AUTOMATIC TECHNOLOGY MAPPING FOR ASYNCHRONOUS DESIGNS

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

Polly Sara Kay Siegel

February, 1995

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

Giovanni De Micheli (Principal Advisor)

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

David Dill (Associate Advisor)

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

Bruce Wooley

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies

Abstract

Asynchronous design styles have been increasing in popularity as device sizes shrink and concurrency is exploited to increase system performance. However, asynchronous designs are difficult to implement correctly because the presence of hazards, which are of little consequence to most parts of synchronous systems, can cause improper circuit operation. Many asynchronous design styles, together with accompanying automated synthesis algorithms, address the issues of design complexity and correctness. Typically, these synthesis systems take a high-level description of an asynchronous system and produce a logic-level description of the resultant design that is hazard-free for transitions of interest. The designer then must manually translate this logic-level description into a technology-specific implementation composed of an interconnection of elements from a semi-custom cell library. At this stage, the designer must be careful not to introduce new hazards into the design. The size of designs is limited in part by the inability to safely (and reliably) map the technology-independent description into an implementation.

In this thesis, we address the problem of technology mapping for two different asynchronous design styles. We first addressed the problem for burst-mode designs. We developed theorems and algorithms for hazard-free mapping of burst-mode designs, and implemented these algorithms on top of an existing synchronous technology mapper. We incorporated this mapper into a toolkit for asynchronous design, and used the toolkit to implement a low-power infrared communications chip. We then extended this work to apply to the problem of hazard-free technology mapping of speed-independent designs. The difficulty in this design style is in the decomposition phase of the mapping algorithm, and we developed theory and algorithms for correct hazard-free decomposition of this design style. We also developed an exact covering algorithm which takes advantage of logic sharing within the design. These algorithms were implemented within the burst-mode technology mapper to produce a technology mapper suitable for speed-independent designs.

Dedication

To my loving husband Kurt.

Acknowledgments

My most heartfelt thanks goes to my advisor, Giovanni De Micheli, for his support and guidance over the years. My associate advisor, Professor David Dill, in addition to being the source of many interesting discussions, has also been a wonderful advisor. I have really appreciated the candor and frankness of his comments, and I have learned tremendously from him. I would also like to thank Professor Bruce Wooley, for serving as my chairman, reading my thesis, and for being an additional source of advice and support during my tenure here. Professor Teresa Meng served as my associate advisor for the early part of my thesis, and I thank her for her help and direction in understanding asynchronous circuits, and for serving on my Orals committee.

I would like to thank Steve Nowick, Peter Beerel, Chris Myers, Jerry Burch, and Ken Yun for many useful discussions related to asynchronous system design and hazards. I would also like to thank the members of the synchronous synthesis group—Jerry Yang, Luca Benini, Claudionor Coelho, Maurizio Damiani, David Filo, Rajesh Gupta, David Ku, Frédéric Mailhot, and Tom Truong—for discussions on various synthesis topics, and for their support and friendship over the years. I would like to express special appreciation to Frédéric Mailhot for helping me understand the CERES technology mapper while I was adapting it for my needs and to Jerry Yang for being a supportive officemate and putting up with my bicycles during most of my years at Stanford.

I would like to thank Lilian Betters for her tremendous support in helping me get through the Stanford bureaucracy and for her friendship.

Alan Marshall and Bill Coates, then of HP Labs, did portions of the design and toolkit described in Chapter 5, and also made my stay at Stanford a lot more enjoyable. Professor Al Davis of University of Utah (formerly of Hewlett-Packard), served as a supportive mentor and friend as he over-saw the asynchronous work being done at HP Labs and Stanford. Phil Johnson, of the Hewlett-Packard Design Technology Center, deserves special mention for the numerous times he helped me on various HP hardware and software questions and problems, but mainly for being such a supportive friend over the years. My former boss, Beatriz Infante (then of Hewlett-Packard), was instrumental in my decision to return to school by awarding my the HP Resident Fellowship, in addition to providing sage advice and true friendship long after our business relationship ended. I would also like to thank my boss, Steven Rosenberg of Hewlett-Packard Laboratories, for the support he has given me, both in terms of equipment and employment during part of my tenure at Stanford. And of course, I must thank Bill and Dave for making it all possible.

In addition to my reading committee, several people gave me valuable feedback on my thesis. Bill Coates was brave enough to read through my thesis from cover to cover, and gave me many valuable comments on its contents, most of which I incorporated into the thesis. Phil Johnson was kind enough to proof one of the last drafts of the thesis, and caught many (potentially) embarassing errors, which I have hopefully corrected. Chris Myers and Peter Beerel gave me comments on the introductory parts of early drafts of the thesis which helped both the organization and the content tremendously.

My many friends have helped me stay sane and enhanced my stay here at Stanford, and I offer special thanks to them. I also would like to thank the Stanford Cycling Team for giving me an outlet for stress reduction during my years at Stanford.

I would like to thank both my parents, Paul and Sylvia Siegel, and my in-laws, Charles and Claire Shoens, for believing in me, and for encouraging me to complete the degree. The most special thanks goes to my husband, Kurt Shoens, to whom this thesis is dedicated. I could not have done this without his support, encouragement and love.

This work was supported by the Semiconductor Research Corporation, Contract nos. 91-DJ-205, 92-DJ-205, and 93-DJ-205, by a trust grant from the Center for Integrated Systems, Stanford University, and by Hewlett-Packard Corporation.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Asynchronous design styles have been increasing in popularity as device sizes shrink and concurrency is exploited to increase system performance. However, asynchronous designs are difficult to implement correctly because the presence of *hazards*, which are of little consequence to most parts of synchronous systems, can cause improper circuit operation. Many asynchronous design styles, together with accompanying automated synthesis algorithms, address the issues of design complexity and correctness. Typically, these synthesis systems [67, 89, 20, 19, 57, 4, 50, 15, 62] take a high-level description of an asynchronous system and produce a logic-level description of the resultant design that is hazard-free for transitions of interest. The designer then must manually translate this logic-level description into a technology-specific implementation composed of an interconnection of elements from a semi-custom cell library. At this stage, the designer must be careful not to introduce new hazards into the design. The size of designs is limited in part by the inability to safely and reliably map the technology-independent description into an implementation.

Automatic technology mapping techniques have been employed over the past decade for synchronous design styles [41, 73, 48]. These algorithms allow translation of a technology-independent logic description into a library-specific (technology-dependent) implementation. However, these techniques by themselves are not suitable for asynchronous design styles because they do not take hazards into account.

This thesis attempts to bridge the gap between synchronous mapping techniques and asynchronous design styles by developing theory and algorithms for automatic technology mapping for some popular asynchronous design styles.

1.1 Asynchronous Design

Asynchronous systems are digital designs that do not use a global clock to synchronize circuit activity. Whereas synchronous systems employ a global clock to synchronize activity within the circuit, asynchronous systems use local handshakes to control the flow of activity. As a by-product of using a global synchronization scheme, logic within synchronous systems can ignore the presence of *hazards,* or glitches, during operation. (The clock logic in synchronous systems is, of course, sensitive to hazards.) Because all parts of asynchronous systems are sensitive to signal transitions at all times, special attention must be paid to hazards during their design, introducing additional complexity to their analysis and synthesis.

Asynchronous designs offer the following advantages over synchronous designs:

1. *Elimination of problems due to clock skew.* Synchronous systems, as they get larger, suffer from timing problems due to clock skew, requiring that delays on the clock lines be carefully balanced to ensure proper synchronization. Because asynchronous systems keep synchronization local, they do not have to worry about such issues.
2. *Elimination of global clock routing difficulties.* In conjunction with clock skew problems, it is difficult to route the clock in large synchronous chips. In addition to the routing difficulties, the clock routing consumes precious space on the chip. Asynchronous systems, lacking a global clock, do not have these problems.
3. *Low power.* Synchronous systems consume power with every clock tick as all registers are operated in lock-step. Special power-down circuitry can be designed to halt the clock during periods of inactivity, but this circuitry consumes additional area and adds complexity to an

already complex design. Because asynchronous systems operate only when there is activity, there is no power consumed (other than because of leakage) when the circuit is inactive, resulting in potentially lower overall power consumption.

4. *Performance.* Synchronous systems are limited by the worst-case performance of the longest path between registers along with the worst-case parameters of the technology in which they are implemented. Asynchronous systems, on the other hand, can operate at the average-case performance of each element in the chip, at a rate which tracks the environmental conditions that affect the performance of the given technology.

5. *Modularity.* The clock speed constrains how portions of a synchronous design can be upgraded. With asynchronous systems, a new faster component can be inserted in place of a slower one and the circuit will continue to work correctly, with an accompanying improvement in performance.

However, despite these many potential advantages, most digital designs are still done synchronously because of the difficulties in designing asynchronous systems:

1. *Sensitivity to hazards*. Because asynchronous systems do not have a global clock, the circuits are sensitive to hazards or glitches at all times. Each step of the design from high-level down to layout must be done in a hazard-free way for the circuit to operate correctly.

2. *No standard design methodology.* There are many different asynchronous design styles and methodologies in use today. However, they suffer from a lack of consistency among the specification styles and differing assumptions among the implementation styles.

3. *Lack of mature automatic synthesis methods.* Some of the proposed asynchronous design methodologies have supporting automatic synthesis tools, some which guarantee correct hazard-free design and others which do not. However, most of these synthesis methods are still experimental and have not yet been tried on large-scale designs. The tools that do exist are more primitive than those that are available for synchronous design, and are not available commercially. Furthermore, some elements of the synthesis path, such as those that this thesis addresses, are just coming into place.

4. *Necessity for custom design.* Most of the asynchronous design styles and methodologies require the use of specialized transistor-level components, requiring the designers to do custom design, where cells must be designed either manually or with automatic module generators. This requires significant expertise and expense. In general, these design styles do not support the use of existing libraries.

The difficulty of doing asynchronous design, coupled with the lack of mature, proven synthesis tools for asynchronous synthesis have held back the growth of asynchronous design activity. This thesis addresses some of these issues by completing the synthesis path to allow truly automatic design using libraries of standard components.

1.2 Automatic Synthesis

Automatic synthesis methods for synchronous systems have been the subject of active research for the past 15 years [25, 35, 11, 9, 27, 1]. Synthesis of designs from a high-level specification down to a transistor-level implementation is fairly mature, and has migrated from being studied by researchers at universities and research labs to being commonly used by ASIC designers in industry.

For synchronous systems, the synthesis tool flow starts with a high-level description of the circuit written in a hardware description language (HDL) such as VHDL [78] or Verilog [80], and produces a netlist suitable for layout, as is depicted in Figure 1.1. A cost metric, such as area or delay, is also supplied to guide the synthesis process at each step, although the expression of the metric changes with the level of design abstraction.

First, the HDL description is refined by an *architectural synthesis* program into a set of unoptimized logic equations expressing the desired behavior [26]. Typically, the HDL description is first compiled into an internal graph-based representation. At this point, compiler optimizations can also be done on the basic code, resulting in a more efficient design. Next, the architectural synthesis performs *scheduling* and *binding* operations, guided by the cost metric. Here the program parti-

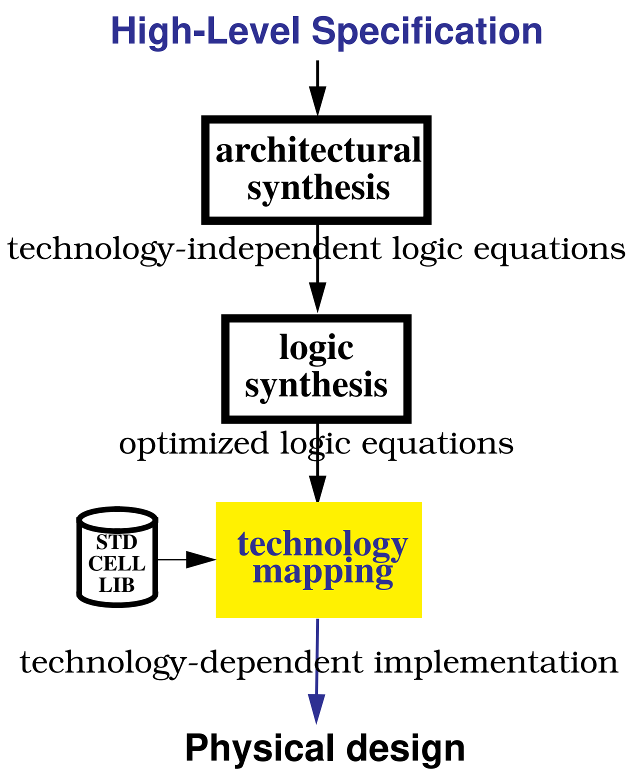


Figure 1.1: Synthesis tool flow

tions the design into *data path* and *control,* and generates a structural model of the data path and a state transition diagram describing the control.

The finite-state machine description is then passed to the *logic synthesis* step. This step determines the number of state variables in the state machine, the state encoding, and the implementation structure based on the cost metric. The cost metric has a strong influence on the results. For example, a cost metric which favors reducing delay may result in the synthesis method producing an encoding that uses a large number of state variables and later translates into a fast but area-inefficient implementation. From here, the unoptimized equations can be passed to a logic optimization program, such as SIS [77], where a more optimal set of logic equations is derived from the initial set of equations. Again, the cost metric is used to drive the optimization in terms of a variable type that makes sense for this level of abstraction. For example, for area cost metrics, minimizing the number of literals typically results in a minimal area circuit.

The final step in the process is the library-binding step, also called *technology mapping*, where the logic equations are mapped into a netlist of library elements from an existing library of standard-cell or gate-array components. Again, the algorithms use the cost metric to guide their decisions; however, at this step the metrics are tied directly to the real delay and area of each component. This netlist is then passed to a place-and-route tool to finish the implementation of the design.

Asynchronous synthesis has advanced to the point where many of the pieces in the tool flow described above are available, depending upon the design style. However, in most of the asynchronous synthesis systems in the research community today, the technology mapping step is still one that is done manually. Some design styles, such as delay-insensitive design styles, require special custom asynchronous components, forcing the design to be done at the transistor-level. Other design styles, such as some speed-independent design styles, allow the use of complex atomic hazard-free gates, but these gates may or may not exist in the given standard-cell library. Still other design styles, such as burst-mode design styles, allow implementation using commonly available gates, but the translation from logic to implementation still requires time consuming manual labor in addition to verification (through simulation) that hazards were not introduced into the implementation during this step. In addition, the manual translation may be inefficient in terms of area or some other cost metric.

Thus, there is strong motivation to investigate the use of automatic technology mapping to complete the synthesis process. Both theoretical and algorithmic contributions will be necessary to ensure that the technology mapping step produces hazard-free implementations that realize the intended logic. These issues are the subject of this thesis.

1.3 Asynchronous Design Styles

Many different asynchronous design styles, along with accompanying manual or automatic synthesis methods, have cropped up over the years. Most of the automatic synthesis work has focused on the design of *control* circuitry rather than *data paths.* This thesis will also take a control orientation, because technology-mapping techniques are generally more suitable for random control

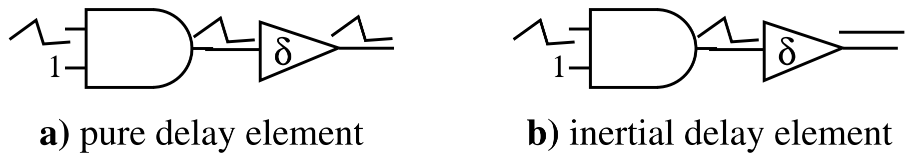


Figure 1.2: Pure versus inertial delays

logic. Because datapath circuitry is typically very regular with many repeated instances of identical blocks, it is implemented in the smallest area with the best performance by using a module generator to automatically tile basic cells together to realize the intended function. This thesis addresses a subset of these design styles, and it is therefore important to explore the breadth of styles to understand where our work fits in. Our exposition will be brief; a more complete discussion of these asynchronous design methodologies can be found in [45], [36], and [83]. The reader interested in work on asynchronous data paths is directed to [79], [52], [28], and [86].

