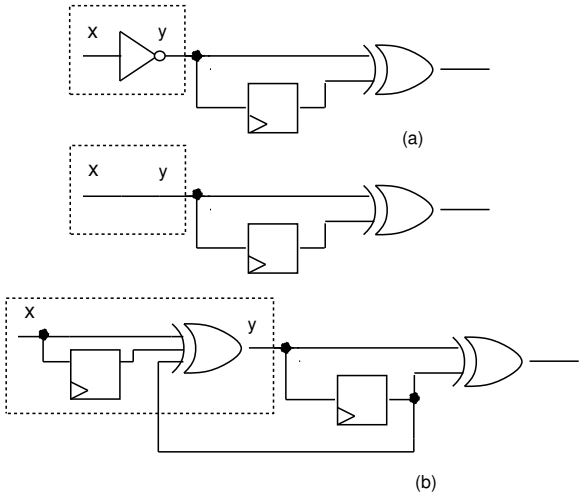


Figure 5.1: Original and optimized versions of the inverter circuit.

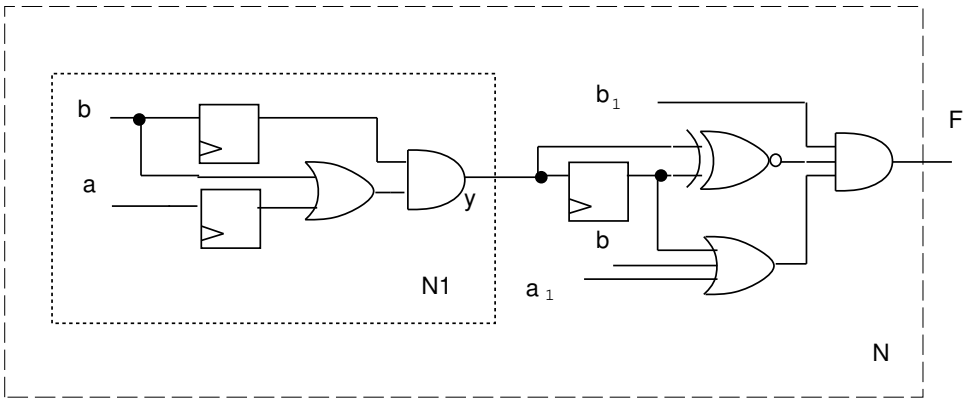


Figure 5.2: Local optimization of $N1$ embedded in a larger network N

For every input sequence, y must then satisfy

$$b_2 b_1 (a_1 + b) = b_1 (y \overline{\oplus} y_1)(b + a_1 + y_1)$$

Assume now that the input pattern $b b_1 b_2'$ is impossible at the network inputs. The above equality must be satisfied only for those input sequences not containing the impossible pattern, hence the constraint on y becomes:

$$b_2 b_1 (a_1 + b)(b b_1 b_2')' \leq b_1 (y \overline{\oplus} y_1)(b + a_1 + y_1) \leq b_2 b_1 (a_1 + b) + b b_1 b_2'$$

□

5.2 Synchronous Recurrence Equations

Definition 5.1 *We call* **Synchronous Recurrence Equation** *(SRE) a Boolean equation of type*

$$\mathbf{F}(\mathbf{x}, \mathbf{x}_1, \cdots, \mathbf{x}_d, \mathbf{y}, \mathbf{y}_1, \cdots, \mathbf{y}_d) = \mathbf{G}(\mathbf{x}, \mathbf{x}_1, \cdots, \mathbf{x}_d) \tag{5.1}$$

where $\mathbf{F}, \mathbf{G}$ *are ordinary pattern functions. The (nonnegative) integer* d *is termed the* **memory depth** *of the equation.*

We call a **feasible solution** *of the SRE a pattern function*

$$\mathbf{f}(\mathbf{x}, \cdots, \mathbf{x}_d, \mathbf{y}_1, \cdots, \mathbf{y}_d) \tag{5.2}$$

and an initial value specification

$$\begin{aligned} \mathbf{y}_{@-d} &= \mathbf{g}_{@-d}(\mathbf{x}_{@-d}, \cdots, \mathbf{x}_{-2d}) \\ &\vdots \\ \mathbf{y}_{@-1} &= \mathbf{g}_{@-1}(\mathbf{x}_{@-1}, \cdots, \mathbf{x}_{@-d+1}) \end{aligned} \tag{5.3}$$

such that if

$$\mathbf{y}_{@n} = \mathbf{f}_{@n}(\mathbf{x}, \cdots, \mathbf{x}_d, \mathbf{y}_1, \cdots, \mathbf{y}_d) \quad \forall n \geq 0 \tag{5.4}$$

then Eq.(5.1) holds true.

The equation resulting from the inverter optimization problem of Example (43) is thus precisely an SRE, where $F = y \oplus y_1$ and $G = x' \oplus x'_1$.

The presence of *don't care* conditions in Example (44), however, does not allow us to cast the optimization problem as an SRE, in the way it is defined above. We extend the definition by replacing Eq. (5.1) with the pair of inequalities:

$$\mathbf{F}_{min}(\mathbf{x}, \mathbf{x}_1, \cdots, \mathbf{x}_d) \leq \mathbf{F}(\mathbf{x}, \mathbf{x}_1, \cdots, \mathbf{x}_d, \mathbf{y}, \mathbf{y}_1, \cdots, \mathbf{y}_d) \leq \mathbf{F}_{max}(\mathbf{x}, \mathbf{x}_1, \cdots, \mathbf{x}_d) \tag{5.5}$$

5.2.1 Optimization of synchronous circuits by recurrence equations

The previous section showed that the degrees of freedom associated with a gate in an acyclic network can be expressed by a recurrence equation in terms of the primary input

variables. This equation can be obtained easily if representations of the functions **F** and $\mathbf{F}^{\mathbf{y}}$ are available. In order to apply recurrence equations in a sequential logic optimization engine, the following problems must be solved:

1) How to find a minimum-cost solution to a recurrence equation;

2) How to keep into account the presence of internal signals during the synthesis step.

The following sections address these two problems in order.

A synchronous network realizing a function as in Eq. (5.2) may in general contain feedback interconnections, as **y** is expressed in terms of the past values $\mathbf{y}_1, \cdots, \mathbf{y}_d$. In this work we focus our attention on simpler solutions, in the form $\mathbf{f}(\mathbf{x}, \cdots, \mathbf{x}_d)$ only. Otherwise, the optimization of a vertex may introduce feedback and methods other than recurrence equation would be needed for its optimization.

Such solutions are hereafter called **acyclic**. It is worth noting that acyclic solutions need no initial value specifications.

5.3 Finding acyclic solutions.

Our solution procedure is essentially divided in two steps. The first step consists of transforming the synchronous synthesis problem into a combinational one, by providing a characterization of the acyclic solutions to an SRE.

We recall that Eq. (5.1) represents a **functional equation**, *i.e.* an equation whose unknown is the function $\mathbf{f}(\mathbf{x}, \cdots, \mathbf{x}_d)$. In turn, **f** is completely described by its truth table; the truth table entries of the function **f** then represent the actual unknowns of the problem. The preliminary step consists of determining a representation of the truth tables corresponding to feasible solutions. The second step consists of the search procedure for optimum solutions. We focus in particular on minimum two-level representations of **f**.

5.3.1 Representing feasible solutions

For the sake of simplicity, we limit our attention to the synthesis of a single-output function f, the generalization to the multiple-output case being conceptually straightforward, but computationally more complex.

The support of f is formed by the $n_i \times d$ variables representing the components of the vectors $\mathbf{x}, \cdots, \mathbf{x}_d$. Any such function can be represented by its truth table, of $2^{n_i \times d}$ entries. These entries are here denoted by $f_j; j = 0, \cdots, 2^{n_i \times d} - 1$ [1]

A function f is completely specified once all f_j's have been assigned a value. At the beginning of the solution process, none of the f_j are known, and there are in general several possible assignments, corresponding to feasible solutions of different cost.

Example 45.

For the problem of Example (43), we seek a function $f(x, x_1)$ of minimum cost that can replace the inverter. The function is entirely described by its truth table, represented in Table (5.1). The entries f_0, f_1, f_2, f_3 represent the unknowns of the problem. Definite feasible solutions are represented by $f_0 = 1, f_1 = 1, f_2 = 0, f_3 = 0$ (corresponding to the original inverter) and by $f_0 = 0, f_1 = 0, f_2 = 1, f_3 = 1$ (corresponding to the simple interconnection). □

For each assignment of $\mathbf{x}, \cdots, \mathbf{x}_d$, Eq. (5.1) specifies a constraint on the possible assignments to $y, \cdots, y_d$. Such constraints can be expressed by means of a **relation table** associated with the SRE. The left-hand side of the table represents the assignments of the inputs $\mathbf{x}, \cdots, \mathbf{x}_d$, while its right-hand side represents the corresponding assignments to $y, \cdots, y_d$ that satisfy the SRE.

x	x_1	f
0	0	f_0
0	1	f_1
1	0	f_2
1	1	f_3

Table 5.1: Symbolic tabular representation of an unknown function $f(x, x_1)$.

Example 46.

The SRE associated with the problem of Example (43) is

$$x' \oplus x'_1 \leq y \oplus y_1 \leq x' \oplus x'_1$$

[1] In this case, the subscript denotes the entry number and is not a time stamp.

Corresponding to the assignment, say, $(x, x_1) = (0, 1)$, the SRE reduces to the constraint $1 \leq y \oplus y_1 \leq 1$, that is, $y \oplus y_1 = 1$.

