TECHNOLOGY MAPPING OF TIMED ASYNCHRONOUS CIRCUITS

by

Curtis A. Nelson

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering

Electrical and Computer Engineering

The University of Utah

December 2004

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Curtis A. Nelson

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Chris J. Myers

Erik Brunvand

Behrouz Farhang-Boroujeny

Reid R. Harrison

Ken Stevens

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Curtis A. Nelson in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Chris J. Myers
Chair, Supervisory Committee

Approved for the Major Department

Marc Bodson
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

This dissertation presents an efficient method for technology-mapping of timed asynchronous circuits. Technology-mapping combines the steps of decomposition, partitioning, and matching/covering to implement a synthesized design in a given technology. This work is applied to timed circuits, which are a class of asynchronous circuits that utilize explicit timing information for optimization throughout the entire design process. This work carries the timing constraints down to the circuit implementation level, giving new insight into the detection and elimination of hazards. In asynchronous circuits, correct operation requires that there are no hazards in the circuit implementation. Therefore, each internal node and output of the transformed circuit following decomposition must be verified for hazard-freedom to ensure correct operation. Current verification algorithms require an explicit state exploration often resulting in state explosion for even modest sized examples. The goal of the hazard verification portion of technology-mapping is to abstract the behavior of internal nodes and utilize the reduced state space to make a conservative determination of hazard-freedom for each node in the circuit. The newly annotated circuit is then mapped to an existing library for implementation. This dissertation explores various complexities of libraries used for matching and examines the hazard covering behavior using a variety of gates. Issues such as short-circuit detection and common-input matching are explored in detail, particularly when libraries contain generalized C-elements. The goal of this research is a hazard-free implementation of the synthesized design in a user-provided technology. Experimental results indicate that this new hazard-verification approach is substantially more efficient than existing timing verification tools and that in most cases hazard-free netlists are produced with modest sized libraries.

To my family, for toughening my skin.

CONTENTS

ABSTRACT . . . . **iv**

LIST OF TABLES . . . . **ix**

LIST OF FIGURES . . . . **x**

ACKNOWLEDGMENTS . . . . **xiii**

CHAPTERS

1. INTRODUCTION **1**

1.1 Synchronous Design Flow 2
1.2 Asynchronous Design Flow 4
1.3 Verification 6
1.4 Synchronous Technology-Mapping 7
1.5 Asynchronous Technology-Mapping 12
1.5.1 Delay-Insensitive Circuits 12
1.5.2 Speed-Independent Circuits 12
1.5.3 Fundamental-Mode Circuits 15
1.5.4 Timed Circuits 17
1.6 Goal 18
1.7 Contributions 18
1.8 Dissertation Overview 19

2. BACKGROUND AND SEMANTICS **21**

2.1 Time Petri Nets 21
2.2 Netlists 23
2.3 Sum-of-Products Representation 23
2.4 Generalized C-elements 24
2.5 Timed Asynchronous Design Flow 26

3. DETERMINATION OF HAZARD-FREEDOM **30**

3.1 State Graphs 31
3.2 Definition of Circuit Correctness 32
3.2.1 Complex Gate Equivalence 32
3.2.2 Acknowledgment Hazards 34
3.2.3 Monotonicity Hazards 35
3.3 Hazard Verification Algorithm 36
3.3.1 Checking Equivalence 36

3.3.2 Finding Stable States 37
3.3.3 Untimed Stabilization 39
3.3.4 Timed Stabilization 40
3.3.5 Acknowledgment Hazards 45
3.3.6 Monotonicity Hazards 46

4. DECOMPOSITION 51

4.1 Decomposition Algorithm 51
4.1.1 Decomposition to Base Functions 52
4.1.2 Insertion of Inverter Pairs 57
4.2 Hazard Annotation 59
4.3 Input Pin Reordering 60
4.4 Library Creation 63
4.4.1 Implementation Library 64
4.4.2 Base Functions 64
4.4.3 Decomposition of Common-Input Cells 65
4.4.4 Input Label Permutations 67

5. MATCHING AND COVERING 69

5.1 Basic Matching 69
5.1.1 Top Level Matching Algorithm 71
5.1.2 Matching Algorithm 73
5.1.3 Simple Matching and Covering Example 74
5.2 Hazard-Aware Matching 78
5.3 Short-Circuit Issues in gC's 83
5.4 Matching Common-Inputs 89

6. EXPERIMENTAL RESULTS 93

6.1 Verification 93
6.1.1 Verification of Benchmark Files 93
6.1.2 Verification Using Decomposed Circuits 95
6.1.3 Timed and Untimed Stabilization 96
6.2 Matching and Covering 97
6.2.1 Implementation Libraries 98
6.2.2 Matching Using Speed-Independent Synthesis 99
6.2.3 Matching Using Timed Synthesis 101
6.2.4 Hazard-Aware Matching 102
6.2.5 Short-Circuit Rejection for gC's 106
6.3 The Cost of Hazard-Freedom 107
6.4 Case Studies 110
6.4.1 False Hazards 110
6.4.2 Nonpropagating Acknowledgment Hazard 112
6.4.3 Nonoptimum Covered Circuit 114

7. CONCLUSIONS **116**
7.1 Summary 116
7.2 Current Relevance 118
7.3 Future Work 119
7.3.1 False Hazards 119
7.3.2 Reordering of Inputs 119
7.3.3 Inertial Delay Models 120
7.3.4 Hazard-Dependent Behavior 120
7.3.5 Acknowledgment Hazards 120
7.3.6 Inverter Pair Insertion 121
7.3.7 Other Input Forms 121
7.3.8 Internal Cycles 121

REFERENCES **122**

LIST OF TABLES

4.1 Truth tables for a gC and a CEL. 54
5.1 Costs for sample library. 76
5.2 Structural matches for Ebergen output b. 77
6.1 Comparison against other timing verification tools. 94
6.2 Comparison for decomposed netlists. 95
6.3 Hazard comparison based on stabilization method. 96
6.4 Hazard reduction for speed-independent synthesis. 100
6.5 Hazard reduction for timed synthesis. 102
6.6 Hazard reduction using various weighting factors. 103
6.7 Comparing w_1,w_2 = (0,1) by wins and losses. 105
6.8 Short-circuit rejection of gC's. 106
6.9 Cost of hazard reduction for untimed synthesis and verification. 108
6.10 Cost of hazard reduction for timed synthesis and verification. 109

LIST OF FIGURES

1.1 Synchronous design flow. 3
1.2 Asynchronous design flow. 5
1.3 Atomic gate model. 6
1.4 Decomposition architectures. 9
1.5 4-input AND function decomposition. 10
1.6 Hazard behavior using a speed-independent model. 14
1.7 Hazard behavior using a fundamental-mode model. 16
2.1 A time Petri net example. 22
2.2 Generalized C-element structure. 24
2.3 Example gC circuit. 26
2.4 Expanded timed asynchronous design flow. 27
2.5 Top level technology-mapping algorithm. 28
3.1 Example CGE circuit. 33
3.2 Explicit state graph. 34
3.3 Circuit to illustrate hazards. 35
3.4 Top level algorithm for verification. 36
3.5 Algorithm for finding stable states. 38
3.6 Untimed stabilization algorithm. 39
3.7 Timed stabilization algorithm. 41
3.8 Algorithm to update the zone. 42
3.9 Zone creation and evolution. 43
3.10 Timed stabilization recursion. 43
3.11 Algorithm to check for acknowledgment hazards. 45
3.12 Algorithm to check for monotonicity hazards. 46
3.13 Algorithm to check for a potential hazard. 47
3.14 Example to illustrate monotonicity hazards. 49
4.1 Generalized C-element structure. 53

4.2 CEL structure. 54
4.3 Decomposition algorithm for f_u. 55
4.4 Decomposition algorithm for an OR function. 55
4.5 Decomposition algorithm for an AND function. 56
4.6 Decomposition algorithm for a literal. 57
4.7 Example decompositions. 58
4.8 Hazard annotations. 60
4.9 Input pin reordering. 61
4.10 Hazard-freedom through pin reordering. 62
4.11 Library cell examples 65
4.12 2-input XOR example. 66
4.13 XOR matching example. 66
4.14 Input permutation example. 68
5.1 Top level matching and covering algorithm. 72
5.2 Matching algorithm. 73
5.3 Decomposition of Ebergen output b. 75
5.4 Example library. 75
5.5 Optimum covering of Ebergen output b. 79
5.6 Final circuit and netlist. 79
5.7 Algorithm for hazard-aware matching. 81
5.8 Hazard-aware matching. 82
5.9 Algorithm for short-circuit detection in gCs. 84
5.10 gC and CEL structures. 85
5.11 A short-circuit example. 85
5.12 Plain gC short-circuit example. 87
5.13 *gc22* short-circuit example. 87
5.14 Segment of Ebergen output x state graph. 88
5.15 XOR decompositions. 90
5.16 *gc22* library element structure. 90
5.17 Algorithm for common-input matching. 91
5.18 Common-input matching example. 92
6.1 Example library cells. 99

6.2 False hazard example using circuit *rpdft*. 111
6.3 Full timed state graph for the region of interest in circuit *rpdft*. 112
6.4 Nonpropagating acknowledgment hazard example. 113
6.5 Nonoptimum covered circuit example. 115

ACKNOWLEDGMENTS

There are many to thank and the first is Chris Myers, my committee chair and patient advisor for the 3+ years of this research project. He has ably and willingly provided proper alignment for my energy forces. He has proven not only to be a superb technical advisor, setting high standards and requiring compliance, but has also extended friendship beyond that expected. He has broadened my horizons, both in technical terms, and also by providing direct support to experience research at Stanford University and the Tokyo Institute of Technology. I am grateful to have been chosen as a student in his asynchronous research group.

Special thanks go to Eric Mercer and Erik Peskin who spent countless hours keeping my gaskets from blowing. Scott Little, Kip Killpack, Dave Walter, and Yanyi Zhou provided necessary technical support, as well as making office hours more enjoyable. I would like to thank Professor Tomohiro Yoneda for allowing us to conduct research in Japan. I also would like to acknowledge the support given by the Semiconductor Research Corporation, especially Ginny Wiggins and her capable staff.

I'd like to thank my employer, Walla Walla College, and the administration who supported my leave and gambled that I am a safe bet for the long-term. I'd also like to acknowledge the support and sacrifices made by my colleagues at the Walla Walla School of Engineering who pitched in at crucial times to lighten my load.

There is no way I could adequately express the debt of gratitude I owe my wife Wanda and daughter Heather for their love and support, and loaning me to the state of Utah for 2+ years.

Finally, I'd like to thank the mountains of Utah, just for being there.

CHAPTER 1

INTRODUCTION

Synchronous systems are those controlled by a central clock. These systems have dominated logic design since the advent of the integrated circuit and the necessary design styles and tools have become mature and well understood. However, with trends in the integrated circuit industry pushing performance to physical limitations, the timing issues involved with global clock synchronization have become increasingly difficult to resolve. As clock speeds increase, the amount of time available per clock period decreases to the point where serious design problems and even outright failure can occur if these issues are not resolved. As a result, logic designers require better CAD tools and more precise modeling. Alternatively, serious consideration is being given to more aggressive design styles.

Asynchronous design styles remove the need to address clocking issues by removing the global clock itself. However, asynchronous designs are difficult to implement reliably because of the presence of *hazards.* In many cases, the advantages inherent to asynchronous design are wasted due to the overhead required to implement hazard-free circuits. *Timed circuits* are a class of asynchronous design that uses explicit timing information in circuit synthesis [1]. The use of explicit timing information has been shown to potentially outperform synchronous designs and other asynchronous design styles as well. This is demonstrated in the Intel RAPPID project in which an asynchronous instruction length decoder for an x86 processor is designed using timed circuits, which are three times faster while using half the power of the comparable synchronous design [2].

Researchers have developed numerous *Computer Aided Design* (CAD) tools to support asynchronous design styles. These tools have focused on the specification and synthesis stages of the design flow. Designs are specified at a high level and synthesis produces a logic description that is guaranteed to be hazard-free. Since logic synthesis is best optimized when it is not constrained by an implementation technology, the resulting logic equations are unlikely to conform directly to available library elements. *Technology-mapping*, also called *library binding*, is the process whereby a *technology-independent* logic representation is mapped to a *technology-dependent* library. For timed asynchronous circuits, this process is often done by hand as there are currently no automated CAD tools available that address removal of hazards created during technology-mapping.

Synchronous designers have complete CAD tool-sets available from a variety of commercial sources and high-level, technology-independent design descriptions are implemented in technology-dependent libraries with little or no custom work needed by the designer. These tools and the algorithms they employ cannot be directly applied to timed asynchronous designs because they have little or no ability to deal with hazards. Although performance pressure is causing designers to expand their consideration of other design methodologies, the adoption of timed asynchronous circuits is unlikely until the supporting CAD tools are complete and have been tested on industry grade designs. In addition, few designers are aware of these alternative methodologies because they remain primarily in the purview of research labs and universities and have seen little exposure to market driven pressures.

This dissertation bridges the gap between synchronous technology-mapping techniques and the technology-mapping of timed asynchronous circuits. The goal is to complete the design flow for timed asynchronous circuits by developing an efficient and verifiable methodology for identifying and eliminating hazards created during technology-mapping.

1.1 Synchronous Design Flow

A typical design flow for synchronous systems is shown in Figure 1.1. The process of automatic synthesis starts with a technology-independent description of the design and ends with a technology-dependent transistor level implementation.

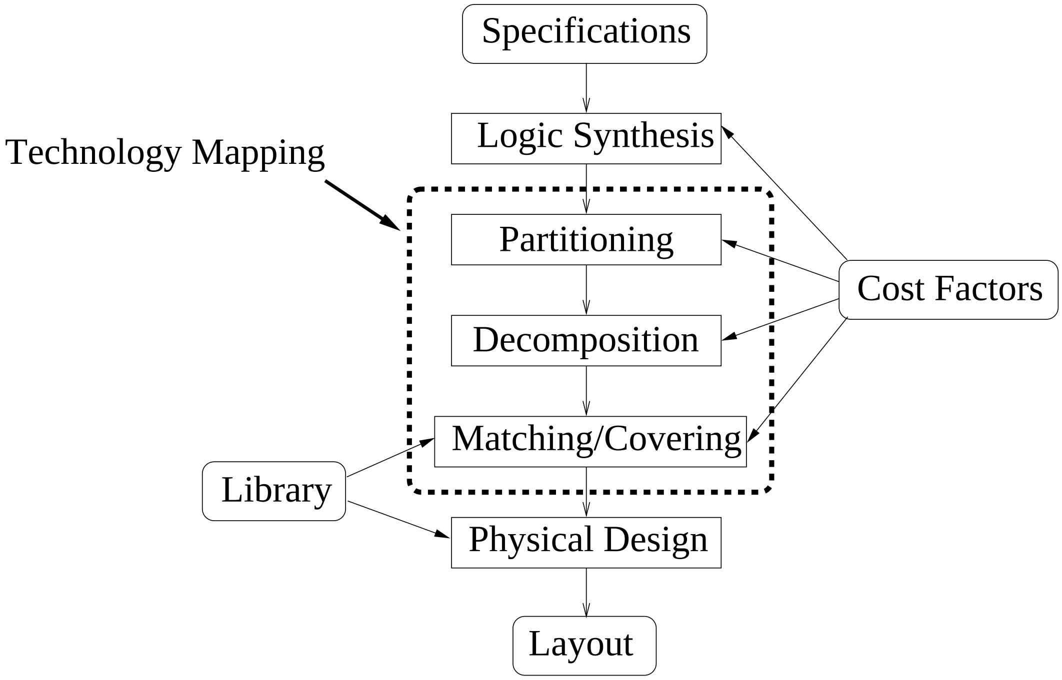


Figure 1.1. Synchronous design flow.

The subject of fully automated systems for synchronous design has been an active area of research since the 1960s. Initial research focused on the problems of minimizing the number of states needed to implement a given *Finite State Machine* (FSM) specification and the decomposition of an FSM into a network of two or more FSMs, which are behaviorally equivalent to the original one. Books by Hartmanis [3] and Kohavi [4] give excellent summaries of the first two decades of research in the optimization of synchronous FSMs.

In the 1980s, the research began to address the issue of structural representations, where the states (and sometimes the inputs and outputs) of the FSM are encoded in such a way that issues of area, delay, power, and other cost factors are adequately addressed. This problem was connected to multiple-valued logic minimization by De Micheli et al. [5] and extensions to this work were done by Villa et al. [6]. A more recent survey on the synthesis of FSMs for synchronous systems is found in [7].

In the mid- to late 1980s, several CAD tools for synchronous design appeared from research labs [8, 9, 10, 11]. As the theory and tools matured in the 1990s, a number of companies such as Synopsis, Cadence, and Mentor Graphics began marketing complete tool sets allowing the designer an automated means of producing fully functional and verifiable designs. The design process typically starts with an abstract high-level behavioral description of the circuit. Over the years, this has migrated from schematics or RTL descriptions to high-level languages such as VHDL [12] or Verilog [13]. These languages allow for better maintenance, portability, and reuse of existing design efforts.

The logic synthesizer takes this high level description and creates a logical representation of the circuit implementation. This process is often guided by a cost metric such as area, power, or delay and the resulting logic equations reflect this metric. For instance, a one-hot encoding scheme may be used in a synchronous state machine to reduce the next-state logic which in turn reduces the combinational path delay. This allows for a faster clock at the expense of increased area. The synthesis engine works best with few (or no) constraints and the resulting logic equations are optimal in a technology-independent sense. These equations from the synthesis engine are then mapped to a specific technology in the technology-mapping phase, which is described later.

1.2 Asynchronous Design Flow

Synchronous design flows cannot be used *as is* for asynchronous circuits because of the possibility of introducing hazards on new internal wires. As a result, the design flow shown in Figure 1.1 must be modified to accommodate the added restrictions placed on asynchronous design. Figure 1.2 shows the additional complexity associated with these modifications.

The particular asynchronous design style chosen determines the algorithms used during synthesis to generate a set of hazard-free logic equations. The synthesis of a timed asynchronous circuit typically begins with a *time Petri net* (TPN) [14] as the design specification. The TPN is then analyzed and a *state graph* (SG) is produced as an intermediate representation of the circuit. The synthesis engine then takes

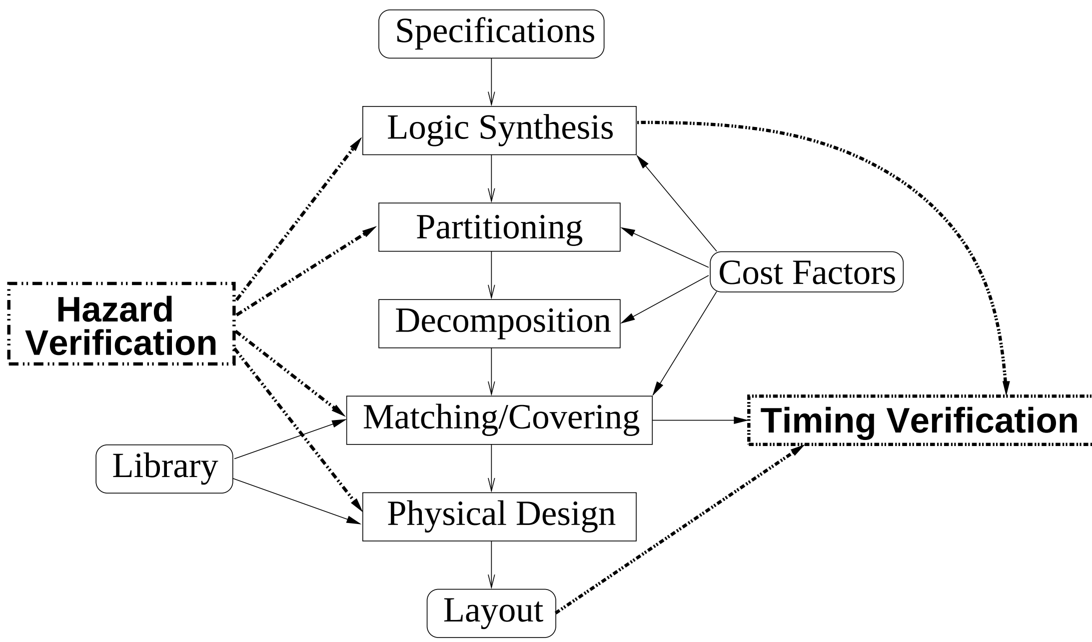


Figure 1.2. Asynchronous design flow.

this state graph as input and produces a hazard-free output, typically in the form of logic equations. The technology-mapper then takes the TPN and the hazard-free logic equations as inputs and outputs a hazard-free netlist composed of cells from an implementation library.

Depending on the particular asynchronous design style, hazards can be introduced during the decomposition stage of technology-mapping because the structure of the circuit changes. Each node in the newly decomposed netlist must be annotated with hazard properties. This information is then passed to the matching/covering stage, which attempts to eliminate the hazards by careful selection of library elements. This work relies on the *atomic gate model*, which allows library elements to encapsulate hazardous nodes within their structure. A complex gate is considered atomic when its Boolean function is evaluated instantaneously and the resulting value is then available at the output after a specified delay as shown in Figure 1.3.

Once the circuit has been mapped to a technology-dependent library, timing reverification must be performed to ensure the final netlist conforms to the original

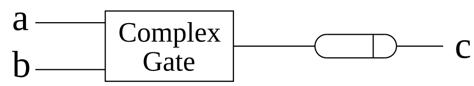


Figure 1.3. Atomic gate model.

timing specifications. Failure to ensure that the completed design meets original timing specifications may result in a circuit whose behavior is not consistent with the initial specifications.

1.3 Verification

While timed asynchronous circuits offer potential advantages over synchronous circuits such as faster operation and lower power, these advantages are often offset by the expense of the circuit overhead needed to eliminate hazards. Hazards are conditions generated by the structure of the circuit or timing relationships between inputs and propagation delays that can cause incorrect behavior. As synthesized hazard-free logic equations are mapped to a given gate library, new internal nodes are introduced in the circuit netlist. Each new internal node as well as the outputs of the circuit must be verified for hazard-freedom to ensure correct operation of the mapped circuit. This verification must be extremely efficient to allow for many alternative designs to be considered during technology-mapping.

Verification algorithms for timed circuits [15, 16, 17, 18, 19, 20, 21, 22, 23] often employ *zones* represented by *difference bound matrices*. When these verification algorithms are applied to gate-level designs, the enumeration of the state-space can can lead to state explosion problems. Yoneda and Schlingloff used a *partial order* method to reduce this problem for timed circuits [24].

The state explosion issue in the verification of timed circuits must also be addressed for gate-level *speed-independent* (SI) asynchronous circuits. In SI circuits, no timing assumptions are made about gates or the environment. Beerel addresses efficient gate-level verification methods for determinate SI circuits [25, 26]. Determinate SI circuits adhere to SI constraints and allow input choice (conditionals) but not output choice (arbitration). This work reduces the state space explosion problem found in earlier methods by examining individual behavior at each internal node and approximating this behavior for each state in the specification. The hazard-freedom of the circuit is then verified by examining this *cube approximation*. When the number of internal signals is high as compared with the number of primary inputs and outputs (a feature common of many circuit design styles), this cube approximation technique has the potential to substantially reduce the complexity of verification as demonstrated in the results shown in [25].

The work described in this dissertation combines the cube approximation method of [25] with the zone-based timing verification methods and develops an efficient gate-level timing verification method for timed asynchronous circuits.

1.4 Synchronous Technology-Mapping

Technology-mapping is the process of binding the technology-independent design to a dependent technology. The synchronous technology-mapper takes as input a technology-independent set of logic equations and a library of cells, and produces a technology-dependent netlist implementing the circuit from cells found in the library. This process is often optimized for a particular cost metric such as area, speed, or power. Solving this problem exactly is intractable [27] so heuristics have been created to break the problem up into smaller parts to get as good an approximation as possible.

Excellent overviews of synchronous technology-mapping algorithms are found in [28, 29]; therefore this section gives only a brief review. Algorithmic mappers as described in [30, 31, 32], generally break the process up into three phases: decomposition, partitioning, and matching/covering.

During partitioning [33], the network is split into single-output cones of logic where each cone represents a partition of the circuit obtained by creating a cut-set at points of multifanout. Each of these subcircuits is called a *subject graph*, which is matched to the various library cells, called *pattern graphs*, in the matching/covering step. The purpose of partitioning then, is to divide the size of the circuit to make the covering function more tractable. In its simplest form, the circuit is split at all points of multifanout and at the inputs to sequential elements. Each subpartition is then covered locally. While this certainly makes the covering problem more tractable, the efficiency obtained by covering across partition boundaries is completely missed and the result is less optimum. Programs such as MIS [9] and SIS [34] optimize the partitioning step to include the possibility of covering across partition boundaries. The final result of this optimization is likely to be a smaller circuit with a corresponding penalty paid in computation time.

The synthesized logic expressions from the initial network are represented as a *directed acyclic graph* (DAG). These expressions are decomposed into a multilevel circuit composed of simple gates called *base functions*. Base functions are typically 2-input AND, OR, NAND, or NOR gates, possibly inverters, and a primitive storage element. The implementation library must include these base functions to guarantee a solution.

Decomposition serves two purposes. First, it guarantees that a solution can be found, that is, a netlist of the logic equations can always be created from the library elements. Second, the finer granularity of a decomposed network allows for more matching options and a chance of a higher quality solution. This is particularly important since optimum netlists may differ depending on the chosen cost metric.

As an example, Figure 1.4 shows two possible decomposition architectures for a 4-input AND gate. The cost metric often determines which architecture is chosen. For instance, the decomposition shown in Figure 1.4(a) has a longer worst case path delay than the decomposition shown in Figure 1.4(b) so it is not desirable if delay is the primary cost metric. However, if explicit or relative timing information is available, it is possible to order the later arriving inputs closer to the output to minimize delay through the decomposition. In addition, the decomposition in Figure 1.4(a) may prove attractive because the architecture is consistent for N-inputs whereas the architecture in Figure 1.4(b) changes depending on the number of inputs.

The matching/covering step [33] finds all possible matches of pattern graphs for each node in the subject graph. The best of these matches is then selected taking

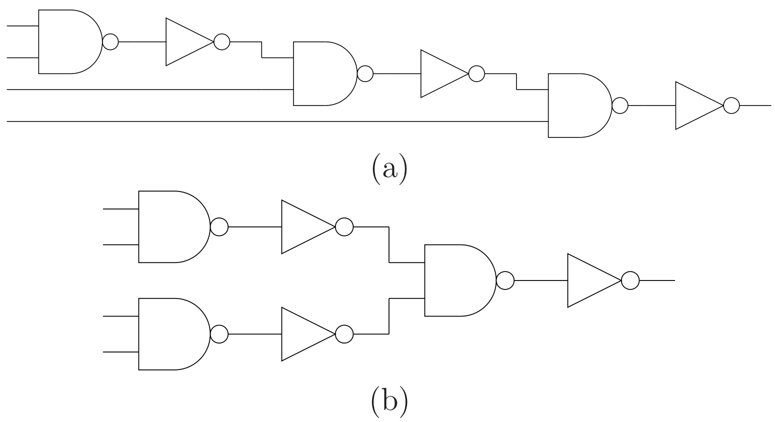


Figure 1.4. Decomposition architectures. (a) A nonsymmetric decomposition. (b) A symmetric decomposition.

into account the selected cost factors. Matching algorithms are classified as either *structural* or *Boolean* and both methods are discussed in more detail below. Both methods fit into the design flow shown in Figure 1.1 and are differentiated primarily by the process in which the subject graph representing the synthesized design is matched to the pattern graphs, representing the available library elements.

Structural matching is straightforward and is the simplest to implement. The structural approach is implemented in programs `Dagon` [30], `MIS` [35], and `Techmap` [36]. A recent and interesting structural pattern matching algorithm [37] uses lookup tables by recognizing that the matches for a node in a subject graph are related to the matches for its children.

For structural matching, the decomposed functions can be viewed as trees, with roots at the outputs and the primary inputs as leaves. Beginning at the leaves, the decomposed subject graph is then compared node-by-node to the available library gates. Because of the semicanonical nature of the decomposition, each node may have several variations that need to be examined. If a library gate can implement the tree to that point, the node is annotated with that information in addition to the cost up to that point. The cost includes the cost of generating the inputs and the cost of the gate. For example, in Figure 1.5(a), the cost at node e is the cost of a 2-input NAND gate where the cost of node g is either the cost of an inverter

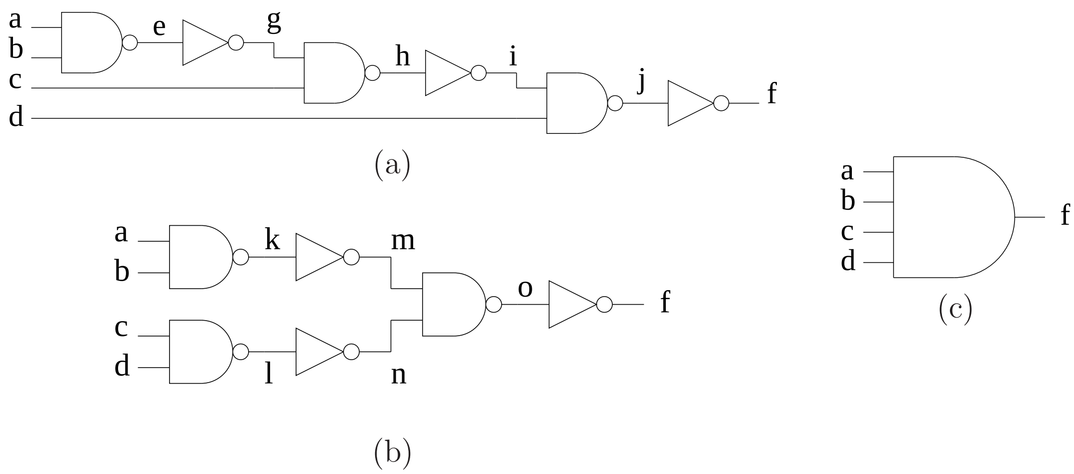


Figure 1.5. 4-input AND function decomposition. (a) A nonsymmetric decomposition. (b) A symmetric decomposition. (c) 4-input AND gate.

in series with a 2-input NAND gate or the cost of a 2-input AND gate.

When the root is reached, the lowest cost for implementing that portion of the circuit is chosen. If there are higher level functions available in the library such as 3-input NAND gates or AND-OR-INVERT gates, it is typically advantageous to enclose as many gates in the subject graph as possible. The complexity of the matching algorithm increases as the base functions used increase in complexity. The additional search space needed for complete matching makes base functions beyond 2-input NAND gates and inverters computationally expensive.

The structural mapping method usually gives good results but the quality can be affected by how the decomposition is done and even by the form of the equations derived to describe the design. In addition, every possible structural representation for a library cell would need to be included in the library for a complete matching process. For example, both of the structural representations shown in Figures 1.5(a) and 1.5(b) must be included as separate library elements. It is difficult (and perhaps intractable) to create *all* possible structural representations of a given Boolean function. As a result, it is unlikely that structural mapping finds all possible matches for a design. However, the structural approach is extremely efficient computationally.

Boolean mapping uses a process similar to structural matching except the nodes are labeled with the Boolean functions they represent rather than the cost. The Boolean approach was first implemented in the program `CERES` [31] and in an industrial tool from Fujitsu [38]. In [33], Mailhot proposed to use a list of possible matching functions at all nodes. This list represents all combinations of intermediate nodes available. For instance, given the 4-input AND function $f = abcd$ shown in Figure 1.5(a), the network consists of the following nodes: $f = \bar{j}$, $j = \overline{id}$, $i = \bar{h}$, $h = \overline{gc}$, $g = \bar{e}$, $e = \overline{ab}$. The library is exhaustively checked for Boolean functions matching each node. For instance, node h would be checked for gates implementing the following functions: $h = \overline{gc}$, $h = \overline{\bar{e}c}$, $h = \overline{\overline{\overline{ab}}c}$. Unlike the structural method, these gates here need not match exactly; they need only be equivalent in a Boolean sense. For instance, the gate $g = a(b|c)$ matches the function $f = ab|ac$. A match of this type would not be found by the structural mapper unless both gates are structurally represented in the library. The Boolean matching method can check for input permutations, inversions, and use *don't care* conditions to further optimize the solution. This method tends to be computationally prohibitive for larger networks, but work by Burch [39] shows implicit methods can be used to produce more reasonable run times.

Looking again at Figures 1.5(a) and 1.5(b), a Boolean matching procedure at the output node f would find both of these decompositions equivalent. However, a structural mapper would match both of these to the 4-input AND gate of Figure 1.5(c) only if their unique structural representations are entered as two separate library cells. Thus, if complete Boolean representation is desired for structural matching, the library must contain every possible structural representation of the function represented. Because of this, structural libraries tend to be limited to a smaller number of functions. The structural matching technique is computationally efficient and the choice of which mapping technique to use often depends as much on reasonable run times as it does on the quality of the final solution.

1.5 Asynchronous Technology-Mapping

Asynchronous design has evolved to the point where several classes or styles of design have become well developed. The particular design style chosen determines the types of hazards generated and the algorithms used to synthesize the design. Four main classes of asynchronous design styles avoid or reduce hazards during synthesis and technology-mapping by simplifying the timing assumptions. These classes are delay-insensitive, speed-independent, fundamental-mode, and timed.

1.5.1 Delay-Insensitive Circuits

Delay-insensitive circuits [40] make no assumptions on the delays of the logic gates or routing wires. This type of design is extremely robust and works correctly under any timing scenarios. However, it is quite restrictive and few designs can be implemented by direct synthesis methods. Often, syntax-directed methods are employed that use nonstandard libraries [41]. This approach renders technology-mapping unnecessary because the language compiler determines the structure and selects the library elements directly.