We can broadly lump asynchronous design styles into five categories in decreasing order of robustness: *delay-insensitive, quasi-delay-insensitive, speed-independent, fundamental-mode,* and *timed.* The last is general enough to encompass both synchronous and asynchronous designs. Although this thesis focuses primarily on fundamental-mode and speed-independent designs, it is instructive to understand why we made that choice. Before we describe the design styles, however, we must briefly discuss the underlying delay assumptions upon which these styles are based.

1.3.1 Delay Models

Each design style makes a different set of assumptions about the delay model of the underlying circuitry. Depending upon the delay assumptions used, the synthesized circuits may be more or less tolerant to perturbations in the actual delays present in the implementation. Robust circuits come at the expense of added circuitry and difficulty in design.

Delays can be classified as either *pure* or *inertial*. Figure 1.2 illustrates the two cases with a basic AND gate followed by a delay element. A *pure* delay element will pass along each signal appear-

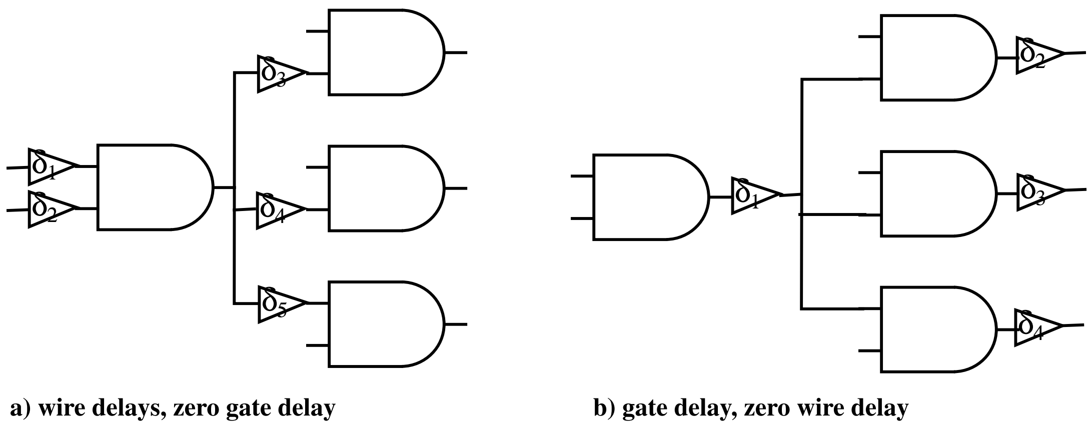


Figure 1.3: Delay models: wire delays versus gate delays

ing at its input directly to its output after some delay δ. As a result, glitches will also be passed along, regardless of their duration. An *inertial* delay element will only pass along pulses that are of a duration greater than the inertial delay. As a result, glitches of short duration will be filtered out by the element. Using a pure delay model as the basis of analysis will result in a more conservative, but more robust, design, and this is the model we assume throughout this thesis. Delays can be associated either with the *wires* or the *gates* in the circuit, or both. *Wire* delays are assumed to be associated with the wires at the inputs of each gate. *Gate* delays are generally attached to the outputs of the gates, with the gate itself acting as a zero-delay function evaluator, as shown in Figure 1.2.

Figure 1.3 illustrates an important but subtle difference between the two delay models for single-output components. If it were not for *wire forks*, or points of multi-fanout, the two models would be equivalent, since delays on the inputs can easily be transformed into delays on the outputs. However, at points of multi-fanout, the two models differ in their treatment of the wire-fork. When wire delays are assumed, a transition at the output of a gate may appear at the inputs of its fanout gates at different times. On the other hand, when gate delays are assumed, points of multi-fanout are assumed to be *isochronic,* i.e., there is a single lumped delay on the output of the gate, as seen in Figure 1.3b, and the output changes will propagate to the inputs of each of the fanout gates at the same time. Some design styles use a combination of wire and gate delays but assume that forks are isochronic, since assuming different delays on each element of a wire fork can complicate the design process considerably.

Delays can further be classified as either *bounded* or *unbounded.* A *bounded* delay element has upper and lower bounds on the time in which the element will respond. An *unbounded* delay element has an unknown positive, but finite, delay. Design styles which use the unbounded delay assumption produce more conservative (less optimized) designs that are more robust in the presence of arbitrary delays.

Design Style	Delay Type
Delay-insensitive	arbitrary wire and gate delays
Quasi-delay-insensitive	arbitrary wire and gate delay w/isochronic forks
Speed-Independent	arbitrary gate delay, zero wire delay w/isochronic forks
Generalized fundamental-mode	bounded wire delays w/assumptions on feedback path
Timed	specific wire/gate/environmental delays

Table 1.1 Taxonomy of design styles by delay assumptions

With this understanding of the various delay models in mind, we can now associate design styles with their delay assumptions, as can be seen in Table 1.1.

1.3.2 Previous Work

Having briefly laid out the basic assumptions surrounding the various asynchronous design styles, we will examine some of the previous work that is relevant to this thesis. We will focus more on research that addressed the implementation level, since that forms the basis for our work.

1.3.2.1 Delay-Insensitive Designs

Delay-insensitive designs are asynchronous designs that are insensitive to both wire and gate delays, where delays are assumed to be arbitrary but finite. To guarantee that a change at the output of a gate is properly received by gates connected to its fanout, a *handshake protocol* must be used, in conjunction with *completion detection* circuitry, by both the sender and receiver to acknowledge receipt of the signal. Failure to provide proper acknowledgment can result in hazards, and thus, faulty circuit operation.

Various researchers have used specialized modules to aid in the creation of delay-insensitive (DI) implementations ([58], [61]), because of the difficulty of implementing DI designs with single-output gates. These modules are designed assuming bounded delays, but once they have been created, the modules can be used within delay-insensitive circuits as basic elements. Martin [51] later proved that the class of purely delay-insensitive designs with single-output gates is limited to those containing only C-elements[1] as multiple-input operators. This motivates the use of *isochronic forks* as a way to create a broader class of circuits, which is the subject of the next subsection.

Brunvand [15] uses `occam`, a language based on Hoare's CSP [39], to describe computation as a set of concurrent processes communicating over fixed channels. A description is then compiled into an implementation composed of delay-insensitive control units utilizing a two-phase handshake protocol, and *bundled data paths*. A bundled data path consists of a group of data wires accompanied by a single control wire that indicates the validity of the data. The data wires are physically laid out such that the data on all wires are guaranteed to be valid before the control signal is asserted, simplifying communication and hazard considerations. As a result of the delay-insensitive protocol, individual components can be enhanced without affecting the correctness of the circuit's behavior.

Ebergen proposed a *trace-theory* based method for delay-insensitive circuit design ([30], [31]). A regular-expression-like syntax is used to describe circuit behavior, where each symbol in the alphabet corresponds to a signal in the circuit. A delay-insensitive circuit described in this language is then automatically translated into an implementation composed of basic atomic elements such as toggles, sequencers and C-elements.

---

1. A two-input Muller C-element is a single-output sequential element with next-state equation C = AB + (A+B)C, where A and B are its inputs, and C is its output. It is used in certain asynchronous design styles as a preferable alternative to an SR-latch because it does not exhibit metastability in any of its states.

1.3.2.2 Quasi-Delay-Insensitive Designs

The main difficulty in delay-insensitive designs, which precludes implementation of many circuits, is that of having to *acknowledge* transitions on each output path of a multi-fanout gate. With purely delay-insensitive designs, each transition on each fork of a multi-fanout gate must be separately acknowledged.

Martin showed that by allowing isochronic forks, a larger class of *quasi-delay-insensitive* circuits can be designed with properties similar to pure delay-insensitive designs [51]. By using the isochronic fork assumption, if one of the forks is acknowledged then all are considered to be acknowledged. This assumption requires both that the differences in the delays in the wire forks are negligible compared to the gate delays and that the switching thresholds in the gates being driven are close to each other. As a result, most researchers make the assumption that certain forks are isochronic, as necessary to ensure that signals are acknowledged.

Martin and Burns developed an automated synthesis procedure for quasi-delay-insensitive designs that takes a design described in CSP and automatically synthesizes equations implementing the circuit. The low-level design is implemented using specialized custom atomic elements along with simple atomic combinational gates. A quasi-delay-insensitive microprocessor was designed in this design style using this synthesis method [50, 53, 18].

Brunvand extended his delay-insensitive work described in [15] to include implementations created out of elements from the Actel family [23] of field-programmable gate-arrays (FPGAs) [14]. He creates macros for some standard self-timed and delay-insensitive components out of the basic Actel multiplexor (MUX) elements, and uses these macros in place of custom CMOS to implement circuits in Actel FPGAs. Although this eases the burden of having to do custom design, he was not able to create some of the more complex delay-insensitive elements, such as specialized arbiters, using the FPGA cells. It is also not clear how he handled enforcement of isochronic fork restrictions without having control over the routing.

Hauck developed a special SRAM-based FPGA specifically for asynchronous designs, called Montage [37]. There are two basic types of blocks: a routing and logic block (RLB) and an arbiter block. Each RLB can implement any 3-input combinational function or any 2-input state-holding element such as a C-element. The arbiter block can implement an arbiter, enabled arbiter or a synchronizer. The FPGA architecture is organized to help meet isochronic fork constraints. An automatic place-and-route program was designed to map asynchronous designs to this architecture.

1.3.2.3 Speed-Independent Designs

Speed-independent designs were originally pioneered by Muller [60]. A speed-independent design is one where gate-delays are assumed to be unbounded but finite, and zero wire delays are assumed. Points of multi-fanout in the design are considered to have zero-delay (i.e., all forks are considered to be isochronic). Designs which adhere to these delay assumptions are quite robust to a wide range of delays. However, the range of designs that adhere to these assumptions, although larger than for delay-insensitive circuits, is still relatively small.

Chu presented an approach for synthesizing speed-independent control circuits from high-level specifications called *signal transition graphs (STGs)* [19]. An STG is a form of an interpreted Petri net, where transitions on a net correspond to transitions of a signal in a control circuit. The STG models both the *environment* and the action of the circuit itself, forming a closed system. Chu proved a hazard-free implementation is derivable from an STG where certain syntactic properties hold, such as *liveness* and *persistency.* Meng devised an automatic synthesis algorithm based on this work, which transforms an STG into a series of Boolean equations implementing each signal [57]. Each equation in the circuit is manually realized by replacing it by an atomic custom CMOS gate which has been manually designed.

Beerel [3] proposed a different synthesis method to generate a speed-independent gate-level implementation from a state graph, where a state graph is a data structure easily derived from an STG. Implementations consist of combinational logic blocks and C-elements, where each combinational logic block is implemented out of simple unlimited-fanin AND and OR gates. However, the synthesis method assumes that such unlimited-fanin gates exist, which is not, in general, a realistic assumption. If the gates are not in the library, then the design cannot be realized. Beerel has considered some heuristics for addressing this issue, but has not formally treated the problem [5].

He also assumes that atomic gates with arbitrary input inversion exist, another unrealistic assumption when using CMOS technology.

Lin and Lin [46] assume the same implementation model as Beerel, but they start from an STG rather than a state graph, reducing the complexity of the synthesis process and allowing larger circuits to be synthesized as a result. Their low-level implementations suffer from the same limitations as those produced using Beerel's method.

Kondratyev et al [43] also give a method for synthesizing gate-level implementations of speed-independent circuits. Because their initial specification is a *change diagram (CD)*, they cannot synthesize circuits with choice, a significant limitation. Additionally, because their method requires the addition of a number of state-holding elements (either C-elements or SR latches), their designs are not area efficient. Recent work has addressed some of those limitations, and their current method takes a state-graph based implementation with choice and translates it into a gate-level circuit, giving performance and area results similar to [3].

1.3.2.4 Generalized Fundamental-Mode Designs

The *fundamental-mode* design style, pioneered by Huffman [40, 83], imposes restrictions on when input changes can occur. In a Huffman machine, the circuit is arranged as a combinational logic block with some subset of the outputs fed back to hold the state, as shown in Figure 1.4. In its most basic form, only a single-input change is allowed on the inputs before the combinational logic must settle. Timing constraints are imposed locally on the feedback path to insure proper hazard-free operation. Additionally, the combinational logic in the feed-forward path must be hazard-free. This mode of operation is quite restrictive, and severely limits the concurrency allowed, thus limiting its usefulness for real designs. Unger [83] suggested a mechanism to extend fundamental-mode to allow multi-input changes. However, this mechanism involved the use of inertial delay elements, which, in addition to having questionable reliability, also slow down circuit operation. Thus, this method of operation never saw wide use.

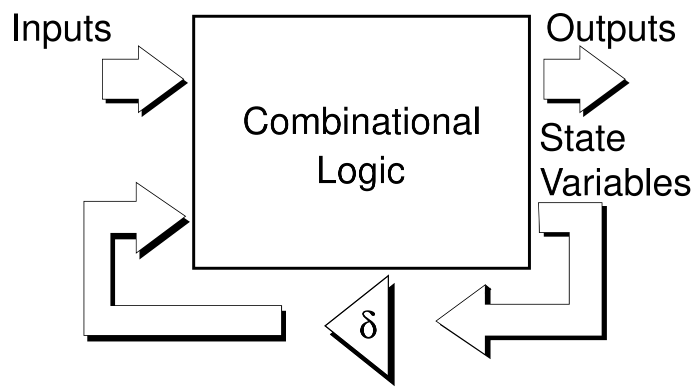


Figure 1.4: Huffman-mode asynchronous finite state machine

Davis [21] pioneered an extension of fundamental mode, called *data-driven mode*, which allowed a limited form of multi-input changes on the inputs to the state machine before the outputs had to settle. This mode of operation eventually evolved into what is now referred to as *burst mode*.

Burst mode is an extension of fundamental mode which relaxes the single-input-change assumption and allows a restricted burst of inputs to change before the state machine changes state. As with single-input-change fundamental mode, there are timing constraints placed on the feedback path to ensure correct operation. This design style has been shown to be practical and has been used to produce several real designs [21, 49, 69]. Although the timing constraints imposed by the restriction on the feedback variables limit the concurrency somewhat, the separation of the feed-forward and feedback paths simplify the specification and the synthesis leading to a practical asynchronous design style that is relatively easy for synchronous designers to use.

Several synthesis methods have been created that support gate-level hazard-free synthesis of burst-mode designs from a high-level specification. In particular, Nowick [67, 64, 66, 69] and Yun [89] devised state-machine synthesis methods which generate burst-mode implementations from high-level *asynchronous finite state machine (AFSM)* descriptions. The result of each synthesis method is an AFSM implementation which is composed of two-level combinational logic and (possibly) latches. Yun [90] later extended burst-mode specifications to allow a wider range of multi-input change input specifications, including conditionals and *don't cares*.