Table (5.2) contains the relation table for this SRE. The second column shows in particular the assignments of y, y_1 that satisfy the SRE, corresponding to each assignment of x, x_1. The relation table for the problem of Example (44) is shown in Table (5.3). □

x	x_1	x_2	y y_1
0	0	–	00, 11
0	1	–	01, 10
1	0	–	01, 10
1	1	–	00, 11

Table 5.2: Relation table for the inverter optimization problem.

a	b	a_1	b_1	a_2	b_2	y y_1
-	-	-	0	-	-	–
-	0	0	1	-	0	0–, –0
-	0	1	1	-	0	01, 10
-	-	1	1	-	1	00, 11
-	1	-	1	-	1	00, 11

Table 5.3: Relation table for the problem of Example (44). Dashes represent *don't care* conditions.

Recall that we are seeking solutions in the form $y = f(\mathbf{x}, \cdots, \mathbf{x}_d)$. Corresponding to each entry of the relation table, we can re-express the right-hand side constraints on $y, \cdots, y_d$ as constraints on the f_j's, as shown by the following example.

Example 47.

For the relation table of Tab.(5.2), corresponding to the assignment (x, x_1, x_2) = $(0, 0, 1)$, the possible assignments for (y, y_1) are either $(0, 0)$ or $(1, 1)$. Since we assume $y = f(x, x_1)$ and $y_1 = f(x_1, x_2)$, we have $y = f(0, 0) = f_0$

and $y_1 = f(0,1) = f_1$. Therefore, the possible assignments for y_1, y are also described by

$$(f_0 + f_1')(f_0' + f_1) = 1 \tag{5.6}$$

The same process is repeated for all rows of the relation table. The resulting constraints on the entry variables f_j are described in Table (5.4). □

x	x_1	x_2	y y_1
0	0	0	$(f_0' + f_0)(f_0 + f_0') = 1$
0	0	1	$(f_0' + f_1)(f_0 + f_1') = 1$
0	1	0	$(f_1' + f_2')(f_1 + f_2) = 1$
0	1	1	$(f_1' + f_3')(f_1 + f_3) = 1$
1	0	0	$(f_2' + f_0')(f_2 + f_0) = 1$
1	0	1	$(f_2' + f_1')(f_2 + f_1) = 1$
1	1	0	$(f_3' + f_2)(f_3 + f_2') = 1$
1	1	1	$(f_3' + f_3)(f_3 + f_3') = 1$

Table 5.4: Transformed relation table for the inverter optimization problem.

A function f represents a feasible solution to an SRE if and only if all the constraints appearing on the right-hand side of the relation table hold true. It is thus possible to represent such constraints by means of their conjunction, *i.e.* by a single equation of type

$$\mathcal{K}(f_j; j = 0, \cdots, 2^{n_i \times d} - 1) = 1 \tag{5.7}$$

Example 48.

In the inverter optimization problem, the conjunction of all the relation table constraints produces:

$$\mathcal{K} = (f_0 \overline{\oplus} f_1)(f_1 \oplus f_3)(f_1 \overline{\oplus} f_2)$$

In the problem of Example (44), by considering a solution in the form $f(b, a_1, b_1)$, we have eight unknowns $f_j; j = 0, \cdots, 7$. It can be verified that they must satisfy

$$\mathcal{K} = (f_1' + f_4')(f_1' + f_6')(f_3 \oplus f_4)(f_3 \oplus f_6)(f_3 \overline{\oplus} f_5)(f_3 \overline{\oplus} f_7) = 1 \tag{5.8}$$

□

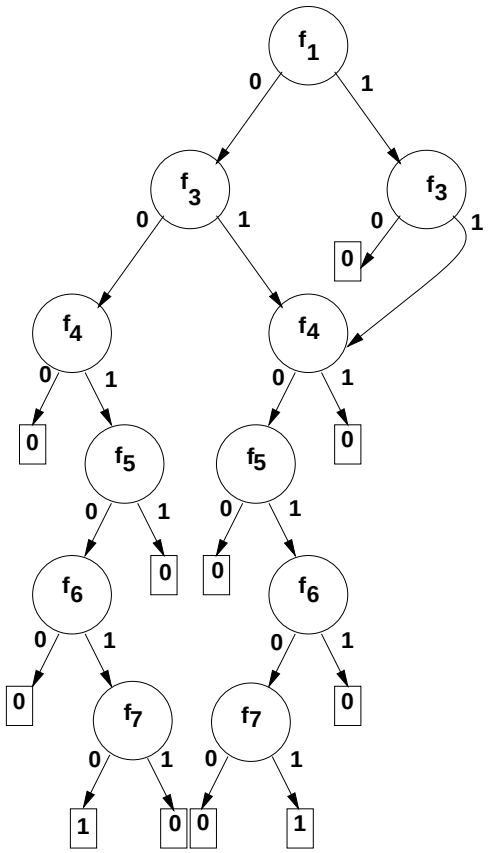


Figure 5.3: BDD representation of $\mathcal{K}$ for the optimization problem of Example (48)

It is in practice convenient to represent $\mathcal{K}$ by means of its Shannon decomposition tree, or its BDD. For Example (48), the BDD of the function $\mathcal{K}$ is reported in Fig. (5.3). It is worth noting that each path of the BDD of $\mathcal{K}$ corresponds to a partial assignment of the entries f_j resulting in $\mathcal{K} = 1$, hence it represents a feasible solution: The BDD of $\mathcal{K}$ is a representation of the set of feasible solutions of the SRE.

Definition 5.2 *We call* **support set** *of $\mathcal{K}$ (indicated by $SUPP_{\mathcal{K}}$) the set of entry variables f_j that appear in its BDD. The don't care* **set** *of $\mathcal{K}$, denoted by $DC_{\mathcal{K}}$, is the set of entry variables that* **do not appear** *in the BDD.*

Entry variables belonging to $DC_{\mathcal{K}}$ can be set arbitrarily in every feasible solution: they represent a *don't care* condition common to all such solutions.

5.4 Minimum cost solutions.

In the previous chapter we mapped the solution problem of a recurrence equation into the following combinational synthesis problem:

given	**determine**
an expression $\mathcal{K}(f_j)$ in terms of the $2^{n_i \times d} - 1$ entries f_j of f;	the minimum-cost function f such that $\mathcal{K} = 1$.

It is worth remarking on the differences between this problem and other combinational synthesis problems considered in the literature. The classic theory of incompletely specified functions [4, 3] considers incomplete specifications in which each entry is either assigned a value, or is a *don't care* . Unspecified entries can be filled in arbitrarily.

A first generalization to the classic theory (the minimization of Boolean Relations [31, 8]) has been considered in the context of optimization of multiple-output combinational circuits [31, 8]. It was shown in particular that for some multiple-output logic optimization problems correlations exist between assignments to the same entries of different incompletely specified functions. Note, however, that different entries of a single function could still be chosen independently. In the present case, instead, assignments to different entries are "correlated" : for example, by looking at the expression $\mathcal{K}$ in Example (48), it is easy to see that f_4 and f_5 must always be assigned opposite values.

We developed an algorithm for solving the combinational problem, with a strategy similar to the Quine-McCluskey algorithm for Boolean functions [4, 3, 6, 31, 8]. It first identifies a set of candidate prime implicants and then synthesizes a minimum-cost solution.

Definition 5.3 *A* **cube** $c(\mathbf{x}, \cdots, \mathbf{x}_d)$ *on the variables of* $\mathbf{x}, \cdots, \mathbf{x}_d$ *is the product of some such variables, in either true or complemented form. The variables appearing in* c *are termed the* **support** *of* c*. The* **size** *of a cube is the number of variables not in the support of* c*: cubes with larger size have fewer literals. A cube of size 0 is a* **minterm.** *A cube* c *is a* **candidate implicant** *if there exists a feasible solution* $f \geq c$*. A candidate implicant is a* **prime** *if there exists a feasible solution* f *for which* c *is prime, i.e. for which there is no implicant* $\widetilde{c}$ *of* f *that strictly contains* c*.*

We represent a feasible solution as sum of implicants. If a cube c is part of a feasible solution f, then for each assignment of $\mathbf{x}, \cdots, \mathbf{x}_d$ such that $c = 1$ it must also be $f = 1$.

Definition 5.4 *We call the set of entry variables f_j for which $c = 1$ the* **span** *SP_c of c.*

In the solution algorithm, cubes are represented by two different encodings. The first encoding is by its logic expression in terms of the variables of $\mathbf{x}, \cdots, \mathbf{x}_d$. The second encoding reflects that a cube represents a set of entries of SP_c. A cube is thus also encoded as a product of entry variables f_j, with each entry variable f_j appearing in true form if it appears in SP_c, and complemented form otherwise.

Example 49.

For the problem of Example (44), the cube $a_1 b_1$ covers the entries f_3 and f_7. It is thus encoded by a product $f'_0 f'_1 f'_2 f_3 f'_4 f'_5 f'_6 f_7$. □

5.4.1 Extraction of primes

This section describes the steps in the construction of the list of candidate primes.

The starting point of the procedure is a list of all candidate implicants of size 0 (*i.e.* , all candidate minterms). By definition, a minterm m is a candidate implicant if and only if there is a solution $f \geq m$. Let f_j denote the (unique) entry of SP_m. Hence, m is a candidate only if there is a solution f such that $f_j = 1$, *i.e.* , if there is a path in the BDD of $\mathcal{K}$ with f_j set to 1.

It is worth noting that, according to this definition, minterms covering elements of $DC_{\mathcal{K}}$ are candidates as well.

The set of candidate minterms can thus be identified by a simple depth-first traversal of the BDD of the function $\mathcal{K}$.

Pairs of minterms that differ in only one literal are merged to form a cube of size 1, by removing the differing variable. Unlike the Quine-McCluskey procedure, however, we must check whether $\mathcal{K}_c = 1$ has a solution before listing the new cube as an implicant.

This test is equivalent to checking whether the function $\mathcal{K}$ contains the product representing SP_c. This test can also be performed by a depth-first traversal of the BDD of $\mathcal{K}$.