1.5.2 Speed-Independent Circuits

Speed-independent circuits [42, 43] are those that rely on a timing model that assumes that gate delays are unbounded and wire delays are negligible. These circuits place no global constraints on the environment. Instead, each input change is *acknowledged* by a primary output. This acknowledgment indicates that the environment can then change some inputs to the circuit. These timing assumptions increase the potential for higher circuit performance by exploiting concurrency.

Decomposition of SI designs is first addressed by Kimura in [44, 45]. Kimura's work investigated various problems encountered in circuit delays when buffers are added to wires. Siegel and De Micheli, in work specifically targeted at *standard C-implementations*, show in [46] that OR functions can be decomposed according to the associative law independent of signal ordering and the resulting network is still hazard-free. AND functions, however, require a set of conditions in order to be decomposed in a hazard-free manner. They suggest adding acknowledgment wire forks defined in [47] and proposed for use during decomposition in [48] for those gates where a hazard-free decomposition does not exist. These cases occur whenever a monotonic transition on an intermediate gate is not acknowledged somewhere else in the circuit.

Some of the most important work in the area of decomposition is that done by Burns [49]. He uses implicit techniques to decide whether a decomposition of a state-holding or combinational element into two elements with an isolated internal signal is correct. He then extends these techniques to determine all legal decompositions in a parameterized family. Cortadella et al. [50] expanded the work of Burns by using Boolean relations to allow for a larger number of decompositions. In [51], Kondratyev et al. also built upon the work of Burns to implement a two-step process for hazard-preserving decompositions. First, they used techniques based on algebraic factorization originally proposed for combinational logic [28]. Then, they inserted new signals in the decomposition to preserve hazard-freedom, based on work done by Vanbekbergen et al. [52] and Cortadella et al. [53].

Once the circuit is decomposed in a SI and hazard-preserving manner, Siegel and De Micheli show in [46] that the partitioning and matching/covering algorithms that is originally described in [54] as pertaining to burst-mode circuits (described in Section 1.5.3) can be used without modification.

Figure 1.6 illustrates how the decomposition of an AND function may or may not be hazard-free under SI assumptions. The example shown is taken directly from [55] and utilizes a *state graph* (formally defined in Chapter 3) shown in Figure 1.6(a) to indicate a sequence of transitions. The implementation for output d, whose behavior is shown in the state graph, is synthesized using a CAD tool and the results are shown in Figure 1.6(b). Suppose this 3-input AND function with one inverted input is not available in the implementation library. Decomposition must then be done in order for the circuit to be implemented. Figures 1.6(c) and 1.6(d) show two logically equivalent decompositions. The new internal signal e must now be checked for potential hazards introduced by the change of the circuit structure during decomposition. For the decomposition in Figure 1.6(c), in state (1110),

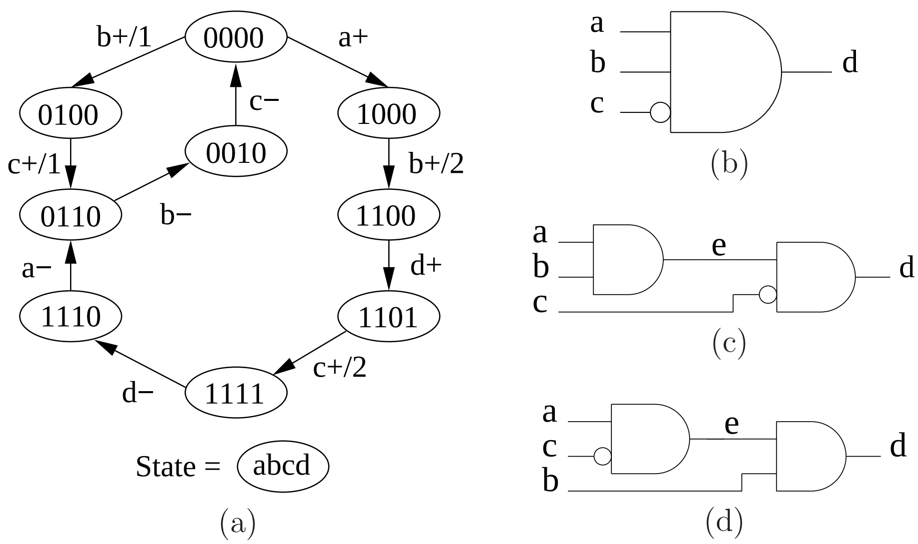


Figure 1.6. Hazard behavior using a speed-independent model. (a) State graph. (b) 3-input AND function. (c) Hazardous netlist. (d) Hazard-free netlist.

inputs a, b, c and internal signal e are high while the output d is low. After a falls, the circuit transitions to state (0110), and e becomes excited to fall. Assume the AND gate generating e is slow. When b falls, the circuit transitions to state (0010). If at this point c falls before e falls, d can become excited to rise prematurely. The result is a hazard on the signal d, and there is a potential for circuit failure.

Next, consider the decomposition shown in Figure 1.6(d), beginning in state (1110) again. This time e begins low, and it does not become excited to change until after a falls, b falls, c falls, and a rises again. At this point, b is already low, which maintains d in its low state until b rises again. There is no sequence of transitions that can cause this circuit to experience a hazard on the output d.

This example illustrates the need to take special care during decomposition of SI circuits. It also hints at how explicit timing information may be used to help determine whether a hazard condition actually manifests or not as seen in the example of Figure 1.6(c). This topic is explored fully in Chapter 3.

For most of the work published in the SI domain, technology-mapping of the circuit is done completely in the decomposition stage, making partitioning and matching/covering unnecessary. The goal of this type of SI work is to prevent the creation of any hazards during decomposition and, if that is not possible, resynthesize and repeat until a hazard-free implementation is found.

1.5.3 Fundamental-Mode Circuits

Fundamental-mode circuits [56, 57] were pioneered by Huffman and impose restrictions on when input changes can occur. This style of asynchronous circuits allows input changes only after the entire circuit has stabilized in response to a previous input change. The most basic fundamental-mode class of circuits is called *single-input change* (SIC) where only one input is allowed to change at a time. This mode of operation severely limits the use of concurrency so it has seen limited use for real designs.

Stevens [58] pioneered an extension to the SIC fundamental mode style of design that evolved into what is now referred to as *burst-mode*. Burst-mode allows a restricted burst of inputs to change rather than a single input. This design style has been shown to be practical in industrial designs such as those found in work done by Davis's group at Hewlett-Packard [59, 60, 61, 62]. A further extension, termed *extended burst-mode* (XBM), was developed by Yun et al. [63] and allows for the use of directed don't cares to specify that an input *may or may not occur* in a given burst. In addition, this style supports *conditional input bursts*, allowing decisions about future behavior to be based on the level of a particular signal.

The timing restrictions on when inputs can change in fundamental-mode circuits act much like the clock in a synchronous system. As a result, these circuits allow the designer to apply basic synchronous techniques for technology-mapping.

Decomposition can be performed by recursively applying De-Morgan's theorem and the associative law to the network. Both of these operations have been shown to be hazard preserving [57, 54, 64], that is, if the original circuit is hazard-free, then the decomposed circuit is also hazard-free. More recently, technology-mapping techniques for XBM machines that optimize for average-case performance have been developed by Chou et al. in [65]. Support for *generalized C-element* (gC) implementations of XBM designs is presented by Yun and Dill [66, 67]. A transistor-level technique for average-case technology-mapping of XBM gC designs is presented by James and Yun [68]. Finally, with the limitations that deep submicron circuit implementation places on cell libraries, Yang [69] investigates direct transistor-level technology-mapping as applied to asynchronous XBM controllers.

The process for technology-mapping of fundamental-mode circuits is virtually identical to that used for technology-mapping of synchronous circuits. The restrictions are that decomposition must be done using only De-Morgan's theorem and associative laws [64], and the library elements used during the matching stage be hazard-free. This result should not be surprising; once any input or burst of inputs change, *all* internal signals and outputs must be allowed to stabilize before another burst of inputs can occur. Fundamental-mode circuits suffer from the same limitation that synchronous circuits do; the reduced ability to exploit concurrency.

Figure 1.7 illustrates how the hazard problem encountered in Figure 1.6(c) for SI circuits cannot occur under fundamental-mode assumptions. Figure 1.7(a) shows a burst-mode state machine with inputs a, b, c, and output d. The output d has again been synthesized with a CAD tool and the results are shown in Figure 1.7(b). Starting with the dark circle in Figure 1.7(a), the circuit state is a, b, c, and e all

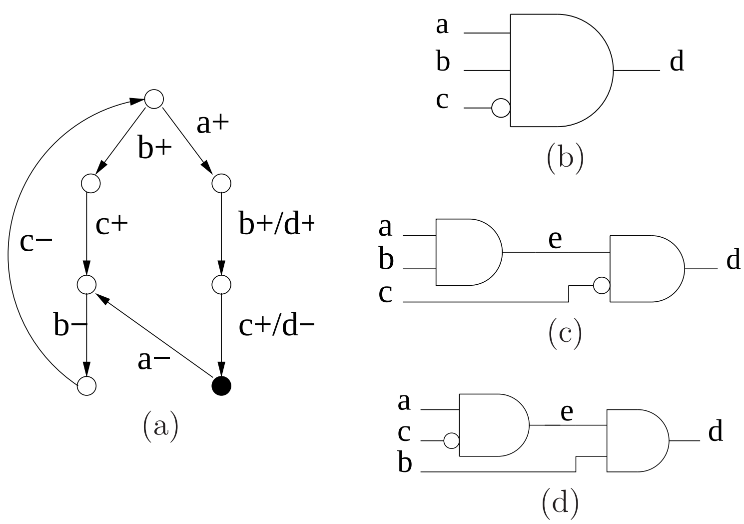


Figure 1.7. Hazard behavior using a fundamental-mode model. (a) Burst-mode state machine. (b) 3-input AND function. (c) Hazardous netlist under the SI model. (d) Hazard-free netlist under the SI model.

high and d low. At this point, a goes low causing the new internal node e to go low after some delay. Next, b goes low after e has stabilized followed by c going low. The output cannot glitch as it did in Figure 1.6(c) because e is already low well before c goes low. In fact, there is no sequence of events that can cause a hazard, under fundamental-mode assumptions, in either of the decompositions shown in Figures 1.7(c) and 1.7(d).

1.5.4 Timed Circuits

It is often the case that hazard conditions found in SI circuits do not actually manifest as glitches in the real circuit implementation due to the actual timing behavior. The reason for this is that internal signals, once enabled, certainly do transition in some finite amount of time. If the time evolution in the state space can be tracked, then it may be possible to identify the stability of some internal signals. This additional stability information may help determine whether or not a hazard condition actually produces a glitch in the circuit implementation.

Timed circuits are a class of asynchronous circuits that place two-sided timing constraints on the inputs and gates. Myers et al. first used timed state space exploration to produce optimized timed asynchronous circuits as described in [70, 71, 1, 16].

Since exhaustive state space methods quickly become intractable, much of the synthesis effort has gone into developing algorithms that examine the state space and utilize explicit timing to determine which states are unreachable. A *reduced state graph* (RSG) is then created that factors in the timing constraints. This RSG is analyzed to determine an optimum solution based on a set of constraints using speed-independent synthesis algorithms.

There is not a large abundance of previous technology-mapping work applied to timed circuits. The work by Lavagno et al. [72, 73, 74] leverages synchronous technology-mapping methods as does the work proposed in this research. Their work uses *signal transition graphs* as the primary design specification and applies to a *bounded wire-delay* (BWD) model. A BWD model assumes unbounded delays for gates and bounded delays for wires. Lavagno et al. performs synthesis on a circuit without regard to hazard creation and determines which nodes are hazardous by performing a dynamic timing analysis. This analysis conservatively detects that a static hazard occurs when the difference between the delays along two paths in one subcircuit is greater than the delay between two transitions. Their algorithms then *add delays* until the hazard disappears. This approach is successfully applied to small circuits where the additional delay elements result in an average delay penalty of 25 percent when comparing the results of speed-independent work done by Beerel [47] against those of Lavagno [75].

The other work in the area of technology-mapping of timed circuits was done by Myers et al. [76], where they use a decomposition and resynthesis approach applied to large fanin gC gates.

1.6 Goal

The goal of this dissertation is to implement hazard-free technology-mapping of gate-level timed asynchronous circuits by carrying the timing information provided with the circuit specifications into the technology-mapping algorithms. This is done by applying the synchronous technology-mapping process of Figure 1.1 to timed asynchronous circuits while modifying algorithms as necessary to deal with the presence of hazards. The basis of this work is the extension of the verification work done in [25] to perform efficient verification of the decomposed netlist. Verification is also done on the final netlist to ensure that any timing changes incorporated during the technology-mapping process do not introduce new hazards. This new research has the potential to markedly increase the size of timed asynchronous circuits that can be technology-mapped in an efficient and hazard-free manner.

1.7 Contributions

This dissertation makes four contributions in the area of technology-mapping of timed asynchronous circuits: adaption of the synchronous technology-mapping flow, efficient gate-level hazard verification using explicit timing, algorithms and methods to utilize gC's in circuit solutions, and evaluating library complexity for implementing hazard-free circuits.

The first contribution is to adapt the synchronous approach to technology-mapping with modifications as necessary to minimize the effects of hazards. This approach allows for the leveraging of existing research in synchronous design. The synchronous technology-mapping methods of design decomposition, partitioning, matching, and covering work well for timed circuit technology-mapping when the issues of hazard elimination are properly addressed.

The second contribution is the development of a new and highly efficient method for using explicit timing information to verify hazard-freedom in gate-level timed asynchronous circuits. Hazard verification is essential for technology-mapping and the algorithms must be extremely efficient to allow for many alternative designs to be considered. The results show that this new method can be substantially faster then previous gate-level timing verification tools.

The third contribution is the development of algorithms that optimize matching and covering in the presence of hazardous nodes. Particular emphasis is given to the use of gC's because of their compact nature and hazard-free operation. These algorithms require that issues of short-circuits and common-input pins be addressed.

The final contribution is the evaluation of library complexity for implementing the timed asynchronous circuit. Typical asynchronous approaches often require the use of specialized libraries. The use of various complexities of libraries in technology-mapping of timed asynchronous circuits gives designers more flexibility for implementing hazard-free circuits.

1.8 Dissertation Overview

Chapter 2 explains the structure of the synthesized equations that are used as inputs to the technology-mapper. Formalism is developed to support the design description methods for time Petri nets, netlists, and sum-of-products representation. Finally, the technology-mapping algorithm design flow is presented in both graphical and algorithmic forms.

Chapter 3 focuses on methods to verify hazard-freedom in a decomposed netlist. First, a state graph is formally defined. Then, the components of circuit correctness are defined including complex gate equivalence, acknowledgment hazards, and monotonicity hazards. Since the decomposition can result in a large netlist, efficient verification algorithms are developed to verify hazard-freedom. These algorithms verify each node in the circuit by first using speed-independent algorithms followed by algorithms using explicit timing information. Each node in the netlist is then annotated with nodes found to be hazardous. The chapter is concluded with examples illustrating concurrency, soft coloring, and path stabilization.

Chapter 4 presents the development of algorithms for the decomposition of the synthesized design. Since many of the existing algorithms for asynchronous technology-mapping decompose directly to library gates, this chapter follows more of the synchronous approach, discussing enhancements applicable to asynchronous designs and in particular, timed methods. These enhancements include unbalanced tree decomposition, input pin reordering, and placement of trigger signals. Issues related to decomposition of library elements are discussed including proper annotation for common-input pins and gC's.

Chapter 5 discusses the creation and application of algorithms that match the decomposed netlist to a general library and then do a covering to create the final netlist. The issue of matching with different cost factors is discussed as well as algorithms to deal with library elements with common-inputs. For gC's, the possibility of creating short-circuits in the transistor stacks is presented and illustrated with examples. Finally, algorithms are developed to address the complex issue of matching in the presence of hazardous nodes.

Chapter 6 tabulates and discusses the results from running numerous examples through the supporting software. Results from running the timed hazard verification algorithms are presented and compared against other timed circuit verifiers. The technology-mapping results are presented using libraries with increasing complexity and variety.

Chapter 7 summarizes the results of this research and discusses the successes and limitations of this technology-mapping approach. A future work section presents a number of ideas for extending this research.

CHAPTER 2

BACKGROUND AND SEMANTICS

This chapter expands on the asynchronous technology-mapping design flow for timed circuits presented in Figure 1.2. The necessary semantics and background terminology to support time Petri nets, netlists, and sum-of-products representations are developed and illustrated with examples. The chapter concludes with a top-level algorithm for timed asynchronous technology-mapping.

2.1 Time Petri Nets

As an intermediate form of circuit representation, the time Petri net models the possible input behaviors and the required output behaviors for timed circuits [14]. Let W be a finite set of wires in a timed circuit. The timed behavior of a circuit is modeled as sequences of rising and falling transitions on W. For any $w \in W$, $w+$ is a rising transition and $w-$ is a falling transition on the wire w. In the following definitions, let $\mathbb{Q}^+$ and $\mathbb{R}^+$ denote the sets of nonnegative rational and nonnegative real numbers, respectively. A W-labeled one-safe TPN is a directed bipartite graph described by the tuple $TPN = \langle W, T, P, F, M_0, s_0, l, u, L \rangle$ where:

- $W = I \cup O$ is the set of wires where I is the set of input wires and O is the set of output wires;
- T is the set of transitions;
- P is the set of places;
- $F \subseteq (T \times P) \cup (P \times T)$ is the flow relation;
- $M_0 \subseteq P$ is the initial marking;

- $s_0 \subseteq W$ is the set of wires that are initially high;
- $l : T \rightarrow \mathbb{Q}^+$ is the lower timing bound function;
- $u : T \rightarrow \mathbb{Q}^+ \cup \{\infty\}$ is the upper timing bound function;
- $L : T \rightarrow W$ is the labeling function.

The state of a TPN is a pair $\langle M, D\rangle$ where M is the current marking (i.e., the subset of places that hold tokens) and $D : T \rightarrow \mathbb{R}^+$ is a clock assignment function assigning nonnegative real valued ages to transitions. With every transition $t \in T$, its associated *preset* is $\bullet t = \{p \in P \mid (p,t) \in F\}$. The *postset* of a transition is defined as $t\bullet = \{p \in P \mid (t,p) \in F\}$. Note that the preset and postset for places are defined in a similar manner. A transition, t, is *enabled* in a state if the members of its preset form a subset of the places in the marking of the state (i.e., $\bullet t \subseteq M$). A transition, t, is *fireable* in a state if it has been enabled longer than its lower timing bound (i.e., $D(t) \geq l(t)$). A transition, t, *must fire* before it has been enabled longer than its upper timing bound (i.e., $D(t) \leq u(t)$).

A TPN for the example presented in Figure 1.6 is shown in Figure 2.1(a), with two alternative netlists shown in Figures 2.1(b) and 2.1(c). In the initial state, transitions $a+$ and $b+/1$ are enabled, and exactly one of these transitions fires

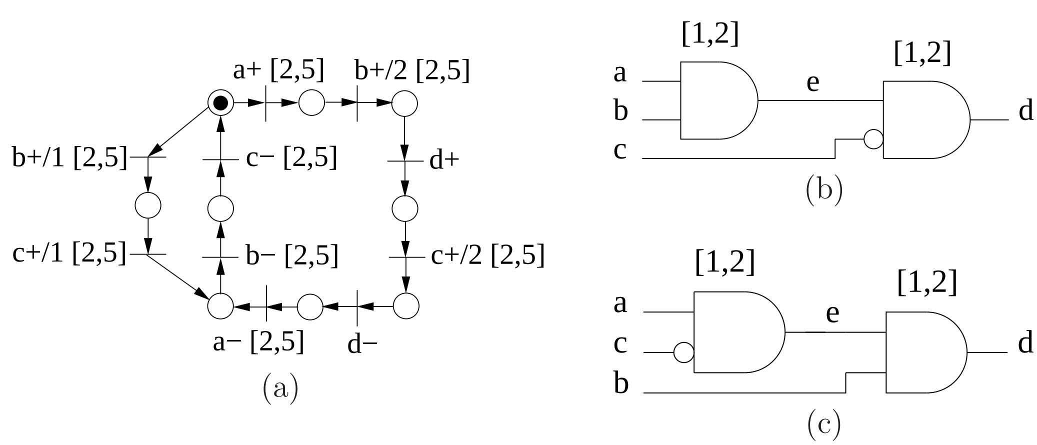


Figure 2.1. A time Petri net example. (a) An example TPN. (b) Hazardous netlist under the SI model. (c) Hazard-free netlist under the SI model.

within two to five time units. The /1 and /2 notations indicate different transitions on the same signal wire. If $a+$ fires, then the $b+/2$ transition becomes enabled and fires within two to five more time units enabling $d+$.

2.2 Netlists

The goal of the verification portion of this research is to verify the correctness of a circuit implementation against a given TPN specification. The circuit to be verified is described using a netlist modeled by a directed graph $NET = \langle V, E \rangle$ where:

- $V = I \cup O \cup N$ is the set of vertices in the circuit;
- $E \subseteq (I \cup O \cup N) \times (N \cup O)$ is the set of edges.

Each vertex $v \in V$ represents a node in the netlist. This set is composed of both the input wires, I, and output wires, O, from the TPN description as well as new nodes internal to the circuit, N. Each $e \in E$ represents a directed connection in the netlist from one node to another node. The set of *fanins* to a node is denoted by $FI(v)$, and the *fanouts* are denoted by $FO(v)$. Each node also has an associated minimum gate delay, min_v, a maximum gate delay, max_v, and an area cost, a_v.

The netlist for a possible circuit implementation of the signal d is shown in Figure 2.1(b). The set of vertices, V, is $\{a, b, c, d, e\}$, and the set of edges, E, is $\{ \ (a, e), \ (b, e), \ (e, d), \ (c, d) \ \}$. An alternative circuit implementation for signal d is shown in Figure 2.1(c). The delays shown on the gates could just as easily be shown on the wires. The primary concern, as discussed later, is with the maximum delay path from primary inputs to outputs.

2.3 Sum-of-Products Representation

The output of the synthesis analyzer is a netlist. In this netlist, each node, u, which is in $N \cup O$ has an associated gate output function $f_u(v_1, \ldots, v_r)$ where $FI(u) = \{v_1, \ldots, v_r\}$. The form of expression of f_u is a set of products, $f_u = \{\{p_1, \ldots, p_n\}\}$. Each *product*, $p \in f_u$, is a conjunction (AND) of literals. A product is also referred to as a *cube*. A *literal* is a variable v_i or its complement $\overline{v}_i$. A *sum-of-products* (SOP) is a set of products that are disjunctively combined.

For example, the function associated with node e in Figure 2.1(b) is $f_e(a, b) = \{\{a, b\}\}$. As another example, consider the function f_u, depicted in set notation as $f_u = \{\{a, b\},\{\overline{a},c\},\{a, c, d\}\}$. For clarity, this thesis often expresses functions in the Boolean formula form. In this form $f_u = ab + \overline{a}c + acd$.

For state-holding netlists, f_u is expressed as $f_u = f_u^{set} + u \bullet f_u^{\overline{reset}}$. Note that in this case, $u \in FI(f_u)$. The conditions which cause f_u to be set are denoted as $f_u^{set} = \{p \in f_u \mid u \notin p\}$. The conditions which cause f_u to be reset are denoted as $f_u^{reset} = \overline{f_u^{\overline{reset}}}$ where $f_u^{\overline{reset}} = \{p - \{u\} \mid p \in f_u \wedge u \in p\}$.

2.4 Generalized C-elements

A state-holding function, f_u, can be implemented using a *generalized C-element* (gC) implementation as shown in Figure 2.2(a). A CMOS implementation of a gC circuit is shown in Figure 2.2(b). The set function is combinational logic that

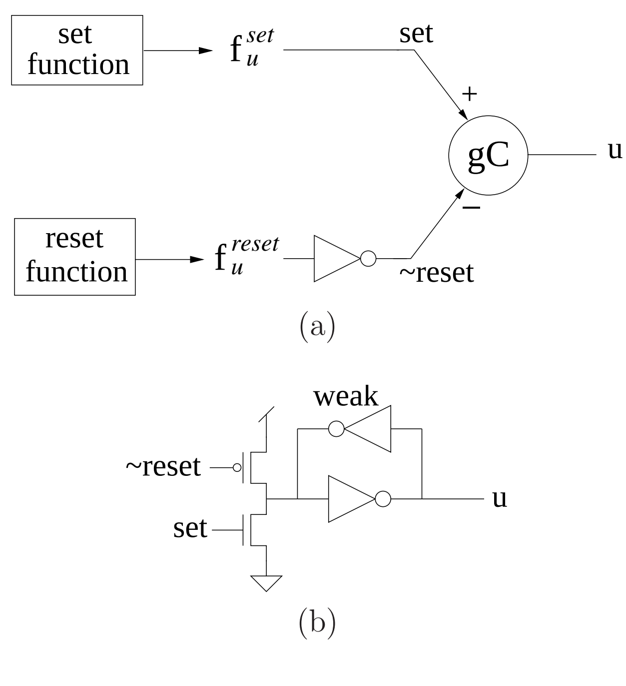


Figure 2.2. Generalized C-element structure. (a) General form of a gC circuit. (b) A CMOS implementation.

implements the function f_u^{set}, and it represents the logic that sets the output of the gC. The reset function is combinational logic that implements the function f_u^{reset}, and it represents the logic that resets the output of the gC. The set and reset functions are naturally decoupled and can be decomposed separately. If $|f_u^{\overline{reset}}| = 0$, that is, if the number of elements of $f_u^{\overline{reset}} = 0$, then the output is not state-holding and the gC is unnecessary.

The state-holding function in a gC is implemented through a head-to-tail inverter structure as shown at the output of Figure 2.2(b). The driving strength of the feedback inverter is weak compared to that of the device stack, allowing for the device stack to drive the output except in the case where the *set* input is low and the ∼*reset* input is high. In this case, the output value is held by feedback through the weak inverter.

Figure 2.3(a) shows an example of a gC circuit where $f_u^{set} = ab + cd$ and $f_u^{reset} = pq + rs$. The CMOS implementation for f_u is shown in Figure 2.3(b). Each product of f_u^{set} is implemented as a series chain of NMOS devices and all such terms are connected in parallel to implement the set portion of the netlist. For each product in f_u^{reset}, each literal is first complemented, and a new product is formed from these complemented literals. Each of these new products is implemented as a series chain of PMOS devices and all such terms are connected in parallel to implement f_u^{reset}.

An attractive feature of gC's is that they can be considerably more efficient than a fully static implementation. The netlist produced by the synthesis analyzer must ensure that there is never a time where both a set and reset term are enabled concurrently or there is a direct short between power and ground. If this is the case, it is equivalent to a signal being both high and low in the same state, or enabled to be falling or rising in the same state. However, if portions of the circuit are implemented with a gC and portions with other logic elements, there is a possibility that timing delays may cause a direct short between power and ground. If this is the case, an alternative state-holding element must be used.

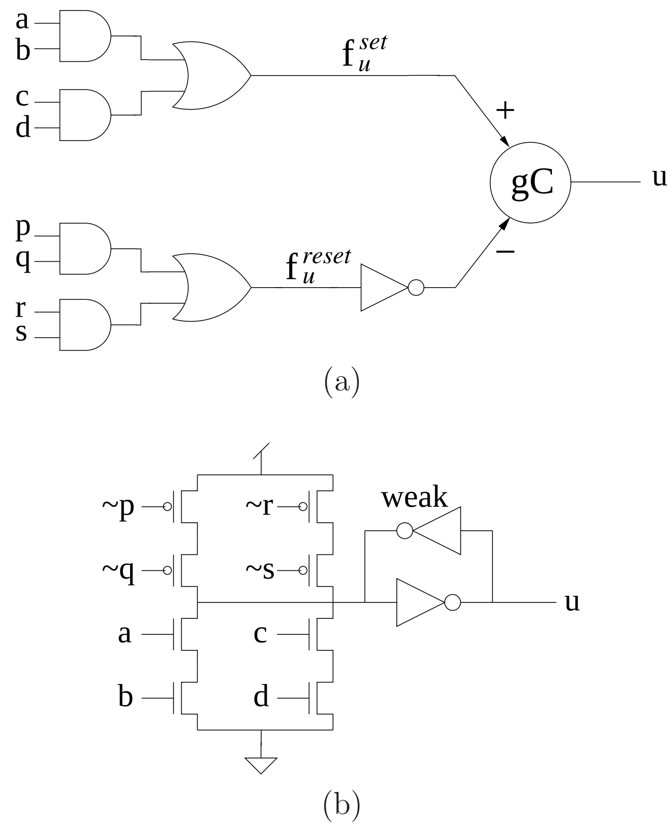


Figure 2.3. Example gC circuit. (a) Circuit of f_u. (b) CMOS implementation.

2.5 Timed Asynchronous Design Flow

The timed asynchronous design flow first presented in Figure 1.2 is shown in expanded form in Figure 2.4. The design flow begins with a time Petri net provided as a user specification. The TPN is then analyzed and a *timed state graph* is constructed. This timed state graph is synthesized using timed state-space techniques resulting in a synthesized netlist, *syn_net*. At this point, the technology-mapping phase begins. The goal of the technology-mapper is to provide a hazard-free netlist composed of library elements. The technology-mapping portion of Figure 2.4 is repeated for every $o \in O$.

The top level technology-mapping algorithm shown in Figure 2.5 takes as inputs the time Petri net, *TPN*, a synthesized netlist, *syn_net*, and a cell library, *LIB*. As discussed earlier, *syn_net* contains a pair of sets, f_u^{set} and f_u^{reset}. If $|\overline{f_u^{reset}}| = 0$, then

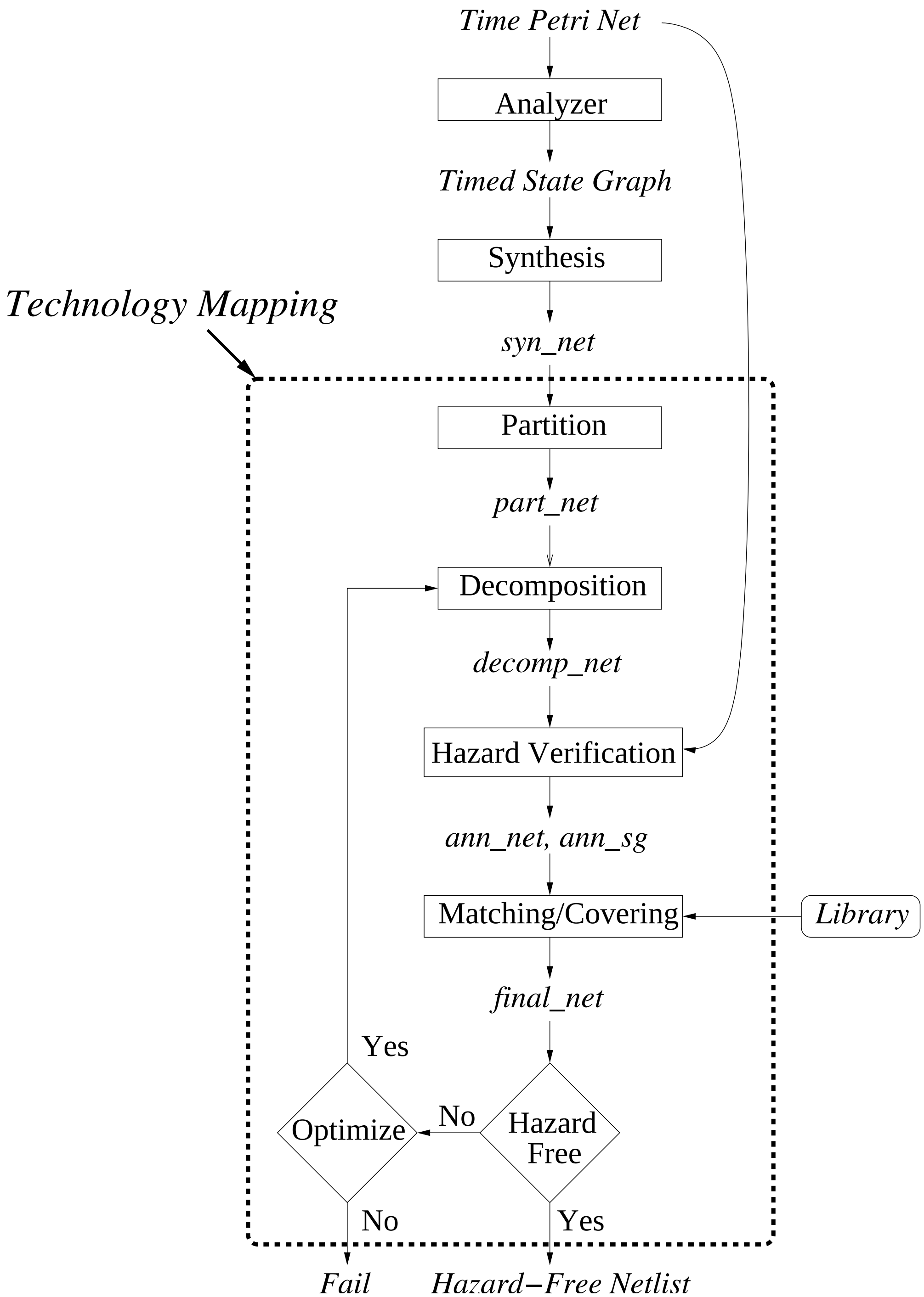


Figure 2.4. Expanded timed asynchronous design flow.

```
techmap(TPN,syn_net,LIB) {
  foreach o ∈ O {
    part_net = partition_netlist_cone(syn_net,o)
    do {
      decomp_net = decomp(part_net)
      (ann_net,ann_sg) = verify(TPN,decomp_net)
      final_net = matcov(ann_net,ann_sg,LIB)
      if (hazard_check(final_net,ann_sg) == NULL) then
        total_net = total_net ∪ final_net
        part_net = NULL
      else
        part_net = optimize(part_net)
        if (part_net == NULL) then
          report 'no hazard-free solution found for output o'
    } while (part_net != NULL)
  }
  return (total_net)
}
```

Figure 2.5. Top level technology-mapping algorithm.

the output is combinational, otherwise the output is state-holding.