These synthesis methods have been used successfully to generate control circuitry for practical asynchronous designs, including the Post Office packet routing chip [21], the cache controller for the STRiP microprocessor [65], a SCSI controller chip [88], and an infrared communications IC chip (the ABCS chip) [49], demonstrating their wide applicability. In the Post Office chip, the synthesis methods produced a set of technology-independent logic equations to implement the next-state and output circuitry. These equations were then mapped by hand to custom gates and were simulated to ensure that there were no glitches. Any glitches that appeared during simulation were manually removed from the circuit. For the STRiP microprocessor, the mapping was done carefully by hand using simple hazard-preserving transformations to ensure that the resulting implementation remained free of hazards. For the ABCS and SCSI chips, the circuitry was mapped automatically by the technology mapper described in this thesis.

1.3.2.5 Timed Asynchronous Designs

Timed asynchronous designs take advantage of explicit timing information during some portion of the synthesis. Borriello, in [8], describes a method to use timing information in the synthesis and optimization of transducers, or synchronous-asynchronous interface logic. He also uses timing information to pad the circuit with delays to avoid hazards in the generated logic.

Lavagno describes an asynchronous design methodology which starts from an STG, and synthesizes an implementation using elements from a library of standard parts [44]. Although the initial implementation is complex-gate speed-independent, he uses timing to do the technology mapping. A procedure is applied to the initial mapped circuit to add delays to avoid hazards in the implementation. This technique not only slows down the circuit, but it also adds area. Furthermore, without having the final routing available, extra delay padding must be used to ensure that hazards don't exist in the final routed circuit.

Moon also addressed the problem of synthesizing a gate-level asynchronous implementation from an STG [59]. In his method he synthesizes a (possibly) hazardous implementation, and then applies hazard reduction techniques to eliminate hazards. The hazard reduction techniques are not guaranteed to work in all cases, however.

Myers proposed a synthesis method that extracts timing information from the environment and makes assumptions about gate delays which are used throughout the synthesis of the design [62]. The designs produced are robust within the given sets of timing constraints, and they are smaller than design styles that only consider implicit timing assumptions. However, doing the timing analysis is computationally expensive, adding an exponential factor to the complexity of the synthesis algorithm.

1.3.3 Design Style Considerations

Each of the design styles we have mentioned has been the subject of active research, and each has its own sets of advantages and disadvantages. Prior to this thesis, no one had addressed the general library binding problem for any of these styles.

There are several asynchronous synthesis tools that have been created to generate burst-mode designs, and application of these synthesis methods to circuits of practical size is not possible until the technology mapping problem has been addressed. There are also numerous synthesis methods available for speed-independent designs, indicating the broad interest in the research community. Although in this thesis we will focus on a subset of the speed-independent implementation styles, the results can be applied to other speed-independent implementation styles with little effort. As a result, we will focus on these two asynchronous design styles: burst-mode and speed-independent.

1.4 Technology Mapping

Technology mapping for synchronous systems is quite mature, having been an active area of research for the past ten years. We would like to briefly review the state-of-the-art in synchronous technology mappers. We will use this work as the starting point for our contributions.

Recall that the technology mapping step in synchronous synthesis takes as input a technology-independent set of logic equations along with a library of elements from a gate-array or standard-cell library, and produces a netlist implementing the equations using cells from the library, optimized for some cost metric such as area or delay. Because solving this problem exactly is compu-

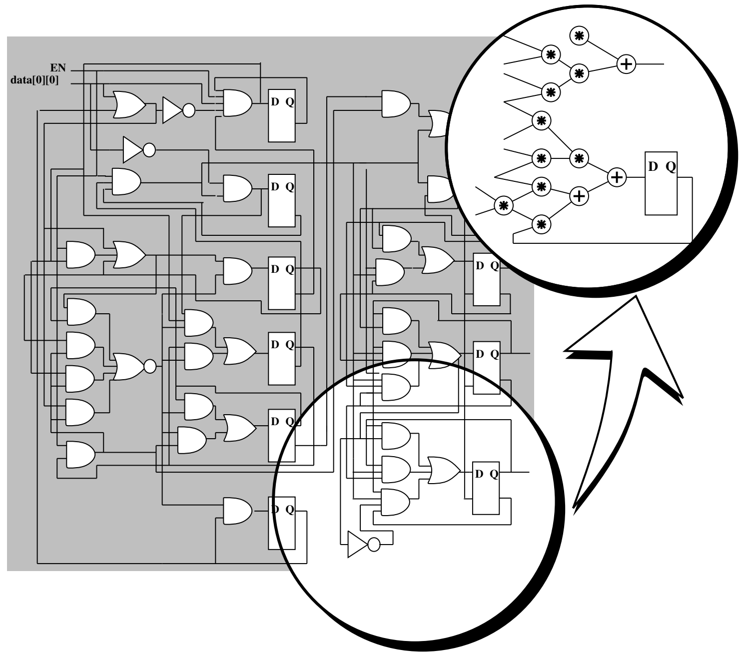


Figure 1.5: Technology mapping: decomposition step

tationally intractable [34], researchers have come up with heuristics to break the problem into smaller pieces to get a good quality approximation to the exact solution. In particular, algorithmic technology mappers [41, 73, 48] generally break the problem into three separate phases: decomposition, partitioning, and matching/covering.

Figure 1.5 (taken from [47]) depicts the decomposition step, in which the initial network, which is represented as a directed acyclic graph (DAG), is decomposed into a multi-level network composed of simple gates (e.g., two-input AND/OR gates or two-input NAND/NOR gates), whose corresponding representative functions are called *base functions*. The technology library must include cells implementing the base functions for a solution to be guaranteed.

Decomposition serves two purposes. First, because the base functions are assumed to be in the technology library, the initial decomposition guarantees that at least one solution (mapping) exists. Additionally, by breaking the network into a network of fine-granularity functions, the solution search space is increased and a better-quality solution will be found in subsequent steps.

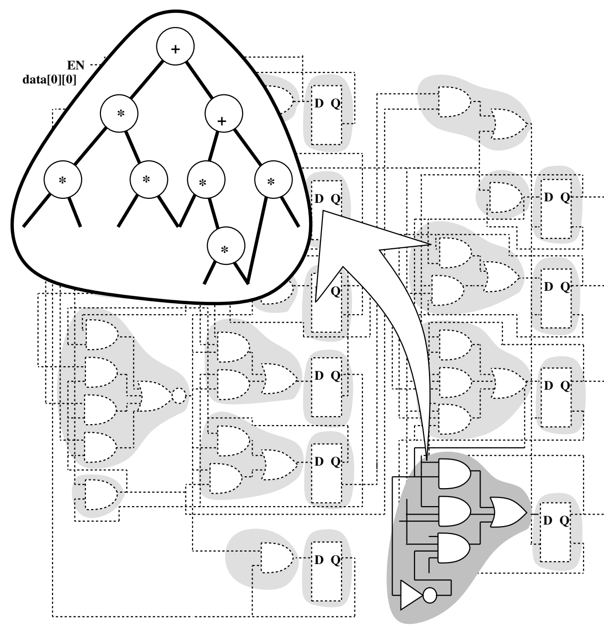


Figure 1.6: Technology mapping: partitioning step

Both the decomposition algorithm and the choice of base function can bias the eventual solution, since there can be many possible decompositions for a given network. Many technology mappers take advantage of this by using different decomposition heuristics depending on which metric is to be optimized.

In the partitioning step, depicted in Figure 1.6 [47], the circuit is split into sets of single-output *cones* of logic, where a cone of logic represents a subnetwork of a partition of the network obtained by cutting the network at various points of multi-fanout. Each cone of logic is represented internally as a *subject graph* which is later covered by elements from the library during the matching and covering step. Partitioning thus reduces the size of the subsequent covering problem to one that is computationally tractable.

Several approaches to partitioning have been proposed. In the simplest form, the network is split at every point of multi-fanout and at each connection to a sequential element, creating small cones of combinational logic. However, this restricts the mapping to operate very locally within the circuit, and improvements that cross multi-fanout boundaries will not be discovered.

In the MIS synthesis program [29], the authors proposed a more global scheme which starts by cutting the points of multi-fanout at primary outputs, to create an initial set of possibly non-disjoint cones of logic spanning the logic in the entire network. Each cone is then mapped in turn, starting at the primary output and stopping when either a primary input or a previously mapped portion of the circuit is encountered. Previously mapped nodes are found at points of multi-fanout; i.e., where the "cones" of logic intersect. In the case where a mapped node is internal to the cell it is bound to, the cell cannot be shared and the algorithm continues to map the circuit, resulting in some duplication of logic. In the case where the mapped node is the output of a mapped cell, the node is marked as a leaf and treated like a primary input for the purposes of further mapping. Although the mechanism has potential to share logic across cones, there is added expense when logic is duplicated across other points of multifanout.

In the matching/covering step, depicted in Figure 1.7 [47], each subject graph is implemented using elements from the technology library. All possible matches to library elements are found for each subnetwork within each subject graph. An optimal set of matching library elements is then selected from the set of possible matches to realize the network from elements of the library. Of all aspects of algorithmic technology mapping, the matching/covering step has received the most attention over the years, since it has the biggest impact on the quality of the results and is potentially very expensive computationally.

Matching algorithms can be classified as either *structural* or *Boolean.* Structural matching was first proposed by Keutzer [41], and has since been used in subsequent technology mappers, such as MIS. With structural matching, each library element is decomposed into a graph composed of base functions, called *pattern graphs.* A library element may have more than one pattern graph associated with it, representing the multiple possible decompositions of that element to allow the covering routine to find the best matching element later on. A *tree-based* matching routine can then recursively determine which pattern graphs are isomorphic to a subgraph of the subject graph. The subgraph is then annotated with the set of possible matches along with their costs. *Dynamic pro-*

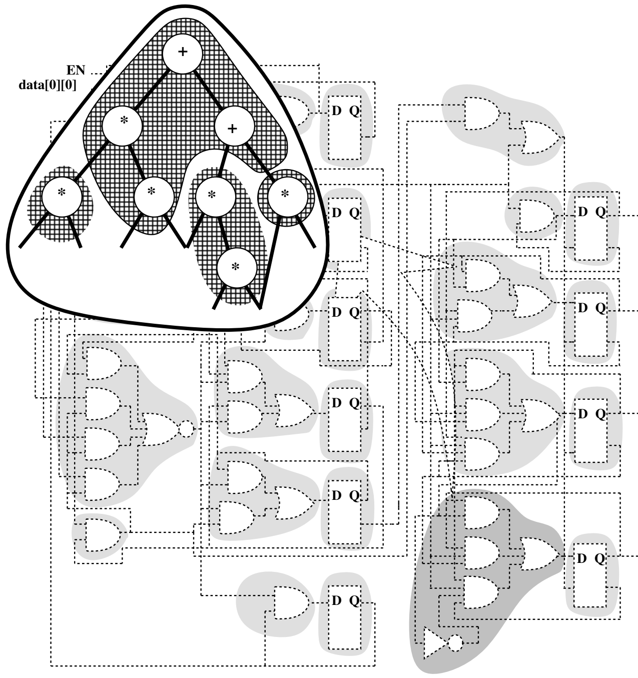


Figure 1.7: Technology mapping: matching/covering step

gramming techniques can then be used to find the optimum cover of the subject graph with elements from the library in linear time. Although structural techniques are computationally feasible, they don't exploit all properties of Boolean functions—such as *don't care* information—during library binding, leading to solutions of inferior quality.

In contrast, by using Boolean techniques in the matching/covering phase, *don't care* information can be exploited to find a better cover for the network at a cost of increased computational complexity [47]. With Boolean matching, each library element's function is represented by an *ordered binary decision diagram* (OBDD), called a *pattern function*. Each subnetwork of the subject graph is also represented by its OBDD, called a *cluster function*. During the matching phase, each cluster function is tested for equivalence against the pattern functions in the library by fixing the variable order in one OBDD and then permuting the order of the variables in the other OBDD until a match is found. The computational complexity of this process can be reduced by exploiting the *symmetries* inherent in each function as a quick filter on the matching. The symmetries can also be used to quickly determine feasible variable ordering, thus simplifying the matching process considerably. This matching algorithm can be extended to use both *controllability don't care* information and *observability don't care* information to increase the set of possible matches. Once the set of matches is found, a covering algorithm, similar to the dynamic programming algorithm used with structural matching, can be used to find an optimal cover.

As we will show later, this algorithmic approach to technology mapping cannot be applied directly to asynchronous designs, since hazards may be introduced in the process. We will use the algorithmic approach as the starting point for our analysis and show how it can be modified to apply to several asynchronous design styles.

1.5 Contributions

The major contributions of this thesis are to address technology mapping for two of the most popular asynchronous design styles: *burst mode* and *speed-independent*. We chose to address burst mode because it retains many of the advantages of asynchronous design while remaining close to synchronous design methodologies. As a result, it should be a good bridge for synchronous designers to cross to aid experimentation with asynchronous design. We chose to address the speed-independent design style because of its potential for robust design and increased concurrency. The difficulty of translating speed-independent designs into a hazard-free implementation remains the biggest obstacle to experimentation with larger designs. Addressing the technology mapping problem removes this obstacle and allows experimentation to get closer to the limits of the underlying synthesis methods.

We can outline our contributions in more detail, as follows:

For burst-mode designs, we analyzed synchronous mapping algorithms for their impact on the hazard behavior of the resulting designs and came up with theory and algorithms for correct hazard-free mapping of burst-mode designs. We implemented the algorithms within the context of an existing synchronous mapper, to create a technology mapping program suitable for burst-mode designs. This technology mapper takes a hazard-free burst-mode logic description and generates a hazard-free implementation composed of parts from a standard-cell or gate-array library. To complete the implementation of this mapper we also had to develop efficient algorithms to analyze the hazard behavior of the library elements and of the subnetworks. As part of joint work with Alan Marshall and Bill Coates of Hewlett-Packard Labs, we incorporated this technology mapper into the STETSON toolkit for asynchronous design, and used it to implement an asynchronous communications receiver chip [49].

We used the burst-mode technology mapper as the starting point for our investigation into technology mapping for speed-independent designs. We came up with new theory and algorithms to allow us to automatically map speed-independent designs into elements of an existing standard-cell or gate-array library. We then applied these algorithms to the implementation of standard benchmark circuits to test our theory on real circuits.

The contributions of this thesis allow the synthesis path to be completed for two of the more important asynchronous design styles, thus removing a key obstacle to acceptance of asynchronous design. With the technology mapper in place, designers can now use asynchronous techniques in place of synchronous techniques to implement real designs, and can thus more easily determine whether asynchronous design can pay off in terms of power savings and performance for their applications.

1.6 Thesis Outline

Having given a short background on asynchronous design styles, we delve into the problem of hazard detection in Chapter 2. First, hazard terminology is introduced, and then previous work on hazard analysis of transistor-level circuits is presented. We then present new work on hazard analysis of MUX-based pass-transistor logic which is essential for being able to use MUX-based field-programmable gate arrays [23] to implement asynchronous circuits.

Chapter 3 revolves around technology mapping for burst-mode designs. We first present our analysis of synchronous technology-mapping techniques, and then present modifications to the matching and covering algorithms to allow us to safely map burst-mode circuits. We then present results from having mapped a number of asynchronous benchmark designs using a mapper we developed for burst-mode designs. As joint work with colleagues from HP Labs, we incorporated this mapper into a toolset for asynchronous design and used the toolset to design a communications receiver integrated circuit, which we describe in the final section.

In Chapter 4 we address the problem of technology mapping for speed-independent designs. Since this design style is more removed from traditional synchronous design, it is more difficult to understand and explain. The difficult problem in addressing this design style was the decomposition step of technology mapping, and we present theorems and algorithms for successfully decomposing these circuits. We present an extension of the covering algorithm to allow us to take advantage of global sharing within the design. We conclude this chapter by describing the results of applying these mapping algorithms to standard asynchronous benchmark circuits.