Example 50.

Consider the BDD of Fig. (5.3). All entries appear in it, except for f_0 and f_2, which represent *don't cares* common to all solutions. There are thus 8 candidate minterms, each corresponding to an entry. These minterms are listed in the first column of Tab. (5.4.1).

Following McCluskey ([40]), they are encoded and sorted by syndrome.

Adjacent candidates are then merged, to form candidates of size 1. For instance, minterm 000 is merged with 001, 010, 100 to form the three candidates $00-, 0-0, -00$.

Each candidate is then tested for $\mathcal{K}_c \equiv 0$. For example, the candidate implicant $10-$, resulting for merging 100 and 101, is not acceptable, because no path in the BDD of $\mathcal{K}$ has $f_4 = f_5 = 1$.

The second column of Tab. (5.4.1) shows the remaining candidates of size 1. □

ba_1b_1 size 0	size 1	size 2
000	00-	- -0 $\surd$
	0-0	0- - $\surd$
001	-00	
010		- -1 $\surd$
100	0-1	
	-01	
011	01-	
101	-10	
110		
	-11 $\surd$	
111	1-1 $\surd$	

Table 5.5: Extraction of the prime implicants for Example XXX

Adjacent implicants produced at this point are then merged to form candidate implicants of size 2 and so on, until no new implicants are generated.

Example 51.

Consider constructing the list of candidates of size 2 of Fig. (5.4.1). First, candidates of syndrome 0 are first compared against those of syndrome 1: Candidate $0--$ is generated first, by merging $00-$ with $01-$. It represents a cube containing f_0, f_1, f_2, f_3. It is an acceptable candidate, because $\mathcal{K}(f_0 = f_1 = f_2 = f_3 = 1) \neq 0$. Candidate $-0-$ is generated next, by merging f_0, f_1, f_4, f_5. It is not an acceptable candidate, because $\mathcal{K}(f_0 = f_1 = f_4 = f_5 = 1) = 0$. Likewise, the candidate $-1-$ is not acceptable, because $\mathcal{K}(f_0 = f_1 = f_6 = f_7 = 1) = 0$. □

After generating implicants of size k it is convenient to discard all implicants of size $k-1$ that are not prime.

An implicant is recognized prime as follows. First, every implicant of size $k-1$ that is not contained in any implicant of size k is immediately recognized prime. Unlike the case of ordinary two-level synthesis, however, it might be the case where an implicant c, although covered by another implicant $\tilde{c}$ of larger size, may still be a prime. This occurs because each candidate can be an implicant of several different feasible solutions. It may then be the case that the solutions f for which c is an implicant do not coincide with the solutions for which $\tilde{c}$ is:

Example 52.

Consider the implicant -11. It is contained in the solution with truth table entries: $f_3 = f_5 = f_7 = 1; f_0 = f_1 = f_2 = f_4 = f_6 = 0$, and it is a prime of this solution, although it is contained in $--1$. Notice that this second candidate is not even an implicant of this solution. □

A second check for primeness is then necessary. In particular, for each prime that has not passed the first check we need to verify if there exists a feasible solution for which it is a prime.

The routine `check_primality` below performs this check. It takes as inputs the BDD representations of $\mathcal{K}$ and of the spans SP_{c_1}, SP_{c_2}, with $c_1 < c_2$, and tests whether there exists a feasible solution containing c_1 and not c_2.

The routine proceeds by identifying those entries f_j such that $c_1 = 0$ and $c_2 = 1$. It

returns TRUE if there is a path in the Shannon decomposition tree of $\mathcal{K}$ that contains c_1 and such that at least one f_j takes value 0.

```
int check_primality(c1, c2,  K)
BDD c1, c2, K;

{

     /* terminal cases */
     if ((K == ZERO_BDD) || (K == ONE_BDD)) return (FALSE);
     if (c_1 == c_2) return(FALSE);

     if (c_1.false == c_2.false == ZERO_BDD)
            /* current entry is present in both implicants */
            return(check_primality(c_1.true, c_2.true, K.true));

     if(c_1.true == c2.true == ZERO_BDD)   {
            /*current entry is absent from either implicant */
           result = check_primality(c1.false, c2.false, K.false);
            if (result == TRUE) return(result);
            return(check_primality(c1.false, c2.false, K.true));
     }
     if((c1.true == ZERO_BDD) && (c2.true != ZERO_BDD))   {
            /* entry present only in c2 */
            /* look for solutions containing only c1 */
            result = cube_in_function(c1.false, K.false));
            if (result == TRUE) return(result);
            /* if not found, continue search */
            return(check_primality(c1.false, c2.true, K.true));
            return((K == ZERO_BDD());
     }
}
```

Table (5.5) indicates with a $\surd$ mark the primes for the optimization problem of Example (44)).

5.4.2 Covering Step.

Once the list of primes has been built, Petrick's method is used to construct the subsequent covering problem [47, 31, 8]. Let N denote the total number of primes $\{c_1, \cdots, c_N\}$. The general solution is written as

$$f = \sum_{r=1}^{N} \alpha_r c_r, \tag{5.9}$$

where the parameter variable α_r is 1 if c_r is present in the solution, and $\alpha_r = 0$ otherwise.

Several cost measures can be applied. One such measure is the number of implicants in the solution: each implicant has then a unit cost. Another measure may consist of the total number of literals in the expression. Each implicant would then contribute with a cost proportional to its size. In either case, the cost W of a solution is expressed by

$$W = \sum_{r=1}^{N} w_r \alpha_r \tag{5.10}$$

where w_r is the cost associated with each prime c_r.

Let $\mathbf{x}^j$ denote the j^{th} assignment (of dimension $n_i \times d$) to the variables $\mathbf{x}, \cdots, \mathbf{x}_d$ (*i.e.* $\mathbf{x}^0$ = 00...0, $\mathbf{x}^1$ = 00...1, ...). For each f_j in $SUPP_{\mathcal{K}}$, it must be

$$f_j = \sum_{r=1}^{N} \alpha_r c_r(\mathbf{x}^j). \tag{5.11}$$

This equation expresses the entries f_j in terms of the parameters α_r. By substituting these expressions in $\mathcal{K}$, we obtain a new expression $\mathcal{K}_\alpha(\alpha_r; r = 1, \cdots, N)$ of the feasible solutions in terms of the variables α_r.

The synthesis problem is thus eventually transformed into that of finding the minimum cost assignment to the variables α_r such that $\mathcal{K}_\alpha = 1$, and it is known in the literature as **Minimum Cost Satisfiability** or **Binate Covering** problem. Its binate nature comes

from the possibility for some of the parameter variables α_i to appear in both true and complemented form in the conjunctive form of $\mathcal{K}_\alpha$, as shown by the following example.

Example 53.

For the optimization problem of Example (44), there are five primes, namely: $c_1 = a_1 b_1, c_2 = b b_1, c_3 = b_1', c_4 = a', c_5 = b_1$. From Eq. (5.11),

$$\begin{array}{llll} f_1 = & \alpha_4 + \alpha_5; & f_5 = & \alpha_2 + \alpha_5; \\ f_3 = & \alpha_1 + \alpha_5; & f_6 = & \alpha_3; \\ f_4 = & \alpha_3; & f_7 = & \alpha_1 + \alpha_2 + \alpha_5. \end{array} \tag{5.12}$$

The equation $\mathcal{K} = 1$ can now be rewritten as

$$\begin{array}{rl} \mathcal{K}_\alpha = & (\alpha_4 \alpha_5' + \alpha_3')(\alpha_4' \alpha_5' + \alpha_3') \\ & [(\alpha_1 + \alpha_5) \oplus \alpha_3][(\alpha_1 + \alpha_5) \oplus \alpha_6] \\ & [(\alpha_1 + \alpha_5) \overline{\oplus} (\alpha_2 + \alpha_5)] \\ & [(\alpha_1 + \alpha_5) \overline{\oplus} (\alpha_1 + \alpha_2 + \alpha_5)] = 1. \end{array} \tag{5.13}$$

There are two minimum-cost solution to this equation, namely $\alpha_5 = 1, \alpha_j = 0; j \neq 5$ and $\alpha_3 = 1, \alpha_j = 0; j \neq 3$. These solutions correspond to the expressions:

$$y = b_1' \; ; \qquad y = b_1;$$

respectively. Notice in particular that if $\mathcal{K}$ is represented in Conjunctive Normal Form (CNF), then also $\mathcal{K}_\alpha$ is automatically expressed in this way. The construction of the function $\mathcal{K}_\alpha$ is thus simplest if a CNF for $\mathcal{K}$ is available.

Figure (5.4) shows the circuit after optimization. The gate producing the signal y has been replaced with a direct wire to b. □

We now contrast this procedure against ordinary two-level optimization for combinational circuits. In the case of the Quine-McCluskey algorithm for ordinary Boolean functions, the covering step is also solved by branch and bound methods. Beginning from a void initial solution, implicants are iteratively added until a complete cover is

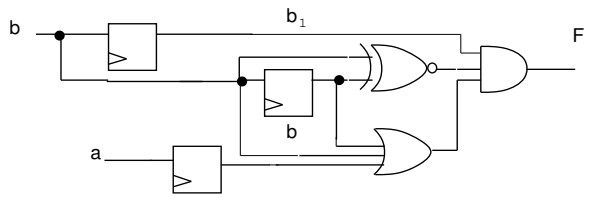


Figure 5.4: Circuit of Example (44) after optimization.

achieved. Different orders of inclusion result in different solutions, among which the one of minimum cost is selected. It is worth noting, however, that at any step the addition of an implicant to a partial cover can only improve the cover and, in the worst case, produce a sub-optimal solution. This is reflected by the unate nature of the function $\mathcal{K}_\alpha$ that would result for this class of problems.