All outputs in the TPN specification are treated independently by first partitioning each $o \in O$ into a single output cone of logic. This partitioning step returns the netlist, *part_net*. The algorithm then loops through the technology-mapping steps attempting to find a hazard-free implementation for output o. First, *part_net* is decomposed into a netlist consisting of base functions. Next, verification is performed on the netlist decomposition and the `verify` function returns a netlist and state graph annotated with hazard information. During the matching/covering phase, the results from verification are used to guide the selection of library elements that best eliminate hazards. This new netlist, *final_net*, is composed exclusively of library cells. Next, *final_net* is again verified for hazards because the timings specified in the TPN for the output firings are approximations until the technology-dependent final netlist is generated. If no hazards are found, the technology-mapping phase for this output is complete. If the circuit is not hazard-free, then a five-step optimization process is employed to make every effort possible to generate a hazard-free netlist.

1. Refine timing. There is an initial chicken-and-egg problem in that the timing specified for the primary outputs in the TPN are only an educated guess since the actual delay timing is not known until the circuit is mapped. If this final verification fails, it is quite likely that the initial timing bounds in the TPN can be tightened, using the minimum and maximum times taken directly from the mapped circuits. The technology-mapping process can then be reiterated until a hazard-free netlist is found, or the timings are as tight as possible and hazards still exist.

2. Pin reordering. Because of the unbalanced decomposition, the wires on *part_net* can be reordered to effect the circuit timing (while maintaining logical equivalence) and the technology-mapping process is repeated until a hazard-free netlist is found or the pin reordering permutations are exhausted. This issue is discussed in depth in Section 4.3.

3. Expand library. First, by employing a more liberal use of the atomic gate assumption, more logic can be placed within a library element, which improves the chances of reducing hazards in the mapped circuit. However, this step places an additional burden on the physical design/layout of these library cells so that they meet the atomic gate assumptions. Second, additional base function library elements with different delays could be added since the hazard verification is performed based on these timings.

4. Add delay. Using the method of Lavagno [74], once hazardous nodes have been determined, it is possible to add delay elements at the appropriate places until the hazard goes away. However, this additional delay results in decreased circuit performance.

5. Answer these questions: Does the hazard actually cause circuit failure? Is the output of *part_net* hazard-free, and if so, does hazardous activity on internal nodes matter? These philosophical questions are expanded on in Section 7.3.

CHAPTER 3

DETERMINATION OF HAZARD-FREEDOM

This chapter presents the pivotal work of efficiently performing gate-level verification for hazard-freedom at each node in the decomposed netlist. A formal presentation of state graphs is first made followed by a discussion of circuit correctness. The circuit correctness section details the topics of complex gate equivalence, acknowledgment hazards, and monotonicity hazards.

At this point, there is sufficient background to discuss the algorithms for determining hazard-freedom. This process amounts to finding stable states using two separate but related techniques: untimed (speed-independent) stabilization and timed stabilization. Once stable states are found, this stabilization information is propagated throughout the state graph. Finally, an examination of the behavior of each node is made to determine whether or not that particular node is free of acknowledgment and monotonicity hazards. Each node is then annotated with its hazard properties and this information is passed on to the matching/covering stage.

The verification method described in this chapter requires that the primary outputs must cut the circuit. In other words, if all primary outputs are removed from the netlist, the netlist would become acyclic. Intuitively, this means there can be no internal cycles in the netlist. Since the goal of this research is to use this verifier as a hazard checker during technology-mapping and the technology-mapper satisfies this restriction, this seems acceptable. However, as detailed in the future work section in Chapter 7, it is desirable to extend this work to handle netlists with reconvergent fanout and internal cycles.

3.1 State Graphs

In order to check correctness, a verification method typically uses a specification such as a TPN and a representation of the circuit implementation such as a netlist and finds all possible states represented using a *state graph* (SG). This verification method then checks the SG (often on the fly as the SG is being generated) for various correctness properties.

A SG is a labeled directed graph whose nodes are *states* and edges are *state transitions*. Formally, a SG is modeled by the tuple $SG = \langle\ S, T, \delta\ \rangle$ where

- S is the set of states;
- T is the set of transitions from the TPN;
- $\delta \subseteq S \times T \times S$ is the set of state transitions.

Each individual state $s \in S$ is modeled as a tuple $s = \langle \nu, z \rangle$ where

- $\nu \subseteq V$ is the set of wires that are high in the state;
- z is a *zone* representing timing relationships.

Timing information is described using zones that are typically represented using *difference bound matrices* (DBMs) [15]. These matrices represent time differences between recently fired transitions. Each entry, z_{ij}, in the matrix represents a timing relationship of the form $\tau_{t_i} - \tau_{t_j} \leq z_{ij}$ where τ_{t_i} is the time at which t_i fires. In other words, z_{ij} represents the maximum amount of time in which t_i fires after t_j. An example zone for the point right after $a+$ in Figure 3.1(a) fires is given below which represents the relationship $2 \leq \tau_{a+} - \tau_{c-} \leq 5$, that is, $a+$ fires between two and five time units after the previous transition, $c-$, fires.

$$\begin{array}{c|cc|} & \tau_{c-} & \tau_{a+} \\ \tau_{c-} & 0 & -2 \\ \tau_{a+} & 5 & 0 \end{array}$$

3.2 Definition of Circuit Correctness

In [25, 26], the following theorem giving sufficient conditions for correctness of a determinate speed-independent asynchronous circuit is presented (reworded here to match the notation used in this dissertation). These conditions are also sufficient for correctness of timed circuits.

Theorem 3.1. *(Sufficient conditions for correctness) Let* $NET = \langle V, E \rangle$ *be a circuit implementing the behavior described by* $TPN = \langle W,T,P,F,M_0,s_0,l,u,L \rangle$*. The* NET *is a correct implementation of the* TPN *if (1) it is complex gate equivalent to the TPN, and (2) it satisfies the acknowledgment and monotonicity properties.*

3.2.1 Complex Gate Equivalence

The first condition for circuit correctness is that the NET be *complex gate equivalent* (CGE) to the TPN. Using a timed state space exploration algorithm such as the ones in [21, 77], it is possible to derive a SG using a TPN to drive the inputs and check the outputs, and a netlist to drive the outputs. However, one of the key results of this research is that this method never explicitly derives this SG. Instead, a SG for a CGE version of the netlist is derived. The CGE circuit for both netlists in Figures 3.1(a) and 3.1(b) is shown in Figure 3.1(c). The SG found using this circuit and the TPN in Figure 3.1(d) is shown in Figure 3.1(e). Each state vector ν is labeled in the state diagram to show the value of all signal wires. The zones calculated during the timed state space exploration algorithm are omitted for clarity. Each edge of the state graph is labeled with a signal transition $t \in T$. The input wire set is $\{a, b, c\}$ and the output wire set is $\{d\}$. There are nine states including 0000 and 1000, and ten state transitions including $(0000, a+, 1000)$. One detail to note is that during the state space exploration to derive this SG, this method checks that the given CGE circuit is equivalent to the desired one. For example, if the CGE circuit given had been the one in Figure 3.1(f), after $a+$ fires, the netlist could produce a $d+$ when one is not expected in the TPN. This complex gate equivalence failure would then be reported to the user.

The number of internal nodes created during decomposition can be large, each

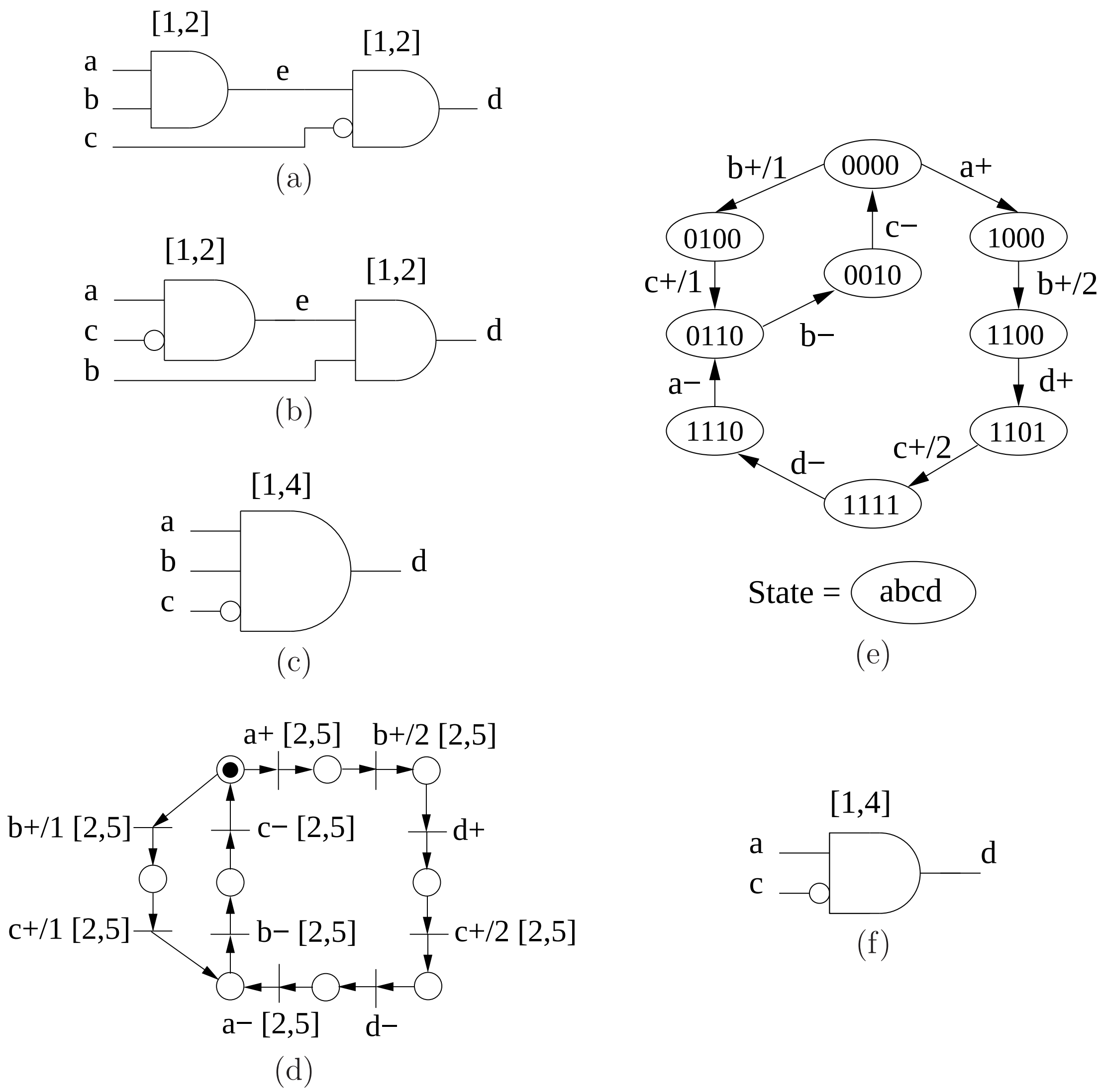


Figure 3.1. Example CGE circuit. (a) Hazardous netlist under the SI model. (b) Hazard-free netlist under the SI model. (c) Correct CGE netlist. (d) Time Petri net. (e) State graph for the correct CGE netlist. (f) Incorrect CGE netlist.

node potentially doubling the state-space. In reality, each newly added node only affects a portion of the state graph. An example of this is shown in Figure 3.2, the explicit SG for the netlist of Figure 3.1(b). Here the number of states increased from 9 to 12 as compared to the SG of Figure 3.1(e). This reflects the fact that the analysis algorithms determined a number of the potential new states to be unreachable. The power of using the CGE state graph is that the behavior of

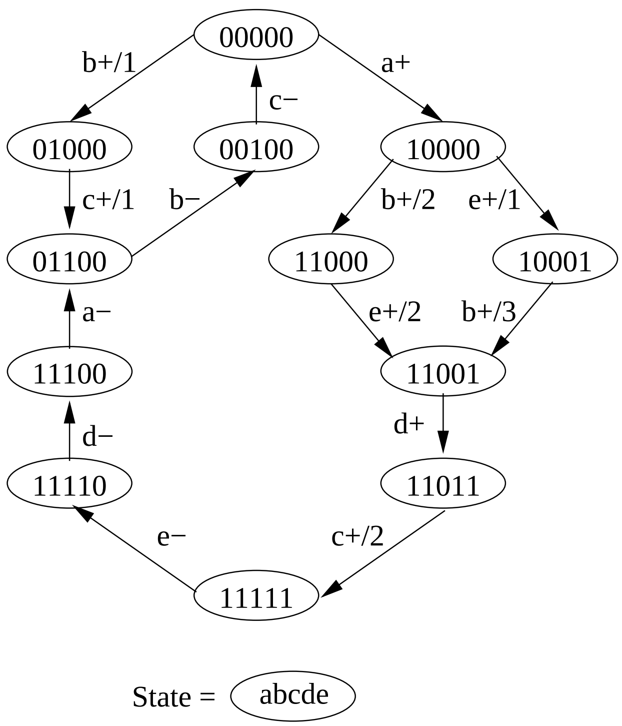


Figure 3.2. Explicit state graph.

internal nodes is abstracted. Adding internal nodes explicitly is unnecessary and keeps the SG from growing in complexity.

3.2.2 Acknowledgment Hazards

The second condition for circuit correctness requires determining if internal nodes adhere to the acknowledgment property. Hazards can manifest in asynchronous circuits due to violations in the *acknowledgment* or *monotonicity* properties [25].

An acknowledgment violation occurs when an internal node becomes excited to change to a new value, but the conditions that caused the excitation change before the node can be shown to have stabilized. An example of a circuit with an acknowledgment hazard is shown in Figure 3.3(a) and the hazard manifests under the timing shown in Figure 3.3(b). This 3-gate circuit implements the function $g = abcd$. The output g should never be enabled to go high because there is no time shown in Figure 3.3(b) where all four inputs are simultaneously high. Previous

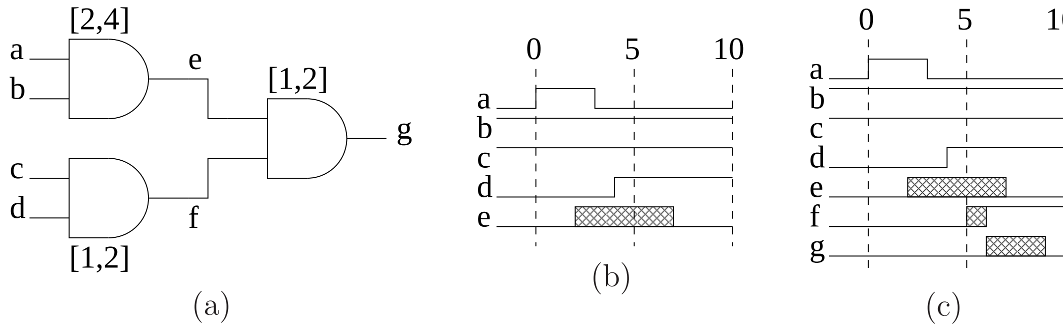


Figure 3.3. Circuit to illustrate hazards. (a) Example circuit. (b) Timing scenario for an acknowledgment hazard. (c) Timing scenario for a monotonicity hazard.

to time zero, input signals a and d are low, and b and c high. This forces the internal nodes, e and f, and the output g to be low. At time zero, input a pulses high for three time units, enabling node e to rise. However, since the delay of the AND gate driving node e has a time delay between two and four time units, it is not certain whether or not node e actually rises before signal a goes low. This represents an uncertainty on node e in response to the pulse on a and the possible failure to *acknowledge* the transition on a indicates an acknowledgment hazard on node e. If input timings and gate delays allow this possible glitch on e to propagate to the output, a *monotonicity* hazard is created on the output g as described in the following section.

3.2.3 Monotonicity Hazards

The third condition for circuit correctness requires the determination of whether or not internal or output nodes adhere to the monotonicity property. A monotonicity violation occurs when an internal or output node is supposed to remain stable but it becomes momentarily excited or when it is supposed to make a transition but it makes the transition nonmonotonically. This occurs when a gate has a *potential hazard* while there is no stable, forcing side-input. For example, a potential hazard exists when the output of an AND gate is supposed to remain stable low or fall, but one input is rising. If a side-input cannot be found that is stable low while the other input is rising, it is possible that the AND gate may momentarily evaluate to 1 causing a glitch on the output of the AND gate.

In the example shown in Figure 3.3, it is possible that the glitch on node e discussed in the previous section can propagate to the output. Consider the timing diagram shown in Figure 3.3(c). After input a goes low at time three, input d rises at time four. This causes node f to be enabled to rise and can do so as early as time five. At time five, both inputs to the AND gate driving the output are in transition and there is the possibility for a glitch on the output. This timing scenario represents a monotonicity hazard on the output g.

3.3 Hazard Verification Algorithm

The correctness conditions presented in Section 3.2 are verified using the algorithm shown in Figure 3.4. This algorithm takes as input a TPN representing the possible input behavior and the required output behavior and a netlist, NET, representing the circuit to be checked. It then determines if the circuit is correct. When the circuit is not correct, this algorithm reports the locations of the errors that it finds. This algorithm is described in detail in the remainder of this section.

3.3.1 Checking Equivalence

The `check_equivalence` function forms a CGE netlist, uses this netlist and the given TPN to derive a SG, and checks if the CGE netlist provides outputs at the times specified by the TPN.

The first step is to derive a CGE netlist in which there are no internal signals. In other words, it derives a netlist that has one gate per primary output signal. The Boolean function for this gate is expressed only in terms of the primary inputs

```
verify(TPN,NET) {
  SG = check_equivalence(TPN,NET)
  find_stable_states(TPN,SG,NET)
  check_acknowledgment(SG,NET)
  check_monotonicity(SG,NET)
}
```

Figure 3.4. Top level algorithm for verification.

and outputs. The delay of this gate is set to the minimum and maximum delay from *any* input to the primary output. Although false paths through the logic may exist, this algorithm need not identify them at this point. Their inclusion results in a higher and thus more conservative maximum delay. At worst, this may result in a node being falsely determined to be hazardous.

The CGE representation for the netlists shown in Figures 3.1(a) and 3.1(b) is shown in Figure 3.1(c). The delay for this gate is set to [1,4], since in both cases there exists a minimum delay path of one time unit and a maximum delay path of four time units.

Using this CGE netlist and the TPN of Figure 3.1(d), the SG of Figure 3.1(e) is found using a timed state space exploration algorithm. During the course of this state space exploration, output firings are checked. If an output fires prematurely, an error is reported to the user. Also, if an output is expected and the circuit does not provide one, an error is reported. In the example of Figure 3.1, if the CGE function $f_d = ac$ is used, after $a+$ and $b+$, a $d+$ would be expected in the TPN of Figure 3.1(d), but the circuit would not produce it. This models a progress condition similar to *completeness with respect to specification* [78] and *strong conformance* [79]. When no errors are detected, `check_equivalence` returns a SG.

3.3.2 Finding Stable States

After the `check_equivalence` step, this algorithm has shown that the circuit is correct at a complex gate level. By hiding the internal signals before finding the state space, the state space is potentially reduced from $O(2^{|I|} * 2^{|O|} * 2^{|N|})$ to $O(2^{|I|} * 2^{|O|})$. When the number of internal signals is large, as is often the case in real designs, this savings can be quite dramatic. However, hazards on internal nodes can still produce incorrect circuit behavior. Therefore, it is now necessary to check that all internal nodes are hazard-free. This is accomplished by determining internal signal behavior implicitly. In particular, this method annotates each state with stability information about each internal signal. The goal of the `find_stable_states` algorithm is to determine in which states and for which state transitions in the complex gate SG that each internal node is stable. This is accomplished by using the predicate `stable`(s,n) for each state $s \in S$ and node $n \in N$ and the predicate `stable`(s,s',n) for each state transition $(s,t,s') \in \delta$. This stability information can then be used to determine if there are any hazards in the given netlist.

The algorithm to find the stability information is shown in Figure 3.5. The algorithm begins by first determining the predicate `eval`(s,n) by finding the Boolean evaluation in each state in the SG for each node in the netlist. This can be accomplished by simply fixing the values for each primary input and output in the netlist to the values given in the state and propagating this information through the netlist. From the SG in Figure 3.1(e) and the netlist in Figure 3.1(b), `eval`(1100,e) and `eval`(1100,d) are determined to both be 1. For node e, the states in the set {1100, 1101, 1111, 1110} evaluate to 1 while the remaining states evaluate to 0.

The algorithm next initializes the stability predicates to FALSE to initially indicate that it is not known whether the internal signals are stable or changing. The goal of the rest of the algorithm is to determine stability of the internal signals, whenever possible. In the next section, a brief review of untimed stabilization is given, which comes from the work in [25] followed by a detailed discussion of timed stabilization, one of the main contributions of this research. The timed stabilization routine does not need to be iterated, so it is executed first. The untimed stabilization routine may require iteration since stabilizations on one node

```
find_stable_states(TPN,SG,NET) {
  foreach s ∈ S and n ∈ V find eval(s,n)
  foreach n ∈ N, s ∈ S, and (s,t,s') ∈ δ,
    stable(s,n) = stable(s,s',n) = FALSE
  stabilize_timed(TPN,SG,NET)
  do
    distribute(SG)
    modified = stabilize_untimed(SG,NET)
  while modified
}
```

Figure 3.5. Algorithm for finding stable states.

of the network can influence stabilizations on other nodes.

3.3.3 Untimed Stabilization

The objective of stabilization is to show that at some points in the SG, the evaluations of some internal node, n, are certain to be stable. The algorithm to determine untimed stability is shown in Figure 3.6. An internal node is considered untimed stable if a change in evaluation on an internal node is acknowledged on a primary output. In other words, for a state transition (s, t, s'), if the transition t could only have occurred if the internal node n is stable at its Boolean evaluation, then it can be said that the transition t has acknowledged that the node n is stable.

To determine if an internal node n is acknowledged to be stable by a state transition (s, t, s'), it is first determined using the function `exists_path`, if a path exists from n to the output transition under consideration. It must then be determined using the function `must_prop` if the value at n must propagate through any possible path to the output. This is done by ensuring that all functions in the path from n to the output have noncontrolling values on the side inputs. Consider the example netlist in Figure 3.1(a) and the state transition (1100,d+,1101). There exists a path between node e and the output d. In state 1100, node e evaluates to 1. This value at e must propagate to the output because d cannot go high until e has gone high. More succinctly, output d switched from low to high as a direct consequence of node e going high and the side input, c being at 0. Therefore,

```
stabilize_untimed(SG,NET) {
  modified = FALSE
  foreach n ∈ N
    foreach (s,t,s') ∈ δ
      if ((exists_path(NET,n,L(t))) and (must_prop(NET,s,n,L(t))) and
          (not stable(s,s',n))) then
        stable(s,s',n) = TRUE
        modified = TRUE
  return modified
}
```

Figure 3.6. Untimed stabilization algorithm.

`stable`(1100,1101,e) is set to TRUE.

The `distribute` function is used to copy this stabilization forward in the state graph until a change in evaluation is encountered. In particular, `stable`(1100, 1101, e) implies the following stability conditions are TRUE: `stable`(1101,e), `stable` (1101,1111,e), `stable`(1111,e), `stable`(1111,1110, e), `stable`(1110,e), and `stable` (1110,0110,e). This distribution of stability information halts when it reaches state 0110 since the Boolean evaluation of e in this state changes from 1 to 0.

The other transition in the SG that could possibly indicate an untimed stabilization for node e is the state transition (1111, $d-$, 1110). In this case, however, the input c is 1 (a controlling value), prohibiting the propagation of node e to the output d. In addition, node e was already shown to be stable high in states 1111 and 1110. Thus, no stabilization can be assumed for the falling transition of e. As explained later, this lack of stabilization on the falling transition of e indicates a hazard on node e.

A similar analysis done on the circuit in Figure 3.1(b) shows that the rising transition on node e is acknowledged by $d+$ and the falling transition is acknowledged by $d-$ since b is high (a noncontrolling value) when d goes low. As a result, this circuit can be shown to be hazard-free under the speed-independent model.

3.3.4 Timed Stabilization

When timing information is taken into account, the hazard found for the netlist shown in Figure 3.1(a) may not actually manifest. If this is the case, then node e is hazard-free. This section describes a new method to determine stabilization using timing information. Timed stabilization attempts to show further stability in the state graph by calculating the maximum possible time through the network to the node of interest, n, and comparing this against the minimum time spent traversing the state graph. When it can be shown that in the worst-case a sufficient amount of time has elapsed, node n can be stabilized.

The algorithm to determine timed stabilization is shown in Figure 3.7. For each node n, the algorithm first measures the longest path delay from any primary input or output to the node n. This must be done because the actual signal that causes

```
stabilize_timed(TPN,SG,NET) {
  foreach n ∈ N
    d = find_max_delay(NET,n)
    foreach s ∈ S
      visit(s) = FALSE
    foreach (s,t_i,s') ∈ δ where s = ⟨v,z⟩
      if (eval(s,n) ≠ eval(s',n)) then
        z' = update_zone(TPN,NET,z,t_i)
        foreach s'' ∈ S
          path(s'') = FALSE
        do_timed(TPN,SG,NET,n,s',z',t_i,d,visit,path)
}
```

Figure 3.7. Timed stabilization algorithm.

n to change evaluation may not be known due to differences in path lengths. For the example netlist in Figure 3.1(a), this delay is determined to be 2. Next, the algorithm initializes the `visit` array that is used to let the recursion know when a state has been visited along multiple paths when determining stabilization of node n. At this point, the algorithm finds state transitions, (s, t_i, s'), where the Boolean evaluation of n changes. This indicates locations in the state graph where the node n becomes unstable. The algorithm then takes the zone z associated with state s and updates it to include the transition t_i. The reason this is done rather than taking the zone associated with s' is that t_i may have been pruned from this zone. It is important that t_i is in the zone that is used for timed stabilization as t_i serves as a reference transition as the algorithm moves forward in the state graph. Finally, the algorithm initializes a `path` array that is used to terminate cycles during the analysis of a path in the SG.

The `update_zone` algorithm shown in Figure 3.8 adds a new transition to a given zone. The first step is to extend the zone to include a new row and column for the new transition, t_i. Next, it searches the zone starting with the transitions that have been added most recently for transitions that enable t_i (i.e., $t_j \in \bullet\bullet t_i$). The first such transition that it finds is the causal transition for t_i. The upper bound of the firing time for t_i should be set in reference to this transition. The upper bound is either taken from the TPN when t_i is a transition on an input wire or it

```
update_zone(TPN,NET,z,t_i) {
  z' = extend(z,t_i)
  found_causal = FALSE
  foreach t_j ∈ z' in reverse order
    if (t_j ∈ ••t_i) then
      if (!found_causal) then
        found_causal = TRUE
        if (L(t_i) ∈ I) then
          z'_ij = u(t_i)
        else
          z'_ij = find_max_delay(NET,L(t_i))
      if (L(t_i) ∈ I) then
        z'_ji = (−1) ∗ l(t_i)
      else
        z'_ji = (−1)∗ find_min_delay(NET,L(t_i))
    else
      z'_ij = ∞
      z'_ji = ∞
  recanonicalize(z')
}
```

Figure 3.8. Algorithm to update the zone.

is taken as the maximum delay in the netlist generating t_i when it is a transition on an output wire. For all transitions that enable t_i, a lower bound must be set between t_j and t_i. For all transitions that do not enable t_i, the timing relationships are initially set to be unbounded. At this point, the zone is recanonicalized using Floyd's all-pairs shortest path algorithm to tighten any loose inequalities. This recanonicalization step is necessary because tightened bounds increase accuracy. In addition, there are often cases where no timing relationship is known between a newly entered transition and the other entries in the zone. Recanonicalization creates these entries in the zone. As an example, the zone found for the state 1110 in the example is shown in Figure 3.9(a). The new zone after adding the transition $a-$ is shown in Figure 3.9(b). The `do_timed` algorithm shown in Figure 3.10 is used to recursively explore the SG, attempting to accumulate sufficient time to stabilize a given node n before reaching a termination condition. This algorithm first marks the current state s as visited in the `visit` and `path` arrays described earlier. Next,

$$\begin{array}{c|c|} & \tau_{d-} \\ \tau_{d-} & 0 \end{array}$$

(a)

$$\begin{array}{c|cc|} & \tau_{d-} & \tau_{a-} \\ \tau_{d-} & 0 & -2 \\ \tau_{a-} & 5 & 0 \end{array}$$

(b)

$$\begin{array}{c|ccc|} & \tau_{d-} & \tau_{a-} & \tau_{b-} \\ \tau_{d-} & 0 & -2 & \infty \\ \tau_{a-} & 5 & 0 & -2 \\ \tau_{b-} & \infty & 5 & 0 \end{array}$$

(c)

$$\begin{array}{c|ccc|} & \tau_{d-} & \tau_{a-} & \tau_{b-} \\ \tau_{d-} & 0 & -2 & -4 \\ \tau_{a-} & 5 & 0 & -2 \\ \tau_{b-} & 10 & 5 & 0 \end{array}$$

(d)

$$\begin{array}{c|cccc|} & \tau_{d-} & \tau_{a-} & \tau_{b-} & \tau_{c-} \\ \tau_{d-} & 0 & -2 & -4 & \infty \\ \tau_{a-} & 5 & 0 & -2 & \infty \\ \tau_{b-} & 10 & 5 & 0 & -2 \\ \tau_{c-} & \infty & \infty & 5 & 0 \end{array}$$

(e)

$$\begin{array}{c|cccc|} & \tau_{d-} & \tau_{a-} & \tau_{b-} & \tau_{c-} \\ \tau_{d-} & 0 & -2 & -4 & -6 \\ \tau_{a-} & 5 & 0 & -2 & -4 \\ \tau_{b-} & 10 & 5 & 0 & -2 \\ \tau_{c-} & 15 & 10 & 5 & 0 \end{array}$$

(f)

Figure 3.9. Zone creation and evolution.

```
do_timed(TPN,SG,NET,n,s,z,t_i,d,visit,path) {
  visit(s) = path(s) = TRUE
  foreach (s,t_k,s') in δ
    z' = update_zone(TPN,NET,z,t_k)
    if (-1*z'_ik > d) then
      if (not visit(s')) then
        stable(s,s',n) = TRUE
    else if (not path(s') and eval(s,n) == eval(s',n)) then
      stable(s,s',n) = FALSE
      do_timed(TPN,SG,NET,n,s',z',t_i,d,visit,path)
  path(s) = FALSE
}
```

Figure 3.10. Timed stabilization recursion.

it considers each state transition (s, t_k, s'). First, it adds the transition, t_k, to the zone. Next, it checks the zone to determine if enough time has accumulated from the reference transition t_i to the new transition t_k such that the node of interest n has certainly stabilized. If it has, it must also check that the state s' has not been visited along a different path. It must be the case that the minimum time upon reaching a state along all paths to that state has exceeded the maximum logic delay d. Therefore, if this state is encountered along a different path and did not stabilize, then this state transition cannot stabilize the node n. If the amount of accumulated delay does not exceed the delay d, then the algorithm must determine if it is going to recurse down this state transition. If this state has been seen previously upon this path, the algorithm has encountered a cycle of states and must not recurse. If the Boolean evaluation of the node n has changed, then again the algorithm must not recurse. If this is a new state on this path and the Boolean evaluation is maintained, then the algorithm recursively visits the state s'. Note that this edge may have been found to be stable along a different path, but it is not stable along the path the algorithm is currently working on. Therefore, the algorithm must say this edge is not stable before recursing. Upon returning from recursion, the path variable is set to false to allow other potential paths to visit the state s.

This algorithm has the potential for requiring the exploration of a large number of paths, although experimental results have not shown this to happen. The further the algorithm recurses through the state graph, the more potential side paths there are to explore. Typically, the length of the paths explored is very short as the recursion terminates quickly. If in the future, examples are found where this is not the case, the algorithm can be changed to limit the path length. This can improve efficiency at the potential cost of more false negative results.

It is now useful to again consider the example netlist in Figure 3.1(a). A change in evaluation on node e occurs between states 1110 and 0110. As mentioned previously, the `do_timed` function is called with the zone shown in Figure 3.9(b). As the SG is traversed, the next transition encountered is $b-$. Since $b-$ fires two to five time units after $a-$, these entries are entered into the appropriate rows and columns as shown in Figure 3.9(c). The timing of the other nondiagonal entries are set to ∞. The zone is then recanonicalized and the resulting zone is shown in Figure 3.9(d). The parameter of interest is the minimum elapsed time between the last transition entered, $b-$, and the initial transition $a-$, which is 2 in this case. Note that lower bounds appear as negative values in a DBM. Since two time units is insufficient time to say with certainty that node e has stabilized, the algorithm considers recursing on state 0010. Since this state has not yet been explored on this path, and since node e still evaluates to 0 in this state, the algorithm recurses to state 0010. Upon recursion, the algorithm adds transition $c-$ to the zone as shown in Figure 3.9(e) and recanonicalizes to obtain the zone shown in Figure 3.9(f). The new minimum time elapsed from $a-$ till $c-$ is four time units. Since this number is larger than the maximum delay of the AND gate (two time units), the algorithm can mark this edge as stabilized. The `distribute` function then copies this stabilization onto states 0000, 0100, and 1000 and edges $(0000, b+/1, 0100)$, $(0100, c+/1, 0110)$, $(0000, a+, 1000)$, and $(1000, b+/2, 1100)$. This is significant in that the hazard condition that existed after untimed stabilization cannot manifest because of the timing relationships between the circuit and the SG.