There were problems we were unable to address in this thesis. Chapter 5 concludes with some ideas of how the contributions of this thesis can be extended to solve these open problems.

Chapter 2

Hazards and

Characterization of Library Elements

One of the biggest difficulties in synthesizing asynchronous circuits is avoiding the introduction of hazards into the circuit during the synthesis process. In this chapter we give a more formal treatment of hazards, first presenting a classification of hazards according to type. Because we are primarily interested in the hazard characterization of library elements (as we will show in the next chapter), we examine combinational hazards and their causes in more detail. After presenting basic terminology and theory related to combinational hazards, we propose a simple model that can be used to analyze the hazard behavior of gate-level library elements. Because this model can also be used to analyze Boolean subnetworks, we defer treatment of hazard analysis of gate-level networks until the next chapter. We then examine where hazards can occur in pass-transistor circuits, focusing specifically on MUX-based FPGA cells. After developing theory supporting their hazard analysis, we present results from analyzing the Act1 FPGA cell and its personalizations for hazards.

2.1 Terminology

Although we assume the reader has a basic familiarity with Boolean terminology, for clarity this section defines some terms that appear frequently in this thesis. For more detail, please see [10] and [55].

A *literal* is an instance of a Boolean variable or its complement. For example, if a is a Boolean variable, in the equation $y = ab + a'c$, each occurrence of a is a literal (i.e., a is a literal and a' is another literal).

An *implicant* is a product of literals within a sum-of-products (SOP) expression. The term *cube*, which refers to the mapping of Boolean variables onto an n-dimensional cube, is used interchangeably with implicant.

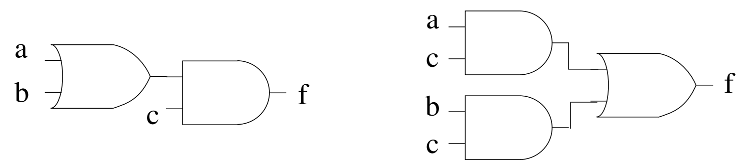
A *prime implicant* is an implicant that is not contained by any other implicant of the function.

A *minterm* is a product of literals that contains all input variables of the function.

Definition 2.1 *A* transition cube*, T[*α*,* β*], is the smallest Boolean subspace that contains* α *and* β*, where* α *is the minterm corresponding to the starting point, and* β *is the minterm corresponding to the endpoint. (This is described as a transition subcube in [6], and is also equivalent to the supercube(*α*,* β*) [74].) The transition cube defined by T[*α*,* β*] includes all minterms that can be reached in a (possibly multi-input change) monotonic transition(s) from* α *to* β.

A function expressed by a two-level sum-of-products equation can be directly mapped into a two-level gate implementation of AND gates feeding into an OR gate, where the inputs to the AND gates have a one-to-one correspondence with the variables in the product terms. We will use SOP expressions and their two-level gate implementations interchangeably throughout this thesis.

A *Boolean Factored Form (BFF)* expression is a factored Boolean expression that maps into a multilevel logic network, where parentheses represent grouping of a set of literals into a single gate. Figure 2.1 illustrates two different BFF expressions for the same function. In Figure 2.1a, the BFF expression is represented by its equivalent OR-AND network. Figure 2.1b shows the same function factored out into a sum-of-products AND-OR network. Although these two representations are *functionally* equivalent—that is, they have the same set of minterms—the two expressions are not *structurally* equivalent, which will be an important attribute when we consider the hazard behavior of a network.



a) f = (a + b) c **b)** f = a c + b c

Figure 2.1: Equivalent functions with different Boolean factored form representations

2.2 Hazards

In the most general sense, a *hazard* is an unwanted transition (or set of transitions) that appears on the output of some logic function in response to a set of changes on its inputs. There are many different classifications of hazards, depending upon the level of abstraction and on the design style. For the first part of this thesis, we are mainly concerned with hazards in combinational logic because we assume that synthesis methods feeding into the library binding step have produced a *hazard-free* design at the logic level. For the two design styles we are concerned with, it is important to understand what types of hazards can be introduced at other levels of abstraction, in order to understand the requirements that must be imposed on the initial circuits to be passed to the technology mapper. In later chapters we will be concerned with sequential hazards which may result from transformations on the combinational logic, but we will defer discussion of this topic to that chapter.

We can classify hazards as either *sequential* or *combinational*. Sequential hazards only exist in a circuit with feedback; if the feedback were to be cut, the hazard would disappear. Combinational hazards are found in combinational networks, where a combinational network can also be a subnetwork of a sequential circuit that is composed only of combinational elements. By definition, if a sequential circuit has a combinational hazard for some set of input transitions, it cannot also have a sequential hazard for those same transitions.

The sequential hazards that occur in generalized fundamental-mode circuits are called *essential hazards* [83, 55]. Essential hazards result from signals that travel through two paths in the circuit, one which is composed of purely combinational logic and the other which has either a sequential component or feedback. Essential hazards can only be eliminated by controlling the delays in the circuit, so that the path through the combinational component propagates to all state-holding elements prior to changes from other state-holding elements. This can be achieved in one of two ways: modification of delays in the implementation (e.g., by ensuring that the fundamental-mode assumptions on the feedback path hold), or modification of state encoding to ensure that signals change in a particular order. We assume that essential hazards are either removed by the state-machine synthesis method and/or by adding delay into the feedback path to ensure that the fundamental-mode assumptions still hold. Therefore, for technology mapping of burst-mode designs we are concerned only with combinational hazards.

For speed-independent designs, we cannot take advantage of timing assumptions on the feedback lines to eliminate hazards, as we can with generalized fundamental-mode designs. As a result, a class of hazards known as *delay hazards* [2] must be considered during synthesis and technology mapping. We explain these hazards and their implications in more detail in Chapter 4.

2.2.1 Combinational Hazards

For combinational hazards, we are often dealing with a portion of a larger circuit. The theory presented in this section assumes arbitrary wire delays in the circuit. In the absence of points of multi-fanout as is generally the case with combinational circuits, it should be clear from the discussion in the previous chapter that the results on combinational hazards also apply to combinational circuitry within larger sequential circuits assuming an arbitrary gate-delay model (as for speed-independent circuits).

There are two basic classes of combinational hazards: *function* and *logic* hazards. Function hazards are a property of the logic function and can only be eliminated through appropriate placement of delay elements, whereas logic hazards are purely a property of the implementation. By defini-

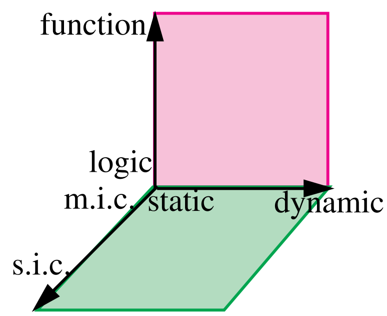


Figure 2.2: Combinational hazard space

tion, if a network has a function hazard for a given transition, then it cannot also have a logic hazard for that same transition.

Within the class of logic hazards, there are *single-input change* (s.i.c.) hazards and *multi-input change* (m.i.c.) hazards. Additionally, each class of hazards (function and logic) includes both *static* and *dynamic* hazards. Given a Boolean function f and a transition between two points α and β in the input space $\{0,1\}^n$, static hazards apply to transitions where $f(\alpha) = f(\beta)$, and dynamic hazards apply to cases where $f(\alpha) \neq f(\beta)$. This partitioning of the class of hazard behaviors can be seen in Figure 2.2. We distinguish between static and dynamic hazards because their treatment is different, as we will show later.

More formally, we can define the terms as follows (these definitions were taken from Bredeson and Hulina [13]):

Definition 2.2 *A Boolean function f contains a* static function hazard *in a transition cube* T[α, δ] *if and only if*

1. $f(\alpha) = f(\delta)$ *and*
2. *there exists at least one input state* $\beta \in$ T[α, δ] *such that* $f(\alpha) \neq f(\beta)$

Static hazards can be further separated into *static 0-hazards* and *static 1-hazards.* A static 0-hazard is a static hazard where $f(\alpha) = f(\beta) = 0$, and the output makes a $0 \rightarrow 1 \rightarrow 0$ transition during the input change from α to β. Similarly, a static 1-hazard is a static hazard where $f(\alpha) = f(\beta) = 1$, and the output makes a $1 \rightarrow 0 \rightarrow 1$ transition during the input change from α to β. For two-level logic, static-1 hazards are more prevalent in sum-of-products circuits and static-0 hazards are more prevalent in product-of-sums circuits.

Definition 2.3 *A Boolean function f contains a* dynamic function hazard *in a transition cube* $T[\alpha, \delta]$ *if and only if*

1. $f(\alpha) \neq f(\delta)$ *and*
2. *there exists at least one pair of input states* $\beta \in T[\alpha, \delta]$, $\gamma \in T[\beta, \delta]$ *such that* $f(\beta) = f(\delta)$ *and* $f(\gamma) = f(\alpha)$.

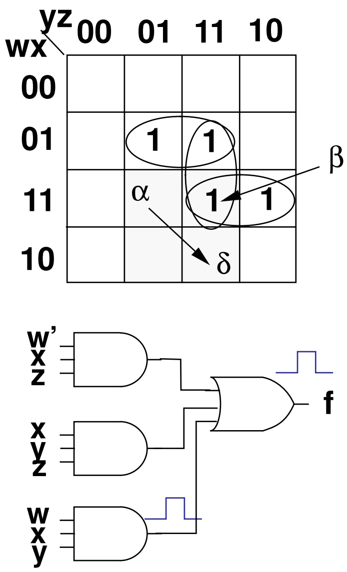
Figure 2.3 illustrates these definitions. The circuit on the left with its accompanying Karnaugh map illustrates a static 0-hazard when traversing between *wxy'z and wx'yz.* We can see that if the delays are such that *y* changes before *x,* then two of the gates will go on momentarily. Because we cannot control this behavior except by adding delays, it is a function hazard. In the second example, we can see an illustration of a dynamic function hazard. With the appropriate delays, it is possible for the circuit to undergo a transition from α to β to γ finally ending up at δ, resulting in the dynamic function hazard, as shown in the figure.

With these definitions in hand, Bredeson proved the following theorem in [13]:

Theorem 2.1 *If a Boolean function* f *contains a function hazard for an input change* α *to* β*, it is impossible to realize a network for* f *which will eliminate possible multiple output changes for the input change* α *to* β.

In other words, Theorem 2.1 implies that to realize a hazard-free network for a given transition cube, that cube must be function hazard-free.

We can now define logic hazards. Again, the definitions are taken from Bredeson and Hulina.



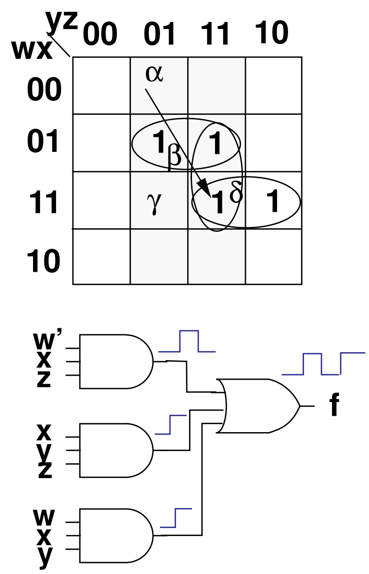


Figure 2.3: Function hazard examples

Definition 2.4 *A network contains a* static logic hazard *for the transition cube* T[α*,* β] *if and only if*

1. *f(*α*) = f(*β*) and*

2. *no static function hazard exists within the transition cube* T[α*,* β] *and*

3. *during the input change* α *to* β*, a momentary pulse may be present on the output.*

Static logic hazards occur in sum-of-products implementations whenever a transition is not properly covered by a single cube, i.e., whenever the implementation does not contain a single gate that maintains the output value throughout the input transition.

In Figure 2.4a, the transition from *w′xyz* to *wxyz* in the Boolean space is not covered by a single gate in the implementation. It is thus possible, through some set of gate delays, for both AND gates to be off momentarily, and for the output to make a transition through 0 before settling at its final value, resulting in a static 1-hazard. This problem can be eliminated if an additional AND gate with inputs *xyz* is added to hold the output high during that transition.

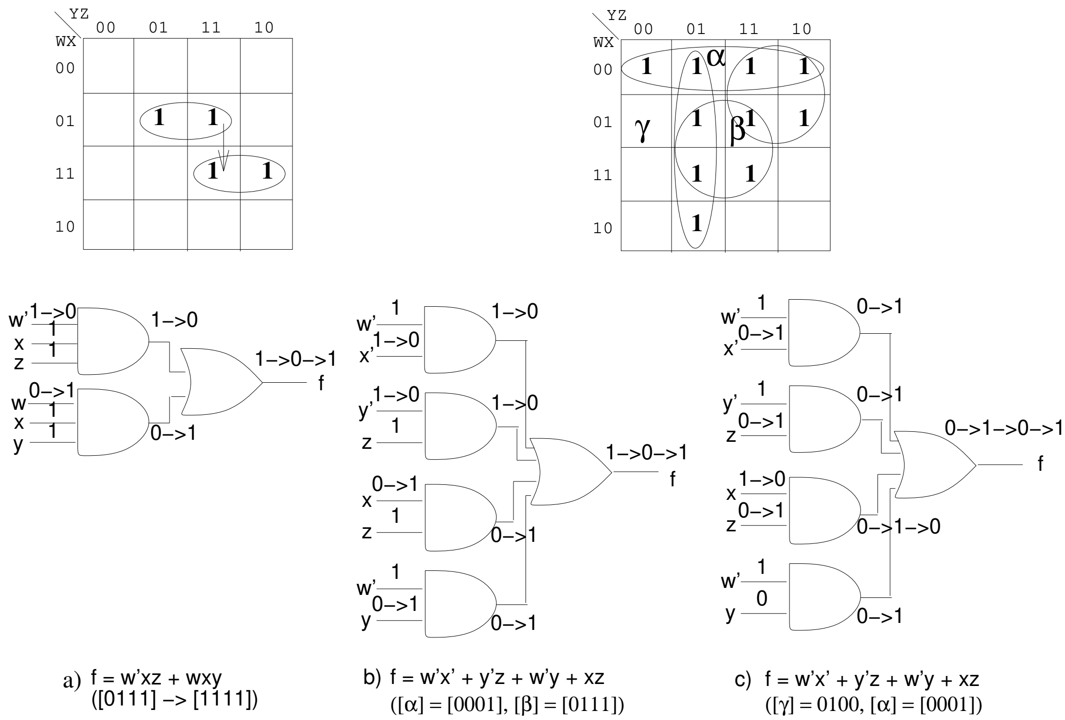


Figure 2.4: Examples of various types of hazards.

Figure 2.4b illustrates a multi-input change static logic hazard. During the transition from point α to point β, there is no single gate that holds the output high during the transition. So if gates $w'x'$ and $y'z$ are sufficiently fast and the gates xz and $w'y$ are sufficiently slow, then the output can momentarily take on a 0 value during the transition, resulting in a static hazard.

For two-level sum-of-products expressions, Eichelberger showed that it is necessary and sufficient that all prime implicants be included to ensure that there are no multi-input-change static logic hazards in the circuit [32]. For multi-level expressions, the conditions are more complex and are discussed in [12].