In our case, the binate nature of the function $\mathcal{K}_\alpha$ has a negative impact on the complexity of the solution procedure. It might in fact be the case that two implicants **may not** appear in the same cover. For our current example, this is precisely the case of c_3 and c_5: No feasible solution can in fact contain c_3 and c_5 together, and this is reflected by the binate clause $[(\alpha_1 + \alpha_5) \oplus \alpha_3]$ in the constraint equation. During the search process, the addition of a prime to a partial cover can then invalidate the cover, thus representing a source of backtrack conditions.

5.5 Recurrence equations for sequential optimization.

The previous sections showed that SREs express the *don't cares* associated with a gate in an acyclic circuit, and outlined an exact two-level synthesis algorithm for SREs. This algorithm, however, is not yet suitable for a direct implementation in a logic synthesis environment, for the following two reasons.

First, since the SRE is expressed in terms of the primary inputs only, we would neglect the possibility of using internal signals for optimizing each gate. Second, the complexity of the synthesis algorithm makes it very inefficient for more than 5-6 input variables. To this regard it is worth observing that the support of $\mathcal{K}$ grows exponentially with the number of these variables.

In order to make recurrence equations useful, we must "translate" the degrees of

freedom they express into *don't cares* in terms of **a limited number** of internal signals. To this regard, the local inputs of the gate represent the obvious choice.

5.5.1 Image of a SRE.

The "translation" problem can be formulated as follows. We are given an SRE in terms of the primary inputs $\mathbf{x}$ of a circuit and an internal variable y. This equation can be put in the form

$$\mathcal{R}(\mathbf{x}, \cdots, \mathbf{x}_d, y, \cdots, y_d) = 1 \tag{5.14}$$

Let $\mathbf{z}$ denote the vector of the local input variables for the gate under optimization. Let n_l denote the dimension of $\mathbf{z}$. We know that the gates driving $\mathbf{z}$ realize pattern functions of the same inputs $\mathbf{x}, \cdots, \mathbf{x}_d$, *i.e.* :

$$\mathbf{z} = \mathbf{G}(\mathbf{x}, \mathbf{x}_1, \cdots, \mathbf{x}_d). \tag{5.15}$$

We want to construct a new SRE :

$$S(\mathbf{z}, y, \cdots, y_d) = 1 \tag{5.16}$$

expressing the same constraints as Eq. (5.14) in terms of the local variables $\mathbf{z}$.

This is accomplished as follows. First, we augment the function $\mathbf{G}$ into a function $\mathbf{H}(\mathbf{x}, \mathbf{x}_1, \cdots, \mathbf{x}_d, y, \cdots, y_d)$ with $n_l + d$ components. The first n_l components of $\mathbf{H}$ coincide with those of $\mathbf{G}$, while the remaining components are the identity functions for $y, y_1, \cdots, y_d$.

As mentioned in sections (4.1-4.2), corresponding to each pattern of depth d of the primary input variables $\mathbf{x}$, the SRE describes the set of acceptable patterns of y with the same depth. The function $\mathcal{R}$ thus describes a set of acceptable patterns of the variables $\mathbf{x}, y$.

Consider now the **image** of the set of **unacceptable** patterns (described by $\mathcal{R}'$) under the function $\mathbf{H}$. It represents a particular set of patterns of $\mathbf{z}, y, \cdots, y_d$. Let $I_{\mathcal{R}'}$ denote this set of patterns.

One pattern is in $I_{\mathcal{R}'}$ if there exists an unacceptable pattern of variables $\mathbf{x}, y$ of depth d that can produce it. Hence, this pattern must be unacceptable. All patterns not in

this set, instead, are perfectly acceptable, as they cannot be the result of an unacceptable pattern.

Hence, we take as function S the following:

```
BDD *SRE_image(R, y, d, function_list)
BDD *R;  /* BDD of the SRE*/
var *y;  /* variable */
int d;   /* depth of the relation */
list *function_list; /* list of local inputs */

{

    function_list = augment(y, d, function_list);
    return (COMPLEMENT(image(COMPLEMENT(R), function_list));

}
```

In the above pseudocode, `function_list` is the list of local inputs of `y`, expressed as functions of the primary inputs. This function list is `augmented` to form the list representing **H**. Then, the actual image computation of $\mathcal{R}$ is carried out, and the result complemented.

5.6 Implementation and experimental results.

We implemented in C the algorithms described in this chapter, and tested them on standard synchronous logic benchmarks. A network is first made acyclic by identifying and breaking feedback paths. Several choices are available in this respect. One possibility consists of identifying a minimum set of feedback vertices and breaking the feedback loops accordingly. This approach grants a minimum number of feedback variables. Another possibility we considered was of breaking feedback loops during a depth-first traversal of the network. We took this second option, thereby granting some minimality of the varibles we have to deal with. The first five columns of Table (5.6) report the statistics of the sequential benchmark circuits considered for test.

For each circuit, the SRE of each gate is built, and mapped into the gate's local inputs. Each gate is then optimized.

The last two columns of Table (5.6) report the final literal count and CPU time after optimization. The results were obtained on a SUN SparcStation LX.

Circuit	inputs	outputs	lits	regs	optl	cpu
s208	11	2	166	8	108	3
s298	3	6	244	14	155	14
s344	9	11	269	15	186	25
s420	19	2	336	16	251	258
s444	3	6	352	21	202	142
s641	35	24	539	19	241	302

Table 5.6: Experimental results for some logic optimization benchmarks.

5.7 Summary.

This chapter showed that the representation of structural *don't cares* at the sequential level requires a new means. In the case of acyclic networks, *don't cares* are represented fully by a **Synchronous Recurrence Equation.** The gate optimization problem is then cast as that of finding a minimum-cost solution to this equation.

A two-step exact solution algorithm has been proposed. The first step transforms the synchronous problem into a combinational one, which we have shown to differ from those previously considered in the literature. An exact algorithm for the latter problem is then presented.

Unfortunately, the algorithm requires treating each truth table entry of a function as an independent Boolean variable. The number of these variables is then exponential in the number of gate inputs, and represents the major bottleneck of the algorithm.

Other sources of complexity are presented by the binate nature of the covering problem and by the complexity of the prime-finding procedure.

Nevertheless, the method is very attractive for the optimization of sequential logic, because it makes available optimal solutions otherwise unreachable. We thus inserted the SRE solving procedure in a sequential logic optimization program. Experimental results show an improvement of about 7% over previously optimized circuits. The CPU penalty, however, is occasionally severe.

The method is thus probably best used only for the optimization of selected gates, for example those in the critical path of the circuit.

Currently, the global feedback function of the optimized network is not changed. This actually represents an unnecessary restriction to optimization: the feedback function can, in principle, be altered, as long as the observable terminal behavior of the entire network is not affected by this change. Further investigation on this aspect could result in algorithms leading to better quality optimization results.

Chapter 6

Cyclic synchronous networks

The simplest approach to the optimization of a cyclic network N consists of optimizing its acyclic portion N_d by the algorithms of Chapters (4-5), and regarding the feedback inputs and outputs as primary inputs and outputs, respectively. Intuitively, as the feedback interconnections are not directly controllable nor observable, this approach neglects some degrees of freedom. For example, some feedback sequences may be never asserted by the network and may therefore be considered as an external controllability *don't care* condition for N_d. Moreover, some values of the feedback input may be never observed at the primary outputs. As these inputs are generated by the feedback ouputs of N_d, these conditions actually represent an external observability *don't care* condition of the feedback outputs of N_d.

In this chapter, we consider capturing the existence of these *don't care* conditions in the form of virtual *external don't care sets* on N_d.

The rest of the chapter is organized in two major sections. The first section focuses on the impossible input sequences caused by the presence of feedback, while the subsequent section is devoted to observability *don't cares* induced by the equivalence of feedback sequences.

6.1 Modeling of cyclic networks.

In Chapters (4-5) we used extensively pattern functions to model the input/output behavior of acyclic networks.

The presence of feedback renders the behavior of cyclic networks more complex. In particular, it may not be captured by a sequence function:

Example 54.

Consider the circuit of Fig. (6.1.a). Corresponding to an input sequence $\ldots 0,0,0,\ldots$, the two sequences $\ldots 0,0,0,\ldots$ and $\ldots,1,1,1,\ldots$ are possible at output y. The two responses are due to the unspecified content of the delay element.

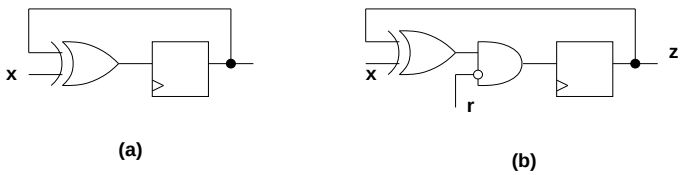


Figure 6.1: Circuit requiring a reset to realize a sequence function.

□

The appropriate modeling of one such network would be by means of a **sequence relation**. Informally, a sequence relation maps a sequence of input symbols into a **set** of sequences of output symbols.

The behavior of a circuit with n_i inputs and n_o outputs is thus captured by a relation

$$\mathcal{F} : (\mathcal{B}^{n_i})^\omega \rightarrow \mathcal{P}((\mathcal{B}^{n_o})^\omega) \tag{6.1}$$

For example, for the circuit of Fig. (6.1.a), $\mathcal{F}(\mathbf{0}) = \{\mathbf{0}, \mathbf{1}\}$.

We denote by $\mathcal{F}_{@n}(s)$ the set of values possibly taken by the network output at time n.

This modeling, however, is too complex in practice. Most circuits are designed with a **reset** sequence, so that their output is uniquely identified, at least for all time-points

$n \geq 0$. [1] Moreover, the outputs are regarded as irrelevant at time $n < 0$. Under these assumptions, the terminal behavior of a cyclic network can be modeled by a sequence function **F** as well, in the sense that for each input sequence s, $\mathbf{F}(s)$ could differ from any of the possible network outputs $\mathcal{F}(s)$ only for $n < 0$.