3.3.5 Acknowledgment Hazards

As mentioned in Sections 3.2.2 and 3.2.3, hazards can manifest in asynchronous circuits due to violations in the *acknowledgment* or *monotonicity* properties. The final step in the verification process is to check each node for violations in these properties.

The algorithm shown in Figure 3.11 uses stability information in each state

```
check_acknowledgment(SG,NET) {
  foreach (n ∈ N)
    foreach (s,t,s') ∈ δ
      if ((eval(s,n) ≠ eval(s',n)) and !stable(s,s',n)) then
        report acknowledgment hazard on n for (s,t,s')
}
```

Figure 3.11. Algorithm to check for acknowledgment hazards.

s and each state transition δ to check for acknowledgment on all excited nodes. The algorithm examines each node n and each state transition (s, t, s') in which n changes Boolean evaluation. If n is not stable before it changes Boolean evaluation, then an acknowledgment hazard is reported.

3.3.6 Monotonicity Hazards

The algorithm to check a netlist for monotonicity violations is shown in Figure 3.12. Monotonicity violations are caused on a node, $n \in (N \cup O)$, by one of its fanins, $v \in \mathrm{FI}(n)$, in a particular state, $s \in S$. For each node, all states in the SG are are applied to the netlist. At this point, the algorithm constructs a cube formed from all $n \in (I \cup O \cup N)$ and applies it to f_n to determine if, given what is known about the current state, the value of node n is being forced to a known value. The function, cube(s), is a cube formed from the values determined on each $n \in (I \cup O \cup N)$ using the state vector and what is known about internal nodes from the `stable` and `eval` predicates. Note that cube(s) represents a set of implementation states. More formally, cube(s) is defined below for each node v:

$$\mathit{cube}(s)(v) = \begin{cases} s(v) & \text{if } v \in I \cup O \\ \mathit{eval}(s, v) & \text{if } v \in N \wedge \mathit{stable}(s, v) \\ X & \text{otherwise} \end{cases}$$

Next, cube(s) is applied to the function f_n. f_n(cube(s)) is written to denote f_n(cube(s)(v_1), cube(s)(v_2), ..., cube(s)(v_r)). In other words, the value of each fanin in the cube is extracted and applied to the function f_n. Since some values

```
check_monotonicity(SG,NET) {
  foreach n ∈(N ∪ O)
    foreach s ∈ S
      if (f_n(cube(s)) = 'X') then
        foreach v ∈ FI(n)
          if (potential_hazard(s,n,v)) then
            report monotonicity hazard on n for (s,v)
}
```

Figure 3.12. Algorithm to check for monotonicity hazards.

applied to the function may be X, the function f_n may return X. For example, if $f_n(a,b) = ab$, $f_n(0,\text{X}) = 0$ while $f_n(1,\text{X}) = \text{X}$.

If the value at node n is determined to be unknown, then all fanins of node n are checked for potential hazards. The algorithmic definition for `potential_hazard` shown in Figure 3.13 is modified from [25] to fit the definitions used in this research. To help in the evaluation of potential hazards, a potential cube, pcube(s,v)(u), is formed as follows:

$$pcube(s,v)(u) = \begin{cases} cube(s)(v) & \text{if } u \neq v \\ eval(s,u) & \text{if } u = v \end{cases}$$

The potential cube is equivalent to cube(s) except at node v, which is set to its final evaluation.

The `bitcomp` function referenced in Figure 3.13 takes as arguments a state, s, and node, $v \in (I \cup O)$, and returns a new state that has the bit v complemented. This new state and the node n then become arguments to the predicate `eval`.

The absence of a potential hazard is determined by examining the conditions that prevent it from occurring. There are four such conditions shown in the algorithm of Figure 3.13 and briefly described below. A potential hazard cannot occur on node n in state s for fanin v:

```
potential_hazard(s,n,v) {
  if (v ∈ N ∧ stable(s,v)) then
    return FALSE
  if ((v ∈(I ∪ O)) ∧ (eval(s,n) ≠ eval(bitcomp(s,v),n))) then
    return FALSE
  if ((f_n(pcube(s,v)) = 'X') ∧ !stable(s,n)) then
    return FALSE
  if (f_n(pcube(s,v)) = eval(s,n)) then
    return FALSE
  return TRUE
}
```

Figure 3.13. Algorithm to check for a potential hazard.

1. if v is an internal node and is stable in state s.

2. if v is a primary input or output and the evaluation of n changes when v is complemented.

3. if $f_n(\text{pcube}(s,v))$ does not indicate that n is being forced to a known value and n is not stable in state s.

4. if $f_n(\text{pcube}(s,v))$ is equal to the Boolean evaluation of n in state s.

Condition 1 implies that potential hazards can only be caused by internal nodes if they are unstable. Condition 2 implies that potential hazards are only caused by external nodes when changing their value does not result in a change in evaluation. Condition 3 indicates that there is no potential hazard when node n is not being forced to some value if n is not stable to begin with. Finally, condition 4 implies there is no potential hazard if setting node v to its final evaluation forces node n to its final evaluation. If all four of these conditions cannot be met, then a potential hazard exists and a monotonicity hazard is reported on node n in state s caused by fanin v.

An example circuit to illustrate monotonicity hazards is shown in Figure 3.14(a). This example is named *rpdft* and is taken from a suite of examples used by the timed automata tool `KRONOS` [80, 81]. The monotonicity hazard occurs on node $b58$ in state 00001 and is caused by the $b49$ fanin. The state graph segment where this hazard occurs is shown in Figure 3.14(b). In state 00101, signal $b49$ is high and signal $b48$ is low. Thus, signal $b58$ is held low by $b49$. When input b falls and the circuit transitions to state 00001, $b48$ is enabled to rise and $b49$ is enabled to fall, both after one gate delay. If $b49$ falls before $b48$ rises, then $b58$ is enabled to rise when it should stay stable low. This represents a monotonicity hazard on node $b58$ caused by its $b49$ fanin. There is an additional monotonicity hazard in this circuit on node $b58$ that is caused by fanin $b48$ in state 01101. Starting in state 01001 as shown in Figure 3.14(c), signals $b49$ and $b58$ are both low while signal $b48$ is high. When signal b rises, a transition is made to state 01101. This enables $b49$ to rise and $b48$ to fall. If $b48$ falls before $b49$ rises, then $b58$ is enabled to rise when it should have stayed low. This again represents a monotonicity hazard, this one caused on node $b58$ by its fanin $b48$ in state 01101. Note that in both these cases, the output is not affected by these internal monotonicity hazards.

Finally, there is an additional monotonicity hazard reported on the output t in state 10001 by its fanin $b58$. Starting in state 00001 as shown in Figure 3.14(d), signal $b58$ is low and signals $b55$ and output t are high. When signal d falls, the circuit transitions to state 10001, signal $b55$ is enabled to fall (through two gate delays) and signal $b58$ is enabled to rise (through three gate delays). However, it is not possible to determine the order in which these internal nodes switch. Thus, in state 10001, neither $b55$ or $b58$ has been stabilized, and it is determined that a monotonicity hazard occurs on the output t. This last monotonicity hazard is discussed in more detail in Chapter 6 because this particular hazard represents a *false negative* hazard. When a full timed state-space exploration is done, it is shown

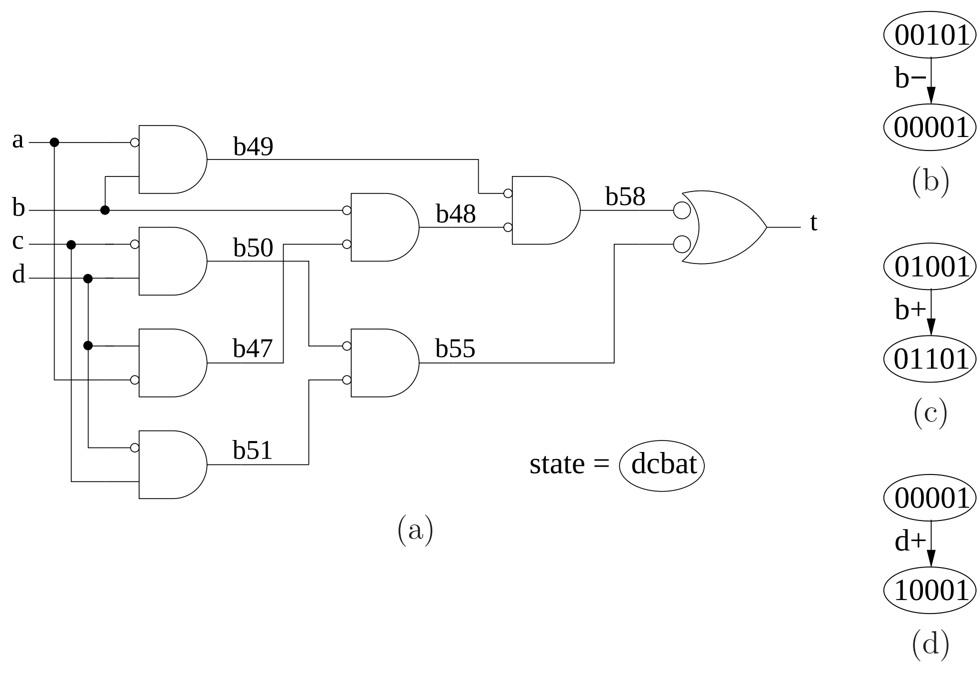


Figure 3.14. Example to illustrate monotonicity hazards. (a) *rpdft* benchmark circuit. (b) First state graph segment showing a monotonicity error. (c) Second state graph segment. (d) Final state graph segment.

that this hazard does not actually exist because the switching order of nodes $b55$ and $b58$ eliminates the possibility of a monotonicity hazard on the output.

CHAPTER 4

DECOMPOSITION

Decomposition transforms the synthesized netlist representation into a logically equivalent network that consists entirely of base functions. This process is often done in synchronous systems by repeatedly applying DeMorgan's laws and the associative law. The same decomposition algorithm that is applied to the subject graph must also be applied to the implementation library resulting in pattern graphs for the library cells. This insures that a subgraph of the subject graph can then be structurally matched to the pattern graphs in the library.

This chapter first presents the decomposition algorithm derived for use with timed asynchronous circuits. A discussion of decomposition to base functions is presented, followed by discussions of unbalanced trees and the use of inverter pairs. Next, the issue of annotating each node with hazard information is presented followed by a section on reordering of the input pins. Finally, issues of library creation are discussed including common-input pins and input name permutations.

4.1 Decomposition Algorithm

If the synchronous approach is applied to timed asynchronous circuits, hazards are likely created on newly formed nodes. These nodes must be annotated with the type of hazard that occurs (as discussed in Chapter 3) so that the matching stage can deal with them appropriately. The timed asynchronous decomposition approach developed in this chapter uses unbalanced trees when decomposing f_u^{set} and f_u^{reset}. This approach allows for input pin reordering in cases where hazardous nodes cannot be properly covered in the following stage of technology-mapping.

Decomposition guarantees a solution in the matching and covering stages as long as the base functions are available in the implementation library. In this case, a trivial solution can always be be found for the subject graph. Different heuristics can be applied to decomposition algorithms depending on the desired optimization parameters. Care must be taken, however, as a structural decomposition may not be unique and the resulting netlist may affect the quality of the final result. Most, if not all, technologies have limitations on the number of fanins allowed per gate. In CMOS technology, for instance, performance is degraded considerably when the gate fanin exceeds four. As a result, implementation libraries typically exclude any gate exceeding this fanin.

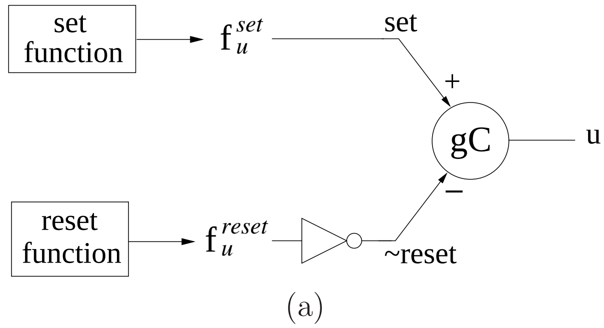
Decomposition is performed using repeated applications of DeMorgan's theorem and the Boolean associative law. Both of these operations are known to be hazard preserving for Huffman style circuits [57]. In other words, if the original circuit is hazard-free, than the decomposed circuit is also hazard-free. For these cases, the technology-mapper must adhere strictly to these Boolean laws. Removing any redundant logic may introduce hazards.

Unfortunately, this same decomposition algorithm as applied to SI circuits (and by extension to timed circuits) is not hazard preserving as illustrated in Figure 1.6. As a result, each internal node created in the decomposition must be checked for hazard-freedom.

4.1.1 Decomposition to Base Functions

The output of synthesis is a function, f_u, first presented in Section 2.3 whose gC implementation form is shown again in Figure 4.1(a). Although highly unlikely, a simple one cell implementation is possible providing that a gC library cell is available that covers the entire subject graph. The whole crux of the technology-mapping problem is that these complex cells are seldom available and f_u must be decomposed into gates simple enough that library cells are guaranteed to be available to match to the decomposed subject graph.

The circuit structure of a gC shown in Figure 4.1(b) is susceptible to short-circuits if both the n- and p- stacks are concurrently enabled. This short-circuit problem is discussed at length in Chapter 5 but it is mentioned here because gC's cannot always be used for state-holding. The use of a gC is preferable due to its



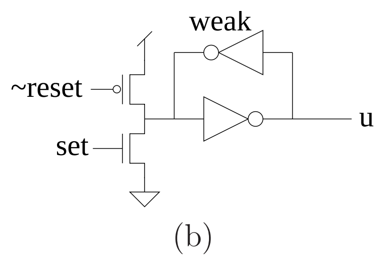
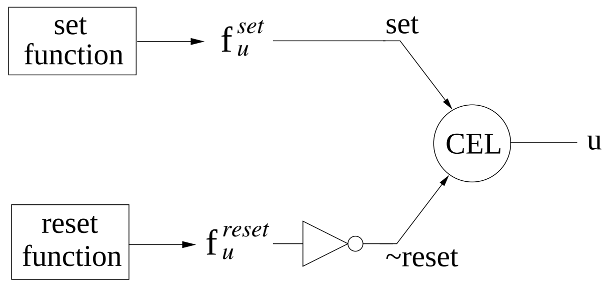


Figure 4.1. Generalized C-element structure. (a) General form of a gC circuit. (b) A CMOS implementation.

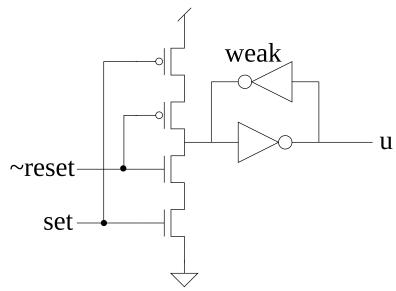
compact structure but if short-circuit problems are present, another element must be used to provide the state-holding function.

One such element that prevents short-circuits is the standard Muller C-element (CEL). The general structure of a CEL is shown in Figure 4.2(a) and a CMOS implementation [82] is shown in Figure 4.2(b). The difference between this gate and the gC can be easily seen from the truth tables shown in Table 4.1. In particular, when the *set* input is equal to 1 and the $\sim reset$ input is equal to 0, this gate holds its previous state while a gC has a short-circuit. Therefore, when a short-circuit condition is detected, the gC gate is replaced with a CEL. Note that when f_u^{set} is of the form ab and f_u^{reset} is of the form $\bar{a}\bar{b}$, the CEL can be used for both this logic and the state-holding function.

Referring again to Figure 4.1(a), f_u contains a set of product terms, f_u^{set}, representing the conditions that cause the output to be set. If the circuit is state-holding, there is a corresponding set of product terms, f_u^{reset}, that represent conditions causing the output to be reset. These two sets of terms are naturally



(a)



(b)

Figure 4.2. CEL structure. (a) General form of a CEL circuit. (b) A CMOS implementation.

Table 4.1. Truth tables for a gC and a CEL.

inputs		outputs	
set	∼reset	gC	CEL
0	0	0	0
0	1	hold	hold
1	0	sc	hold
1	1	1	1

decoupled and can be decomposed separately. The decomposition is done as a tree structure, with the root at the output and the inputs as leaves and is performed from the tree root back towards the leaves.

Figure 4.3 shows the top level algorithm used to perform decomposition on f_u. The decomposition algorithm takes as input the partitioned netlist, f_u, and returns a new netlist composed of base functions. The first step in the algorithm is to

```
decomp(f_u) {
  if (|f_u^{\overline{reset}}| > 0) then
    return gc(or_decomp(f_u^{set}),inv(or_decomp(f_u^{reset})))
  else
    return or_decomp(f_u)
}
```

Figure 4.3. Decomposition algorithm for f_u.

determine if the netlist f_u is state-holding. A state-holding netlist implies that the number of elements in $f_u^{\overline{reset}}$ is nonzero. If this is the case, the function `gc` is called, a gC is placed at the root, and the cones of logic representing the set and reset portions of f_u are decomposed separately. Note that a gC is placed in the decomposition and later checked for short-circuit conditions. If any are found, the gC is replaced with a CEL.

Since f_u^{set} and f_u^{reset} are both disjunctively combined sets of products, the `or_decomp` function is called for both arguments of the `gc` function. However, an inverter is placed on the output of the reset portion by the function `inv`. This state-holding structure is implemented as shown in Figure 4.1(a). If $f_u^{\overline{reset}}$ has no product terms, the circuit is combinational and the state-holding function is unnecessary. In this case, f_u is passed to the `or_decomp` function.

The `or_decomp` function of Figure 4.4 takes a set of product terms as inputs. If there is only one product term, a call is made to the `and_decomp` function and this product term is decomposed into an unbalanced tree structure. If there are

```
or_decomp(products) {
  select product from products {
    if (|products| == 1) then
      return and_decomp(product)
    else
      return inv(inv(nand2(inv(and_decomp(product)),
                           inv(or_decomp(products - {product})))))
  }
}
```

Figure 4.4. Decomposition algorithm for an OR function.

multiple product terms, then one of the product terms is selected, a call is made to the `and_decomp` function followed by a call to the `inv` function, thus completing one argument of the `nand2` function call. Next, the selected product term is removed from *products*, `or_decomp` is called on the remaining products, `inv` is called on this result, thus completing the other argument of the `nand2` call. Finally, the `inv` function is called twice.

Timed circuit decomposition shown in the `and_decomp` function of Figure 4.5 uses unbalanced trees for two reasons. First, as mentioned in Chapter 1, an unbalanced decomposition of a product term has a regular structure regardless of the number of literals, whereas a balanced structure varies depending on the number of literals. Second, unbalanced trees facilitate the search for a minimal delay implementation through input pin reordering, an optimization technique discussed in Section 4.3 that can potentially reduce both the circuit delay and the number of hazards in the decomposed netlist. One potential drawback of the unbalanced architecture involves the case where all inputs arrive simultaneously. Assuming 2-input gates, if the number of inputs, n, is greater than one, the number of gate delays in the balanced architecture is the ceiling of $log_2(n)$, where in the unbalanced architecture the number is *n-1*. This difference becomes more significant with increasing n.

The `and_decomp` function of Figure 4.5 takes a single product term with an arbitrary number of literals and decomposes it into an unbalanced tree structure.

```
and_decomp(product) {
  select literal from product {
    if (|product| == 1) then
      return lit(literal)
    else
      return inv(inv(inv(nand2(lit(literal),
                         and_decomp(product − {literal})))))
  }
}
```

Figure 4.5. Decomposition algorithm for an AND function.

If *product* contains only one literal, a call is made to `lit`. The function, `lit`, shown in Figure 4.6, inserts an inverter pair at the leaf if the *literal* is positive, otherwise it inserts a single inverter. If *product* contains multiple literals, then one argument to the `nand2` function call is implemented by selecting one of the literals and passing it to the `lit` function. The second argument is implemented by removing this literal from *product* and recursively calling the `and_decomp` function. Upon return from `nand2`, the `inv` function is called three times.

Figure 4.7 shows the results of running three examples through these decomposition algorithms. Figure 4.7(a) shows the decomposition of the combinational function $f_u = abcd$. Figure 4.7(b) shows the decomposition for the combinational function $f_u = a\bar{b}\bar{c} + wxyz$. Finally, Figure 4.7(c) shows the decomposition for a state-holding function with $f_u^{set} = ab + cd$ and $f_u^{reset} = e$.

Excluding the input and output structures, the internal structures between 2-input nand gates in the examples of Figure 4.7 have a string of two or three inverters. The decomposition algorithms actually place a string of four inverters where two are shown but one inverter pair is removed during implementation because it contributes nothing to the circuit solution or optimization.

4.1.2 Insertion of Inverter Pairs

The algorithms described in the previous section employ the use of inverter pairs in the decomposition [35, 32, 29]. This clever technique was first used in the MIS program [35]. The goal of using inverter pairs is to increase the granularity of the netlist thereby increasing options for the matching/covering stage.

The addition of inverter pairs at selected nodes does not affect the Boolean

```
lit(literal) {
  if literal is positive then
    return inv(inv(literal))
  else
    return inv(literal)
}
```

Figure 4.6. Decomposition algorithm for a literal.

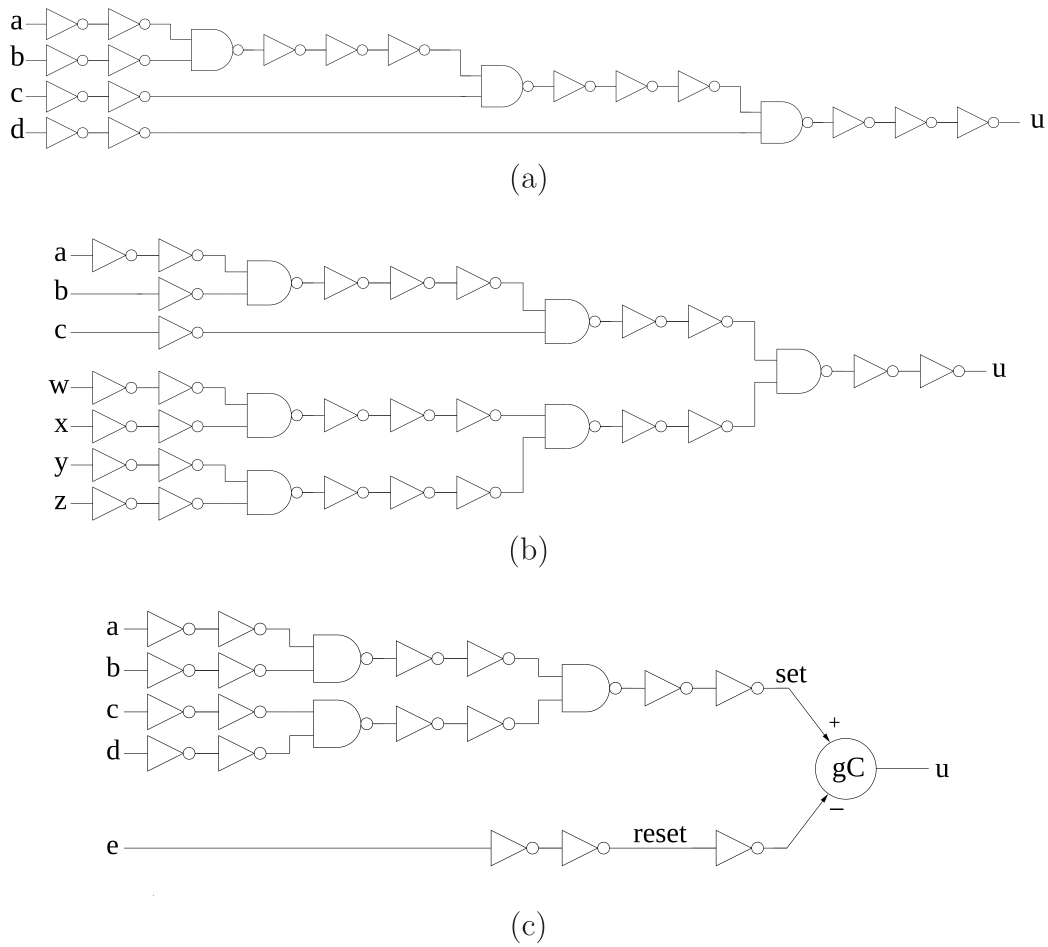


Figure 4.7. Example decompositions. (a) Combinational function $f_u = abcd$. (b) Combinational function $f_u = a\bar{b}\bar{c} + wxyz$. (c) State-holding function $f_u^{set} = ab + cd$ and $f_u^{reset} = e$.

function being implemented and is straightforward to implement in both the subject and pattern graphs. The dynamic programming algorithm used in the matching stage can now take advantage of a wider range of matching options. Any inverter pair chosen as the best match at a node during the matching phase can safely be removed from the covered netlist since its actual implementation is a wire. The implementation cost of a matched network, starting from an unmatched network with inverter pairs, has lower (or at most equal) cost to that of a matched network derived without inverter pairs.

There are three potential drawbacks from using inverter pairs. First, the hazard verification is done on the unmatched network, consisting only of base functions, and the increase in the number of nodes due to inverter pair insertion increases the computational cost of the verification algorithm. Second, there is an additional computational cost in the matching stage because more library cells are considered as potential matches due to the finer granularity of the network. Finally, the additional hazards that the inverter pair nodes are likely to create must be considered in the matching algorithm, possibly making this step more difficult and time consuming. Providing that the matching and covering stage can properly deal with all hazardous nodes, the covered netlist has equal to or fewer library elements (after removing inverter pairs). Thus, the timed technology-mapping algorithm has more potential for implementing a hazard-free netlist when using inverter pairs.

4.2 Hazard Annotation

Once decomposition is complete, each node is verified for hazard-freedom using the algorithms presented in Chapter 3. If hazards are found to exist, each node is annotated accordingly. It is important that this annotation indicates the type of hazardous behavior because acknowledgment and monotonicity hazards may be treated differently in the matching stage.

Figure 4.8 shows three outputs from a file named *Ebergen*, which is taken from a suite of examples used by the timed automata tool `KRONOS`. These outputs are analyzed and synthesized using the ATACS tool. The synthesized equations are verified using the algorithms in Chapter 3 and decomposed using the algorithms in Section 4.1. Each node in Figure 4.8 has been annotated with its hazard properties where an M indicates that a monotonicity hazard exists on that node and an A indicates that an acknowledgment hazard exists. Note that the presence of hazards is dependent upon explicit timing delays associated with the base functions in the decomposition. For the hazards shown in Figure 4.8, *inverter* delays are [1,4], *nand2* delays are [2,4], and gC delays are [4,6] time units. Note that no hazardous nodes are present in Figure 4.8(a), only monotonicity hazards are present in Figure 4.8(b), and only acknowledgment hazards are present in Figure 4.8(c).

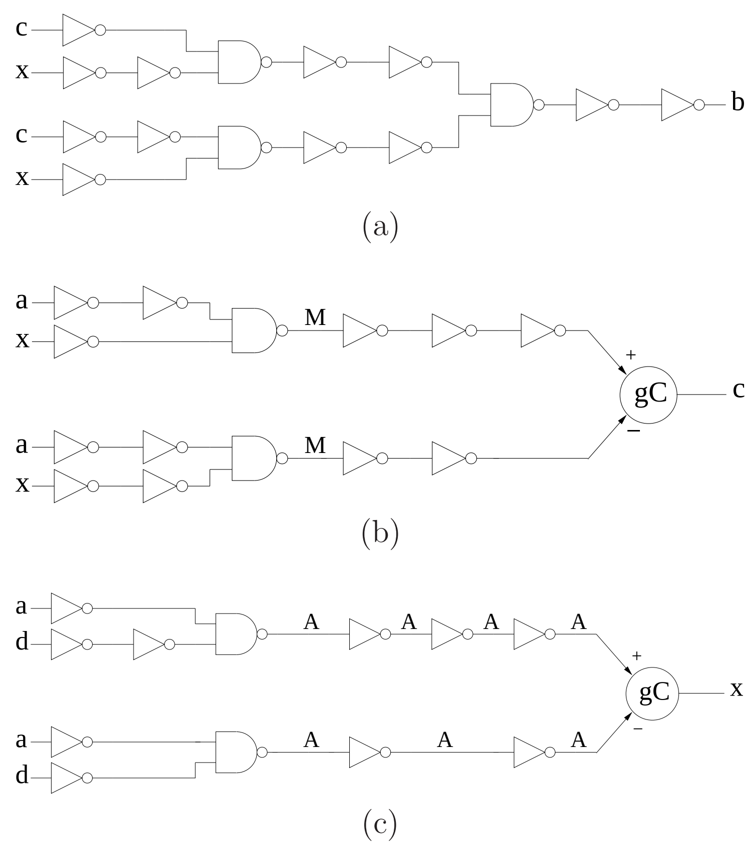


Figure 4.8. Hazard annotations. (a) Ebergen output b. (b) Ebergen output c. (c) Ebergen output x.

4.3 Input Pin Reordering

The unbalanced tree structure used during decomposition provides a means for finding minimal delay through a netlist, and also allows for the possibility of considering many different decompositions through reordering of the input pins. To take advantage of this, later arriving inputs are placed closer to the output. The two examples shown in Figure 4.9(a) and 4.9(b) indicates how input pin reordering reduces the maximum delay through the circuit. Suppose the delay on the *nand2* gates is $[2, 4]$ time units and the delay on the *inverter* gates is $[1, 3]$ time units. For the arrival times of the primary inputs shown in Figure 4.9(a), the output u switches somewhere in the range of $[12, 22]$ time units. Figure 4.9(b) shows how this delay time can be reduced by reordering the inputs so the later arriving input,

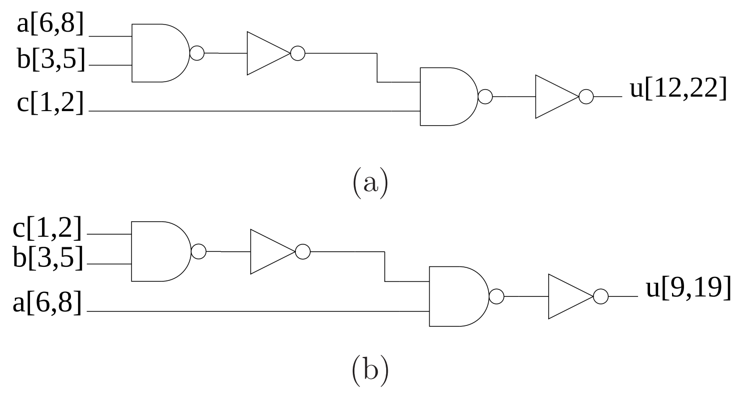


Figure 4.9. Input pin reordering. (a) Delay calculations for a 3-input AND function. (b) Pin reordering to improve delay times.

a, is placed closer to the output. Now the switching time of output u has decreased to $[9, 19]$ time units.

Not only can this pin reordering technique be used in timed asynchronous designs to minimize delay in the decomposed network, but it can also be used to alter the time at which internal and output nodes switch. This technique is illustrated in the netlists that are first presented in Figures 1.6(c) and 1.6(d) and shown again in Figures 4.10(a) and 4.10(b) with gate delays added. Using the gate delays of [5,8], node e is hazardous in Figure 4.10(a) but is found to be hazard-free in Figure 4.10(b).

Referring to Figure 4.10(a), beginning in state (1110), inputs a, b, c, and internal signal e are high while the output d is low. After a falls, the circuit transitions to state (0110), e becomes excited to go low and does so [5,8] time units later. Signal b then falls [2,5] time units after a fell and the circuit transitions to state (0010). Signal c now falls [2,5] time units after b, which is [4,10] time units after the AND gate driving signal e is excited to fall. Since the AND gate driving node e takes [5,8] time units to evaluate, it is possible that the AND function driving the output d can become excited to rise prematurely. The result is a hazard on the signal d, and there is a potential for circuit failure.

The netlist of Figure 4.10(b) shows how, after input pin reordering, the later

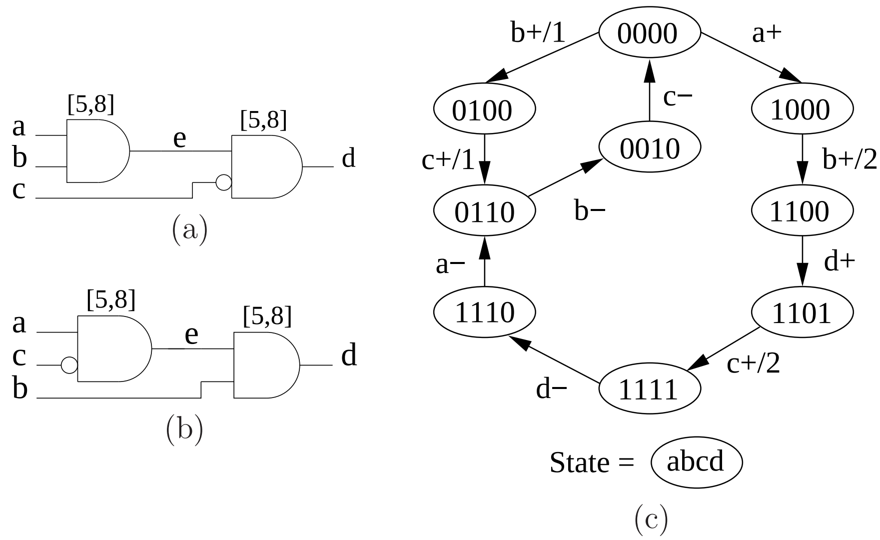


Figure 4.10. Hazard-freedom through pin reordering. (a) Hazardous netlist. (b) Hazard-free netlist. (c) State graph.

arriving input, c, is placed further from the output in the decomposition. This maintains the logical equivalence to the netlist of Figure 4.10(a) while at the same time changing the timing behavior. Beginning again in state (1110), this time e starts low, and it does not become excited to change until after a falls, b falls, c falls, and a rises again. At this point, input b is already low, which maintains d in its low state until b rises again. There is no sequence of transitions that can cause this circuit to experience a hazard on the output d. Thus, the hazard shown in Figure 4.10(a) has been eliminated by the reordering of the inputs.