Definition 2.5 *A network contains a* dynamic logic hazard *for the transition cube* T[α, β] *if and only if*

1. *$f(\alpha) \neq f(\beta)$ and*
2. *no dynamic function hazard exists within the transition cube* T[α, β] *and*
3. *during the input change α to β, a momentary "0" output and a momentary "1" output may appear.*

Dynamic logic hazards are applicable to both single-input change and multi-input change situations. A dynamic hazard occurs when, during an expected $0 \rightarrow 1$ ($1 \rightarrow 0$) transition of the output, a $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$ ($1 \rightarrow 0 \rightarrow 1 \rightarrow 0$) transition occurs. For single-input change conditions, this corresponds to a situation where a literal and its complement fan out to several paths. For multi-input change conditions, a dynamic hazard can occur when a single gate is turned on momentarily during the transition. For example, in Figure 2.4c a transition between points γ and α can result in gate xz turning on and off before gate $y'z$ turns on, resulting in a dynamic hazard.

2.3 Hazard Characterization of Library Elements

In this section, we examine the characterization of Boolean networks implemented in two specific technologies: static CMOS logic and pass-transistor logic. As we will show in subsequent chapters, replacement of a Boolean subnetwork by an element from a standard-cell or gate-array library cannot be done without knowledge of the hazard behavior of the library element to ensure that hazards are not introduced during technology mapping. An understanding of the underlying technology in which the library elements are implemented is necessary to properly characterize the hazard behavior.

For static CMOS gates, we briefly introduce the model which we use for subsequent characterization; analysis algorithms are deferred until the next chapter, because, in addition to library elements, this analysis applies to Boolean subnetworks. We go into more detail on pass-transistor logic in this chapter, because the details are specific to library cells. We are interested in this type of logic because of the popularity of MUX-based FPGA technologies. As a result, we tailor our

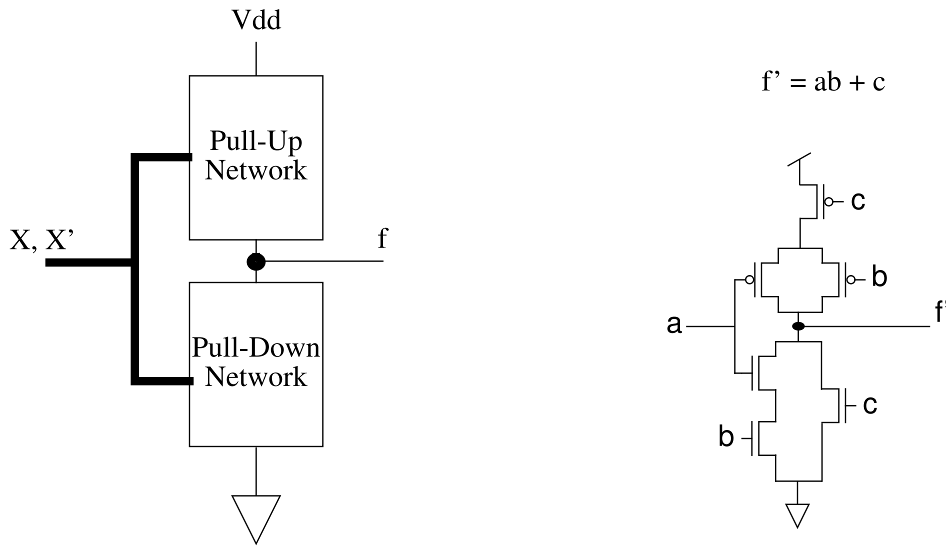


Figure 2.5: Static CMOS networks

discussion to this application, although the theory we develop applies to general pass-transistor networks.

Other types of active logic, such as Domino and CVSL, can be modeled for the purposes of hazard analysis by the gate-level formalisms used for static CMOS logic. Therefore we omit discussion of these logic families. (Information on these logic families can be found in [85].)

Throughout the remainder of this chapter we assume the reader has a basic familiarity with MOS technology.

2.3.1 Static CMOS Logic

Static CMOS logic gates are implemented as an interconnection of P-type and N-type MOS transistors. Each transistor in the network acts like a switch which conducts when the transistor is on, by effectively shorting the drain and source of the transistor, and which does not conduct when the transistor is off, isolating the drain and source nodes. A gate implementing the complement of a function *f* can be implemented by connecting a *pull-down* network of N-type transistors and a *pull-up* network of P-type transistors together to form the desired function. The pull-down network

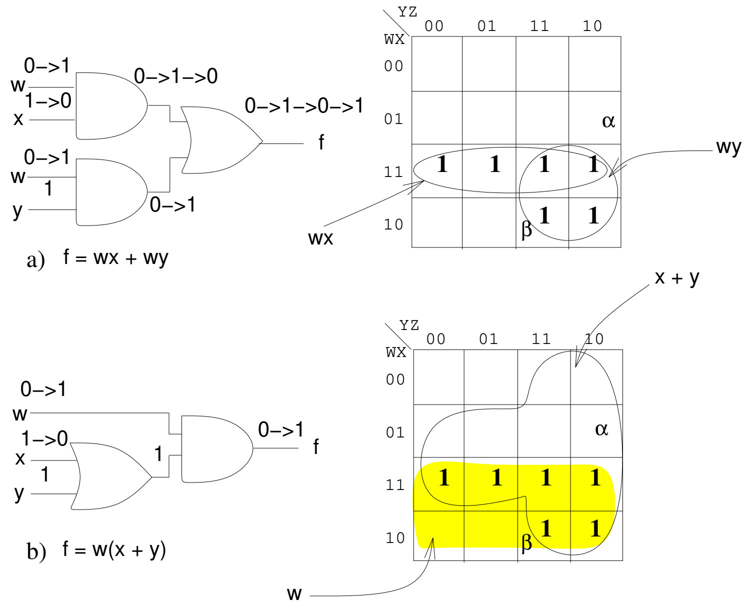


Figure 2.6: Different structures for the same function can result in different hazard behaviors.

consists of N-type MOS transistors interconnected to implement the complement of the function, and the pull-up network consists of P-type MOS transistors interconnected to implement the function's dual, connected as shown in Figure 2.5.

In steady-state operation, either the pull-up network is "on", connecting the output to the voltage rail to maintain a logic "1", or the pull-down network is "on", connecting the gate's output to ground to maintain a logic "0". Thus, with static CMOS logic, the output node is *always* driven, negating any storage effects due to capacitance on the output node.

Because the output is always driven, its logical value is always defined. As a result, each product term in the Boolean function defining a gate's operation can be mapped to a cube in the Boolean n-space. Boolean factored form (BFF) is a convenient form which allows us to express both the structure and functionality of an interconnection of gates (as mapped to cubes in the Boolean n-space), and also allows us to reason about the hazard behavior of the individual library elements.

For example, the BFF expression for the library element in Figure 2.6a indicates that the library element is implemented as a sum of two cubes, resulting in a dynamic logic hazard when inputs *w* and x change with $y = 1$. However, if this element were implemented as shown in Figure 2.6b, then there would be no dynamic hazard for that input burst.

Note that other representations are possible as long as they adequately express both the structure and the functionality of the library element. Boolean factored form is simply a convenient notation which meets these requirements for CMOS circuits. This implies that the structure of each library element must be accurately abstracted from the transistor-level description of the library element, and represented as its equivalent Boolean factored form in the library description.

We examined some typical commercial standard-cell and gate-array libraries to see how many elements contained logic hazards. In this examination, we extracted the Boolean Factored Form description for each cell from the library's databook and used that representation for hazard analysis. Table 2.1 shows the elements in those libraries that contained logic hazards. The LSI and CMOS3 libraries are commercial CMOS ASIC libraries [38]. The GDT library is a CMOS standard-cell library that was produced specifically for a particular chip, and includes many complex AND-OR-INVERT (AOI) and OR-AND-INVERT (OAI) gates. For these libraries, the only library elements that contained logic hazards were multiplexors.

	Hazardous Elements			
Library	**Types**	**Number**	**Total Elements**	**% Hazardous**
LSI9K	Muxes	12	86	14%
CMOS3	Muxes	1	30	3%
GDT	None	0	72	0%

Table 2.1 Static CMOS Libraries and their hazardous elements

2.3.2 Pass-Transistor Logic

Functions can also be implemented in pass-transistor logic. This style of circuit design is characterized by connecting a set of branches together in a wired-OR configuration, where each branch is composed of a series interconnection of transistors, as shown in Figure 2.7. A branch is "on" when the gates of all transistors are high, and the input to the branch is then passed to the output. For example, in the figure above, when a is high and b is low, signal e is passed to the output. Drive

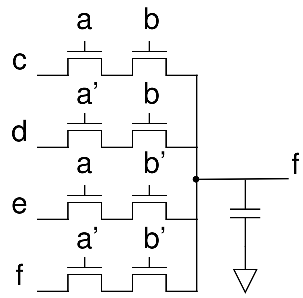
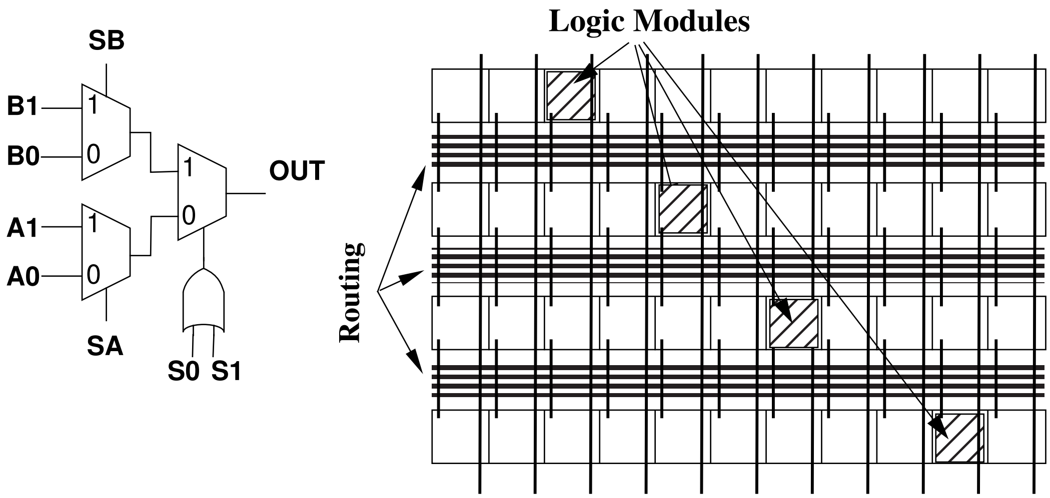


Figure 2.7: Example of a pass-transistor network

fights can result if two branches are on simultaneously. Conversely, if no branch is driving, the capacitance on the output node will hold the node’s previous value. Typically, functions designed with pass transistors are designed in such a way that these two cases never occur in steady-state, i.e., the pass-transistor network always has exactly one branch driving the output.

There is a large body of work on the analysis of pass-transistor networks, which we will briefly review before extending this theory to apply to hazard analysis. Our ultimate goal is to apply this general pass-transistor theory to analysis of MUX-based FPGA technology, where the basic FPGA cell is implemented as an interconnection of pass transistors. As with gate-level combinational hazard analysis, because we are examining circuits with no points of multi-fanout, assuming arbitrary transistor delays is equivalent to assuming arbitrary wire delays.

FPGA technology is particularly important because implementation of digital circuits using FPGAs has become popular due to quick turnaround time and inexpensive prototyping cost. The basic cell is *personalized* to realize any one of a number of macros realizing different combinational functions, in effect creating a virtual library of combinational components which can then be used to implement a given netlist. To use this technology in the implementation of hazard-free control circuits, we must be able to characterize the hazard behavior of the various personalizations of the basic cell. Although we concentrate on hazard analysis of the basic cell of a specific manufacturer, the results will apply to any FPGA that has a basic cell implemented with pass-transistor technology.



a) Actel Act1 basic module **b)** FPGA layout

Figure 2.8: Actel FPGA

In the remainder of this section, we first present a little more background on the FPGA we are concentrating on. Next, after briefly reviewing some of the previous work on the analysis of pass-transistor networks, we present some terminology that will be used throughout this section. We then present some formalisms for hazard characterization of pass-transistor-based cells, and use these formalisms to theorize where hazards can occur. Finally, we use this theory to analyze the hazards in personalizations of the basic cell provided by the Actel Act1 FPGA, and we present those results prior to concluding this chapter. Although we only analyze the Act1 cell, the results can easily be applied to other Actel gate arrays, such as Act2 and Act3.

2.3.2.1 Background

Actel FPGAs are *antifuse-programmable* gate arrays [33]. Each FPGA consists of a number of instances of the basic logic module, where the basic cell can implement any one of a number of Boolean functions. The basic Actel Act1 logic module is shown in Figure 2.8a. It consists of a four-input MUX controlled by three control lines. Unprogrammed instances of the basic logic module are tiled together in rows separated by horizontal routing channels, and overlaid by vertical routing, as shown in Figure 2.8b.

Programming the FPGA to implement a given netlist involves both cell personalization and routing. Once each gate in a netlist has been mapped to an instance of the basic module on the FPGA, each instance is *personalized* to realize the function it is to implement. To aid the placement and routing process, a given function, e.g. an AND3 with one inverted input, may have several different personalizations which implement that function. Choice of the personalization is decided by the router at the place-and-route phase, and is not under the control of the technology mapper.

Personalization involves a combination of *bridging* (physically shorting together) two or more inputs within a given cell, and connecting one or more of the inputs to voltage rails (Vdd if the input is to be connected to logic “1”, and GND, if the input is to be connected to logic “0”). In many FPGA technologies, this bridging is done over a path with non-negligible delay which, as we will show later, can cause hazards.

2.3.2.2 Previous Work

We want to develop a model for analyzing hazards in pass-transistor networks, similar to the model we used to analyze static CMOS networks. We build upon work done by previous researchers to develop our model. We briefly review that work in this section.

Radhakrishnan *et al* [72] presented a mathematical model for analysis of pass-transistor circuits, and proposed both a Karnaugh-map based manual synthesis method and a Quine-McCluskey-like synthesis method for synthesizing pass-transistor based circuits. Pedron and Stauffer extended the mathematical model to include a larger class of pass-transistor circuits, and proposed a synthesis method for generating pass-transistor realizations [71]. An alternate graph-based mathematical model was used by Brzozwski and Yoeli to analyze a larger class of CMOS circuits [17]. Each of these formalisms is capable of modeling a tristated output; i.e., the case where a node is not being actively driven, but a capacitance maintains its previous value. These models were used only in analysis of the *functionality* of the circuits, and not the *hazard behavior* of the circuit.

Sasi used the model originally proposed by Radhakrishnan to analyze CMOS complementary logic, pseudo-nMOS logic and static CVSL logic for hazards [75]. He proved that the minimal transistor representation of a complementary CMOS gate does not always guarantee a hazard-free design. He only performed the analysis for single-input change hazards, however.

Other researchers have used a ternary simulation model to detect hazards in pass-transistor circuits. Yoeli and Rinon originally applied ternary algebra to switching networks for static hazard analysis [87]. Eichelberger extended this work by using this ternary model for simulation of gate-level networks to detect both single input-change and multi-input change hazards [32]. Bryant later applied ternary algebra to a switch-level model of a network, and implemented switch-level simulation algorithms in the program MOSSIM, which he used to detect races in switch-level networks [16].

Simulation approaches to hazard detection suffer from having to exhaustively simulate the network to ensure that there are no hazards. For individual gates, this may be a reasonable approach, but for larger circuits it is prohibitive. Furthermore, a model for compactly representing the hazard behavior of a network is still required for later comparison of hazard behaviors. It is for this reason that we have chosen an algebraic approach to detecting hazards in pass-transistor networks. We use the formalisms employed by [72], [71], and [75] in our analysis.