We now cast these assumptions (namely, not observing the outputs at time $n < 0$, and applying a reset to the circuit) in a more formal setting.

6.1.1 The reset assumption.

We assume that a particular input pattern is applied at time $n < 0$, so that the value taken by all delay elements at time 0 is known. We assume that this pattern has a finite length, and thus occupies some time interval $[-R, -1]$.

All sequences not containing this pattern are regarded as impossible input sequences for the circuit. To describe this set, it is convenient to introduce a sequence function **R**:$(\mathcal{B}^{n_i})^\omega \rightarrow (\mathcal{B}^{n_o})^\omega$. The function $\mathbf{R}(s)$ takes value **1** if the sequence s contains the reset pattern, and takes value **0** otherwise.

Not observing the outputs at time $n < 0$ can be modeled by a suitable sequence function as well. We introduce a function **O**: $(\mathcal{B}^{n_i})^\omega \rightarrow (\mathcal{B}^{n_o})^\omega$. The function $\mathbf{O}(s)$ takes value $\mathbf{1}_{@n}$ for $n < 0$, and takes value $\mathbf{0}_{@n}$ otherwise.

Reset and initial non-observation are taken into account by assuming that

$$\mathbf{DC} \geq \mathbf{R} + \mathbf{O} \tag{6.2}$$

It could be shown easily that, under the assumptions of reset and the non-observation at $n < 0$, it is possible to construct a function **F** whose values coincide with the network outputs for $n \geq 0$. In other words, there exists a function **F** that represents adequately the terminal behavior of a cyclic network for the time-points $n \geq 0$ of interest.

[1] This is usually accomplished by adding a dedicated reset input, as shown in Fig. (6.1.b). We regard these additional inputs just as ordinary primary inputs.

6.2 Feedback and external controllability *don't cares*

In this section, we consider modeling the effect of feedback as an added controllability *don't care* on the inputs of the acyclic portion of the network. We explain the methods by analyzing the case of the circuit of Fig. (6.2) .

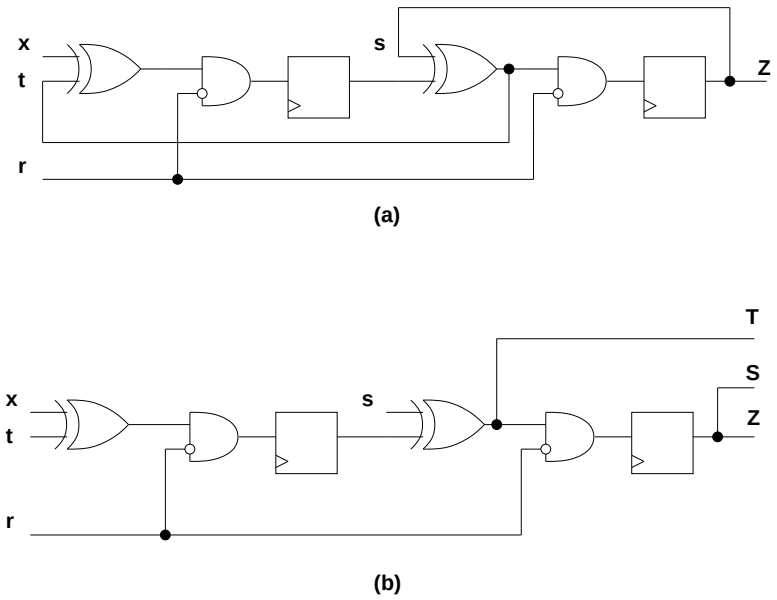


Figure 6.2: Working example for the computation of controllability *don't cares* .

The circuit of Fig. (6.2.a) implements a very simple finite-state machine, namely, a two-bit shift-register. The circuit was realized from its state diagram after an inappropriate encoding of the states.

Inputs x and r denote the primary and reset inputs, respectively. Reset is applied at time -1, zeroing the content of both registers at time 0.

As far as external *don't cares* are concerned, no input sequence with $r_{@-1} = 1$ is applied. We also assume, for simplicity, that no other external *don't cares* are present.

The circuit contains two feedback lines, labeled s and t, respectively. Cutting these feedback lines produces the circuit of Fig. (6.2.b). The outputs of the acyclic subnetworks are described by the pattern functions

$$\begin{aligned} Z &= r'_1\{s_1 \oplus [r'_2(x_2 \oplus t_2)]\} \\ S &= r'_1\{s_1 \oplus [r'_2(x_2 \oplus t_2)]\} \\ T &= s \oplus [r'_1(x_1 \oplus t_1)] \end{aligned} \tag{6.3}$$

Hence, the output at each time-point n depends on the values of x, r, s, t at time $n, n-1, n-2$. Because of feedback, some patterns of values of x, r, s, t over a time span of length 2 may be impossible. In order to simplify the circuit, it is thus necessary to identify which patterns of x, r, s, t can occur.

Because we do not observe the outputs at time $n < 0$, we are actually interested in the patterns that can occur over the time intervals $[n-2, n]$ only for $n \geq 0$.

Consider first the time-interval $[-2, 0]$. We already know that the value of $r_{@-1}$ is 1. We also know that the values of s and t at time 0 are linked to their past values by Eq. (6.3). Hence, only some combinations of these values is possible, namely, those satisfying Eq. (6.3) at time 0.

We describe this set by a characteristic function

$$\begin{aligned} C_{<0>} = \; & r_1 \cdot (s\overline{\oplus}(r'_1\{s_1 \oplus [r'_2(x_2 \oplus t_2)]\})) \cdot \\ & (t\overline{\oplus}(s \oplus [r'_1(x_1 \oplus t_1)])) = \\ & r_1 s' t' \end{aligned} \tag{6.4}$$

In general, we describe the set of **possible** patterns in the interval $[-2, 0]$ by means of a pattern function $C_{<0>}$. The subscript $< i >$ hereafter is used to indicate an iteration count. The pattern function $C_{<0>}$ takes value 1 at time 0 corresponding to those patterns in the interval $[-2, 0]$ that can occur at the network inputs. In other words, taking

$$C_{<0>,@0} = r_{@-1} s'_{@0} t'_{@0} \tag{6.5}$$

provides a set of possible patterns over the interval $[-2, 0]$. Eq. (6.5) correctly indicates that, as a consequence of reset, only $s = 0$ and $t = 0$ are possible combinations.

Eq. (6.4) represents only a first estimate of the set of possible sequences in that interval. We have not taken into account, for example, that the values of s and t at time -1 are fixed by the feedback network as well, and that therefore only those sequences that satisfy Eq. (6.3) at time -1 are possible.

A more refined expression of $C_{<0>}$ would then be

$$\begin{aligned} C_{<0>} = \; & r_1(s\overline{\oplus}(r'_1\{s_1 \oplus [r'_2(x_2 \oplus t_2)]\})) \cdot \\ & (t\overline{\oplus}(s \oplus [r'_1(x_1 \oplus t_1)])) \cdot \\ & (s_1\overline{\oplus}(r'_2\{s_2 \oplus r'_3(x_3 \oplus t_3)]\})) \cdot \\ & (t_1\overline{\oplus}(s_1 \oplus [r'_2(x_2 \oplus t_2)])) \end{aligned} \tag{6.6}$$

Further refinements can be obtained by adding similar expressions regarding s_2 and t_2, and so on, the limit being fixed by the complexity of the resulting expression. Notice also that the expression (6.6) contains literals outside the interval $[-2, 0]$ of interest, namely, x_3, t_3, r_3. The removal of these spurious literals by existential quantification [2] produces the following final expression:

$$C_{<0>} = r_1 s' t' [r_2 s'_1 t'_1 + r'_2(s_1 \oplus t_1 \overline{\oplus} x_2 \oplus t_2)] \; . \tag{6.7}$$

Notice that estimate (6.7) is smaller (*i.e.* more accurate) than (6.4).

The next step of the process consists of finding the set of possible patterns of inputs in the interval $[-1, 1]$, that is, the possible combinations of values of $r_{@-1}, r_{@0}, r_{@1}$, $s_{@-1}, s_{@0}, s_{@1}, \cdots$.

Consider first a simpler problem, namely, finding the patterns of one of the signals, say, t, over the time span $[-1, 1]$. We are given the patterns over the time span $[-2, 0]$. Because the two time intervals overlap, all we need to do is express the dependency of $t_{@1}$ from $t_{@0}, t_{@-1}$:

$$\begin{aligned} & t_{@-1} = t_{1,@0} \\ & t_{@0} = t_{0,@0} \\ & t_{@1} = T_{@1} = T_{-1,@0} = (r'\{s \oplus [r'_1(x_1 \oplus t_1)]\})_{@0} \; . \end{aligned} \tag{6.8}$$

Eq. (6.8) expresses the values of $t_{@-1}, t_{@0}, t_{@1}$ by means of some pattern functions of t, the first and second one being in particular the retiming by one and the identity function, respectively. It is thus possible to obtain the combinations of values of $t_{@-1}, t_{@0}, t_{@1}$ by computing the image of $C_{<0>}$ according to these functions.

[2] Existential quantification is used because we are interested in patterns that can occur, hence, that can happen for some value of x_3, t_3, r_3.