This example shows that the placement of signal c further from the output has eliminated the hazard condition. Signals such as c are called *trigger signals* and placing them further from the output likely decreases hazardous activity in the netlist. Trigger signals are those signals whose firing can cause the circuit or gate to become excited. Multiple trigger signals may exist and if so, they are all assumed to arrive concurrently. All other signals applied to a gate (or circuit) are called *context signals* and are assumed to be stable before a trigger signal activates. As just shown, trigger signal placement in the unbalanced decomposition tree can have a direct affect on the circuit delay and hazard properties.

Synchronous designs are delay optimized when the trigger signals are placed as close to the output as possible. SI designs need trigger signals placed as close to leaves of the decomposed network as possible to ensure that all other signals that affect the transition of the output are stable before the trigger signal enables the output to change.

Referring to the state graph of Figure 4.10(c), input $c+/2$ is the trigger signal for d-. When this trigger signal is placed closer to the output as in Figure 4.10(a) a hazard occurs because node e has not had time to stabilize before c rises. However, the circuit of Figure 4.10(b) is hazard-free because the trigger signal is placed closer to the inputs and its transition is the last one to occur before d falls.

This example illustrates conflicting priorities for timed asynchronous circuits. For minimal delay, trigger signals should be placed as close to the output as possible since they are the latest arriving signals. For maximal hazard-freedom, trigger signals should be placed as close to the leaves as possible to ensure that all other signals affecting the output have stabilized.

4.4 Library Creation

Libraries have been created for both synchronous and asynchronous technology-mapping and it is a combination of these libraries that this work uses to bind the subject graph to a specific technology. There are several issues that are important in regards to the selection and implementation of a cell library.

First, prefiltering must be done when preparing the library for use in matching the elements to the subject graph. For instance, library cells expecting a clocking signal (in the case of synchronous latches and flip-flops) along with its associated setup and hold times, will be treated using the same atomic gate model used for all library cells, that is, the Boolean function of the cell evaluates instantly and the value is available at the output after a specified delay. In other words, inputs such as clocking signals will be treated like any other inputs to the cell.

Second, special annotations must be included with library cells that cannot be represented in a tree structure. Examples of these functions are exclusive-or, exclusive-nor, and many gC gates. These cells are referred to as *common-input* cells where equivalent subnetworks must be found that drive the common-input pins, otherwise the cells cannot be utilized in the implementation the subject graph. This issue is addressed in Section 4.4.3.

Third, the structural approach to technology-mapping requires that the matching/covering stage find isomorphic pattern matches between the subject graph and the pattern graphs. This necessarily requires that the pattern graphs be decomposed in a structurally equivalent manner to which the subject graph is decomposed.

4.4.1 Implementation Library

An *implementation library* must be available to the technology-mapper to ensure that the decomposed netlist can be successfully implemented. Each cell of the implementation library is modeled as an atomic gate. The combinational and sequential cells are single output logic functions. The sequential elements are either CEL's or gC's with internal feedback for state-holding purposes.

Each library cell is a netlist composed of a node for each fanin to the cell and a single output node. This output node has the Boolean function of the cell, f_u, associated with it. If this cell is a state-holding element, then $f_u = f_u^{set} + u \bullet \overline{f_u^{reset}}$. The pattern graphs for each cell are generated using the same algorithms used for decomposing the subject graph. Each library cell is decomposed once into a pattern graph by the algorithms presented in Section 4.1 and then subsequently fetched from cache.

Figures 4.11(a) and 4.11(b) show two examples of library cells. Figure 4.11(a) is a combinational netlist implementing the XOR function where $f_u = \bar{a}b + a\bar{b}$. Figure 4.11(b) is a state-holding function where $f_u^{set} = pq$ and $f_u^{reset} = \bar{p}\bar{r}$.

4.4.2 Base Functions

The implementation library is required to contain the base functions used in the subject graph decomposition. These three functions are the inverter, 2-input NAND, and a CEL. With two small additions, the base library could provide simple optimizations to the matching/covering stage. First, a plain gC such as that shown

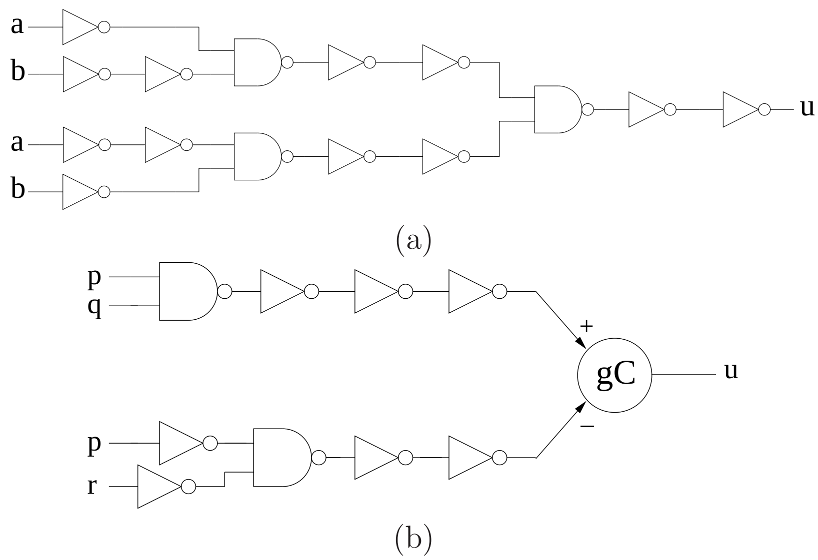


Figure 4.11. Library cell examples. (a) 2-input XOR function. (b) gC function.

in Figure 4.1(b) could replace the CEL for state-holding. This would reduce the size of the device level implementation for f_u providing short-circuit conditions are not present. Second, a zero-cost buffer in the form of an inverter pair whose implementation is a wire would allow for the removal of unused inverter pairs in the subject graph.

4.4.3 Decomposition of Common-Input Cells

Library cells with common-input pins cannot normally be represented as trees. However, a tree representation can be made by splitting the leaves of the inputs that are common and making a special notation for those input variables that are associated with more than one leaf vertex. Since the decomposed network is a directed acyclic graph, the matching stage must ensure that equivalent subnetworks exist for these input variables. Thus, all library cells with common-inputs must carry a special annotation to indicate that equivalent subnetworks (or equivalent primary inputs) must be applied to common-input variables. This problem is not unique to timed circuit design. However, many gC's have this common-input pin structure so this situation occurs frequently in timed circuits.

An example of one such decomposition is shown in Figure 4.12. Again, all

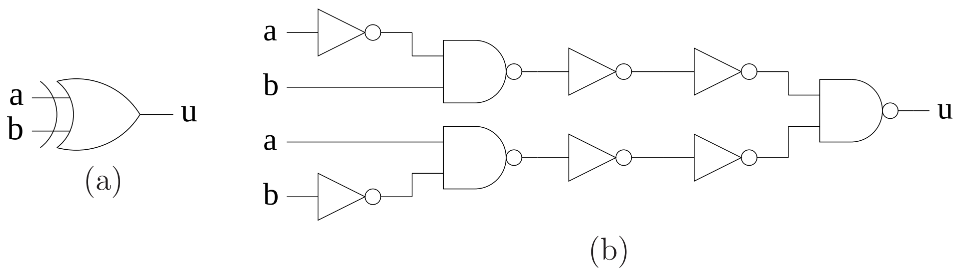


Figure 4.12. 2-input XOR example. (a) XOR circuit symbol. (b) XOR decomposition.

inverter pairs on the inputs and output have been omitted for this example. Figure 4.12(a) shows the schematic symbol and Figure 4.12(b) shows the library cell decomposition of the function $f_u = a\bar{b} + b\bar{a}$. Both of the inputs labeled a in the pattern graph of Figure 4.12(b) *must be mapped to the same physical wire* of the subject graph in order for this library cell to be mapped correctly.

The portion of the circuit shown in Figure 4.13(a) enclosed in the rectangle can be properly covered by the library cell shown in Figure 4.12(b) because the subnetworks driving nodes $a1$ and $a2$ are identical as are the subnetworks driving

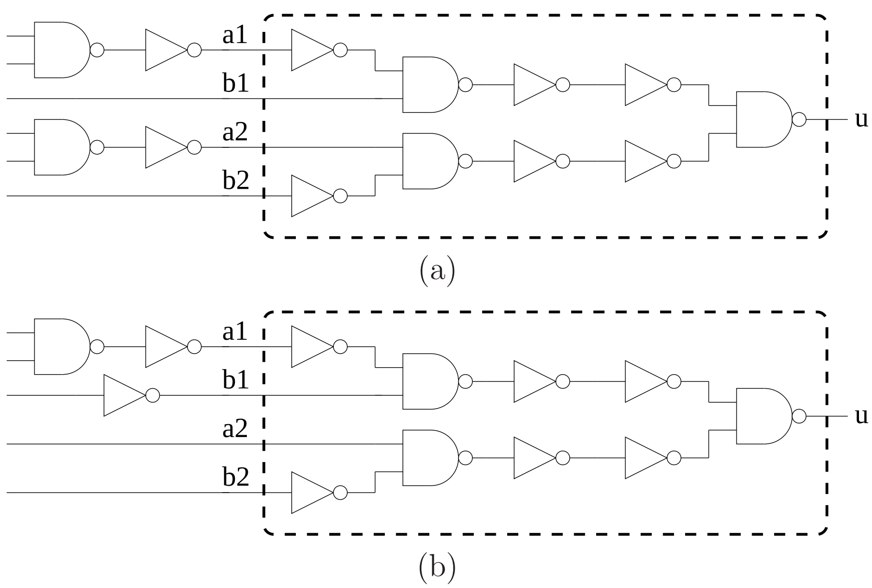


Figure 4.13. XOR matching example. (a) A circuit that matches. (b) A circuit that does not match.

nodes $b1$ and $b2$. However, Figure 4.13(b) cannot be covered by the library element of Figure 4.12(b) because the subnetworks driving nodes $a1$ and $a2$ are not identical, nor are the subnetworks driving nodes $b1$ and $b2$.

The previous discussion concerning common-input pins applies as well to many of the gC library elements. These elements, by their very nature, require some portions of both the set and reset cones of logic in the subject graph to be mapped to the element. The same restriction applies, that is, subnetworks driving common-input pins on the gC must be equivalent.

An example of a gC that addresses the issue of common-input matching is shown in Figure 4.14. Figure 4.14(a) shows the pattern graph for a gC library element with $f_u^{set} = pq$ and $f_u^{reset} = \bar{p}\bar{r}$. Figure 4.14(b) shows a subject graph that is the Ebergen output x. The dotted rectangular area indicates that part of the subject graph that is mapped to the pattern graph of Figure 4.14(a). Note that the pattern graph of Figure 4.14(a) is a structural match for the subject graph of Figure 4.14(b). However, the common-input requirement (on input p) for this library cell does not match because the p input on the set portion of Figure 4.14(a) is matched to the input a through one inverter while for the reset portion it is matched to the a input. However, this library cell *could* have matched had its inputs been permuted.

4.4.4 Input Label Permutations

Consider again the subject graph of Figure 4.14(b) where this time a match is made to the pattern graph of Figure 4.14(c). The structure is identical to the pattern graph of Figure 4.14(a) and the only difference is that the order of the inputs on the set and reset portions of the pattern graph have been permuted. Now, when this pattern graph is matched to the subject graph the common-input pin p is considered a match because it matches the d input on top (through the inverter pair) and the d input on the bottom.

This example shows that each library cell with common-inputs needs to have a unique input pattern in order that all such cases can be accommodated. This permutation creates a potentially exponential increase in the number of library cells when common-input elements are present. However, the size of the libraries used

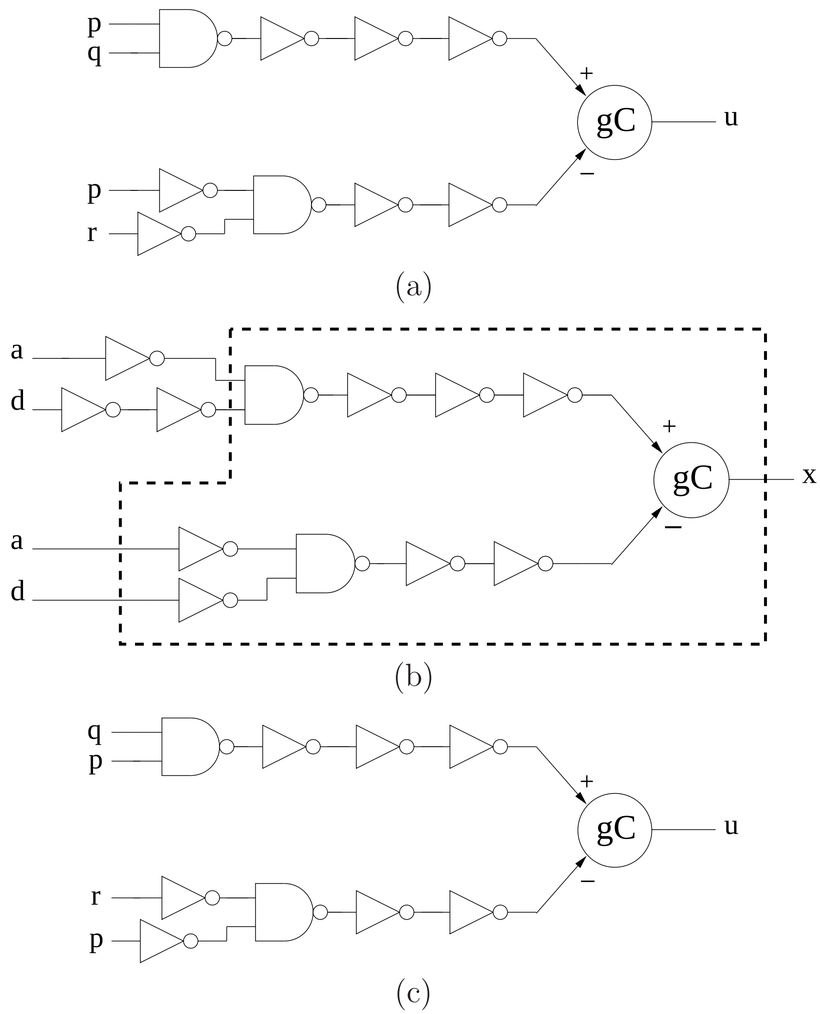


Figure 4.14. Input permutation example. (a) A pattern graph with $f_u^{set} = pq$ and $f_u^{reset} = \bar{p}\bar{r}$. (b) Ebergen output x. (c) Pattern graph of with permuted inputs.

for matching is relatively small and experimental results show that the size of the library has a minor effect on computation time for technology-mapping [35].

CHAPTER 5

MATCHING AND COVERING

This chapter addresses the issues involved in matching and covering a timed asynchronous circuit to a technology-dependent library. Synchronous matching and covering algorithms are well documented in [28, 33, 29] and a brief explanation is first given. Then algorithms for matching are presented and explained and an example is used to illustrate. Next, cost factors, especially those related to hazards, are presented followed by a section on hazard-aware matching. Finally, algorithms involving short-circuit checks and common-input matching are presented and discussed.

Several departures from the synchronous design flow arise due to the nature of timed asynchronous technology-mapping. The first, and most important, is that of matching and covering the decomposed netlist such that all monotonicity and acknowledgment hazards are eliminated in the final netlist. The second issue is the need to check for potential short-circuits in the p- and n- transistor stacks when gC's are selected from the library. The final issue is the matching of library cells with common-input requirements. This last issue is not unique to asynchronous designs; it must also be dealt with in synchronous designs for library cells such as XOR functions and their derivatives. However, it occurs much more frequently in asynchronous designs, particularly when gC's are present in the library.

5.1 Basic Matching

The seminal work in tree-based matching and covering is due to Keutzer [30]. In this work, both the subject and pattern graphs are required to be acyclic and rooted, with the root being associated with the subnetwork and cell outputs. When the decomposition creates a tree structure, all input variables are associated with different leaf vertices except in the pattern graphs for common-input library cells. For those library cells that are represented as rooted trees, the tree matching and tree covering problems can be solved in linear time [28].

The structural matching problem for general library functions, represented as directed-acyclic-graphs, is thought to be intractable [27]. Nevertheless, efficient matching algorithms have been developed because the number of library cells considered as potential matches is typically small. Experimental results show that the size of the library has a minor effect on overall computation time for technology-mapping [35].

A typical approach to match an implementation library to a design is to match subnetworks of the subject graph to individual library cells [30, 31, 32]. This matching process is applied to the subject graph for each library cell, and an optimum covering is chosen based on a cost parameter such as area or delay.

The matching/covering stage takes as inputs the annotated netlist from the hazard verification process along with a technology-dependent library and creates a new netlist composed exclusively of library cells. The matching algorithm finds the best match at every node from the available library cells, taking into consideration a primary and secondary cost factor chosen from hazard-freedom, area, or delay. Since a suboptimized netlist in regards to area or delay still functions correctly provided it is hazard-free, this research focuses on hazard elimination as a primary cost objective. Correct circuit operation cannot be guaranteed when hazardous nodes are present in the final netlist. The matching algorithm determines which library cell best matches the circuit at each node and annotates the node with the selected cell. When calculating the best match, the matching algorithm must take into account the number and types of hazardous nodes that are encapsulated by the cell as well as hazardous nodes left exposed where the leaves of the library cell are mapped in the subject graph.

After the circuit is matched, the covering algorithm determines the best cover using dynamic programming methods [83, 30] and writes a new netlist composed of cells from the library. It is still possible that the final netlist may contain hazardous nodes. First, it may not be possible to eliminate all hazardous nodes through the matching/covering process. The success of this step is dependent upon the number and topology of the hazardous nodes, and the available library cells. Second, the newly covered netlist alters the timing of the decomposed netlist because the structure of the circuit will likely change. It is certainly possible that this newly covered circuit may now exhibit other hazards due to the new timing landscape. As a result, hazard reverification must be performed on the covered netlist.

If this final verification shows no hazards, then the technology-mapping process is complete and the library netlist is a hazard-free implementation of the original circuit specification. If not, then various optimizations are performed, as discussed in Chapter 2. The technology-mapping process is then repeated until a hazard-free netlist can be produced or it can be shown that one is not possible.

5.1.1 Top Level Matching Algorithm

The top level algorithm shown in Figure 5.1 takes as inputs the decomposed and hazard-annotated netlist, *decomp_net*, a state graph, *SG*, that has been annotated with the `stable` and `eval` predicates, and a cell library *LIB* that is a set of nets representing the library cells. The objective of this algorithm is to return a covering of *decomp_net* such that this covering is free of hazards.

This algorithm works its way through the netlist topologically, beginning at the leaves and working towards the root. For each internal and output node n, the netlist is partitioned at n so that pn represents a netlist from the connected leaves to n. The cost at node n is then set equal to ∞. Next, each library element $l \in LIB$ is decomposed into a netlist and considered as a possible match. The `match` function returns the cost of matching dl to a subgraph rooted at n, where the parameters *cost* and *new_cost* contain information regarding area, delay, acknowledgment hazards, and monotonicity hazards. If the `match` function cannot find a structural match between the library element, dl, and the partitioned netlist, pn, then a cost of ∞ is returned. In this case, the algorithm rejects this library element and considers the next library element.

After the `match` function returns the cost of matching dl to pn, the `matcov`

```
matcov(decomp_net,SG,LIB) {
  foreach n ∈ (N ∪ O) in topological order {
    pn = partition(decomp_net,n)
    cost(n) = ∞
    foreach l ∈ LIB {
      dl = decomp(l)
      new_cost = match(dl,pn,cost)
      if (new_cost != ∞) then
        new_cost = new_cost + haz_aware_cost(l,n)
        if (new_cost < cost(n)) then
          if ((common_input(dl,pn)) ∧
              (!gC(l) || !short_circuit(SG,l,dl,pn))) then
            cost(n) = new_cost
            best_match(n) = l
    }
  }
  return cover(decomp_net,best_match)
}
```

Figure 5.1. Top level matching and covering algorithm.

function adds the cost of the library cell l to *new_cost*. The cost of the library cell is calculated by calling the `haz_aware_cost` function, which determines the area, delay, and hazard costs of selecting l. If these newly computed costs are less than the previous costs, the algorithm updates the *cost* and *best_match* values at node n, providing certain conditions are met. First, cells with common-input requirements must pass the common-input test (discussed in Section 5.4). If this check passes, then the *cost* and *best_match* are updated if the library element is a combinational function or it is a gC and passes the short-circuit check (discussed in Section 5.3). At least one match is guaranteed at each $n \in (N \cup O)$ because all nodes in the decomposition are driven by a circuit element in the *base function* set and all such functions are required to be elements of *LIB*.

After all library cells have been checked, the `cover` function takes *decomp_net* and *best_match* and creates a new netlist composed of cells from the library. The algorithms for optimum tree covering of timed asynchronous circuits are not unique to timed asynchronous technology-mapping. The covering is computed using dynamic programming techniques and is well documented in [83, 30].

5.1.2 Matching Algorithm

The basic algorithm for matching is taken from [28] and modified as shown in Figure 5.2 to reflect the features unique to this research. The `match` algorithm is linear in the size of the graphs [28]. The algorithm of Figure 5.2 finds isomorphic pattern matches between a pattern graph and a subnetwork of the subject graph, beginning at the root and working towards the leaves. The decomposed netlist used for matching includes three base functions: 2-input NANDS, inverters, and a CEL. Each vertex of the subject and pattern graphs is associated with either a 2-input NAND or C-element and has two children (inputs), or an inverter with one child. Each vertex is identified by its *type* where a call to *type(v)* returns the type of base function associated with this vertex.

The algorithm in Figure 5.2 is invoked with u as the root of the pattern graph and v as a vertex of the subject graph. If the vertex of the pattern graph is a leaf, then a path exists from that leaf in the pattern graph to the root of the subject graph and the cost associated with node v is returned. When both vertices are not leaves, they must have an equal number of children that must recursively match for a match to be possible.

```
match(u,v,cost) {
  if (type(v) ≠ type(u)) return(∞)
  if (v is a leaf) return(∞)
  if (u is a leaf) return cost(v)
  if (type(v) == inverter) then
    u_c = partition(u,child_u) ; v_c = partition(v,child_v)
    return match(u_c,v_c,cost)
  else
    u_l = partition(u,left_child_u) ; u_r = partition(u,right_child_u)
    v_l = partition(v,left_child_v) ; v_r = partition(v,right_child_v)
    if (type(v) == nand2) then
      return min[(match(u_l,v_l,cost) + match(u_r,v_r,cost)),
                 (match(u_r,v_l,cost) + match(u_l,v_r,cost))]
    else
      return (match(u_l,v_l,cost) + match(u_r,v_r,cost))
}
```

Figure 5.2. Matching algorithm.

After it has been determined that u is not a leaf, a check is made on the subject graph. If v is a leaf, then a corresponding leaf has not been reached in the pattern graph and a match is not possible. A cost of ∞ is returned meaning no match is found. A cost of ∞ is also returned if the types of base functions driving u and v are not identical.

At this point, a check is made on which base function is driving nodes u and v. If `type`(v) returns an inverter, than the match algorithm is called recursively with the children of the current vertices. If `type`(v) returns a 2-input NAND function, then the match function is called recursively four times. The first call matches the left children of u and v, and the second call matches the right children of u and v. The third call matches the right child of u and the left child of v, and the fourth call matches the left child of u and the right child of v. The minimum cost of these two match pairs is then returned. This ensures an optimum matching by checking the pattern graph against both sides of the 2-input NAND function in the subject graph. The *cost* parameter is passed through the recursion levels and is continuously updated with costs associated with area, delay, acknowledgment hazards, and monotonicity hazards whenever a leaf of the pattern graph is reached. These cost parameters are discussed in more detail in Section 5.2.

If `type`(v) is neither an inverter or a 2-input NAND function, then u and v are both roots and the algorithm assumes a CEL. Since the children of a gC are not symmetric (one is the *set* function, the other is the *reset* function), only one pair of matches is done, once down the set portion of the gC, and the other down the reset portion.

5.1.3 Simple Matching and Covering Example

The following example illustrates the algorithms of Figures 5.1 and 5.2 as applied to a simple circuit. Issues involving optimum matching in the presence of hazards (Section 5.2), short-circuits (Section 5.3), and common-inputs (Section 5.4) are addressed in their respective sections.

The example used for illustration is the combinational output b from the Ebergen example first presented in Chapter 4 and shown again in Figure 5.3. Each

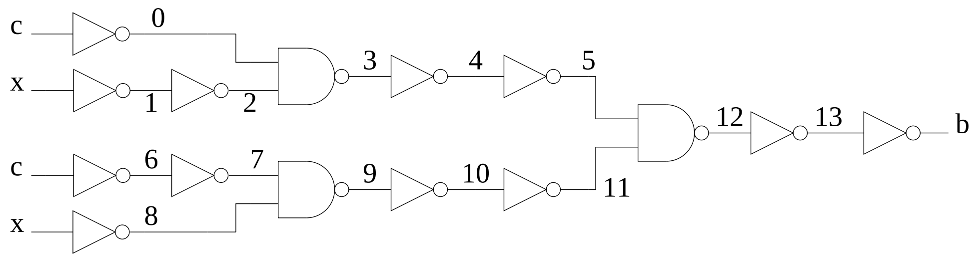


Figure 5.3. Decomposition of Ebergen output b.

node has been annotated with a node number to facilitate explanation. The combinational output b implements the function $f_b = a\overline{b}+\overline{a}b$. When common-inputs are considered in Section 5.4, the match algorithm shows that this circuit matches a 2-input XOR gate.

The example library shown in Figure 5.4 contains eight combinational logic elements. These elements have been structurally decomposed in a fashion identical to the decomposition done on the subject graph. Table 5.1 shows the logic function for each library cell and the area and delay costs associated with each element.

The area cost of matching a cell is the cost of the cell plus the sum of the area

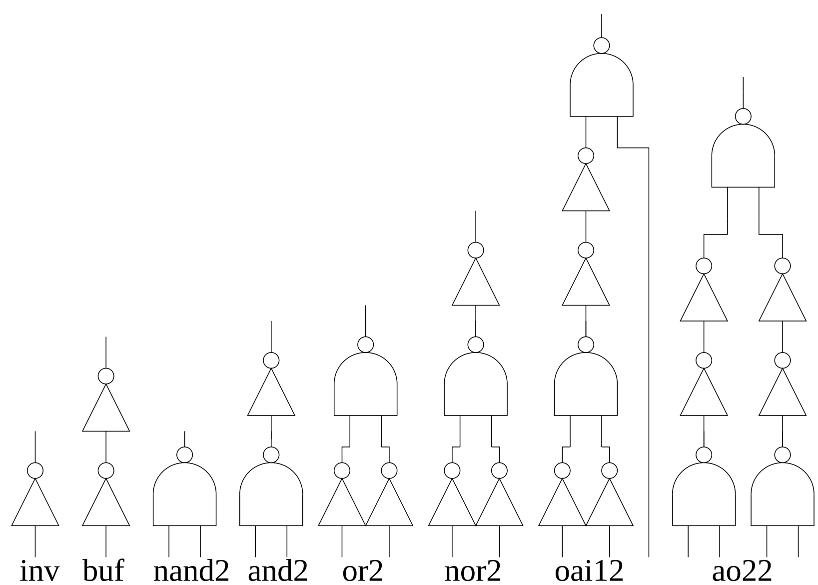


Figure 5.4. Example library.

Table 5.1. Costs for sample library.

Cell	Cost Parameters Area	Min Delay	Max Delay	Function
inv	16	4	6	!a
buf	0	0	0	a
nand2	24	5	7	!(ab)
and2	32	6	8	ab
or2	32	6	8	a|b
nor2	24	5	7	!(a|b)
oai12	40	9	11	!(a(b|c))
ao22	48	8	10	ab|cd

costs of all the leaf vertexes that the inputs of this cell map to in the subject graph. The minimum delay cost of a cell is calculated by adding the smallest delay seen at any leaf vertex to the minimum delay of the matching cell. Likewise, the maximum delay cost is calculated by adding the largest delay seen at any leaf vertex to the maximum delay of the matching cell.

The matching algorithm examines all nodes in the subject graph beginning at the leaves (inputs) and working towards the root (output). The first nodes to be matched in Figure 5.3 are nodes 0, 1, 6, and 8. The only library cell that structurally matches at these four nodes is the *inv*. Since these nodes are driven by primary inputs, which have no area cost, the area cost at these nodes is simply the area of *inv*, 16. Also, the inputs have no delay cost so the delays for these nodes are those for a single *inv*, 4 and 6. The cells from the library that have an isomorphic pattern match to a subnetwork of the subject graph are tabulated in Table 5.2. The cells shown in bold are selected as the best match and the running total of area and delay is also shown in Table 5.2.

The next nodes to be matched are nodes 2 and 7. There are two possible matches at these nodes, *inv* and *buf*. The *inv* area cost is that of itself and the cost at the previous node, which is 16 in both cases for a total area cost of 32. The time delay of the *inv* cell is that of itself plus the time delays at the previous node, which is 8 for minimum delay and 12 for maximum delay. The match for the *buf*

Table 5.2. Structural matches for Ebergen output b.

Node Number	Matching Cells	Cost Parameters		
		Area	Min Delay	Max Delay
0	**inv**	16	4	6
1	**inv**	16	4	6
6	**inv**	16	4	6
8	**inv**	16	4	6
2	inv, **buf**	0	0	0
7	inv, **buf**	0	0	0
3	**nand2**, or2	40	5	13
9	**nand2**, or2	40	5	13
4	inv, and2, **nor2**	40	5	13
10	inv, and2, **nor2**	40	5	13
5	**buf**, inv	40	5	13
11	**buf**, inv	40	5	13
12	nand2, or2, oai12, **ao22**	80	8	16
$13^\star$	**inv**, and2, nor2	96	12	22
$13^\star$	inv, and2, **nor2**	104	10	20
b	inv, **buf**	80	8	16

has all costs of zero and is the obvious choice for minimizing both area and delay.

Next, nodes 3 and 9 are matched. The library cells *inv*, *buf*, *and2*, and *nor2* are rejected because `type`(v) is not equal to `type`(u) at the root. The *nand2* cell as well as the *or2* cell both match the subnetwork of the subject graph rooted at these nodes. The *oai12* and *ao22* cells do not match because as the `match` function recurses toward the leaves, a leaf of v is reached before a leaf of u is reached. This causes the match function to return a cost of ∞ and these cells are not considered as potential matches.

The area costs for the *nand2* is the sum of the area cost of its two inputs plus its own area cost, or $16 + 0 + 24 = 40$. The minimum delay is that of the input attached to the buffer, 0, plus the minimum delay of the *nand2*, which is 5 for a total of 5. The maximum delay is that of node 0, which is 6, plus that of the *nand2*, which is 7, for a total of 13. Using a similar analysis for the *or2* gate, the area = 48, minimum delay = 6, and maximum delay = 14. Thus, the *nand2* is the best match at nodes 3 and 9 for both area and minimum delay and the costs are shown in Table 5.2.

This matching procedure is continued up to and including the output *b*. Referring to Table 5.2, the best cell match found at each node is the same for both area and delay minimization, except at node 13. Here, an *inv* is found to provide the minimum area while a *nor2* gate provides the minimum delay. However, this difference is negated by the selection of a buffer as the best match at the *b* output.

The next step in the technology-mapping algorithm is to cover the matched circuit and write a netlist of library cells composed of the best matches found during the matching stage. The covering algorithm begins at the root of the subject graph, working towards the leaves. If *buf* cells are encountered during covering, they are removed from the final netlist. For example, the best match at output *b* is the *buf* cell. This double inverter cell is removed from the covering, and node 12 becomes the logical equivalent of output *b*. Note that the *buf* cells can remain in the covered netlist if they serve some electrical purpose such as signal buffering.

Figure 5.5 shows the optimum covering for the circuit of Figure 5.3. Note that in this example, the covered netlist is optimum for both area and delay. The final covered circuit is shown in Figure 5.6(a) and an Espresso-type netlist is shown in Figure 5.6(b). This netlist must now be reverified for hazard-freedom because the timing from the decomposition, where the hazard check is first made, has been altered. Note that in the original circuit decomposition shown in Figure 5.3, the absolute min/max delays from any input to the output is [30,50]. In the covered netlist of Figure 5.5(a), these delays have been reduced to [8,16].