2.3.2.3 Terminology

In this subsection we develop the formalisms on which we base our analysis of the hazard behavior of pass-transistor circuits.

Definition 2.6 *A* pass transistor *is a MOS transistor operated as a switch, where the transistor drain (source) is connected to the signal to be passed along, the transistor gate is connected to the control input, and the output signal is taken from the transistor source (drain).*

Initially we will not distinguish between a single N-type or P-type MOS transistor and a pass "gate" composed of a complementary pair of transistors connected by complementary control variables.

Definition 2.7 A pass network *is an interconnection of pass transistors which realizes a particular switching function f(X), where X = {x_1, x_2,…, x_n} is the set of inputs to the function.*

Each terminal of a pass transistor in a pass network is driven by VDD (logical "1"), GND (logical "0"), some complemented or uncomplemented primary input, or by the source or drain of another terminal in the network.

Definition 2.8 A branch *of a pass network implementing the switching function f(X) is a series connection of pass transistors where the drain or source of the transistor at one end of the series is connected to an input source selected from the set {0, 1, x_i, x_i'}, and the other end of the series connection is connected to the primary output.*

A given pass network may be composed of several branches connected together to implement the given switching function. Figure 2.9 shows a pass network with two branches.

Definition 2.9 A pass variable *is the input to a branch of the pass network. A pass variable may be chosen from the set {0, 1, x_i, x_i'}.*

Definition 2.10 A control variable *is an input to the gate of a transistor in the pass network. When the control variable has a value equivalent to a logic "1", the pass gate conducts.*

If we look at the top branch in the Figure 2.9 we see that *c* is the pass variable, and *b'* and *a* are the control variables.

Definition 2.11 A pass implicant *is a Boolean switching function which denotes the function of a branch of a pass network, and includes information about both the pass variable and the switching function. A pass implicant is denoted* $C_i[P_i]$, *where* C_i *is a product term composed of control variables which control the pass transistors in that branch, and* P_i *is the pass variable which will get passed to the output if the product term is true.*

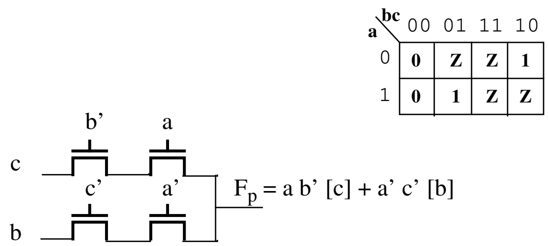


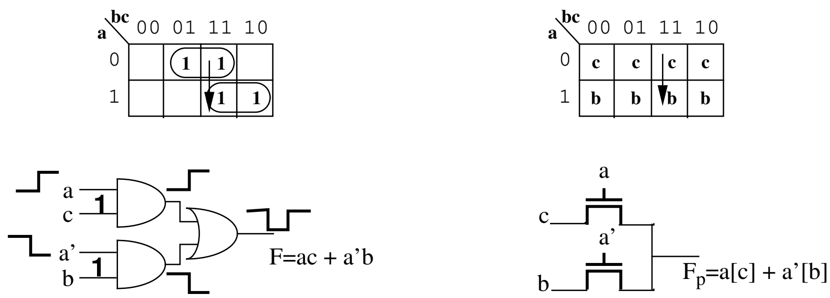
Figure 2.9: Illustration of pass-transistor terminology

This is similar to an *implicant* in Boolean algebra, except that instead of passing a constant "1" if the implicant is true, the value of the pass variable is passed if the implicant is true. The circuit in the figure thus has two pass implicants: *a b' [c]* and *a' c' [b]*.

Definition 2.12 *A pass function, F_p: $B^n \rightarrow \{0, 1, Z\}$, is a sum of pass implicants.*

The pass function thus represents a wired-OR connection of a set of pass-transistor branches. Note that a pass function differs from a Boolean function in that both the ON-set (when the function is TRUE) and the OFF-set (when the function is FALSE) must be specified, due to the presence of the high-impedance state when no branch is driving the output. With Boolean functions (representing static CMOS logic), we only need to indicate the ON-set, since the output is always being driven.

Given this terminology, we say that a Boolean function F is equivalent to a pass function F_p if each value in the input space of F is explicitly specified by F_p, i.e., *ON-set(F) = ON-set(F_p), and OFF-set(F) = OFF-set(F_p)*. Clearly, there may be several possible pass-gate implementations of a given function F, even considering a fixed pass network configuration. In the example shown in Figure 2.9, we can see that $F_p \neq F$, since the output is not explicitly driven for several states in the input space.



Hazardous implementation **No static logic 1-hazard**

Figure 2.10: Example of gate-level versus transistor-level hazards for a MUX.

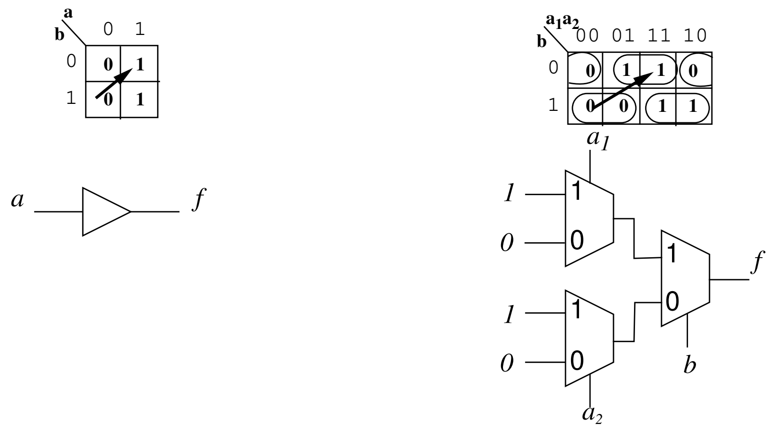
2.3.2.4 Hazard Analysis of MUX-based Pass-Transistor Networks.

Before we present a formal hazard analysis of pass-transistor networks for MUX-like configurations we will present some examples to illustrate the issues.

Due to charge storage, pass-transistor networks cannot be analyzed for hazards in the same manner as gate-level networks. To see, we use a MUX implemented as an interconnection of simple logic gates, shown in Figure 2.10, as an example. In Figure 2.10, there is a static logic 1-hazard present in the gate-level diagram (with *a* changing, and *b* and *c* remaining high). In the pass-transistor design, there will not be a hazard, because if both branches of the network are turned off, the output capacitance will hold the output at the previous value of 1. There are additional problems imposed by the pass-transistor design that might not exist in a gate-level design, however.

When dealing with a fixed logic module composed of pass transistors, such as the Actel Act1 logic module, there are many possible personalizations for a given function. Because the transistor layout is fixed, any cell can be personalized at the same cost to implement the desired function. However, the routing costs may vary depending upon the personalization that is chosen, and so the costs of implementing a netlist are related solely to routing and are thus difficult to compute and compare. It is easy to postulate many hazardous implementations for a given Boolean function in this technology, even when an equivalent gate-level implementation may be hazard-free.

For example, the gate-level implementation for the function in Figure 2.11a is simply a buffer. Figure 2.11b shows a personalization of the Actel cell which realizes the buffer. In this implementa-



F = a

$F_{pt} = ba_1[1] + ba_1'[0] + b'a_2[1] + b'a_2'[0]$

a) **b)**

Figure 2.11: Dynamic hazard with constant MUX inputs

tion, the control inputs to the two intermediate MUXes are bridged together and connected to the signal *a*. We distinguish the two connections to *a* with subscripts. This implementation has a dynamic logic hazard for the transition 01→10, caused by the bridging. The dynamic hazard can occur if a_1 changes, followed by *b* changing, followed by a_2.

Although no node in a pass network that exhibits charge storage can exhibit a static 1-hazard as it might in a gate-level implementation, static logic 1-hazards are still possible when multiple control inputs change, or when a control variable changes at the same time as a pass variable.

In our analysis we assume that bridging has some non-negligible delay, i.e. within the module, wire delays are significant due to routing and anti-fuse resistance. (This is not what we assume in a gate-level model.) Therefore, we must treat bridged inputs as separate instances of the variables. We call these variable instances *transient variables* and denote them by subscripting the steady-state variable name by a unique instance number.

Given these assumptions, we must define a function that expresses the transient behavior of the configuration due to bridging delays. We can then analyze this function to characterize the hazard behavior of the network.

Definition 2.13 *The* transient pass function*, F_{pt}, is derived from the pass function, F_p, by subscripting each instance of every pass variable in the function to distinguish its unique occurrence. In the steady state, i.e., when no pass variables or control variables are changing, $F_{pt} \equiv F_p$.*

We equivalently refer to F_p as the *steady-state function*, and F_{pt} as the *transient function.* We saw an example of a transient pass function in Figure 2.11, where the input *a* had two separate instances, as reflected by the subscripts in the equation.

The following theorem shows that by distinguishing the instances of each specific input variable we can convert logic hazards in the steady-state space into function hazards in the transient space.

Theorem 2.2 *If F_p is the pass function corresponding to an implementation of the Boolean function F such that each point in the Boolean space defined by ON-set(F) $\cup$ OFF-set(F) is represented by a pass implicant of F_p and such that ON-set(F_p) $\cap$ OFF-set(F_p) = $\emptyset$ (i.e., for every input pattern, F_p has a well-defined and non-conflicting output value), then F_p has a logic hazard for a function-hazard-free transition cube of F if and only if F_{pt} has a function hazard over the same transition cube.*

Proof:

$\Rightarrow$ (By contradiction.) Suppose the pass-transistor network represented by F_p has a hazardous implementation for some function-hazard-free transition. Then a glitch may appear on the output during that transition, since a logic hazard exists. Furthermore, suppose that the transient function, F_{pt}, does not have a function hazard for that transition. Then the hazard on F_p must show up as logic hazard in F_{pt}. Because of the charge storage on the output node, even if all branches of the function are off momentarily during the transition in the inputs, the output will retain the value prior to the change; therefore, no glitch can result from this situation. Furthermore, our assumptions imply that there are no states where two branches are simultaneously driving the output to opposite values; therefore, no glitch can result from metastability. Given that the original pass function was function-hazard free for this transition, the only other possible cause of the glitch then is that multiple inputs of F_{pt} are changing such that a glitch appears. But because the output value after each transition of a transient variable is represented by a minterm in F_{pt} this implies that there is a function hazard in F_{pt} for that transition, and thus F_{pt} cannot have a logic hazard for that same transition, contradicting our assumptions.

$\Leftarrow$ Suppose the transient pass function corresponding to the implementation has a function hazard in F_{pt} over a function-hazard-free transition of F_p. Then, during a transition from the starting point to the endpoint, it is possible, due to delays, that the function hazard will be exercised resulting in a glitch on the output. However, by assumption F_p was function-hazard free implying that the glitch could only be caused by a logic hazard. ❑

This theorem implies that at least two variables in the transient space must change for a logic hazard in the original network to be possible (because function hazards only apply to multi-input changes). Furthermore, at least one variable must have multiple instances for there to be a function hazard in a transition cube of F_{pt} which is not also a function hazard in the corresponding transition cube of F_p.

For the transition shown in the example illustrated in Figure 2.11, we can see that a function hazard exists over the transient transition cube and thus there is a dynamic hazard.

The corollary below follows directly from the theorem:

Corollary 2.2.1 *Any personalization of a MUX-based FPGA that does not use bridging of inputs or control variables is logic-hazard free.*

From the theorem, we see that to determine whether or not a given personalization has logic hazards, we need only generate the set of function-hazard-free transitions for the given function, and then examine its transient transition cube to see if there are function hazards within that transition cube. A procedure for generating the set of function-hazard-free transition cubes will be described in the next chapter.

2.3.2.5 Actel Library

The Act1 basic cell implements all functions of two variables, most functions of three variables and some functions of four variables. For a given function, there are many possible implementations given the basic Act1 cell. However, the Actel router chooses among a restricted set of personalizations during placement and routing of a given netlist.

We examined selected Act1 macros for logic hazards, using the information on the transistor-level design of the basic cell that can be found in the databook. Table 2.2 lists the statistics for the cells we analyzed.

Number of Inputs	Number of Macros Analyzed	Number of Hazard-Free Macros	Number of Hazardous Macros	Number of Hazardous Macros w/Hazard-Free Personalizations
1-input functions	4	4	0	n/a
2-input functions	14	14	0	n/a
3-input functions	32	16	10	8
4-input functions	22	17	5	2
Total	90	51	15	10

Table 2.2 Hazards statistics of cells from Actel Act1 library

The table indicates that for many of the more complicated functions, the macros have hazards in them. Furthermore, for quite a few of the macros that have hazards, a hazard-free implementation can be found by disallowing some of the personalizations, possibly at the expense of routing efficiency.

Taking a more detailed look into the breakdown of specific macros, Table 2.3 shows a function-by-function listing of the personalizations and their hazards. An asterisk by the macro name represents all types of that gate; e.g., AND* represents all AND gates: 2-input ANDs, 3-input ANDs and 4-input ANDs.

Macro	Number of Personalizations	Number of Hazard-Free Personalizations	Number of Hazardous Personalizations
AND*, OR*, NAND*, NOR*	varies	all	0
BUF*, INV*	varies	all	0
XOR, XNOR, XA1, XA1A	varies	all	0
XO1	4	0	4
AO1	10	4	6
AO1A	8	3	5
AO1B	7	3	4

Table 2.3 Hazards in Actel cells

Here we can see that it is the more functionally complex macros that have hazards. The only macro that cannot be implemented by any of the existing hazard-free personalizations is XO1 (which implements the function $F = (A \oplus B) + C$).

The number of hazards present in the Actel realizations of various Boolean functions may suggest a modification to the technology mapping procedure presented the next chapter to take advantage of the *don't-care* transitions in the design. The results of this work also suggest that with appropriate modifications to Actel's personalizations which implement the various macros, the FPGA technology would be more suitable for technology mapping of asynchronous designs.

2.4 Summary

In this chapter, we introduced basic hazard terminology, classifying the types of hazards that exist and explaining how they can occur. We briefly examined the hazard-behavior of library elements constructed out of static CMOS logic, and showed how these networks can be abstracted by their Boolean factored form representation for the purposes of hazard analysis. We also examined the hazard-behavior of library elements constructed out of MUX-based FPGA cells, and presented the following contributions:

- A model suitable for hazard-analysis of MOS pass-transistor networks. This model was adapted from models presented in [72] and [71].
- Theorems on the existence of hazards in pass-transistor networks and application of these theorems to MUX-based design.
- Evaluation of the Actel Act1 MUX-based gate array library for hazards. Although we only analyzed the Actel Act1 FPGA, the results of the theory apply to other Actel families, such as the Act2 and Act3 FPGAs.

We use the terminology and results presented in this chapter as the basis for some of the work presented in the next chapter, where we present efficient algorithms for analysis of gate-level networks based on their BFF representation. We extend the terminology further when we discuss speed-independent circuits.

Chapter 3

Technology Mapping for Burst–Mode Asynchronous Designs

In this chapter, we look at the problem of technology mapping for asynchronous designs. In particular, we concentrate on the *generalized fundamental-mode* asynchronous design style [67], since we can easily separate the combinational portions of the design from the storage elements, as with synchronous design styles. First, we present some background information on fundamental-mode operation and hazards. In Section 3.2, we examine each step of algorithmic technology mapping for its influence on the hazard behavior of the modified network. We then present modifications to an existing synchronous technology mapper, CERES, to adapt it to work for generalized fundamental-mode designs. In Section 3.3, we present efficient algorithms for hazard analysis that are used by the modified technology mapper during the mapping process. Section 3.4 presents the results obtained by applying the technology mapper to some benchmark circuits. As joint work with researchers from HP Labs, the mapper was incorporated into the STETSON toolkit for asynchronous design and used to design an infrared communications chip [49]. Details are presented in Section 3.5. Finally, Section 3.6 concludes with a summary of contributions for this chapter.