Now, for the complete problem, we compute the image of $C_{<0>}$ according to the following equalities:

$$\begin{aligned}
&r_{@-1} = r_{1,@0}\\
&r_{@0} = r_{0,@0}\\
&r_{@1} = r_{-1,@0}\\
&s_{@-1} = s_{1,@0}\\
&s_{@0} = s_{0,@0}\\
&s_{@1} = S_{@1} = S_{-1,@0} = (r'\{s \oplus [r'_1(x_1 \oplus t_1)]\})_{@0}\\
&t_{@-1} = t_{1,@0}\\
&t_{@0} = t_{0,@0}\\
&t_{@1} = T_{@1} = T_{-1,@0} = (s_{-1} \oplus [r'(x \oplus t)])_{@0}\\
&x_{@-1} = x_{1,@0}\\
&x_{@0} = x_{0,@0}\\
&x_{@1} = x_{-1,@0}
\end{aligned} \tag{6.9}$$

In this case, we obtain

$$Img(C_{<0>}) = s'_{-1}s't'r_1[t_{-1}\overline{\oplus}(r'x)] \tag{6.10}$$

The new estimate of the set of possible patterns is then

$$\begin{aligned}
&C_{<1>} = C_{<0>} + (Img(C_{<0>}))_1 =\\
&\left(r_1s't'[r_2s'_1t'_1 + r'_2(s_1 \oplus t_1\overline{\oplus}x_2 \oplus t_2)]\right) + \left(r_2s'_1t'_1s'[t\overline{\oplus}(r'_1x_1)]\right) =\\
&r_1s't'[r_2s'_1t'_1 + r'_2(s_1 \oplus t_1\overline{\oplus}x_2 \oplus t_2)] + r_2s'_1t'_1s'(t'x'_1 + tr'_1x_1)
\end{aligned} \tag{6.11}$$

The final estimate is obtained by the following **fixed-point** iteration:

$$C_{<k+1>} = C_{<k>} + (Img(C_{<k>}))_1 \tag{6.12}$$

For the circuit of Fig. (6.2), the final result, after four iterations, is:

$$\begin{aligned}
C = \quad & r'_2r'_1(t\overline{\oplus}x_1)(t_1\overline{\oplus}x_2)(s\overline{\oplus}t_1)(s_1\overline{\oplus}t_2)+\\
& r'_2r_1s't'(s_1 \oplus t_1\overline{\oplus}x_2 \oplus t_2)+\\
& r_2s'_1t'_1r'_1s'(t\overline{\oplus}x_1)+\\
& r_2s'_1t'_1r_1s't' \ .
\end{aligned} \tag{6.13}$$

Eq. (6.13) provides much information about the behavior of the circuit. The function C indicates in particular that, when no reset is applied for two clock periods (line corresponding to the term beginning with $r_1' r_2'$), then t and s represent the input x delayed by 1 and 2 clock periods, respectively. Viceversa, if a reset is applied (entries labeled r_1), then $s = 0$ and $t = 0$.

The complement of C represents a set of impossible input patterns for the circuit and can be used for its optimization. We conclude this section by showing that using the *don't care* derived so far it is possible to obtain the correctly encoded shift-register.

To simplify the presentation, rather than using the full *don't care* , we consider its restriction to the variables in the expression of Z:

$$DC = \forall_{s,s_2,t,t_1}(C') = [r_2' r_1' (s_1 \overline{\oplus} t_2) + r_2' r_1 + r_2 s_1']' \ . \tag{6.14}$$

The expression of DC was obtained from C' by *consensus* over the variables not appearing in Z, namely, s, s_2, t, t_1.

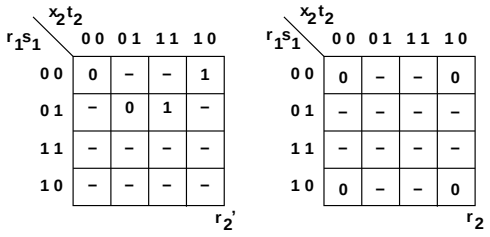


Figure 6.3: Karnaugh map of the function Z and of its *don't cares* for the two-bit shift-register problem

Fig. (6.3) shows the Karnaugh map of the function Z along with its *don't cares* . A minimum-cost realization is provided by the expression $r_2' r_1' x_2$. This expression is precisely the one realized by the properly-encoded shift-register, shown in Fig. (6.4).

6.2.1 *Don't cares* and state-space traversals

The problem of finding the input controllability *don't cares* for a circuit with feedback bears an evident similarity with the identification of the set of reachable states in a finite-state machine [48]. In the present iteration, the role of "states" is played by the patterns

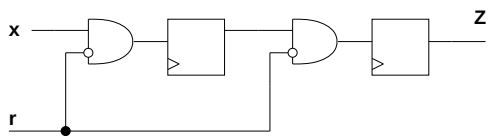


Figure 6.4: The correct version of the two-bit shift-register, as obtained by optimization with controllability *don't cares* .

of input and feedback variables.

Two aspects that distinguish the two problems are, however, worth remarking. First, we consider values on the feedback wires over a predefined, but otherwise *arbitrarily long* interval, rather than values at individual time-points. Second, we can extract a set of impossible patterns of input **and** feedback values, as opposed to focusing on impossible feedback combinations only.

We also remark on a difference of the present approach with respect to traditional finite-state machine optimization. Traditional methods determine the set of unreachable states and use this set as *don't care* for the combinational portion of the circuit. In the case of the circuit of Fig. (6.2), every state is reachable, so there is no *don't care* available for the optimization of the shift-register.

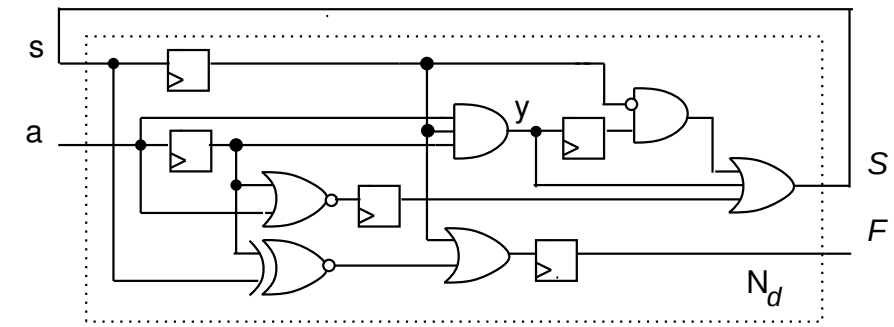
Notice also, however, that in order to determine the set of reachable states, one would have to work with only three binary variables (the two state bits plus the input variable). Moreover, convergence would be reached after just two iterations. With the present method, we needed four iterations to converge and we had to operate on 9 variables.

6.3 Perturbation analysis of cyclic networks.

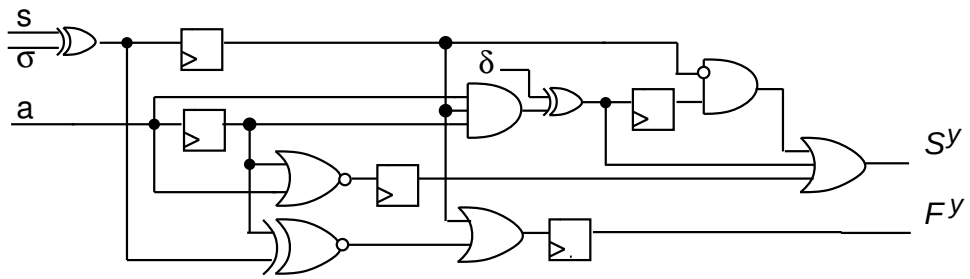
The algorithms of the previous section still regard the feedback output as a primary output. The functionality of the network realizing the feedback output is thus left unchanged by logic optimization, *modulo* the external controllability *don't cares* . On the other hand, the feedback outputs of a network are not primary outputs, and it may be possible to change their value corresponding to an input pattern that occurs, as long as this modification is not observable at the true primary outputs of the network.

We thus consider in this section associating an observability *don't care* with the

feedback outputs of the acyclic portion of a cyclic network. Again, we explain our methods referring to a working example, namely, the circuit of Fig. (6.5).



(a)



(b)

Figure 6.5: a) A cyclic network decomposed into a acyclic subnetwork and a feedback interconnection. b) Perturbed network for the AND gate optimization.

Consider the problem of optimizing the AND gate with output y in the circuit of Fig. (6.5.a). The modification of the gate introduces a perturbation that may affect the function realized at the feedback output. If the optimization algorithms of chapters (4)-(5) are used, then this perturbation is bound in such a way that neither the primary nor the feedback outputs are ever affected.

On the other hand, we may wish to let some error propagate through the feedback, as long as it never affects the the actual output.

In order to derive the bounds on the perturbation δ for the AND gate, it is convenient to compare the behavior of the original network with the network modified by **two**

perturbation signals, at the logic gate and feedback input, respectively, as shown in Fig. (6.5.b). The perturbation signals are here denoted by δ and σ, respectively. The behavior of the perturbed network is described by the output function $F^y(\delta, \sigma)$ and by the feedback function $S^y(\delta, \sigma)$.

In particular, $F = F^y(0,0)$ and $S = S^y(0,0)$. As σ represents the perturbation of the feedback input, it obeys the recurrence

$$\sigma = S^y(0,0) \oplus S^y(\delta, \sigma) \stackrel{\text{def}}{=} E^S \ . \tag{6.15}$$

The errors in functionality between the original and perturbed networks is described by the function

$$E = F^y(0,0) \oplus F^y(\delta, \sigma) \ . \tag{6.16}$$

The function g^y can replace f^y if

$$E_{@n} \leq CDC^{ext}_{@n} + ODC^{ext}_{@n} = DC^{ext}_{@n} \ \ \forall n \geq 0. \tag{6.17}$$

In the rest of this section we show that Eq. (6.17) can be decomposed into two separate bounds on δ and σ, respectively. As $\sigma_{@n}$ models the perturbation of the feedback function at time n, its bound represents implicitly an observability *don't care* set for S at time n. Moreover, we show that this bound is independent from δ, and can thus be computed once and for all before the optimization of the network.