5.2 Hazard-Aware Matching

When there is more than one structural match at node n, cost factors are used to help determine which library element to choose. Historically, area and/or delay have driven this selection decision while logical effort [84], power, and reliability are other viable cost parameters. Since asynchronous circuits may fail when hazards are present, this research uses hazard-freedom as the primary cost factor and considers

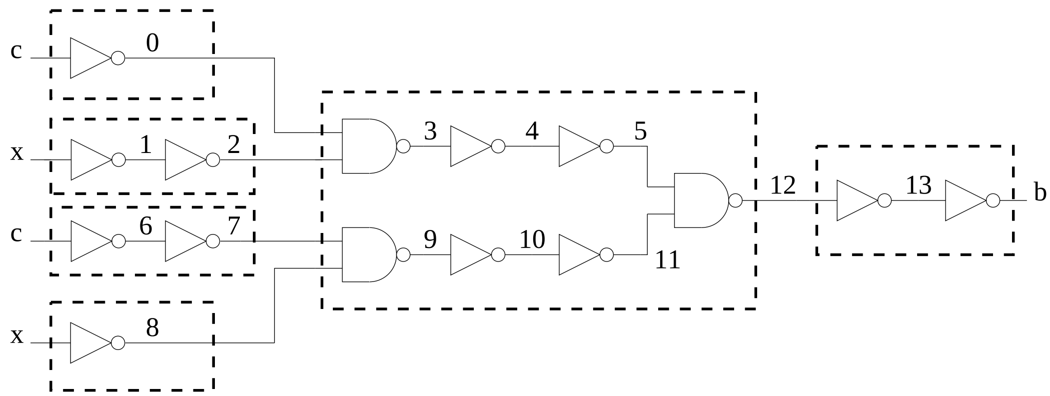


Figure 5.5. Optimum covering of Ebergen output b.

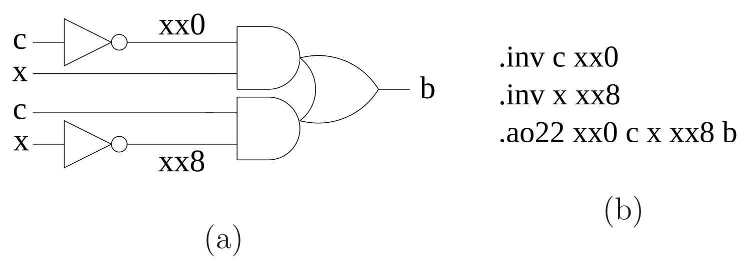


Figure 5.6. Final circuit and netlist. (a) Final mapped circuit. (b) Espresso type netlist.

area and delay as secondary cost factors. For all three cost parameters of hazard, area, and delay, the costs are cumulative from the reachable leaves of the subject graph up to node n. The cost array referenced in the matching algorithms represents the costs associated with selecting the current library element for the given node n. The cost array is composed of the following entries that are used to help guide which library element, l, to choose:

- *area* is the area cost of l plus the sum of areas at the nodes in the subject graph at the leaves of l;
- *mindel* is the minimum delay of l plus the minimum delay of any node in the subject graph at the leaves of l;

- *maxdel* is the maximum delay of l plus the maximum delay of any node in the subject graph at the leaves of l.

In addition, the cost array contains the following parameters associated with the hazard characteristics of mapping l to the subject graph:

- *ackhaz* is 1 if there is an acknowledgment hazard on the root node being covered by l, plus the sum of the acknowledgment hazard costs at the nodes in the subject graph at the leaves of l;

- *monohaz* is 1 if there is a monotonicity hazard on the root node being covered by l, plus the sum of the monotonicity hazard costs at the nodes in the subject graph at the leaves of l.

The equation used to calculate the final hazard cost of implementing l is then:

$$hazcost = w_1 * monohaz + w_2 * ackhaz \tag{5.1}$$

where w_1 and w_2 are coefficients used to optimize the matching process in order to reduce or eliminate hazards in the covered netlist.

Hazard-freedom requires that any hazards present in the subject graph be eliminated in the matching and covering stages. This presents a unique challenge, because acknowledgment and monotonicity hazards must be treated in different ways. Encapsulating an acknowledgment hazard within an atomic gate effectively eliminates that node and the hazard with it. However, the same approach may not work for removal of monotonicity hazards because these hazards are caused by a gate input with a potential hazard, while there is no forcing side-input. Thus, to remove monotonicity hazards during matching, the input with the potential hazard should be left exposed while the other gate input(s) should be encapsulated. This approach does not guarantee that the monotonicity hazard is removed. It simply attempts to alter the timing on a side-input such that it may be found to be forcing in the state for which the monotonicity hazard exists.

The different constraints for acknowledgment and monotonicity hazards cannot always be met and Equation 5.1 has been developed in an attempt to optimize the result. Equation 5.1 parameterizes the effects of hazards in the netlist by weighting the *ackhaz* and *monohaz* variables.

The algorithm for cost determination during hazard-aware matching is shown in Figure 5.7. This algorithm is called from the higher-level matching algorithm of Figure 5.1 and is invoked whenever a library element is found to match a subgraph rooted at n. This algorithm updates the cost parameters *ackhaz* and *monohaz* during the matching of the library element, l, and returns these two parameters, as well as the cost parameters of l. The algorithm of Figure 5.1 then updates the cost array and a decision is made on whether this library cell is the best match from the perspective of minimizing hazards.

The algorithm of Figure 5.7 takes as inputs a node n from the subject graph where the root of l is located, and the library element l. The first step of the algorithm determines if an acknowledgment hazard exists at n. If so, then *ackhaz* is set to one if the library element is not a *buffer*. When buffers are found to be the optimum library element at n during matching, they are removed in the final netlist, and any acknowledgment hazards at n are removed with them. The *monohaz* parameter is set to one if there is a monotonicity hazard present at n. Finally, the algorithm returns all costs associated with a match of l, rooted at n.

Once the number of exposed hazards is calculated for a library cell during matching, Equation 5.1 determines the cost if this cell is chosen as the best match.

```
haz_aware_cost(l,n) {
  ackhaz = monohaz = 0
  if (ack(n) ∧ !buffer(l)) then
    ackhaz = 1
  if (mono(n)) then
    monohaz = 1
  return (area(l), mindel(l), maxdel(l), ackhaz, monohaz)
}
```

Figure 5.7. Algorithm for hazard-aware matching.

The coefficients w_1 and w_2 are used to determine how best to implement the equation over a wide range of examples. Figure 5.8 illustrates how the algorithm of Figure 5.7 matches to a circuit where both acknowledgment and monotonicity hazards are present. This example is the u output from a file named *elatch* where $f_u^{set} = d\bar{c}$ and $f_u^{reset} = \bar{d}b$. After hazard verification, it is found that there are acknowledgment hazards present on nodes $0, 1, 2, 7, 8, 9$ and monotonicity hazards present on nodes 3 and 10. The monotonicity hazard on node 3 is caused by both of its fanins, nodes 1 and 2, and the monotonicity hazard on node 10 is also caused by both of its fanins, nodes 7 and 9.

Figure 5.8(a) shows the matching that occurs when w_1 is set to -1 and w_2 is set to +1 in Equation 5.1. These coefficients instruct the hazard-aware matching

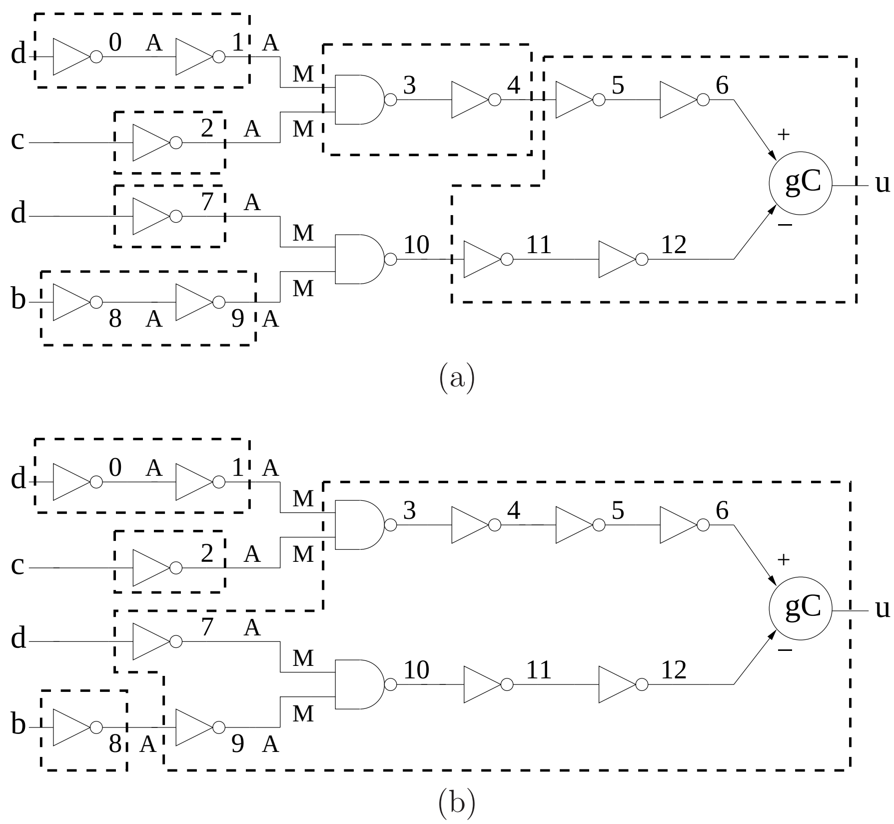


Figure 5.8. Hazard-aware matching. (a) Matching for w_1, w_2 = (-1,+1) and (-1,0). (b) Matching for w_1, w_2 = (0,+1) and (0,0).

algorithm to reward a match (a lower number is better) that leaves monotonicity hazards exposed and penalizes a match that leaves acknowledgment hazards exposed. Figures 5.8(a) also represents the matching that is found when w_1 is set to -1 and w_2 is set to 0. Figure 5.8(b) shows the resulting matches for w_1, w_2 equal to (0,+1) and (0,0). Note that when both coefficients are set to zero, the resulting matching does not take hazards into account and a secondary cost function, such as area, is used to drive the matching decisions.

When hazard verification is done on the resulting covered circuit of Figure 5.8(a), acknowledgment hazards are still present on nodes 2 and 7, and the monotonicity hazard reported on node 10 is also still present. In addition, a new monotonicity hazard has been created on node 4, due in part to the asymmetric path delays caused by the new circuit topology. For Figure 5.8(b), verification results indicate that there are still acknowledgment hazards present on nodes 2 and 8, and a new monotonicity hazard has been created on the output, u.

Chapter 6 presents the results of hazard-aware matching using several variations on w_1 and w_2 for a number of example circuits.

5.3 Short-Circuit Issues in gC's

The design passed to the technology-mapper from synthesis could possibly be covered by one gC element, provided it is available in the library. This is quite unlikely, except for small designs or highly customized libraries. A much more likely case involves the covering of a portion of the design with a gC and the remaining portion with other logic elements. Once the circuit covering is broken up in this manner, timing issues create the possibility of short-circuits in the transistor stacks of the gC.

The power of gC's is their relatively low area and delay costs compared to the cost of implementing the equivalent logic with other library elements. This cost savings is due in part to the ability to make direct connections to the gates of the p- and n- devices. As a result, any gC mapped to a portion of the decomposed network must be checked to ensure potential short-circuit conditions are not introduced. If they are, that particular gC must be rejected as a potential match.

The short-circuit algorithm shown in Figure 5.9 takes as inputs a state graph, SG, a library cell, l, a netlist, dl, representing the decomposition of l, and a netlist, pn, representing the decomposed circuit to be matched. The first step in the algorithm is to find the product of f_u^{set} and f_u^{reset}. The purpose of this step is to identify conditions that can potentially short the transistor stack.

As an example of this product, consider the plain gC shown in Figure 5.10(a). The plain gC is defined in product form as $f_u^{set} = a$ and $f_u^{reset} = \overline{b}$. The product of these two products returns $a\overline{b}$ representing a short-circuit condition when $a = 1$ and $b = 0$. This short-circuit condition can readily be seen in Figure 5.10(b) when there is a simultaneous high on the gate of the n- transistor and a low on the gate of the p- transistor.

A more interesting example is shown in Figure 5.11 where the function f is

```
short_circuit (SG,l,dl,pn) {
  u = head(l)
  shortsop = find_prod(f_u^set, f_u^reset)
  if (|shortsop| == 0) return FALSE
  shortsop = translate(shortsop,dl,pn)
  foreach s ∈ SG {
    foreach product ∈ shortsop {
      short_circ = TRUE
      foreach lit ∈ product {
        if (sig(lit) ∈ I ∪ O) then
          if (s(sig(lit)) ≠ val(lit)) then
            short_circ = FALSE
        else
          if (stable(s,sig(lit)) ∧
            (eval(s,sig(lit)) ≠ val(lit))) then
            short_circ = FALSE
      }
    }
    if (short_circ) return TRUE
  }
  return FALSE
}
```

Figure 5.9. Algorithm for short-circuit detection in gCs.

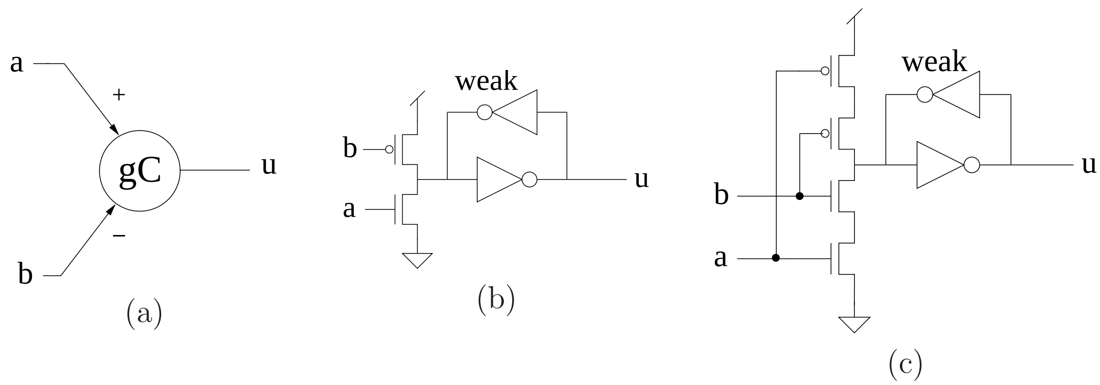


Figure 5.10. gC and CEL structures. (a) Plain gC circuit symbol. (b) CMOS implementation of a gC. (c) CMOS implementation of a CEL.

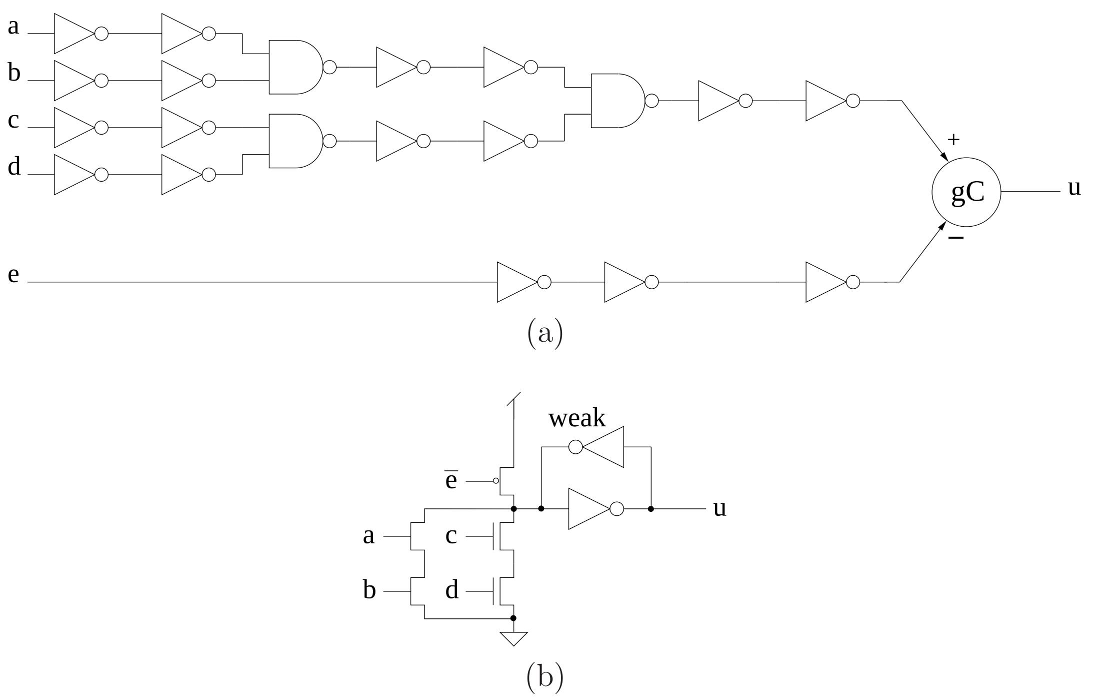


Figure 5.11. A short-circuit example. (a) Logical decomposition of $f_u^{set} = ab + cd$ and $f_u^{reset} = e$. (b) CMOS implementation.

represented as $f_u^{set} = ab + cd$ and $f_u^{reset} = e$. Figure 5.11(a) shows the logical decomposition of this gC and Figure 5.11(b) shows a transistor-level CMOS implementation. For this example, the `find_prod` function returns *abe* and *cde*. Either of these products has the potential for causing a short-circuit in the gC.

If the number of products returned from the `find_prod` function is zero, then there is no possibility of a short-circuit when the library element l is mapped to the partitioned net, *pn*. An example library cell where this occurs is the CEL whose transistor-level structure is shown in Figure 5.10(c). For the CEL, $f_u^{set} = ab$ and $f_u^{reset} = \bar{a}\bar{b}$. `find_prod` returns an empty set indicating that there is no possible combination of inputs where both the p- and n- transistor stacks can be enabled simultaneously.

The short-circuit algorithm next calls the `translate` function, which maps the leaf names from the pattern graph onto the node names in the subject graph. This step is necessary because the final netlist is composed of library cell expressions whose signal names must be a mapping to the physical nodes of the subject graph. The algorithm next checks to see if any state, $s \in SG$, is contained in a product term of *shortsop*. If any one of these product terms produces a possibility of a short-circuit, then the algorithm returns TRUE indicating that a short-circuit condition is possible and this cell must be rejected as a potential match.

The algorithm of Figure 5.9 relies on the `stable` and `eval` predicates that are computed by the hazard checking algorithms in Chapter 3 and stored with each state $s \in SG$. The `sig` function in Figure 5.9 takes a literal as an argument and returns the node (or primary input) in the subject graph where this literal is mapped. For example, when the plain gC of Figure 5.12(a) is matched to the circuit of Figure 5.12(b), `sig`(a) returns node 6 and `sig`(b) returns node 11. The function `val` also takes a literal of a product as an argument and returns a 1 or 0 based on whether the literal is positive or negative, respectively. If `sig`(*lit*) is an external signal, the algorithm checks if the value in the state vector is equal to the value of the literal in the product. If not, then this product cannot cause a short-circuit. If `sig`(*lit*) is an internal signal, and it is stable in state s, the algorithm checks if the

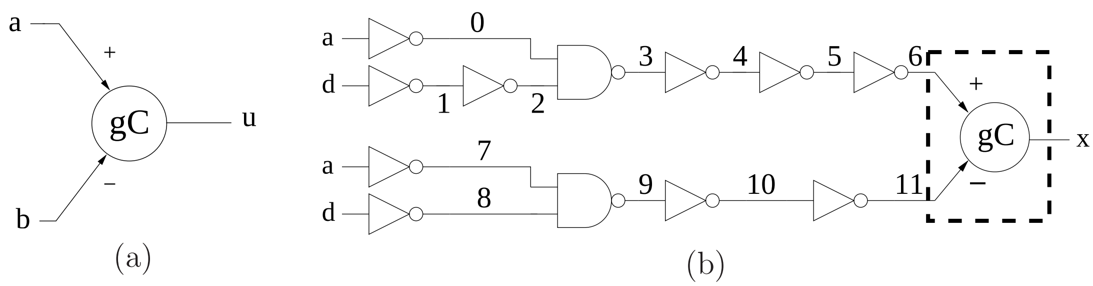


Figure 5.12. Plain gC short-circuit example. (a) Plain gC. (b) Ebergen output x matched to plain gC.

evaluation of this node in state s is equal to the value of the literal. If not, then again no short-circuit condition exists for this product.

Two examples of how gC elements can structurally match a portion of a netlist but are rejected because of short-circuit problems are shown in Figures 5.12 and 5.13. The subject graph in both examples is the Ebergen output x where $f_u^{set} = \bar{a}d$ and $f_u^{reset} = \bar{a}\bar{d}$. The netlist of Figure 5.12(b) is shown superimposed with the plain gC element of Figure 5.12(a) and the netlist of Figure 5.13(a) is shown superimposed

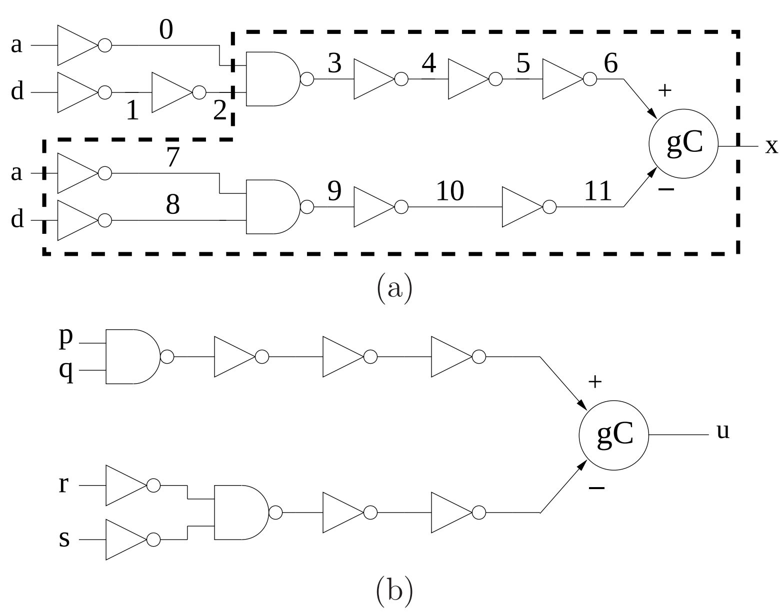


Figure 5.13. *gc22* short-circuit example. (a) Ebergen output x matched to a *gC22*. (b) *gC22* element decomposition.

with the pattern graph of a *gc22* library element, whose pattern graph is shown in Figure 5.13(b).

A portion of the state graph for the subject graph in Figures 5.12 and 5.13 is shown in Figure 5.14. Note that each state is annotated with stability information that is previously determined during hazard verification. The first set of brackets next to each state is stability information relating to nodes 6 and 11 and applies to the example of Figure 5.12. The second set of brackets indicates stability information for nodes 0 and 2 and applies to the example of Figure 5.13. As an example of what this stability information means, in state 10101, the [01][0U] notation indicates that node 6 is stable low and node 11 is stable high for the first set of bracketed items. For the second set of bracketed items, node 0 is stable low and node 2 is unstable.

When the algorithm of Figure 5.9 is applied to Figure 5.12, *shortsop* contains the product $a\bar{b}$. The algorithm next attempts to find a state in which node 6 could be high coincident with node 11 being low. After a search of a portion of the state graph shown in Figure 5.14, state 00101 indicates both nodes are unstable. Since instability can indicate either a low or high level on the node, it is possible that in this state a short-circuit condition could occur. Thus, this match is rejected. For the example in Figure 5.13, after the algorithm in Figure 5.9 is applied, *shortsop*

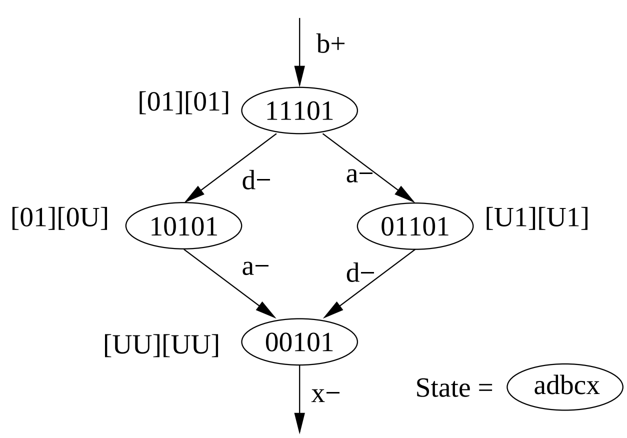


Figure 5.14. Segment of Ebergen output x state graph.

contains the product $pq\bar{r}\bar{s}$. A search of the portion of the state graph in Figure 5.14 now tries to identify bit patterns that could have a binary condition of 1100 for node 0, node 2, primary input a, and primary input d, respectively. The only state where this condition can occur is state 00101. Thus, the *gc22* library element must also be rejected as a potential match.

The rejection of a potential cover is a conservative estimate in cases where stability is concerned. Nodes under consideration could be stable but the abstraction algorithms of Chapter 3 may provide insufficient information to determine stability with certainty. Thus, any indication of instability when an internal node is considered for short-circuits must be treated conservatively.

5.4 Matching Common-Inputs

The final issue addressed by the matching algorithm is that of matching library elements with common-inputs. Common-inputs occur when a library cell decomposition places the same input on different leaves. Normally, this cannot happen in tree-based matching, with the exception of primary inputs, because a common-input on an internal node of the subject graph would fanout to multiple nodes. However, tree-based decompositions can accommodate common-inputs providing equivalent subnetworks drive the internal nodes where the common-inputs are placed.

An example of a common-input library cell is that of the XOR function. In a decomposition allowing reconvergent fanout, the decomposition shown in Figure 5.15(a) is acceptable. However, in a tree-based system, the decomposition required is shown in Figure 5.15(b). This places an additional constraint on the matching algorithm in that both nodes labeled a in Figure 5.15(b) must be driven by equivalent subnetworks, as must both nodes labeled b.

This common-input situation occurs fairly frequently in timed asynchronous technology-mapping, particularly when gC's are present in the library. The requirement that equivalent subnetworks drive common-inputs in tree-based designs often results in many gC's of this type being rejected as potential matches. However, gC's with common-inputs provide a significant benefit in that they eliminate the

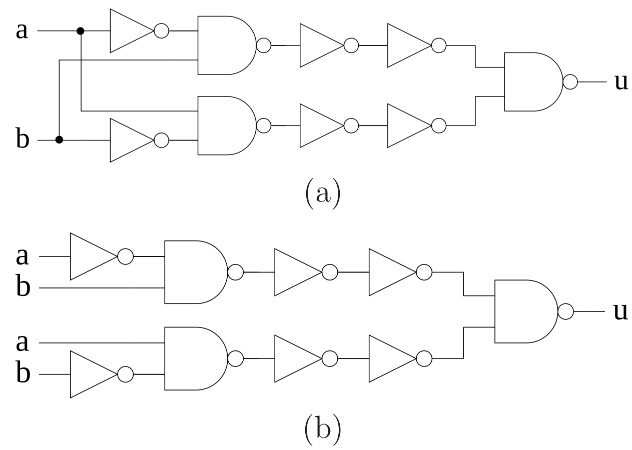


Figure 5.15. XOR decompositions. (a) Reconvergent fanout XOR decomposition. (b) Tree-based XOR decomposition.

possibility of short-circuits in the transistor stack. This is shown in Figure 5.16 where $f_u^{set} = ab$ and $f_u^{reset} = \bar{a}\bar{c}$. Figure 5.16(a) shows the logical decomposition and Figure 5.16(b) shows the CMOS level implementation. It is easily seen in Figure 5.16(b) that the common-input a prevents any short-circuit from occurring in the transistor stack.

Once it has been determined that the subnetworks driving common-input pins are equivalent, only one instance of the subnetwork needs to be included in the covered netlist. This is equivalent to gate-sharing and decreases the number of cells

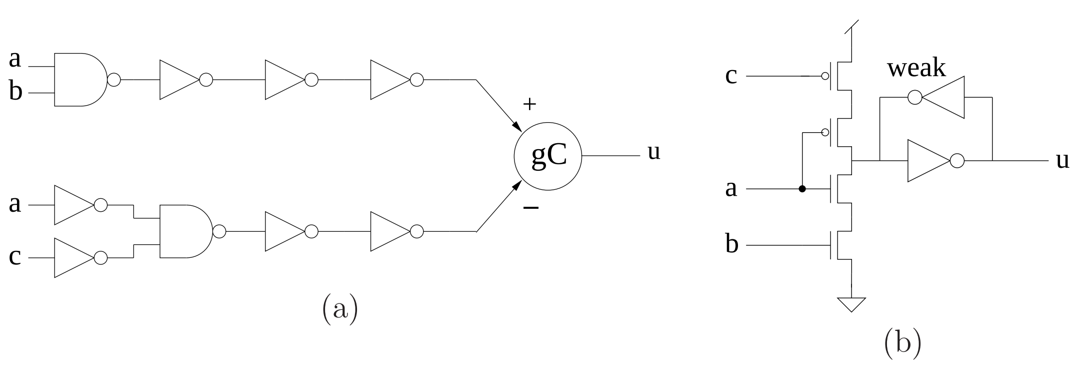


Figure 5.16. *gc22* library element structure. (a) Logical decomposition. (b) CMOS level implementation.

in the final netlist.

The algorithm for matching common-inputs is shown in Figure 5.17. The algorithm takes as inputs a decomposed form of a library element, *dl*, and a netlist, *pn*. Note that when this algorithm is called from the `matcov` algorithm of Figure 5.1, *dl* has already been matched to a subgraph of *pn*, rooted at the head of *pn*.

The first step of the algorithm of Figure 5.17 is to create a product, *inputprod*, formed from the input set of the library element, *dl*. For instance, the input set for the gC element shown in Figure 5.18(a) is $\{\{p\},\{q\},\{r\}\}$ and $inputprod = pqr$. The next step is to make a node list of all the nodes in *pn* that map to each literal of *inputprod*. For example, the input literal p of Figure 5.18(a) maps to internal node 0 and primary input a in Figure 5.18(b). These two nodes are then checked to see if the subcircuits driving them are equivalent. If they are not equivalent, as is the case here, then this library cell is rejected as a potential match.

As is illustrated in Chapter 4, the permuting of inputs potentially increases the number of matches when using libraries with common-input cells. For instance, the pattern graphs of Figure 5.18(a) and 5.18(c) are structurally equivalent, the only difference being that the inputs have been permuted. Now, when the common-input algorithm of Figure 5.17 is run using the pattern graph of Figure 5.18(c), the node list created for the p input of Figure 5.18(b) is internal node 2 and primary input d.

```
common_input (dl,pn) {
  inputprod = find_inputs(dl)
  foreach lit ∈ inputprod {
    nodelist = make_nodelist(lit,dl,pn)
    if (|nodelist| ≠ 1) then {
      node0 = nodelist(0)
      foreach node ∈ {nodelist - nodelist(0)}
        if !(struct_match(node0,node)) then
          return FALSE
    }
  }
  return TRUE
}
```

Figure 5.17. Algorithm for common-input matching.

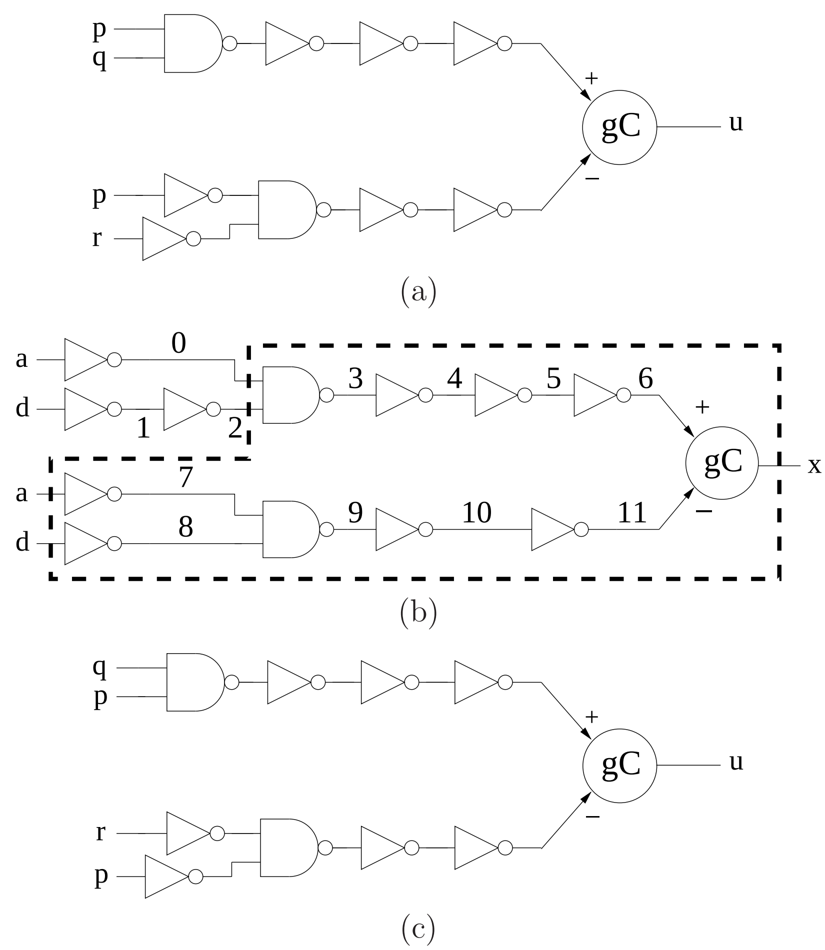


Figure 5.18. Common-input matching example. (a) Pattern graph with $f_u^{set} = pq$ and $f_u^{reset} = \bar{p}\bar{r}$. (b) Ebergen output x. (c) Pattern graph with permuted inputs.

Once the double buffer driving node 2 is removed, these two nodes are equivalent and the algorithm returns TRUE.

CHAPTER 6

EXPERIMENTAL RESULTS

This chapter presents the results achieved by running a number of examples through the algorithm implementation. The first results presented are those relating to verification runs measured against other industry tools. These are followed by results of the matching and covering phases of the technology-mapping flow using increasing complexities of libraries. Finally, this chapter concludes with several short case studies, each elaborating on an interesting aspect of the completed work.