3.1 Background

Before delving into the technology mapping problem, it is important to review the general design style that we are addressing. This section also reviews some terminology related to Boolean algebra and hazards.

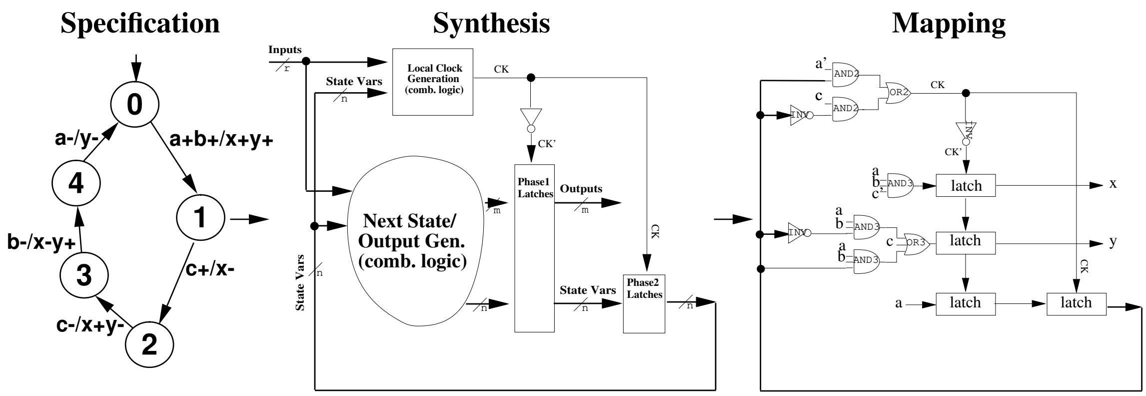


Figure 3.1: Burst-mode state machine and control synthesis tool flow

3.1.1 Design Style

We described the multi-input change fundamental-mode design style in some detail in Chapter 1, but we will review it here. Recall that for single-input change fundamental-mode designs the inputs must remain stable until the outputs and feedback variables have settled in response to the previous single-input change. Because the operation of such state machines are asynchronous, single-input change hazards in the combinational logic can cause incorrect circuit behavior and are thus to be avoided.

Several popular asynchronous design styles extend the single-input change fundamental-mode assumption [55] to allow multiple-input change bursts in specific states. The synthesis methods ([67], [89]) that incorporate this burst-mode—or g*eneralized fundamental-mode*—design style produce logic under the assumption that a burst of input changes can occur in any order and that the outputs and feedback variables of the combinational portion will settle before the next set of input changes are applied. As in the case of single-input change fundamental-mode operation, no hazards can be tolerated during the input bursts. In this design style both single-input change hazards and multi-input change hazards must be considered.

Figure 3.1 shows a simple burst-mode specification, along with the architecture to which it will be mapped by an automatic synthesis method [67]. The output of the automatic synthesis method is a set of technology-independent combinational logic equations. These equations, along with a set of latches, are then used to implement the state machine. The job of the technology mapper is then to implement the combinational portions of this design using, for example, parts from a library of standard cells, in such a way that no hazards are introduced during the mapping process, as can be seen from the example in the Figure.

This chapter addresses the problem of technology mapping for the generalized fundamental-mode design style. Because single-input change fundamental-mode is a subset of generalized fundamental-mode, the single-input change case is automatically handled if we take care of the more complicated burst-mode design style.

3.1.2 Hazards

For the technology mapping problem, we are interested in the hazards present in the combinational portion of both the mapped and unmapped network. We assume that the synthesis method has produced an initial design that is either free from sequential hazards or only has sequential hazards that are not exercised by any legal set of transitions. As a result, we can focus solely on combinational hazards during the mapping process. As we will show later, we can ignore function hazards during the technology-mapping process, allowing us to consider only the logic hazard behavior of subnetworks and library elements. For the generalized fundamental-mode asynchronous design style, it is important to consider all logic hazards, and thus we must make sure that new logic hazards are not introduced during the technology mapping operation.

3.2 Technology-Mapping Procedure

This section describes the approach taken by many recent synchronous algorithmic-based technology mapping programs. In it, we examine each step of the synchronous technology mapper for its effects on the hazard behavior of the transformed network. We then propose modifications to the synchronous technology mapper to allow correct mapping of asynchronous fundamental-mode networks.

The technology mapping step takes as input a technology-independent description of the logic to be implemented. The input to the mapping step typically comes from a logic optimization tool such as MIS [11] in the synchronous case, or an asynchronous logic optimizer [68] in the asynchronous case. Thus, the technology mapping step is primarily concerned with mapping the logic to a technology-specific library in an optimal way, and not with the logic optimization process itself.

To adapt a synchronous technology mapper to the generalized fundamental-mode asynchronous design style, each step in the technology mapping process must be examined and modified to ensure that the step does not introduce new hazards into the network.

3.2.1 Hazard Analysis of the Technology-Mapping Algorithms

As mentioned in Chapter 1, the approach to technology mapping taken by heuristic algorithmic technology mappers, such as DAGON, SIS, and CERES, divides the problem into three major steps: decomposition, partitioning, and matching/covering. First, the initial network, which is represented as a directed acyclic graph (DAG), is decomposed into a multi-level network composed of simple gates (e.g., 2-input AND/OR gates or 2-input NAND/NOR gates), whose corresponding representative functions are called *base functions*. Next, the circuit is partitioned into sets of single-output *cones* of logic, where a cone of logic represents a subnetwork of a partition of the network obtained by cutting the network at points of multi-fanout. The mapper then treats each cone independently. All possible matches to library elements are then found for subnetworks within each logic cone. Finally, an optimal set of matching library elements is selected from the set of matches to realize the network.

Thus, the basic procedure is as follows:

```
procedure tmap(network, library) {
    decomposed_network = tech_decomp(network);
    cones = partition(decomposed_network);
    foreach output in cones {
        find_best_cover(output, library);
    }
}
```

Throughout this chapter, we will use procedure `tmap` and *synchronous mapping procedure* interchangeably.

We now consider how this algorithmic framework (embodied in procedure `tmap`) applies to asynchronous networks.

3.2.1.1 Decomposition

The decomposition step transforms the network into an equivalent network composed of two-input, one-output base gates. This process can be performed by recursively applying DeMorgan's theorem and the associative law to the network. Both operations have been shown to be hazard-preserving for all logic hazards assuming an arbitrary wire-delay model [83]. Thus, the modified network composed of two-input, one-output gates has identical hazard behavior to that of the original network.

Note that within SIS's technology mapper, some simplification is also done during the decomposition step. This can introduce some static 1-hazards if redundant cubes are eliminated by the simplification algorithm. Therefore, we define a procedure, `async_tech_decomp`, that decomposes a circuit using only the associative and DeMorgan's laws. This procedure must be used by the asynchronous technology mapper during the decomposition step.

3.2.1.2 Partitioning

The partitioning step breaks the decomposed network at points of multiple fanout into single-output cones of logic. This heuristic simplification is required to convert a multi-output logic network into a collection of single-output cones of logic, reducing the complexity of the network covering problem ([41], [48]). Given that we start with a hazard-free network and preserve this behavior (within the partitions) in the covering step, the partitioning step does not change the network topology and thus does not alter the hazard behavior of the network.

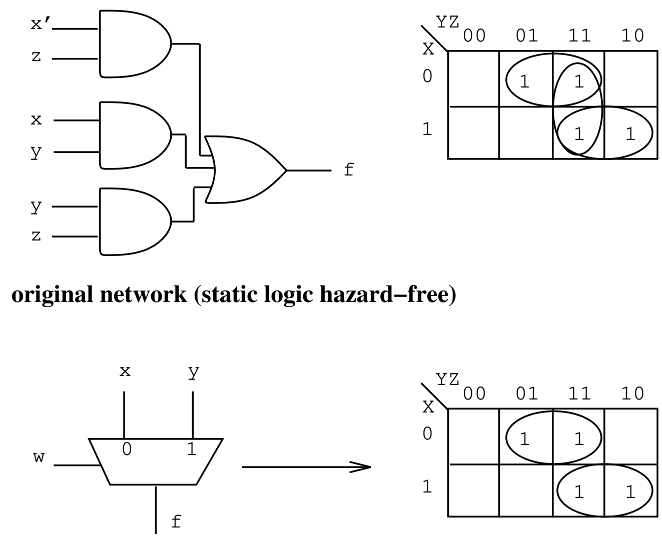
3.2.1.3 Matching and Covering

The matching and covering steps involve identifying equivalence between a subnetwork and a library element, and replacing that subnetwork with the equivalent library element. This step must not introduce any new hazards into the design.

Different algorithms are used for the matching and covering step. Keutzer [41] and Rudell [73] use tree pattern matching techniques for matching a library element to a portion of the circuit. SIS generates a complete set of patterns consisting of different decompositions of two-input, one-output gates for each library element. Standard pattern matching techniques are used to compare the library element with a portion of the network to be mapped. As long as the subnetworks represented by the patterns do not have hazards, these techniques work to preserve the hazard behavior of the circuit, provided that the initial circuit decomposition is hazard-preserving and that the library elements do not have hazards. However, if a hazardous library element is selected, then the hazard behavior of the subcircuit must be examined before the match is accepted.

Some mappers, such as CERES, use Boolean techniques to detect equivalence between a subnetwork of the circuit to be mapped and a library element. These techniques have been shown to give better results than tree matching [47]. However, because they decouple the structure of the subnetwork from the matching process, we cannot reason about the transformations of one subnetwork into the other. However, we can build on theorems presented by Unger in [83] to show that if we replace a portion of the circuit with an equivalent circuit with similar hazard behavior, the resulting circuit is still hazard-free for the transitions of interest.

Figure 3.2 shows a simple case in which the hazard behavior of the design must be taken into account to get a correct hazard-preserving cover. With Boolean matching, structural information is



original network (static logic hazard−free)

mapped network (static 1−hazard)

Figure 3.2: Illustration of logic static 1-hazard.

not taken into account during the mapping process. In this case, the better match from a synchronous point of view yields a cover that has more hazards than the original network.

Theorem 3.1 *Given a sum-of-products circuit that has a given hazard behavior, and a library that consists only of ANDs, ORs, NANDs, NORs, and INVs, if we map this circuit with the synchronous mapping procedure to a (possibly multilevel) circuit implemented with the given library elements, the resulting mapped network will have the same hazard behavior as the original network.*

Proof: The decomposition process makes use of DeMorgan's law and the associative laws to transform the initial SOP expression into a multilevel circuit composed of two-input one-output gates. These operations are hazard-preserving, as shown by Unger, so the resulting decomposed network has the same hazard behavior as the original expression. Since replacing any portion of the circuit with a single AND, OR, NAND or NOR gate is equivalent to applying the associative law and/or DeMorgan's law, these operations are also hazard-preserving. Additionally, mapping of the INV to a portion of the circuit cannot introduce new hazards. The covering process simply selects the best set of these mappings that covers the entire network, which is equivalent to successively replacing individual portions of the network with a library element, which we have just shown to be hazard-preserving. Therefore, the resulting mapped network has the same hazard behavior as the original network. ❑

We can extend this result to include other classes of library elements as well. However, the operations cannot be extended quite so simply when Boolean matching is used. In particular, with Boolean matching we have a problem with static 1-hazards: if a redundant cube is required to eliminate static hazards in the original design, then the matching must not eliminate that redundant cube. Additionally, we may have problems with multi-input-change (m.i.c.) dynamic hazards.

We must first restate Lemma 4.5 from Unger [83], before we can solve the problem of how to handle more general hazardous elements during the matching step.

Lemma 4.5 (Unger) *A transformation of a circuit C that consists of applying a hazard-preserving transformation to a subcircuit of C is itself hazard-preserving.*

This leads to the following key theorem:

Theorem 3.2 (Replacement Theorem) *Given a multilevel network with a given hazard behavior, replacement of a subcircuit C by an equivalent subcircuit C^* that contains a subset of the hazards present in C will not introduce new hazards into the network.*

Proof: If C^* has identical logic hazard behavior to C, then for any set of inputs that might cause a hazard in C, the same inputs will cause a hazard in C^* and will have the same effect on the overall network. Quite clearly then, if there is a hazard that is present in C and missing from C^*, and if the signal transition(s) that excited the hazard in C resulted in a hazard that was visible at the outputs of the network, then replacement of C by C^* will eliminate the hazard in the network, since C^* does not have the original hazard. Therefore, the resulting network has a subset of the hazards present in the original network, and thus the replacement does not introduce any new hazards. ❑

Even though transforming a network according to Theorem 3.2 may eliminate some hazards in the mapped network that were present in the unmapped network, the operation of the network.will not be adversely affected.

The following corollary is a direct result of Theorem 3.2.

Corollary 3.2.1 *Given a logic-hazard-free combinational network, replacement of a subnetwork by a logic-hazard-free equivalent subnetwork will result in a logic-hazard-free network.*

3.2.2 Modified Technology-Mapping Procedure for Generalized Fundamental-Mode Asynchronous Designs

Given the theorems in the previous section, if we start with a set of hazard-free library elements, the covering step and the resulting cover do not introduce new hazards. However, even if we have hazardous library elements, we may still want to use them, since they can potentially give us better matches. To use them, however, we need to make sure that the hazards they contain are a subset of the hazards in the portion of the network that is being matched.

As the starting point for the technology mapping procedure, let us assume that the initial design has no hazards for the transitions of interest. The problem of technology mapping, then, is to map the set of logic equations representing the design to an implementation composed of elements from a specific library, such that the implementation is hazard-free for the transitions of interest.

We can modify our synchronous technology mapping procedure to work for asynchronous fundamental-mode designs (both single- and multi-input change) as follows:

```
procedure bm_tmap(network, library)
    augment_library_with_hazard_info(library);
    decomposed_network = async_tech_decomp(network);
    cones = partition(decomposed_network);
    foreach output in cones {
        find_best_async_cover(output, library);
    }
}
```

The covering routine for `bm_tmap` has modifications in the matching routine as follows:

```
procedure async_matching_routine(subckt, library) {
    best_match = nil;
    if (matching_elements = find_matches(library, subckt)) {
        foreach match in matching_elements {
        if (has_hazards(match)) {                          /* library element has hazards */
            if (hazards(match) ⊆ hazards(subckt))
                accept_match(matching_elements, match);
            else
                reject_match(matching_elements, match);
        }
        else                                               /* library element is hazard-free */
            accept_match(matching_elements, match);
        }
        best_match = compute_best_match(matching_elements);
    }
    return best_match;
}
```

3.2.2.1 Representing the Structure of Library Elements

The functionality of each library element can be expressed in Boolean factored form (BFF), as was described in Chapter 2. We use the BFF expression as an accurate and convenient representation for both the functionality and structure of the particular library element. This BFF expression is then analyzed for logic hazards when the library is read into the mapper, and the logic hazard behavior of each library element is added as an annotation to the library element for later use during the matching phase.