In order to derive this bound, we introduce the following auxiliary error functions:

$$E^F_\delta \stackrel{\text{def}}{=} F^y(\delta, \sigma) \oplus F^y(0, \sigma) \qquad E^F_\sigma \stackrel{\text{def}}{=} F^y(0, \sigma) \oplus F^y(0,0); \tag{6.18}$$

$$E^S_\delta \stackrel{\text{def}}{=} S^y(\delta, \sigma) \oplus S^y(0, \sigma) \qquad E^S_\sigma \stackrel{\text{def}}{=} S^y(0, \sigma) \oplus S^y(0,0). \tag{6.19}$$

The following theorem allows us to split the problem of bounding σ and δ into two smaller subproblems, concerning σ and δ separately, and represented by Eq. (6.20) and (6.21), respectively.

Theorem 6.1 *If the perturbations δ, σ, resulting from changing f^y into a different local function g^y, are such that*

$$E^F_{\delta,@n} \leq DC^{ext}_{@n} \ \forall n \geq 0 \tag{6.20}$$

$$E^F_{\sigma,@n} \leq DC^{ext,@n} \ \forall n \geq 0, \tag{6.21}$$

then Eq. (6.17) holds and g^y can replace f^y.

Proof.

It is sufficient to observe that Eqs. (6.20)-(6.21) imply

$$E^F = E^F_\delta \oplus E^F_\sigma \leq E^F_\delta + E^F_\sigma \leq DC^{ext} \tag{6.22}$$

□

Eq. (6.21) could in particular be resolved by the methods of chapter (4) into a bound

$$\sigma \leq CDC^{ext} + ODC^s_{<0>} \ . \tag{6.23}$$

Eq. (6.23) can be interpreted as follows: if for each input sequence z the perturbation σ is such that

$$\sigma_{@n} \leq CDC^{ext}_{@n} + ODC^s_{<0>,@n} \tag{6.24}$$

then Eq. (6.21) certainly holds.

Notice that $ODC^s_{<0>}$ would be just the observability *don't care* of the feedback input s computed in the **acyclic** subnetwork N_d, assuming as external *don't care* specifications the vector:

$$\mathbf{DC}^{ext} = \begin{pmatrix} CDC^{ext} + ODC^{ext} \\ CDC^{ext} + 1 \end{pmatrix} . \tag{6.25}$$

Again, the subscript $< 0 >$ indicates that the function obtained at this point is actually the basis of an iteration process. The necessity and details of this process are explained next.

6.3.1 An iterative procedure for external observability *don't cares* .

The bound (6.23) represents the extent to which the feedback input can be changed, without changing the network behavior.

The feedback function can then be modified, as long as the introduced functional error results in a perturbation σ obeying Eq. (6.23).

Because the errors on the feedback line obey Eq. (6.15), we must then ensure that

$$E^S \leq CDC^{ext} + ODC^s_{<0>} \tag{6.26}$$

as well. Equation (6.26) represents a constraint on the feedback output of the perturbed network. It is formally identical to Eq. (6.17), and Theorem (6.1) could be applied again, to obtain another pair of sufficient conditions:

$$E_{\delta}^{s} \leq CDC^{ext} + ODC_{<0>}^{s} \tag{6.27}$$

$$E_{\sigma}^{s} \leq CDC^{ext} + ODC_{<0>}^{s} \tag{6.28}$$

In turn, Eq. (6.28) can be resolved by the methods of chapter (4) into a second bound placed on σ. This second bound must again be regarded as a constraint on the function E^S, and so on.

We thus face the problem of determining a self-consistent bound $ODC^s \leq ODC_{<0>}^{s}$ on σ, namely a bound such that

$$\sigma_j \leq ODC_j^s; j = 1, \ldots, P \tag{6.29}$$

implies

$$E^s \leq CDC^{ext} + ODC^s \ . \tag{6.30}$$

This bound is determined by another fixed-point iteration as follows.

Beginning from $ODC_{<0>}^{s}$, we take as new estimate $ODC_{<1>}^{s}$ of ODC^s the observability *don't care* of s, using as external *don't care* specifications

$$\mathbf{DC}^{ext} = \begin{pmatrix} CDC^{ext} + ODC^{ext} \\ CDC^{ext} + ODC_{<0>}^{s} \end{pmatrix}. \tag{6.31}$$

The constraint $\sigma \leq ODC_{<1>}^{s}$ is then a sufficient condition for Eq. (6.28). Then, we intersect this estimate with $ODC_{<0>}^{s}$, to ensure that the new estimate will be a subset of the old one. The process is repeated until convergence. More in detail, the iteration scheme is thus as follows:

- Initialize at 1 the external observability *don't care* of the feedback output S;
- Repeat :
  - Compute $ODC_{<k+1>}^{s}$ using the algorithms of chapter (4) on the feedback input s.

- $ODC^s_{<k+1>} = ODC^s_{<k+1>} ODC^s_{<k>}$;

- Until $ODC^s_{<k+1>} = ODC^s_{<k>}$.

Theorem (6.2) below proves the correctness of the method. We first show an application of the iteration to the circuit of Fig. (6.5).

Example 55.

We derive here the observability *don't care* set ODC^s for the cyclic network of Fig. (6.5). The longest path is $P = 2$, and we assume $ODC^{ext} = 0$.

The functions realized by the network are

$$\begin{aligned} F &= s_2 + s_1 \overline{\oplus} a_2 \\ S &= a'_1 a'_2 + a a_1 s_1 + a_1 a_2 s'_1 s_2 \end{aligned} \tag{6.32}$$

Initially, the feedback output S is taken unobservable. By applying the algorithms of chapter (4), $ODC^s_{<0>} = (s_{-1} \overline{\oplus} a) s' s_1$.

The function $ODC^s_{<0>}$ at this point indicates that, whenever an input pattern $s_1 s' a s_{-1}$ or an input pattern $s_1 s' a' s'_{-1}$ is applied at the inputs of N_d, a perturbation σ can be introduced without being observable at the primary output at future times $n+1, n+2$. Notice, however, that it might change the feedback output, and then affect the primary outputs in this way. For this reason, the set of patterns that allow the introduction of a perturbation needs further shrinking.

Notice also that $ODC^s_{<0>}$ depends on s_{-1}. This dependency can be resolved by using the equality

$$s = S(a, s) = a'_1 a'_2 + a a_1 s_1 + a_1 a_2 s'_1 s_2 \ . \tag{6.33}$$

Carrying out this substitution produces $ODC^s_{<0>} = a_1 s' s_1$. This first estimate is taken as external observability *don't care* for the feedback output S. The second estimate of ODC^s is obtained by regarding $ODC^s_{<0>}$ as an external observability *don't care* of S and by applying the algorithms of chapter (4).

The intersection of this estimate with $ODC^s_{<0>}$ eventually yields $ODC^s_{<1>} = a'a_1s's_1$. This is also the final estimate, as a new iteration would produce the same result. □

Theorem 6.2 *For each arbitrary input sequence z, suppose that a perturbation sequence σ is injected, such that*

$$\sigma_{@j} \leq CDC^{ext}_{@j}(z) + ODC^s_{@j}(z); \quad j = -P, \ldots, -1; \tag{6.34}$$

then:

$$\sigma_{@n} \leq CDC^{ext}_{@n}(z) + ODC^s_{@n}(z) \ \forall n \geq 0; \tag{6.35}$$

and Eq. (6.21) holds.

Proof.

It is sufficient to recall that, because of the convergence of the algorithm,

$$\sigma_j \leq CDC^{ext} + ODC^s_j; j = 1, \cdots, P \tag{6.36}$$

implies

$$\sigma = E^S \leq CDC^{ext} + ODC^s \tag{6.37}$$

□

6.4 Experimental results.

We report in this section on optimization results obtained by applying the *don't care* extraction techniques of chapters 4 and 6.

We considered the synchronous benchmarks reported in Table (4.2). As no information about reset sequences or reset states is available for these benchmarks, a reset sequence consisting of r consecutive zeros was selected for each circuit. The parameter r was then assigned the values $P, P+1, P+2$, with P denoting the longest path (in terms of register count) in the circuit; feedback *don't care* conditions were extracted and

Circuit	$r = P$			$r = P + 1$			$r = P + 2$		
	lit.	reg.	CPU	lit.	reg.	CPU	lit.	reg.	CPU
S208	72	8	16	58	8	21	52	8	24
S298	109	12	27	102	12	44	99	12	51
S344	131	15	31	127	16	41	122	15	49
S444	144	19	29	131	18	41	127	17	51
S526	216	20	31	188	21	34	149	21	41
S641	209	14	53	187	15	64	150	14	88
S820	260	5	59	255	5	69	243	5	73
S832	261	5	65	245	5	98	245	5	188
S1196	554	16	194	531	15	278	521	15	456
S1238	625	16	238	609	15	277	522	14	402
S1494	582	6	91	569	6	191	565	6	236
S9234.1	747	176	785	462	174	987	398	177	1686

Table 6.1: Optimization results

logic optimization of the acyclic portion were performed for each of these values. Delay elements were assigned finite cost, equivalent to 4 literals. It was thus in principle possible to trade off combinational complexity by the addition of delay elements.

6.5 Summary

In this chapter we explored the possibility of taking into account the presence of feedback in cyclic networks. We do so by adding suitable external *don't care* conditions to the acyclic portion of the network. In particular, we presented fixed-point algorithms for the computation of external controllability and observability *don't care* conditions. These *don't cares* need be computed only once at the beginning of the optimization process.

Chapter 7

Conclusions

In this work, we presented a suite of new algorithms for the structural optimization of logic networks, both at the combinational and sequential logic level. The paradigm common to these algorithms is optimization by local re-design of an original network: Each vertex of the network, corresponding to a logic gate, is iteratively visited and optimized, until no improvement occurs in the network.