6.1 Verification

6.1.1 Verification of Benchmark Files

The gate-level timing verification method described in this dissertation has been implemented and tested on numerous examples. Table 6.1 compares the new gate-level timing verification method using standard benchmarks against results for the timed automata tool `KRONOS` [80], a conservative approximation method described in [85], and the `ATACS` explicit state timing verifier [86]. All runtimes are specified in CPU seconds. For `KRONOS` runtimes, an entry with a question mark indicates the amount of time after which the verification ran out of memory. The runtimes for `KRONOS` and Pena's methods are taken from their papers while the runtimes for `ATACS` and the new method are from a 900 MHz Pentium 4 with 256MB of memory. For the new method, an entry of n/a indicates that this example has an internal cycle and cannot be analyzed using the new method. For the smaller examples, the new method has comparable and usually better runtimes than the other methods. However, for larger examples with more concurrency such as *trimos-send*, the new method is more than two orders of magnitude faster than `KRONOS`, 25 times faster

Table 6.1. Comparison of standard benchmarks against other timing verification tools.

		KRONOS	PENA	ATACS		New Method		
Example	Gates	Time(s)	Time(s)	Time(s)	Mem(MB)	Time(s)	Mem(MB)	Hazards
alloc-outbound	11	0.09	3	0.33	5.6	0.09	2.9	0/0
chu133	9	0.63	1	0.16	3.0	0.11	2.2	1/1
converta	12	0.19	12	0.24	3.8	0.11	1.8	2/2
dff	6	0.19	3	0.12	2.5	n/a	n/a	3/?
ebergen	9	0.14	1	0.15	3.0	0.13	1.8	3/3
half	7	0.41	1	0.13	2.2	0.08	1.5	1/1
mp-forward-pkt	10	0.24	5	0.17	3.5	0.10	2.5	0/0
nowick	10	0.05	3	0.20	3.8	0.10	2.0	0/0
rcv-setup	6	0.22	1	0.16	3.2	0.08	1.8	0/0
rpdft	8	2.93	2	0.30	4.0	0.10	1.9	1/2
sbuf-ram-write	17	31.77	415	0.32	5.8	0.20	3.7	1/2
sbuf-read-ctl	10	0.13	2	0.14	3.3	0.10	2.5	0/0
sbuf-send-ctl	13	54	0.49	0.65	6.1	0.10	2.8	1/1
sbuf-send-pkt2	13	0.07	103	0.42	6.6	0.10	3.1	0/1
vme	12	0.39	30	0.39	4.9	n/a	n/a	1/?
mr1	16	607.43	317	0.30	5.1	n/a	n/a	0/?
tsend-bm	12	589.56	46	5.32	8.6	n/a	n/a	1/?
mmu	22	595.09?	480	0.53	7.1	n/a	n/a	0/?
mr0	20	593.24?	48	0.55	7.1	n/a	n/a	0/?
ram-read-sbuf	17	678.48?	550	0.34	6.0	0.18	3.4	0/0
trimos-send	24	580.33?	127	10.7	25.0	4.87	3.6	5/5

than Pena's tool, and twice as fast as the explicit state method in ATACS. In addition, the new method shows some reduction in memory usage as compared to the ATACS explicit state timing verifier. This reduction in run time and memory usage is directly related to the reduced complexity of the state graph, as stated earlier.

Since the goal of this portion of the research is to determine which gates have hazards on their outputs, the explicit method in ATACS is configured to continue after finding one hazard and identify all hazards. It should be noted that KRONOS did not check for hazards, but instead is only checking conformance while Pena's tool halts after a hazard is found. The last column of the table indicates the number of gates that have hazards found by the explicit state method and the new method. Despite being a conservative approximation, the new method found the exact number of hazards in most cases. However, in three examples, *rpdft*, *sbuf-ram-write*, and *sbuf-send-pkt2*, the new method found one additional false hazard. These false hazards and why they occur are explained by example in Section 6.4.1.

6.1.2 Verification Using Decomposed Circuits

The key advantage of this new method is its ability to be able to efficiently verify circuits with a large number of internal signals. In order to demonstrate this, a few benchmark circuits derived from a variety of sources are selected and shown in Table 6.2, and gate-level circuits are derived for them that use only 2-input NAND

Table 6.2. Comparison for decomposed netlists.

		ATACS		New Method		
Example	Gates	CPU Time(s)	Mem(MB)	CPU Time(s)	Mem(MB)	Hazards
scsiSV	18	1.35	7.9	0.13	1.3	0/0
slatch	29	33.5	53.4	0.15	1.8	0/0
lapbsv	37	20.0	41.5	0.17	1.3	0/0
elatch	38	183	229	0.28	1.8	0/0
cnt3	80	>1000	>256	0.24	1.7	?/15
srgate	85	>1000	>256	0.29	2.3	?/0
selopt	164	>2000	>256	0.90	3.3	?/46
cnt11	213	>2000	>256	1.20	4.8	?/78

gates and inverters. In all the examples, the new method is still able to check for hazards in 1.2 seconds or less while, for the largest examples, the explicit state method cannot complete.

6.1.3 Timed and Untimed Stabilization

The new verification method uses a combination of timed and untimed algorithms to determine hazard-freedom for each node and each output in a netlist. It is found that stabilizations in the state graph due to timing occur much more frequently than do stabilizations using untimed (speed-independent) methods. These results are shown in Table 6.3 and they are not surprising because the delays used in the base-function library elements are fairly small (but physically practical) so circuit delays in the decomposition, up to the node of interest, are often small and stabilization through the state graph occurs reasonably quickly.

The surprising result here is how effective the timed stabilization algorithms are.

Table 6.3. Hazard comparison based on stabilization method.

Example	Gates	One method Untimed	One method Timed	Timed/ Untimed
alloc-outbound	11	6	0	0
chu133	9	7	1	1
converta	12	8	2	2
ebergen	9	4	3	3
half	7	7	1	1
mp-forward-pkt	10	6	0	0
nowick	10	7	0	0
ram-read-sbuf	17	13	0	0
rcv-setup	6	5	0	0
rpdft	8	8	2	2
sbuf-ram-write	17	12	2	2
sbuf-read-ctl	10	6	0	0
sbuf-send-ctl	13	11	1	1
sbuf-send-pkt2	13	11	1	1
trimos-send	24	18	5	5

The numbers in the last three columns indicate how many nodes in each circuit are found to be hazardous. When untimed stabilization alone is used, in all cases but one, over half the nodes are hazardous. When timed stabilization only is used, this number is reduced considerably. The last column indicates the results achieved by first running timed stabilization, followed by untimed stabilization. Note how timed stabilization alone gives identical results to the case where timed stabilization is followed by untimed stabilization. This is a potentially significant finding in that it says, at least for these examples, that there is no need to do untimed stabilization. Since untimed stabilization may need to be iterated (unlike timed stabilization), a cost savings in computation time, without apparent loss of accuracy, occurs if timed stabilization is run by itself.

6.2 Matching and Covering

The matching and covering algorithms require implementation libraries that contain cells with known area and delay parameters. In this section, definitions are given for libraries that are created and used to test the new algorithms. Then, results are presented of how effective the new algorithms are at producing hazard-free implementations with increasing complexities of libraries. Next, results are presented showing the effect of various parameter shifts in trying to find an optimum matching and covering while doing hazard-aware matching. Finally, the cost of using hazard-freedom as a primary metric is measured against area and delay optimized implementations.

The matching and covering results are divided into three parts. First, an untimed synthesis and untimed verification of the decomposition is followed by an untimed verification of the covered netlist. This data represents the speed-independent implementation of the example suite. Second, an untimed synthesis and untimed verification of the decomposition is followed by a timed verification of the covered netlist. This gives some insight into how effective the algorithms are at producing hazard-free circuits in the presence of a large number of hazards. Third, a timed synthesis and timed verification of the decomposition is performed followed by a timed verification of the covered netlist. This data represents the thrust of this research, that is, taking timing into all steps of the technology-mapping process in an attempt to show that some or all speed-independent hazards will not manifest due to timing.

6.2.1 Implementation Libraries

In order to determine how best to find hazard-free coverings for decomposed netlists, four different libraries are created with increasing order of complexity. Each library adds a basic new structure and each succeeding library is a superset of the preceding library. These four libraries are the base-function, combinational, single-stack, and common-input libraries. It should be noted that a fifth library, a full-implementation library, could be created that contains some additional cells that have been customized to fully implement f_u^{set} or f_u^{reset} or both. Many of these cells would rely heavily on the atomic gate assumption and would likely embed inverters and additional logic. These types of cells place additional design burdens on the physical design and layout process. Since one of the goals of this work is to use standard library cells whenever possible, every effort is made to avoid the use of these customized cells, noting that it is possible to eliminate all hazards from a circuit implementation by using library elements that are functionally equivalent to the synthesis output.

The first library consists of the base-function set of an inverter, 2-input NAND, CEL, plus a plain gC to be used where short-circuits are not present. Using this library creates a covered netlist that is equivalent to the decomposition, leaving all internal nodes exposed. The resulting verification of the covered netlist mirrors the hazards found during the verification of the decomposed netlist.

The combinational library adds to the base-function library a basic set of combinational cells. These include 2-, 3-, and 4-input cells for AND, OR, NAND, and NOR functions. In addition, there are a variety of other combinational functions such as XOR, XNOR, AOI, and OAI. The total number of cells in this library including common-input permutations is 82.

The single-stack library adds to the combinational library a set of cells that implement single-stack gC's. All single-stack cells are implemented, from one input to each of the set and reset stacks to four inputs each. This library can implement large portions of the decomposition but is subject to short-circuit problems. An example cell is shown in Figure 6.1(a). There are an additional 15 cells added to the combinational library but there are no additional common-input cells added.

The common-input library adds to the single-stack library a set of gC cells with up to four inputs each on the set and reset stacks, just like the single-stack library elements. The difference here is that from one to all four of the inputs can be common between the set and reset stacks. An example cell is shown in Figure 6.1(b). With the number of common-input permutations necessary to do efficient matching, the number of cells in this library increases to 3133.

6.2.2 Matching Using Speed-Independent Synthesis

Table 6.4 compares the reduction in the number of hazards as the complexity of the library available for matching increases. The example suite is a subset of those used in Tables 6.1 and 6.2. These examples are synthesized and the decomposition is verified speed-independently using the `ATACS` tool. The covered netlist is then verified in two separate runs, once speed-independently and once using timing. The numbers in Table 6.4 represent the number of hazardous nodes left in the covered netlist after final verification. It should be noted that two examples, *srgate* and

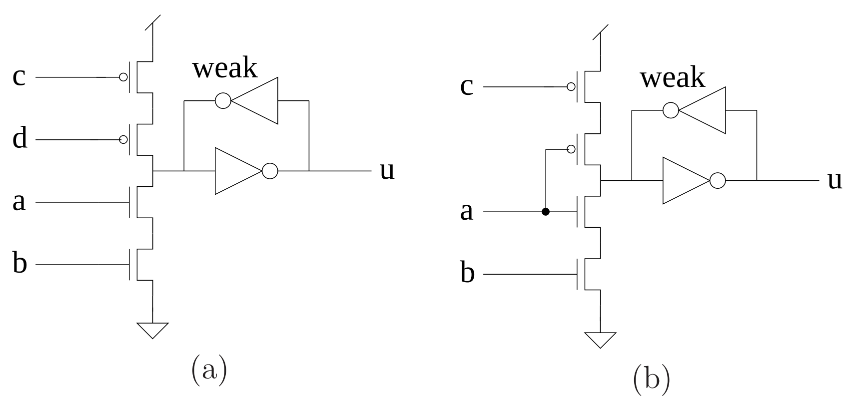


Figure 6.1. Example library cells. (a) Single-stack. (b) Common-input.

Table 6.4. Hazard reduction for speed-independent synthesis.

	(Untimed Verification)				(Timed Verification)			
Example	Sim	Com	SS	CI	Sim	Com	SS	CI
alloc-outbound	14	0	0	0	0	0	0	0
chu133	26	4	4	4	4	0	0	0
converta	45	5	5	5	0	0	0	0
ebergen	26	4	4	3	7	2	2	0
half	17	2	2	2	0	0	0	0
mp-forward-pkt	21	2	2	2	0	0	0	0
nowick	24	2	2	2	0	0	0	0
ram-read-sbuf	31	2	2	2	0	0	0	0
rcv-setup	21	2	2	2	0	0	0	0
rpdft	43	5	5	5	0	0	0	0
sbuf-ram-write	33	4	4	4	0	0	0	0
sbuf-read-ctl	14	2	2	2	0	0	0	0
sbuf-send-ctl	25	4	4	4	5	0	0	0
sbuf-send-pkt2	40	6	6	6	0	0	0	0
trimos-send	121	26	26	28	42	8	8	5
scsiSV	23	4	4	4	0	0	0	0
slatch	14	6	6	6	0	0	0	0
lapbsv	26	2	2	2	0	0	0	0
elatch	24	8	8	8	0	0	0	0
cnt3	60	5	5	4	12	0	0	0
cnt11	177	17	17	15	52	2	2	2

selopt, are unable to synthesize in under 1000 CPU seconds and are not shown in Table 6.4.

The data in Table 6.4 are divided into two sets. The four abbreviations in the second row of Table 6.4 for each data set represent the library used. For example, the *Sim* column represents data taken using the simple library. Both sets of data are compiled with the hazard-aware match coefficients w_1, w_2 = (-1,+1) as explained in Section 5.2. This combination of coefficients is the case where the matching algorithm works to avoid encapsulating nodes with monotonicity hazards while at the same time avoiding leaving any nodes with acknowledgment hazards exposed.

As expected, the speed-independent synthesis creates a hazard landscape with a considerable number of hazardous nodes in the decomposition, as evidenced by the number of hazards in the leftmost *Sim* column. The purpose of this data comparison is to see how effective the hazard-aware matching algorithms perform in the presence of a significant number of hazardous nodes when timing is not used. The numbers in the four leftmost columns of Table 6.4 indicate the number of hazardous nodes remaining in the mapped netlist when timing is not used in verifying the mapped netlist. The numbers in the four rightmost columns indicate the number of hazardous nodes remaining in the mapped netlist when timing is used in verifying the covered mapped netlist.

The data for untimed verification in Table 6.4 show that it is nearly impossible to produce a hazard-free circuit without using timing. In only one instance, *alloc-outbound*, did the algorithms produce a hazard-free circuit with speed-independent verification. The data for timed verification in Table 6.4 show that in all but two examples, using timing information produces a hazard-free circuit. Another interesting aspect to note is that nearly all hazard reduction from the initial hazard landscape occurs when the combo library is used. The data for both timed and untimed verification shows that the single-stack library has absolutely no effect on reducing hazards. While this is true for these examples, this library did, on occasion, reduce the number of gates in the final netlist over the number from the combo library, although there is no reduction in the number of hazards. The common-input library has a positive, although minimal, effect in reducing the number of hazardous nodes in both cases.

6.2.3 Matching Using Timed Synthesis

The data in Table 6.5 are compiled using timing in all steps of synthesis and technology-mapping. The libraries and examples are the same as those used in Table 6.4. In addition, the two files *srgate* and *selopt*, are able to synthesize when timing is used. Once again, the numbers in Table 6.5 represent the number of hazardous nodes left in the covered netlist after final verification of the covered netlist. Due to timing, only a handful of circuits have hazardous nodes in the

Table 6.5. Hazard reduction for timed synthesis.

Example	Simple	Combo	S-Stack	Com-Inps
alloc-outbound	0	0	0	0
chu133	0	0	0	0
converta	0	0	0	0
ebergen	7	2	2	0
half	0	0	0	0
mp-forward-pkt	0	0	0	0
nowick	0	0	0	0
ram-read-sbuf	0	0	0	0
rcv-setup	0	0	0	0
rpdft	0	0	0	0
sbuf-ram-write	0	0	0	0
sbuf-read-ctl	0	0	0	0
sbuf-send-ctl	5	0	0	0
sbuf-send-pkt2	0	0	0	0
trimos-send	21	3	3	0
scsiSV	2	0	0	0
slatch	0	0	0	0
lapbsv	2	0	0	0
elatch	0	0	0	0
cnt3	21	0	0	0
cnt11	52	0	0	0
srgate	0	0	0	0
selopt	126	19	19	17

decomposition (as revealed by mapping with the *Simple* library) and a completely hazard-free mapping is found in all cases except for one, *selopt*. It is clear when comparing Tables 6.4 and 6.5, that utilizing timing during synthesis and technology-mapping is crucial to the goal of producing hazard-free implementations when using technology-dependent libraries.

6.2.4 Hazard-Aware Matching

The results shown in Tables 6.4 and 6.5 are computed based on hazard optimization using fixed values of w_1 and w_2 shown in Equation 5.1. These weighting factors are used in an effort to find a hazard-free solution when the final netlist has remaining hazards. Table 6.6 shows the results for various values of w_1 and w_2 when the example circuits are synthesized and verified speed-independently. The numbers in Table 6.6 indicate the number of hazardous nodes remaining in the netlist following final verification. Since there is little difference between the results using the combo and the common-input libraries of Table 6.4, Table 6.6 shows the results using the common-input library.

It should be noted that data using timed verification are not shown in Table 6.6 because there is virtually no effect on the number of hazards in the final netlist

Table 6.6. Hazard reduction using various weighting factors.

	(w_1,w_2)					
Example	(0,0)	(-1,0)	(0,1)	(1,1)	(-1,1)	(-1,2)
alloc-outbound	7	3	0	0	0	0
chu133	4	10	4	4	4	4
converta	11	16	5	2	5	5
ebergen	5	8	3	5	3	3
half	4	8	2	2	2	2
mp-forward-pkt	5	11	2	4	2	2
nowick	4	2	2	2	2	2
ram-read-sbuf	7	12	2	2	2	2
rcv-setup	2	2	2	2	2	2
rpdft	4	9	4	4	5	4
sbuf-ram-write	5	13	4	5	4	4
sbuf-read-ctl	4	4	2	4	2	2
sbuf-send-ctl	2	13	2	2	4	4
sbuf-send-pkt2	8	12	6	6	6	6
trimos-send	24	45	28	24	28	24
scsiSV	5	8	4	4	4	4
slatch	3	8	3	6	6	6
lapbsv	2	10	2	2	2	2
elatch	6	11	7	8	8	8
cnt3	11	20	6	4	4	4
cnt11	20	40	15	17	15	15
total	143	265	105	109	110	105

based on the coefficients. This is not surprising since timed stabilization has shown all along that it verifies hazard-free in most examples. The analysis now moves column by column through Table 6.6.

The first column, where w_1,w_2 = (0,0) represents the case where there is absolutely no hazard awareness during matching. The matching then is optimized based on a secondary cost factor, in this case area. Good results under these conditions are pure luck and more often than not the number of hazards in the final circuit is high in comparison to other coefficient pairs. The second column, where w_1,w_2 = (-1,0), represents the case where decisions are based only on monotonicity hazards. The minus sign on the w_1 coefficient indicates that leaving monotonicity hazards exposed (the lower the number the better) is desirable. The results here, and also in column one, indicate that ignoring acknowledgment hazards when making matching choices yields poor results. Column three, w_1,w_2 = (0,1) is the case where nodes causing monotonicity hazards are ignored and matches that leave acknowledgment hazards exposed are penalized. In most cases, but not all, this column is a definite improvement over columns one and two. Column four sets both w_1 and w_2 to one, which penalizes exposed hazardous nodes and also penalizes exposed monotonicity nodes. In other words, this column attempts to encapsulate as many hazards as possible. The results here are mixed. In some cases there is an improvement over column three, in other cases there are worse results. Column five is a duplicate of the results of the speed-independent verification portion of Table 6.4. Column six, where w_1,w_2 = (-1,2), gives twice the weight to discouraging leaving acknowledgment hazards exposed as opposed to leaving monotonicity hazards exposed. In general, the results of this column are a slight improvement over that of column five, indicating again that perhaps the removal of acknowledgment hazards is more important than tinkering with monotonicity hazards.

In all the results of Table 6.6, it has become clear that there is no holy grail for optimizing the matching of all circuits with one set of coefficients. Some weights work better on some circuits than other weights do. The total number of hazardous nodes shown in the last row gives a little more insight into which column is best, although these numbers can be skewed by examples with a large number of hazards. On average, the lowest number of hazards is seen for (0,1) and (-1,2) coefficient pairs, with the next best pairs being (1,1) and (-1,1). Clearly, the first two columns are cases to avoid.

Another way to interpret the results of Table 6.6 is to consider in how many examples one pair of coefficients outperforms another pair. Table 6.7 shows how the w_1,w_2 = (0,1) compares to the other five coefficient pairs. A win means that the final verification produced fewer hazardous nodes.

The following four observations are made from comparing the data in Tables 6.6 and 6.7:

- The results of columns one and two in Table 6.6 clearly indicate that acknowledgment hazards *must be encapsulated*.
- The strong showing of the w_1,w_2 = (-1,2) pair suggests that more weight be given to addressing acknowledgment hazards than addressing monotonicity hazards.
- The strongest coefficient pair of w_1,w_2 = (0,1) suggest that perhaps monotonicity hazards could be ignored altogether.
- Since no one coefficient pair always produced the best results, and since the hazard verification algorithms have proven to be fast and efficient, it may be

Table 6.7. Comparing w_1,w_2 = (0,1) by wins and losses.

(w_1,w_2)	Wins	Losses
0,0	13	2
-1,0	19	0
1,1	7	3
-1,1	4	1
-1,2	3	2

prudent to try multiple runs during technology mapping and pick the best result.

6.2.5 Short-Circuit Rejection for gC's

It is pointed out in Section 5.3 that all gC library elements must pass a short-circuit check to be considered as a best match. The data shown in Table 6.8 indicate the percentage of gC elements that structurally match a portion of the subject graph but are rejected due to short-circuit problems.

Table 6.8. Short-circuit rejection of gC's.

Example	State Outputs	Struct Matches	SC Reject	Reject (%)
alloc-outbound	3	5	3	60
chu133	2	4	2	50
converta	3	14	1	7
ebergen	2	8	4	50
half	2	4	3	75
mp-forward-pkt	3	9	8	89
nowick	3	5	4	80
ram-read-sbuf	4	19	12	63
rcv-setup	1	4	4	100
rpdft	0	0	0	100
sbuf-ram-write	3	10	2	20
sbuf-read-ctl	3	7	4	57
sbuf-send-ctl	3	7	4	57
sbuf-send-pkt2	3	5	3	60
trimos-send	6	24	18	75
scsiSV	3	8	8	100
slatch	2	4	3	75
lapbsv	3	7	7	100
elatch	2	8	6	75
cnt3	4	21	13	62
cnt11	8	49	0	0
srgate	1	1	1	100
selopt	5	21	21	100
total	69	244	131	53.7%

Nearly all of the example circuits in Table 6.8 have multiple state-holding outputs. The exceptions are *rcv-setup* and *srgate*, each of which has one state-holding output, and *rpdft*, which has no state-holding outputs. There are six examples where every gC element that matches structurally is rejected because of short-circuit problems. In these cases, all of the state-holding outputs are implemented with CEL's. In one example, *cnt11*, there are no short-circuit problems found for any of the structural matches. Of the eight outputs in this *cnt11*, all are implemented with some type of gC gate.

6.3 The Cost of Hazard-Freedom

Although hazard-freedom must be the driving cost factor behind this research, it is interesting to evaluate the penalty paid for this focus. To measure what is sacrificed to gain hazard-freedom, the example suite is run through the implementation, letting area and delay be the primary cost factors. This gives optimum area and delay numbers, relegating hazard-freedom to a secondary consideration for the purposes of comparison.

Table 6.9 shows the cost of reducing hazards measured against area and delay for speed-independent synthesis and verification. The leftmost column of data represents the initial number of hazards in the decomposition. The next four columns, under the title of *Area Emphasis*, show the increase in area needed to reduce the number of hazards in the final netlist. Under the subtitle of *Hazards*, the leftmost column, (*Area*), indicates the number of hazards left when area is the primary cost factor. The *Haz* column indicates the number of hazards left when hazard reduction is the primary cost factor and minimizing area is the secondary cost factor. The next two columns show the increase in area as the penalty paid to achieve this reduction in the number of hazards. Note that in all cases except one, *elatch*, the number of hazards remains the same or is reduced. The average reduction in the number of hazards is 28.2 percent (142 to 102) at an average area increase of 5.5 percent (6748 to 7116). The four rightmost columns show the cost in circuit delay in order to achieve a reduction in the number of hazards. Here,

Table 6.9. Cost of hazard reduction for untimed synthesis and verification.

Example	Init Haz	Area Emphasis: Hazards Area	Area Emphasis: Hazards Haz	Area Emphasis: Area Area	Area Emphasis: Area Haz	Delay Emphasis: Hazards Del	Delay Emphasis: Hazards Haz	Delay Emphasis: Delay Del	Delay Emphasis: Delay Haz
alloc-outbound	14	7	0	344	376	4	0	42	44
chu133	26	4	4	172	172	4	4	30	30
converta	45	10	4	392	416	9	4	37	44
ebergen	26	5	5	216	240	6	4	26	32
half	17	4	2	144	148	4	2	16	20
mp-forward-pkt	21	5	2	280	312	5	2	38	44
nowick	24	4	2	432	440	5	2	40	40
ram-read-sbuf	31	7	2	428	488	7	2	52	60
rcv-setup	21	2	2	112	112	2	2	19	19
rpdft	43	4	4	216	232	8	5	19	21
sbuf-ram-write	33	5	4	304	320	5	4	44	48
sbuf-read-ctl	14	4	2	264	272	4	2	38	40
sbuf-send-ctl	25	2	2	280	280	4	2	41	43
sbuf-send-pkt2	40	8	6	388	396	11	7	52	56
trimos-send	121	24	24	804	804	32	24	90	102
scsiSV	23	5	4	264	312	9	4	30	36
slatch	14	3	3	156	156	3	3	22	22
lapbsv	26	2	2	212	212	6	2	28	30
elatch	24	6	7	184	232	6	7	24	28
cnt3	60	11	6	352	384	11	6	43	52
cnt11	177	20	15	804	812	23	16	102	109
total	825	142	102	6748	7116	168	104	833	920

the two columns under the subtitle of *Hazards* show that again, in all cases except for *elatch*, the number of hazards is reduced or remains the same. The average reduction in the number of hazards is 38.1 percent (168 to 104) at an average delay increase of 10.4 percent (833 to 920).

Table 6.10 shows the same type of data as that seen in Table 6.9, only this time the results are computed using timed synthesis and verification. As seen in Table 6.5, the number of initial hazards is considerably fewer than found in speed-independent analysis. There are eight example circuits in Table 6.10 with

Table 6.10. Cost of hazard reduction for timed synthesis and verification.

		Area Emphasis				Delay Emphasis			
	Init	Hazards		Area		Hazards		Delay	
Example	Haz	Area	Haz	Area	Haz	Del	Haz	Del	Haz
alloc-outbound	0	0	0	328	328	0	0	40	40
chu133	0	0	0	160	160	0	0	26	26
converta	0	0	0	368	368	0	0	37	37
ebergen	7	0	0	172	172	0	0	26	26
half	0	0	0	104	104	0	0	14	14
mp-forward-pkt	0	0	0	272	272	0	0	37	37
nowick	0	0	0	424	424	0	0	39	39
ram-read-sbuf	0	0	0	320	320	0	0	43	43
rcv-setup	0	0	0	112	112	0	0	19	19
rpdft	0	0	0	216	216	0	0	19	19
sbuf-ram-write	0	0	0	292	292	0	0	40	40
sbuf-read-ctl	0	0	0	228	228	0	0	33	33
sbuf-send-ctl	5	0	0	244	244	0	0	36	36
sbuf-send-pkt2	0	0	0	388	388	0	0	52	52
trimos-send	21	0	0	444	444	0	0	54	54
scsiSV	2	0	0	280	280	0	0	32	32
slatch	0	0	0	152	152	1	1	22	22
lapbsv	2	0	0	200	200	0	0	28	28
elatch	0	0	0	192	192	0	0	22	22
cnt3	21	0	0	352	352	0	0	43	43
cnt11	52	2	0	720	752	0	0	100	100
srgate	0	0	0	336	336	0	0	31	31
selopt	126	19	17	672	744	22	17	84	96
total	236	21	17	5584	5698	23	18	771	783

initial hazards and a hazard-free implementation is found in all but two of these using area as the primary cost factor. When using hazard reduction as the primary cost factor, Table 6.10 shows there is only one example, *selopt*, that can not be implemented hazard-free. Here, there is also an area penalty paid to reduce the number of hazards. Note that when no initial hazards are present, the algorithms optimize on the secondary cost factor. Thus, the area numbers are identical when no initial hazards are present in the decomposition. In the case of the four rightmost columns in Table 6.10, the delay numbers are identical except in the one case where hazards are reduced, *selopt*. Note also that a hazard is created in the *slatch* example where one is not present initially. This is due, in both cases, to a monotonicity violation that is created as a result of the mapped circuit.

The area numbers for Tables 6.9 and 6.10 are computed by summing the individual areas of all outputs for the given example. This summation represents the total area for the mutually exclusive cones of logic for each output because gate-sharing is not implemented. The delay numbers for Tables 6.9 and 6.10 are computed by summing the maximum individual delays of all outputs for the given example. This summation represents the cumulative delay for the mutually exclusive cones of logic for each output. These numbers, in no way, take into account any parallel or concurrent operation of the individual outputs of any one example. It should be noted that the hazard-aware matching coefficients w_1,w_2 are set to (0,1) when compiling the data in Tables 6.9 and 6.10. This pair of coefficients gives the best results for the matching algorithms as seen in Table 6.7.

Clearly, the data in Tables 6.9 and 6.10 shows that the initial number of hazardous nodes in the decomposition plays a large role in determining whether or not a hazard-free solution can be achieved, and the area and delay penalty paid for doing so. Tables 6.9 and 6.10 also show that timed synthesis and verification outperform speed-independent synthesis and verification in terms of creating hazard-free implementations.

6.4 Case Studies

There are three short case studies presented in this section, each illustrating an interesting aspect of the technology-mapping work: false hazards, nonpropagating internal hazards, and a nonoptimized hazard cover.

6.4.1 False Hazards

It is observed in Sections 6.1.1 and 6.1.2 that false negative hazards, although infrequent, do occur. This is a result of the abstraction method, which limits the visible states to those contained in the Complex Gate Equivalent state graph.

Between any two states in this state graph, a number of internal signals can be undergoing an ordered sequence of transitions. The stabilization algorithms do not always find internal nodes to be stable and if more than one input to a gate is unstable in the same state and a forcing side input cannot be found, than a monotonicity hazard is reported.

The example used in Section 3.2.3 is shown again in Figure 6.2, where the false hazard reported on output node t in state 10001 caused by fanin $b58$ is now investigated. Starting in state 00001, signal $b58$ is low and signals $b55$ and output t are high. When signal d rises and and the circuit moves to state 10001, signal $b55$ is enabled to fall (through 2 gate delays) and signal $b58$ is enabled to rise (through 3 gate delays). However, the algorithm is not able to determine the order in which these internal nodes actually switch.

After the timed and untimed stabilization has been completed for the circuit in Figure 6.2, the stability information for state 10001 is shown on each internal node. Note that all external signals are stable at the values in the state vector. For the internal signals, a U indicates that this node is unstable and a 0 indicates that the node has stabilized at that value. Note also that all gates have at least one stable input except for the gate driving the output t. The algorithm to check for monotonicity hazards in Chapter 3 indicates that the output t has a monotonicity

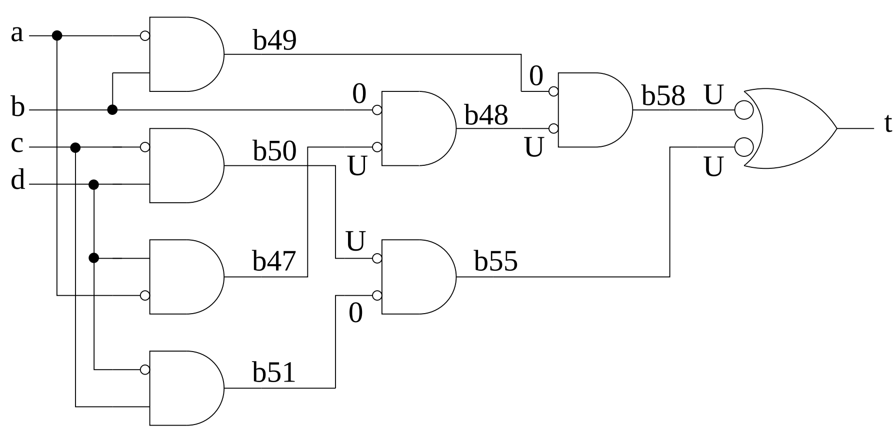


Figure 6.2. False hazard example using circuit *rpdft*.

hazard in state 10001 and it is caused by the $b58$ input.

To explore why this hazard is false, the full timed state graph for the region of interest must be examined. This state graph is shown in Figure 6.3. Note that between the time signal d rises and state 10001 is entered, and signal b rises when state 10001 is exited, internal signals $b47$, $b50$, $b58$ rise, and signals $b55$ and $b48$ fall. It is clear from this state graph that signal $b55$ falls before $b58$ rises. Thus, there is no actual state in this state graph where the ambiguity in Figure 6.2 is present.

6.4.2 Nonpropagating Acknowledgment Hazard

The intent of the verification portion of this research is to identify nodes where hazardous behavior is occurring. Then, the matching and covering phase of the design flow tries to find a circuit covering such that these hazardous nodes are removed. However, it is known that hazardous activity on internal nodes does not necessarily mean that the circuit fails. In other words, if hazards on internal nodes do not propagate to the output, the circuit as a whole may not be hazardous.