This paradigm presented two main problems. First, we do not now *a priori* what functional changes can be made to a vertex. We thus needed a formal description of the re-design space for each vertex of the network. Second, we needed to develop algorithms for **extracting quickly and using efficiently** this re-design space.

In Chapter 2 we introduced *perturbation theory* as a formal model for reasoning on local perturbations of a network. There, we also restricted our attention to combinational networks.

The design space for the improvement of a single vertex of a combinational function is described by a Boolean function, describing implicitly a so-called *don't care* set for the vertex. Based on perturbation theory, we were able to develop a new algorithm for extracting the *don't care* function associated with each vertex of the network. The efficiency of the algorithm is based on the optimal use of local rules.

It is often the case where the *don't care* function associated to a vertex is too complex to be represented in practice. Perturbation theory allowed us to develop new approximation techniques. Unlike previous approaches, these techniques may, in the limit , yield the

exact results. Moreover, we were able to compare **quantitatively** these approximations against previously published ones.

When the simultaneous optimization of more vertices is considered, the interplay of the local perturbations on each function make the accurate description and the optimal use of the *don't cares* very complex.

One possible strategy for overcoming these problems is to make conservative approximations on these *don't cares* . This strategy was considered, for example, in [33, 36]. Perturbation theory allowed us the analysis of this case, and to evaluate the merits and costs of the different approximation styles. We also developed a new optimization strategy for multiple-vertex optimization. This strategy is presented in Chapter 3. It is based on the identification of special sets of vertices, whose optimization is likely to be "simple". We showed that the joint optimization of these subsets of vertices can be carried out by classical two-level synthesis algorithms. We were thus able to achieve a higher optimization quality (with respect to single-vertex optimization) with only a very reasonable increase in the CPU time.

In Chapter 4 we turned our attention to the case of synchronous networks. The complete description of the re-design space associated with a single vertex suddenly becomes much more complex. In particular, it is no longer possible to describe this space by means of a function. On the other hand, this description is the only one that can currently be exploited with efficiency. Therefore, Chapter 4 presents a way of extracting a subset of the design space that can be expressed by a pattern *don't care* function. In Chapter 5, we introduce the notion of recurrence equations as a means for specifying the *don't cares* for a vertex in a general acyclic network. Chapter 6 extends the methods to cyclic networks. A cyclic network is decomposed into an acyclic portion, containing all the logic and delay elements, plus a set of feedback interconnections. The presence of feedback induces an added controllability and observability *don't cares* to the acyclic portion of the network. These external *don't cares* are computed once and for all before the optimization of the acyclic portion.

Bibliography

[1] D.C. Ku and G.De Micheli. *High Level Synthesis of ASICs Under Timing and Synchronization Constraints*. Kluwer Academic, 1992.

[2] D. Ku, D. Filo, and G. De Micheli. Control optimization based on resynchronization of operations. In *Proceedings of 1991 Design Automation Conference*, 1991.

[3] W. Quine. The problem of simplifying truth functions. *American Mathematical Monthly*, 59(8):521–531, 1952.

[4] E. J. McCluskey. Minimization of Boolean functions. *Bell Syst. Tech J.*, 35(5):1417–1444, November 1956.

[5] S. J. Hong, R. G. Cain, and D. L. Ostapko. MINI: A heuristic approach to logic minimization. *IBM Journal of Research and Development*, 1974.

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

[7] R. Rudell. *Logic Synthesis for VLSI Design*. PhD thesis, U. C. Berkeley, April 1989. Memorandum UCB/ERL M89/49.

[8] R.K. Brayton and F. Somenzi. An exact minimizer for boolean relations. In *Proc. ICCAD*, pages 316–319, November 1989.

[9] M. Pipponzi and F. Somenzi. An iterative algorithm for the binate covering problem. In *European Design Automation Conference*, pages 208–211, March 1990.

[10] R. Ashenhurst. The decomposition of switching functions. In *Proceedings of the International Symposium on Switching Theory*, pages 74–116, April 1957.

[11] A. Curtis. *New Approach to the Design of Switching circuits*. Van Nostrand, 1962.

[12] E. L. Lawler. An approach to multilevel boolean minimization. *ACM Journal*, 11(3):283–295, July 1964.

[13] E. S. Davidson. An algorithm for nand decomposition under network constraints. *IEEE Trans. on Computers*, C-18(12), December 1968.

[14] R.Karp. Combinatorics, complexity, and randomness. *Comm. of the ACM*, 29(2):98–111, February 1986.

[15] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A multiple-level logic optimization system. *IEEE Trans. on CAD/ICAS*, 6(6):1062–1081, November 1987.

[16] K. A. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. Multilevel logic minimization using implicit don't cares. *IEEE Trans. on CAD/ICAS*, 7(6):723–740, June 1988.

[17] K. C. Chen and S. Muroga. SYLON-DREAM: A multi-level network synthesizer. In *Proc. ICCAD*, pages 552–555, November 1989.

[18] G. De Micheli, R. K. Brayton, , and A. Sangiovanni-Vincentelli. Optimal satte assignment for finite-state machines. *IEEE Trans. on CAD/ICAS*, pages 269–285, July 1985.

[19] S. Devadas, H.-K. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. Mustang: State assignment of finite-state machines targeting multilevel logic implementations. *IEEE Trans. on CAD/ICAS*, 7(12):1290–1300, December 1988.

[20] J. Hartmains and H. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice-Hall, Englewood Cliffs, N.J., 1966.

[21] S. Devadas and A. R. Newton. Decomposition and factorization of sequential finite-state machines. *IEEE Trans. on CAD/ICAS*, 8:1206–1217, November 1989.

[22] S. Malik, E. M. Sentovich, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Retiming and resynthesis: Optimizing sequential networks with combinational techniques. *IEEE Trans. on CAD/ICAS*, 10:74–84, January 1991.

[23] S. Dey, F. Brglez, and G. Kedem. Partitioning sequential circuits for logic optimization. In *Proc.* 3^{rd} *Int'l Workshop on Logic Synthesis*, 1991.

[24] C.E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing synchronous circuitry by retiming. In *Proc.* 3^{rd} *CALTECH Conference on Large Scale Integration*. Computer Science Press, Rockville, 1983.

[25] G. De Micheli. Synchronous logic synthesis: algorithms for cycle-time minimization. *IEEE Trans. on CAD/ICAS*, pages 63–73, January 1991.

[26] J. Kim and M. Newborn. The simplification of sequential machines with input restrictions. *IEEE Trans. on Computers*, C-20:1440–1443, 1972.

[27] S. Devadas. Optimizing interacting finite-state machines using sequential *don't cares* . *IEEE Trans. on CAD/ICAS*, 10(12):1473–1484, 1991.

[28] J.-K. Rho, G.Hachtel, and F. Somenzi. Don't care sequences and the optimization of interacting finite state machines. In *Proc. ICCAD*. IEEE Computer Society Press, 1989.

[29] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Computers*, 35(8):677–691, August 1986.

[30] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. DAC*, pages 40–45, June 1990.

[31] F. Somenzi and R.K. Brayton. Boolean relations and the incomplete specification of logic networks. In *IFIP VLSI 89 Int. Conference*, pages 231–240, 1989.

[32] M. F. Brown. *Boolean Reasoning*. Kluwer Academic, 1990.

[33] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney. The transduction method - design of logic networks based on permissible functions. *IEEE Trans. on Computers*, 38(10):1404–1424, October 1989.

[34] G. D. Hachtel, R. M. Jacoby, and P. H. Moceyunas. On computing and approximating the observability *don't care* set. In *International Workshop on Logic Synthesis*, 1989.

[35] P. McGeer and R. K. Brayton. The observability *don't care* set and its approximations. In *ICCD, Proceedings of the International Conference on Computer Design*, September 1990.

[36] H. Savoj and R. K. Brayton. The use of observability and external don't cares for the simplification of multi-level networks. In *Proc. DAC*, pages 297–301, June 1990.

[37] A.C.L. Chang, I. S. Reed, and A. V. Banes. Path sensitization, partial boolean difference, and automated fault diagnosis. *IEEE Trans. on Computers*, 21(2):189–194, February 1972.

[38] H. Savoj, R. K. Brayton, and H. Touati. Extracting local don't cares and network optimization. In *Proc. ICCAD*, pages 514–517, November 1991.

[39] M. Damiani and G. De Micheli. Efficient computation of exact and simplified observability *don't care* sets for multiple-level combinational networks. In *International Workshop on Logic and Architecture Synthesis*, April 1990.

[40] Edward J. McCluskey. *Logic Design Principles With Emphasis on Testable Semicustom Circuits*. Prentice-Hall, 1986.

[41] R.L. Rudell and A.L. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Trans. on CAD/ICAS*, 6(5):727–750, September 1987.

[42] S. W. Jeong and F. Somenzi. A new algorithm for the binate covering problem and its application to the minimization of boolean relations. In *Proc. ICCAD*, pages 417–420, 1992.

[43] M. Damiani and G. De Micheli. *don't care* conditions in combinational and synchronous logic networks. *IEEE Trans. on CAD/ICAS*, March 1993.

[44] B. Trakhtenbrot and Y. Barzdin. *Finite Automata: Behavior and Synthesis*. North Holland, Amsterdam, 1973.

[45] G. W. Smith and R. B. Walford. The identification of a minimal vertex set of a directed graph. *IEEE Transactions on Circuits and Systems*, CAS-22:9–15, January 1975.

[46] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[47] S. R. Petrick. A direct determination of the irredundant forms of a boolean function from the set of prime implicants. *AFCRC-TR-56-110 Air Force Cambridge Research Center*, April 1956.

[48] O. Coudert and J.C. Madre. A unified framework for the formal verification of sequential circuits. In *Proc. ICCAD*, pages 126–129, November 1990.