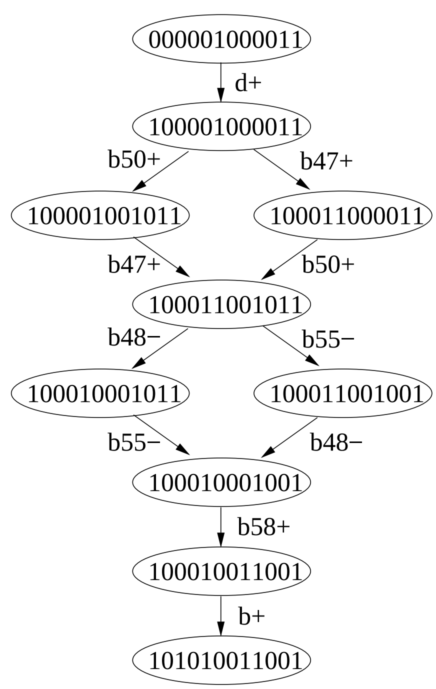


Figure 6.3. Full timed state graph for the region of interest in circuit *rpdft*.

An example of one such circuit (there are many) is shown in Figure 6.4(a). This circuit is called *half* and is taken from the examples used by the `KRONOS` tool. Figure 6.4(b) shows the state graph for the *half* circuit. Note that in the state graph, stability information for internal nodes *x94* and *x96* is placed in square brackets next to each state. After circuit verification, it is found that node *x96* has an acknowledgment hazard between states 0101 and 0111. This is seen in the

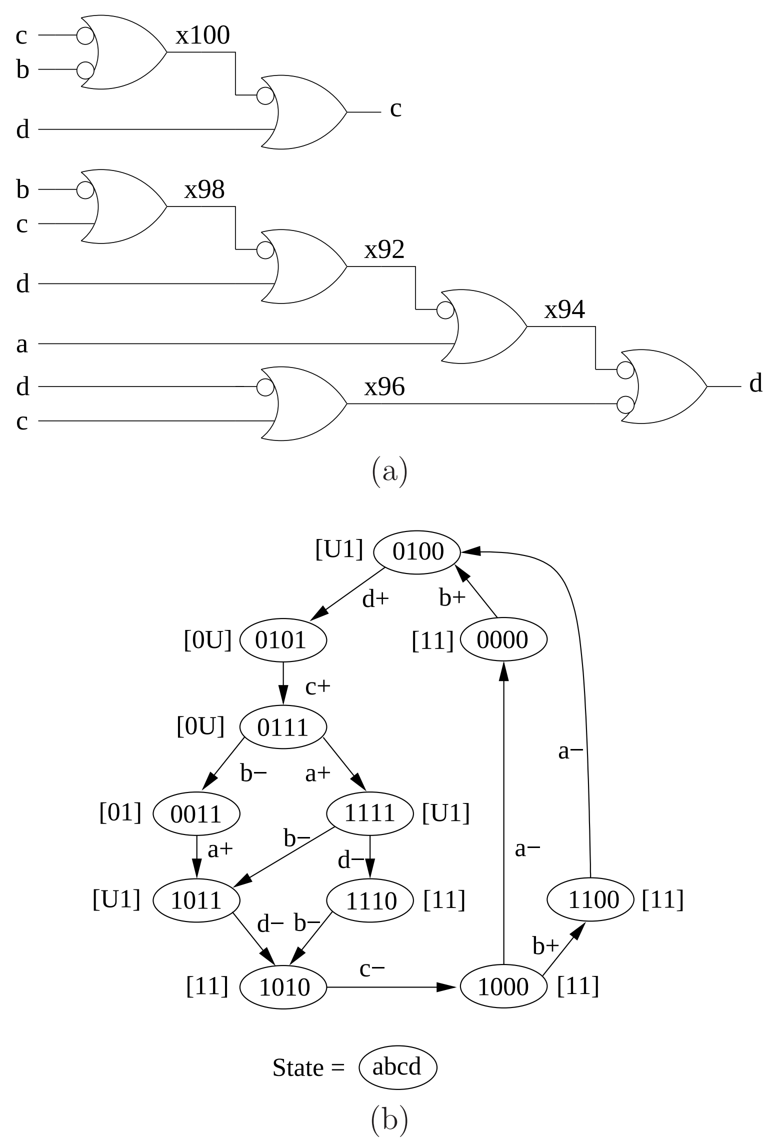


Figure 6.4. Nonpropagating acknowledgment hazard example. (a) *half* circuit. (b) *half* state graph.

state graph by noticing that node *x96* is unstable in both of these states but its evaluation changes, that is, *x96* evaluates to 0 in state 0101 but evaluates to 1 in state 0111. As is shown in the algorithm of Figure 3.11, an acknowledgment hazard is reported between two states under these conditions.

After circuit verification, it is also found that the output d is hazard-free. In other words, the acknowledgment hazard on node *x96* did not propagate to the output. This is because the output is held in a high state by node $x94$, which is stable at 0 during the state transition from 0101 to 0111.

The point of this example is that it may be possible to declare some circuits hazard-free even when there is hazardous activity on internal nodes. One reason this is often the case is because of blocking side inputs such as in the example of Figure 6.4. However, during matching and covering, the circuit may be broken up in such a way that the internal nodes with hazardous activity can propagate their hazardous behavior to the output. This topic is mentioned in the future work section of Chapter 7 because it may be possible to develop algorithms that identify nonpropagating hazardous activity that has no effect on primary outputs.

6.4.3 Nonoptimum Covered Circuit

It is suspected that requiring hazard-freedom to be the primary cost factor during the matching and covering of circuits often leads to results that are nonoptimum from an area and/or delay perspective. This turns out to occur in some cases, but not as many as expected. Frequently, the examples that are run through the supporting software have identical coverings independent of which cost factor is primary and which is secondary. It is seen that circuits with a large number of hazardous nodes are much more likely to have a hazard-based cover that is area or delay suboptimal. One example, with a small number of nodes and hazards, is shown in Figure 6.5, and illustrates the difference in matching when optimizing for area vs. optimizing for hazard-freedom.

This example is the output *req* from a file named *alloc-outbound*. After verification, the circuit shown in Figure 6.5(a) is found to have two nodes with acknowledgment hazards and one node with a monotonicity hazard. After setting w_1,w_2 to

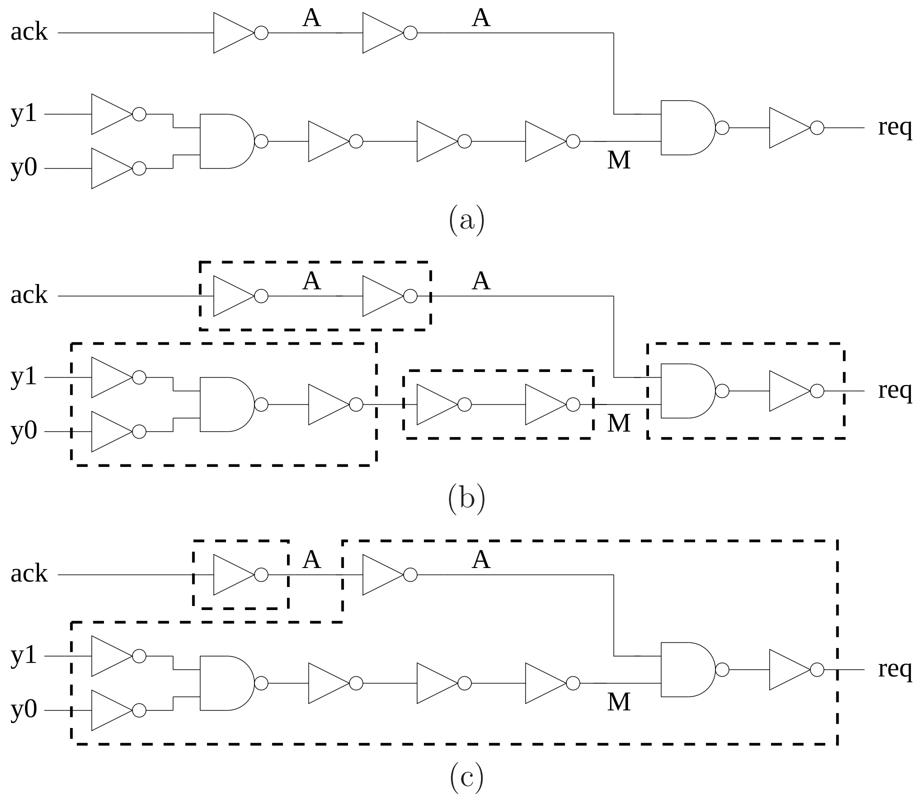


Figure 6.5. Nonoptimum covered circuit example. (a) Example circuit *req*. (b) Cover based on hazard-awareness. (c) Cover based on area minimization.

-1,1 in Equation 5.1, the hazard-aware matching algorithms of Section 5.2 produce the covering shown in Figure 6.5(b). Note that the two acknowledgment hazards are ignored because the buffer is removed from the final netlist. Also note that the monotonicity hazard is left exposed in the covering. Verification on this newly covered netlist is found to be hazard-free.

The covering shown for the circuit in Figure 6.5(c) represents area optimization, and there is no hazard-awareness when choosing the covering elements. This covering is done by setting w_1,w_2 to 0,0 in Equation 5.1. This area covering leaves one acknowledgment hazard exposed and verification performed on the covered circuit indicates this acknowledgment hazard is present in the covered netlist. In addition, a new monotonicity hazard has been created on the output *req*.

CHAPTER 7

CONCLUSIONS

In asynchronous circuits, hazards must be avoided and care must be taken during technology-mapping to not introduce hazards in the design. Therefore, an asynchronous technology-mapper requires a method to rapidly determine when a gate-level transformation of the netlist has introduced a hazard and to provide a method to safely map this netlist to a hazard-free implementation. The technology-mapper described in this dissertation demonstrates that synchronous technology-mapping can be used with small modifications to accommodate the presence of hazards. This approach makes the efficient technology-mapping of gate-level timed asynchronous circuits possible.

7.1 Summary

This work adapts the synchronous design flow for use in the `ATACS` technology-mapping tool. Algorithms are developed for the decomposition of a netlist that uses unbalanced trees and inverter pairs for the netlist representation. Next, algorithmic definitions are created for acknowledgment and monotonicity hazards. Then, the theory and algorithms are developed and implemented for an efficient gate-level verifier that checks each node in a netlist for hazards, and annotates these nodes with this hazard information. This method uses a cube approximation of the internal signal behavior in order to avoid generating an explicit state graph representing the switching behavior of the internal signals. The experimental results for this verifier show that this new method can be substantially faster than previous gate-level timing verification tools. While this new verification method is conservative and thus can report some incorrect hazards, the number of such false negative results is small. This hazard verification method scales well to larger circuits in that it can verify examples with more than 150 gates in less than a second while previous methods fail to complete.

The synchronous technology-mapping algorithms for matching and covering are then developed to provide optimum choices when hazards are present in the netlist. Since asynchronous circuits are sensitive to glitches at all times, the primary objective is to map the netlist decomposition to library cells in a hazard-free manner. There is particular interest in implementing libraries with gC cells because of their compact nature and wide use in asynchronous designs. All circuit implementations using gC's require checking for short-circuit conditions in the mapped circuits. The number of gC library cells rejected as potential matches due to short-circuit issues is significant, 53 percent, on the example circuits that are tested.

The next step is to add gC cells with common-inputs to the library. These types of cells have no short-circuit issues but the common-inputs must be mapped to equivalent subnetworks. It is found that for circuits where hazards are present, increasing the complexity and number of cells in the library, in most cases, reduces the number of hazards present in the final netlist. The use of hazard-freedom as a primary requirement on occasion leads to solutions that are not area or delay optimized. However, the number of such circuits is small and a correct solution is usually found. There is a small number of circuits where a hazard-free mapping could not be found without adding custom cells to the library. These custom cells are typically multiple-stack gC's that cover the entire set or reset portion of the circuit, which is equivalent to providing a library cell that implements the synthesis output. These types of cells place a large burden on the physical design process because of a greater amount of embedded logic and the more liberal use of the atomic gate assumption.

The new algorithms developed during the course of this research include gate-level hazard verification using explicit timing, algorithms to identify acknowledgment and monotonicity hazards, algorithms to perform short-circuit checks for libraries containing gC elements, and algorithms for hazard-aware matching and covering.

There are four significant contributions of this research to the body of knowledge concerning technology-mapping of timed asynchronous circuits. First, it is shown that the synchronous technology-mapping flow can be adapted for use in timed asynchronous circuits. Second, efficient gate-level hazard verification algorithms using timing are developed that allow for alternative implementations to be considered. Third, algorithms and methods are developed to utilize gC's in circuit solutions. Fourth, the investigation of a range of technology-dependent libraries for circuit implementation shows that nearly all circuits can be implemented in a technology-dependent, hazard-free manner.

7.2 Current Relevance

As technology pushes deeper into submicron territory, some of this work ebbs in its usefulness while other parts become more applicable. For instance, current device geometries and the laws of physics make library elements with 4-device stacks impractical. The resulting reduction in library components of this type likely means a more difficult implementation of a hazard-free circuit. On the other hand, smaller and faster circuits means that an atomic gate may be easier to build, allowing for a richer set of library cells with various encapsulated logic.

The use of area and delay as primary or secondary cost factors is also becoming somewhat outdated. Designs are seldom as performance driven today as they are power driven [87]. The design goal is to get the most performance within a power budget. This leads to the possibility of power dissipation, loading, and perhaps logical effort [84], as serious contenders for consideration as primary cost factors.

The synchronous designer can benefit from this work. By following the synchronous technology-mapping flow, it is likely that synchronous designers can utilize some of the algorithms developed to enhance their designs, as long as the circuit can be cut at the primary outputs. For instance, the reduction of hazards is directly related to the reduction of undesirable transitions (read glitches), and can have a significant affect in reducing power dissipation in synchronous circuits. In addition, gC's are a superset of domino and footed domino circuits, a widely used design architecture, which also needs to be short-circuit aware. Short-circuit checks can help determine if the footer gate is necessary and if not, additional stack depth can be used to increase the amount of implemented logic.

Designers adhering to speed-independent styles may also benefit from this work. Speed-independent libraries are often customized and large, with many elements encompassing large amounts of logic. This places a significant design burden on the physical design effort and requires liberal use of the atomic gate assumption. This work makes it possible to explore the implementation of speed-independent designs with more standard libraries.

7.3 Future Work

This research has produced a design flow that can, in nearly all cases, provide hazard-free technology-mapping of timed circuits. There are still issues left that would be interesting to investigate.

7.3.1 False Hazards

There appears to be sufficient information to determine whether a reported hazard is false or not. When an acknowledgment hazard is found on a node n, the state transition, (s,t,s'), where the hazard occurs is reported. For monotonicity hazards, the state s and input v that cause the monotonicity violation on node n are reported. In either case, this information can be used to create an error trace from the initial state. This error trace can then be used to perform a guided simulation of the circuit to detect if the hazard can occur or not. While in theory, this simulation could result in a full state space exploration, it is likely only to require exploration of a small subset of the state space to determine if the hazard is false or not.

7.3.2 Reordering of Inputs

Although it is mentioned briefly in this research, the effects of reordering the inputs on the decomposition have not yet been fully investigated. The examples presented indicate that the reordering step has an effect on the netlist timing that likely leads to a change in the hazard landscape. It may also be possible to examine the nature of input transitions and use the work in [64] to determine a potentially hazard-free decomposition. This future work would also include the identification of context and trigger signals for the netlist under consideration and place these signals wisely in the decomposition [76].

7.3.3 Inertial Delay Models

The use of inertial delay models could possibly lead to circuits with fewer hazards. There have been a number of monotonicity hazards close to where the primary inputs are applied, particularly where asymmetry exists in the number of inverters. The new method of internal node abstraction requires that any timing violation be treated as a hazard. However, under an inertial delay model, timing violations that are short could possibly be filtered out.

7.3.4 Hazard-Dependent Behavior

The algorithms developed in this work determine hazard-freedom for each node individually without regard to the hazard behavior of predecessor nodes in the circuit. It would be interesting to investigate further whether there is a connection between a node that is hazardous and successor nodes. For instance, any acknowledgment hazard at the input to an inverter causes the node at the output of the inverter to be hazardous as well. The verification results in this regard always agree with a full state-space verification so it is assumed that independent treatment of nodes is acceptable. Any stabilization due to timing does not change but untimed stabilization could produce interesting results.

7.3.5 Acknowledgment Hazards

There seems to be some promise in further investigating whether or not acknowledgment hazards that do not manifest on a primary output are problematic. The speed-independent model has been used that reports an acknowledgment hazard if a transition on *any internal node* is not acknowledged by a primary output. However, internal transitions that are blocked from reaching the output by forcing side-inputs may not cause incorrect circuit behavior.

7.3.6 Inverter Pair Insertion

The presence of monotonicity hazards indicate timing relationships between inputs to a gate that behave like race conditions. Lavagno et al. [74] encountered similar conditions in their bounded wire delay model and inserted delays at the appropriate places until the hazards disappeared. It would be interesting to consider the same technique with an eye toward removing the monotonicity hazard by inserting time delay elements in the fanin causing the hazard. This technique is viable because the efficient gate-level hazard verification can quickly determine if, and when, the hazards disappear.

7.3.7 Other Input Forms

As it currently stands, this work is limited to designs specified as time Petri nets. The size and number of examples could be widely expanded by providing support for other input specifications of which several are level-ruled Petri nets, signal transition graphs, and VHDL netlists.

7.3.8 Internal Cycles

The range of examples could also be increased if was possible to provide algorithmic support to handle circuit representation forms that are not tree-based. These examples would not require the cutting of circuits at primary outputs. Rather, circuits could contain internal cycles and reconvergent fanout.

REFERENCES

[1] C. J. Myers, *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Dept. of Elec. Eng., Stanford University, Oct. 1995.

[2] K. S. Stevens, S. Rotem, R. Ginosar, P. Beerel, C. J. Myers, K. Y. Yun, R. Koi, C. Dike, and M. Roncken, "An asynchronous instruction length decoder," *IEEE Journal of Solid-State Circuits*, vol. 36, pp. 217–228, Feb. 2001.

[3] J. Hartmanis and R. Stearns, *Algebraic Structure Theory of Sequential Machines*. Prentice-Hall, 1966.

[4] Z. Kohavi, *Switching and Finite Automata Theory*. McGraw-Hill, 1978.

[5] G. D. Micheli, R. Brayton, and A. Sangiovanni-Vincentelli, "Optimal state assignment for finite state machines," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, pp. 269–295, IEEE Computer Society Press, 1985.

[6] T. Villa, A. Saldanha, R. Brayton, and A. Sangiovanni-Vincentelli, "Symbolic two-level minimization," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, pp. 692–708, IEEE Computer Society Press, 1997.

[7] T. Villa, T. Kam, R. Brayton, and A. Sangiovanni-Vincentelli, *Synthesis of FSMs: Logic Optimization*. New York, New York: Kluwer, 1997.

[8] R. K. Brayton, R. Camposano, G. D. Micheli, R. H. J. M. Otten, and J. van Eijndhoven, "The yorktown silicon compiler system," in *Silicon Compilation* (D. Gajski, ed.), Addison Wesley, 1988.

[9] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Eang, "Mis: A multiple-level interactive logic optimization system," *IEEE Transactions on Computer-Aided Design*, vol. 6, pp. 1062–1081, Nov. 1987.

[10] J. Darringer, D. Brand, W. Joyner, and L. Trevillyan, "LSS: A system for production logic synthesis," *IBM J. Res. Develop.*, Sept. 1984.

[11] D. Gregory, K. Bartlett, A. de Geus, and G. Hachtel, "SOCRATES: A system for automatically synthesizing and optimizing combinational logic," in *23rd Design Automation Conference*, pp. 79–85, IEEE/ACM, 1986.

[12] D. Subcommittee, "Ieee standard vhdl language reference manual," Technical Report IEEE Std 1076-1987, University of California, Davis, Mar. 1988.

[13] D. Thomas and P.R.Moorby, *Verilog Hardware Description Language*. Dordrecht, Netherlands: Kluwer, 1991.

[14] P. Merlin and D. J. Faber, "Recoverability of communication protocols," *IEEE Trans. on Communication*, vol. COM-24, no. 9, pp. 1036–1043, 1976.

[15] D. L. Dill, "Timing assumptions and verification of finite-state concurrent systems," in *Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems*, 1989.

[16] C. J. Myers, T. G. Rokicki, and T. H.-Y. Meng, "POSET timing and its application to the synthesis and verification of gate-level timed circuits," *IEEE Transactions on Computer-Aided Design*, vol. 18, pp. 769–786, June 1999.

[17] T. G. Rokicki, *Representing and Modeling Circuits*. PhD thesis, Stanford University, 1993.

[18] T. G. Rokicki and C. J. Myers, "Automatic verificaton of timed circuits," in *International Conference on Computer-Aided Verification*, pp. 468–480, Springer-Verlag, 1994.

[19] W. Belluomini and C. Myers, "Verification of timed systems using posets," in *International Conference on Computer Aided Verification*, Springer-Verlag, 1998.

[20] W. Belluomini, *Algorithms for Synthesis and Verification of Timed Circuits and Systems*. PhD thesis, Department of Computer Science, University of Utah, Sept. 1999.

[21] W. Belluomini and C. J. Myers, "Timed state space exploration using posets," *IEEE Transactions on Computer-Aided Design*, vol. 19, pp. 501–520, May 2000.

[22] W. Belluomini and C. J. Myers, "Timed circuit verification using tel structures," *IEEE Transactions on Computer-Aided Design*, vol. 20, pp. 129–146, Jan. 2001.

[23] E. Mercer, *Correctness and Reduction in Timed Circuit Analysis*. PhD thesis, Department of Computer Science, University of Utah, Dec. 2002.

[24] T. Yoneda and B. Schlingloff, "Efficient verification of parallel real-time systems," in *Formal Methods in System Design* (C. Courcoubetis, ed.), Kluwer Academic Publishers, 1997.

[25] P. A. Beerel, T. H.-Y. Meng, and J. Burch, "Efficient verification of determinate speed-independent circuits," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, pp. 261–267, IEEE Computer Society Press, Nov. 1993.

[26] P. A. Beerel, J. R. Burch, and T. H.-Y. Meng, "Checking combinational equivalence of speed-independent circuits," *Formal Methods in System Design*, Mar. 1998.

[27] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W.H. Freeman and Company, 1979.

[28] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York, New York: McGraw-Hill, Inc., 1994.

[29] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. Boston: Kluwer Academic Publishers, 1996.

[30] K. Keutzer, "DAGON: Technology binding and local optimization by DAG matching," in *24th Design Automation Conference*, pp. 341–347, IEEE/ACM, 1987.

[31] F. Mailhot, "Algorithms for technology mapping based on binary decision diagrams and on boolean operations," *IEEE Transactions on Computer-Aided Design*, vol. 12, pp. 599–620, May 1993.

[32] R. Rudell, *Logic Synthesis for VLSI Design*. PhD thesis, U. C. Berkeley, Apr. 1989.

[33] F. Mailhot, *Technology Mapping for VLSI Circuits Exploiting Boolean Properties and Operations*. PhD thesis, Stanford University, Dec. 1991.

[34] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," Tech. Rep. UCB/ERL M92/41, University of California, Berkeley, May 1992.

[35] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Technology mapping in mis," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, pp. 116–119, IEEE Computer Society Press, 1987.

[36] C. R. Morrison, R. M. Jacoby, and G. D. Hachtel, "TECHMAP: Technology mapping with delay and area optimization," in *Logic and Architecture Synthesis for Silicon Compilers* (G. Saucier and P. M. McLellan, eds.), pp. 53–64, North-Holland, 1989.

[37] M. Zhao and S. S. Sapatnekar, "A new structural pattern matching algorithm for technology mapping," in *Proc. ACM/IEEE Design Automation Conference*, pp. 371–376, IEEE Computer Society Press, 2001.

[38] H. Sato, N. Takahashi, Y. Matsunaga, and M. Fujita, "Boolean technology mapping for both ecl and cmos circuits based on permissable functions and binary decision diagrams," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, pp. 286–289, IEEE Computer Society Press, 1990.

[39] J. R. Burch and D. F. Long, "Efficient boolean function matching," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, pp. 408–411, IEEE, Nov. 1992.

[40] F. U. Rosenberger, C. E. Molnar, T. J. Chaney, and T.-P. Fang, "Q-modules: Internally clocked delay-insensitive modules," *IEEE Transactions on Computers*, vol. C-37, pp. 1005–1018, Sept. 1988.

[41] J. C. Ebergen, *Translating Programs into Delay-Insensitive Circuits*, vol. 56 of *CWI Tract*. Centre for Mathematics and Computer Science, 1989.

[42] D. E. Muller and W. S. Bartky, "A theory of asynchronous circuits," in *Proceedings of an International Symposium on the Theory of Switching*, pp. 204–243, Harvard University Press, Apr. 1959.

[43] D. E. Muller, "Asynchronous logics and application to information processing," in *Proceedings of a Symposium on the Application of Switching Theory to Space Technology*, pp. 289–297, Stanford University Press, 1962.

[44] I. Kimura, "Extensions of asynchronous circuits and the delay problem I: Good extensions and the delay problem of the first kind," *Journal of Computer and System Sciences*, vol. 2, pp. 251–287, Oct. 1968.

[45] I. Kimura, "Extensions of asynchronous circuits and the delay problem II: Spike-free extensions and the delay problem of the second kind," *Journal of Computer and System Sciences*, vol. 5, pp. 129–162, Apr. 1971.

[46] P. Siegel and G. D. Micheli, "Decomposition methods for library binding of speed-independent asynchronous designs," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, pp. 558–565, Nov. 1994.

[47] P. Beerel and T.-Y. Meng, "Automatic gate-level synthesis of speed-independent circuits," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, pp. 581–587, IEEE Computer Society Press, Nov. 1992.

[48] P. A. Beerel and T. H.-Y. Meng, "Logic transformations and observability don't cares in speed-independent circuits," in *Proceedings of TAU 1993*, Sept. 1993. Participant's proceedings.

[49] S. M. Burns, "General condition for the decomposition of state holding elements," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, IEEE Computer Society Press, Mar. 1996.

[50] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, E. Pastor, and A. Yakovlev, "Decomposition and technology mapping of speed-independent circuits using Boolean relations," *IEEE Transactions on Computer-Aided Design*, vol. 18, Sept. 1999.

[51] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev, "Logic decomposition of speed-independent circuits," *Proceedings of the IEEE*, vol. 87, pp. 347–362, Feb. 1999.

[52] P. Vanbekbergen, B. Lin, G. Goossens, and H. de Man, "A generalized state assignment theory for transformations on signal transition graphs," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, pp. 112–117, IEEE Computer Society Press, Nov. 1992.

[53] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "A region-based theory for state assignment in speed-independent circuits," *IEEE Transactions on Computer-Aided Design*, vol. 16, pp. 793–812, Aug. 1997.

[54] P. Siegel, G. D. Micheli, and D. Dill, "Automatic technology mapping for generalized fundamental-mode asynchronous designs," in *Proc. ACM/IEEE Design Automation Conference*, pp. 61–67, June 1993.

[55] C. J. Myers, *Asynchronous Circuit Design*. John Wiley & Sons, July 2001.

[56] D. A. Huffman, "The synthesis of sequential switching circuits," *J. Franklin Institute*, March, April 1954.

[57] S. H. Unger, *Asynchronous Sequential Switching Circuits*. New York: Wiley-Interscience, John Wiley & Sons, Inc., 1969.

[58] K. Stevens, "Private communication," Sept. 2000. Ken Stevens is with Intel Corporation.

[59] B. Coates, A. Davis, and K. Stevens, "The Post Office experience: Designing a large asynchronous chip," *Integration, the VLSI journal*, vol. 15, pp. 341–366, Oct. 1993.

[60] A. Davis, B. Coates, and K. Stevens, "The Post Office experience: Designing a large asynchronous chip," in *Proc. Hawaii International Conf. System Sciences*, vol. I, pp. 409–418, IEEE Computer Society Press, Jan. 1993.

[61] A. Davis, B. Coates, and K. Stevens, "Automatic synthesis of fast compact asynchronous control circuits," in *Asynchronous Design Methodologies* (S. Furber and M. Edwards, eds.), vol. A-28 of *IFIP Transactions*, pp. 193–207, Elsevier Science Publishers, 1993.

[62] A. Davis, "Synthesizing asynchronous circuits: Practice and experience," in *Asynchronous Digital Circuit Design* (G. Birtwistle and A. Davis, eds.), Workshops in Computing, pp. 104–150, Springer-Verlag, 1995.

[63] K. Y. Yun, D. L. Dill, and S. M. Nowick, "Practical generalizations of asynchronous state machines," in *Proc. European Conference on Design Automation (EDAC)*, pp. 525–530, IEEE Computer Society Press, Feb. 1993.

[64] P. S. K. Siegel, *Automatic Technology Mapping for Asynchronous Designs*. PhD thesis, Stanford University, Feb. 1995.

[65] W.-C. Chou, P. A. Beerel, and K. Y. Yun, "Average-case technology mapping of asynchronous burst-mode circuits," *IEEE Transactions on Computer-Aided Design*, vol. 18, pp. 1418–1434, Oct. 1999.

[66] K. Y. Yun and D. L. Dill, "Automatic synthesis of extended burst-mode circuits: Part i (specification and hazard-free implementation)," *IEEE Transactions on Computer-Aided Design*, vol. 18, pp. 101–117, Feb. 1999.

[67] K. Y. Yun and D. L. Dill, "Automatic synthesis of extended burst-mode circuits: Part ii (automatic synthesis)," *IEEE Transactions on Computer-Aided Design*, vol. 18, pp. 118–132, Feb. 1999.

[68] K. W. James and K. Y. Yun, "Average-case optimized transistor-level technology mapping of extended burst-mode circuits," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 70–79, 1998.

[69] J.-L. Yang, *Transistor-Level Technology Mapping for Extended Burst-Mode Asynchronous Designs*. PhD thesis, University of Utah, Dec. 2003.

[70] C. J. Myers and T. H.-Y. Meng, "Synthesis of timed asynchronous circuits," *IEEE Transactions on VLSI Systems*, vol. 1, pp. 106–119, June 1993.

[71] C. J. Myers, T. G. Rokicki, and T. H.-Y. Meng, "Automatic synthesis of gate-level timed circuits with choice," in *Advanced Research in VLSI*, pp. 42–58, IEEE Computer Society Press, 1995.

[72] L. Lavagno, *Synthesis and Testing of Bounded Wire Delay Asynchronous Circuits from Signal Transition Graphs*. PhD thesis, University of California, Berkeley, Nov. 1992.

[73] L. Lavagno, N. Shenoy, and A. Sangiovanni-Vincentelli, "Linear programming for hazard elimination in asynchronous circuits," *Journal of VLSI Signal Processing*, vol. 7, pp. 137–160, Feb. 1994.

[74] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli, "Synthesis of hazard-free asynchronous circuits with bounded wire delays," *IEEE Transactions on Computer-Aided Design*, vol. 14, pp. 61–86, Jan. 1995.

[75] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli, "Algorithms for synthesis of hazard-free asynchronous circuits," in *Proc. ACM/IEEE Design Automation Conference*, pp. 302–308, IEEE Computer Society Press, 1991.

[76] C. J. Myers, P. A. Beerel, and T. H.-Y. Meng, "Technology mapping of timed circuits," in *Asynchronous Design Methodologies*, IFIP Transactions, pp. 138–147, Elsevier Science Publishers, May 1995.

[77] E. Mercer, C. Myers, and T. Yoneda, "Improved poset timing analysis in timed petri nets," in *The Tenth Workshop on Synthesis and System Integration of MIxed Technologies (SASIMI 2001)*, October 2001.

[78] J. Ebergen and S. Gingras, "A verifier for network decompositions of command-based specifications," in *Proc. Hawaii International Conf. System Sciences*, vol. I, IEEE Computer Society Press, Jan. 1993.

[79] G. Gopalakrishnan, E. Brunvand, N. Michell, and S. Nowick, "A correctness criterion for asynchronous circuit validation and optimization," *IEEE Transactions on Computer-Aided Design*, vol. 13, pp. 1309–1318, Nov. 1994.

[80] M. Bozga, H. Jianmin, O. Maler, and S. Yovine, "Verification of asynchronous circuits using timed automata," in *Electronic Notes in Theoretical Computer Science* (O. M. Eugene Asarin and S. Yovine, eds.), vol. 65, Elsevier Science Publishers, 2002.

[81] S. Yovine, "Private communication," July 2002. Sergio Yovine is with VERIMAG.

[82] M. Shams, J. Ebergen, and M. Elmasry, "A comparison of CMOS implementations of an asynchronous circuits primitive: the C-element," in *International Symposium on Low Power Electronics and Design*, pp. 93–96, Aug. 1996.

[83] A. Aho and S. Johnson, "Optimal code generation for expression trees," *Journal of the ACM*, vol. 23, pp. 499–501, June 1976.

[84] I. Sutherland, B. Sproull, and D. Harris, *Logical Effort: Designing Fast CMOS Circuits*. Morgan Kaufmann Publishers, Inc., 1999.

[85] M. A. Peña, J. Cortadella, A. Kondratyev, and E. Pastor, "Formal verification of safety properties in timed circuits," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 2–11, IEEE Computer Society Press, Apr. 2000.

[86] C. J. Myers, W. Belluomini, K. Killpack, E. Mercer, E. Peskin, and H. Zheng, "Timed circuits: A new paradigm for high-speed design," in *Proc. of Asia and South Pacific Design Automation Conference*, pp. 335–340, Feb. 2001.

[87] K. Stevens, "Private communication," Aug. 2004. Ken Stevens is with Intel Corporation.