3.2.2.2 Modification to the Matching Algorithm

We must now modify the matching algorithm to take the logic hazard information into account. If a hazardous library element is selected as a match for a particular subnetwork, then the subnetwork of interest must be examined to see if the same logic hazards exist. We are not, however, interested in the function hazard behavior of either the library element or the subnetwork. Because a function hazard is purely a property of the function itself and is thus independent of implementation, we know that the transitions produced by the burst-mode state-machine synthesis method must be function-hazard-free for the machine to work.

The procedure for doing the comparison is much easier than the initial hazard analysis of each library element because we already know which transitions are of interest. For each logic hazard in the library element, we must look at the subnetwork to see if the same logic hazard exists. As soon as we find a hazardous transition in the library element that is not in the subnetwork we can stop, because that library element cannot safely be used. At this point the library element is eliminated from consideration as a match for this subnetwork.

Table 3.1 summarizes the (logic) hazardous elements that are present in the libraries we examined for hazards in the previous chapter. For the standard cell libraries, only the multiplexors, which represent a small fraction of the library, contain hazards. So, for those libraries, most matching elements are logic-hazard-free and the normal synchronous algorithms can be used with negligible overhead. Recall that examination of a subset of the Actel Act1 library discovered hazards in many of the more complex functions, primarily in the AOI and OAI macros. (Many of these elements had several instances with different drive capability. For brevity, only one is shown.)

The remaining problems we must solve, then, are how to efficiently characterize the hazard behavior of the library elements, and how to easily determine whether a subnetwork has the same hazards. The next section addresses these problems.

	Hazardous Elements			
Library	**Types**	**Number**	**Total Elements**	**% Hazardous**
LSI9K	Muxes	12	86	14%
CMOS3	Muxes	1	30	3%
GDT	None	0	72	0%
Actel	AOIs, OAIs, Muxes	24	84	29%

Table 3.1 Libraries and their hazardous elements

3.3 Hazard Analysis Algorithms

This section describes the hazard analysis algorithms used by the modified technology mapping procedure to characterize both the hazard behavior of the library elements during initialization and the hazard behavior of the subnetwork during the matching and covering process. These analysis algorithms can also be extended to hazard-removal algorithms.

The analyses are complicated by the fact that we do not know the transitions of interest because the technology mapping step does not have any information about the circuit's environment. As a result, we will attempt to infer the transitions of interest from information that we already know about the circuit. In particular, we can extract the function hazard behavior from the circuit. Furthermore, the delay model assumed by burst-mode circuits implies that the order of a multi-input-change burst cannot be fixed, and as a result we know that the transitions of interest must happen over function-hazard-free transition cubes for the original circuit to be hazard-free. Therefore, our algorithms will take advantage of this by limiting the search for hazards to those cubes that are function-hazard free. This significantly improves the efficiency of the hazard analysis procedures and the tractability of performing the analysis.

Unless otherwise noted, we present the algorithms for m.i.c. logic hazards, because these algorithms will also identify the s.i.c. hazards.

3.3.1 Static Logic Hazard Analysis of Combinational Logic

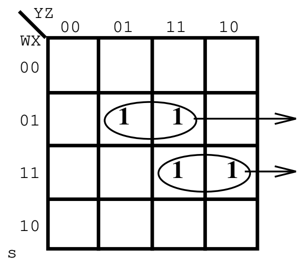
3.3.1.1 Static Logic 1-Hazard Analysis

From Theorem 4.3 of Unger [83], we know that a multi-level expression can be transformed into a sum-of-products expression in a static hazard-preserving manner using the associative, distributive and DeMorgan laws. Therefore, without loss of generality, we will assume that this has been done and we will work with the two-level SOP form of the expression for static 1-hazard analysis.

Any missing prime implicants in the implementation of a given function uniquely identify the static logic 1-hazards present in the circuit. Since our goal is not to generate all prime implicants of the function, but to identify the static logic 1-hazards, we would like to come up with an efficient procedure that doesn't involve prime implicant generation, since generating all prime implicants can be quite expensive. If all cubes in the network are prime, we can simply identify the cube adjacencies that are not covered[1]. If we encounter a cube that is not prime, we then expand the cube into a prime and add it to the list of cubes to be checked by the adjacency checking algorithm, after having flagged the hazard. The algorithm works as follows:

```
procedure static_1_analysis(multilevelExpr)
    SOPexpr = xformTwolevel(multilevelExpr);
    foreach cube in SOPexpr {
                    /* Any uncovered non-primes represent hazards; look at those first */
        if (not prime(cube)) {
            if (prime(cube) not in SOPexpr) {
                addToHazards(primeCube, static1Hazards);
            }
            primeCube = replace_with_prime(cube, SOPexpr);
        }
    }
                                            /* Generate all cube adjacencies */
    foreach cube1, cube2 in SOPexpr {
        if (numAdjVars(cube1,cube2) == 1) {
            generateAdjCubes(cube1,cube2,cubeAdjacencies);
        }
    }
                              /* If the adjacency isn't covered, we have a hazard */
    foreach cube in cubeAdjacencies {
        if (not cubeContainedInExpr(cube, SOPexpr)) {
            addToHazards(cube, static1Hazards);
        }
    }
    return static1Hazards;
}
```

1. Two cubes are said to be *adjacent* if their Hamming distance is 1. In other words, their binary representations differ in exactly one coordinate.



f = w'xz + wxy

USED[w'xz] =1101
PHASE[w'xz]=0111

USED[wxy] =1110
PHASE[wxy] =1111

CONFLICTS[w'xz,wxy]=([1101]&[1110])&([0111]XOR[1111])
=([1100])&([1000])
=[1000]

USED[adjacency]=(USED[w'xz]|USED[wxy])&~CONFLICTS[w'xz,wxy]
=([1101]|[1110])&[0111]
=([0111])
PHASE[adjacency]=(PHASE[w'xz]|PHASE[wxy])&~CONFLICTS[w'xz,wxy]
=([1111])&[0111]
=([0111])
adjacency = xyz

Figure 3.3: Detection of static 1-hazards.

If we only want to test for s.i.c. static logic 1-hazards, the problem is simpler—we need only check that each cube adjacency is covered by some cube in the expression.

Implementation. The data structure we use to represent the logic equation and its cubes is similar to the meta-product structure used in [24]. Two bit-vectors (USED and PHASE) are used to represent each cube, with each bit position in the bit-vector corresponding to a unique variable. For a given cube, if a bit is set in the USED vector, then its corresponding variable appears in the cube. Similarly, for the same cube, for each variable that appears in the vector, the corresponding bit in the PHASE vector is set if the variable appears in its uncomplemented form in the cube. If the corresponding bit is not set, then the variable appears in its complemented phase. Figure 3.3 illustrates the mapping of a function to this data structure.

The complexity of generating the list of cube adjacencies is $O(n^2)$ in the number of cubes in the expression. For every pair of cubes in the equation, the cube adjacency is generated by simple bit operations. Obviously, only a small subset of the cube pairs will be found to be adjacent.

Two cubes are adjacent if and only if a single bit is set in the expression:

$$CONFLICTS = (CUBE1_{USED} \wedge CUBE2_{USED}) \wedge (CUBE1_{PHASE} \oplus CUBE2_{PHASE})$$

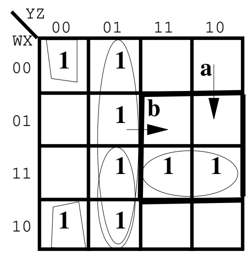
We can decompose this expression to gain some insight. For two cubes to be adjacent, they must share exactly one variable that differs in its phase in the two cubes. So the first part of the expression indicates that variables are present in both cubes. If the two cubes share no variables, then they are not adjacent, so there cannot be a static hazard as a result of those two cubes. The resulting bit vector, then, represents the variable(s) that the two cubes have in common. The second part of the expression represents the variables that differ in phase in the two cubes. The CONFLICTS bit vector will thus have a bit set for each variable that is present in both cubes, but differs in phase between the two. If there is more than one variable that is present in both in differing phases, then the two cubes are not adjacent. If there is only a single bit set in the vector, then the two cubes are adjacent. Figure 3.3 illustrates the formation of the CONFLICTS vector from the cubes of a function.

Once two cubes have been found to be adjacent, the adjacency is easily generated by generating the OR of the two cubes while masking out the literal that expresses the adjacency. This is done for both the USED and PHASE vectors, as can be seen in the figure.

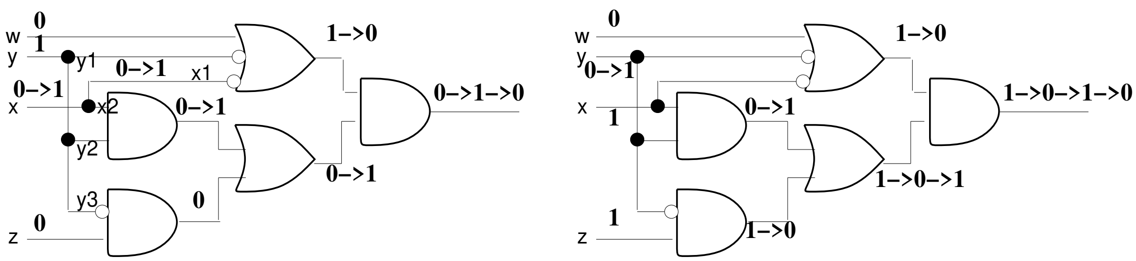
The result of the cube adjacency generation routine is a list of adjacent cubes. Typically, the list of adjacent cubes is much smaller than the $n(n-1)/2$ cube pairs.

3.3.1.2 Static Logic 0-Hazard Analysis

Static 0-hazards are present when both a product term in the sum-of-products form of an expression contains a vacuous term (i.e. a term that contains a variable and its complement, such as $xx'y$, and thus contributes nothing to the expression in the steady-state), and the circuit can be sensitized to view that term at its outputs. The example shown in Figure 3.4a (taken from [55] page 91) shows a circuit where a static 0-hazard is present when $w=0$, $y=1$, $z=0$, and x is changing.



f = (w + y' + x')(xy + y'z)



a) Static 0–Hazard
[0010]–>[0110]

b) s.i.c. Dynamic Logic–Hazard
[0101]–>[0111]

Figure 3.4: Static 0-hazards and s.i.c. dynamic hazards.

The detection of static 0-hazards simply requires that the different paths through the circuit be distinguished so as to mark vacuous terms in the SOP form of the expression. These vacuous terms typically represent the reconvergence of a variable and its complement in a multilevel network, which can lead to hazards caused by different delays through the different paths. The detection procedure is a subset of the detection procedure for single-input change dynamic hazards, which is described in the next section.

3.3.2 Dynamic Logic Hazard Analysis of Combinational Logic

To properly analyze the dynamic logic hazard-behavior of a given multilevel network, it is necessary to uniquely identify the individual paths each signal takes. Past work has used ternary simulation to identify dynamic hazards for specific transitions [32]. However, we are interested in characterizing the dynamic logic hazard behavior of an entire subnetwork or expression, rather than the dynamic logic hazard behavior of a specific input burst. Applying ternary simulation to a network to characterize its dynamic hazard behavior is exponential in the number of input variables, and is thus impractical in the general case. Therefore, we have formulated more efficient hazard analysis techniques for use during the technology mapping process, because hazard analysis is performed frequently during the course of operation.

The remaining subsections focus on the dynamic hazard analysis techniques, starting with the more complicated multi-input change dynamic logic hazard detection procedure, and then describing a simpler procedure for single-input change dynamic logic hazard detection.

3.3.2.1 Multi-Input Change Dynamic Logic Hazard Analysis of Two-Level Networks

Before analyzing the dynamic logic hazard behavior of a network, we review a few definitions.

Recall the definition of a dynamic hazard (reworded to fit the purposes of this section):

Definition 3.1 *Let* α, $\beta \in \{0,1\}^n$ *be points in the input space such that* $f(\alpha) = 0$, *and* $f(\beta) = 1$. *Then, a* dynamic hazard *exists if during an input burst which results in a transition between* α *and* β *the output may go through a* $0 \rightarrow 1 \rightarrow 0$ *transition before settling at 1.*

We must further refine this definition to distinguish between dynamic *function* hazards and dynamic *logic* hazards, as we explained in Chapter 2. Dynamic function hazards are purely a property of the function itself and thus are independent of implementation. We do not need to take dynamic function hazards into account, since, for any acceptable match, both the subcircuit to be mapped and any matching library element have the same function hazards. However, because dynamic logic hazards are a property of the implementation, we must characterize them for all library elements, and for any subnetwork that is matched by a hazardous library element to compare the hazard behaviors.

Let us recall the definition of *transition cube* from the previous chapter:

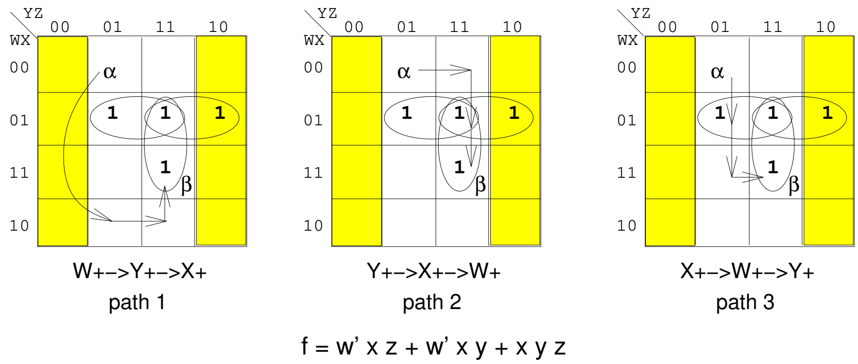


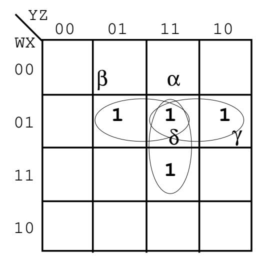
Figure 3.5: Hazards in transition cubes.

Definition 2.1 *A* transition cube*, T[α, β], is the smallest Boolean subspace that contains α and β, where α is the minterm corresponding to the starting point, and β is the minterm corresponding to the endpoint. The transition cube defined by T[α, β] includes all minterms that can be reached in a (possibly multi-input change) monotonic transition(s) from α to β.*

Example 3.1

In Figure 3.5, the transition cube T[α, β] is shown in the non-shaded areas of the Karnaugh maps. Within that transition cube, the input variables may change in any order, thus tracing a path between α and β. For example, variables can change in the order W↑ ⟶ Y↑ ⟶ X↑ tracing path 1 as shown. Or, the variables could change in the order Y↑ ⟶ X↑ ⟶ W↑, tracing path 2, depending upon the delays in the network. The first path exercises no hazards, whereas the second path exercises a dynamic logic hazard. A third path, represented by the transitions X↑⟶ W↑⟶ Y↑ excites a dynamic function hazard. Note that this transition cube is not function hazard-free, a property which will later be important for our dynamic logic hazard detection algorithm. ❑

Given these definitions, we can present the necessary conditions for existence of a dynamic logic hazard in a two-level SOP circuit (similar conditions were also presented in [6], [13], and [32]):



f = w' x z + w' x y + x y z

Figure 3.6: Dynamic hazards within a transition cube.

Theorem 3.3 *Given a two-level sum-of-products representation for a circuit, an implementation of a function f has a dynamic logic hazard for a given monotonic transition* $\alpha \rightarrow \beta$*, where* $\alpha, \beta \in \{0,1\}^n$*, and* $f(\alpha) = 0$ *and* $f(\beta) = 1$ *if and only if*

1. *There is no function hazard over the transition cube* T[α, β].
2. *There exists a cube* $c \in f$ *that intersects* T[α, β] *but does not contain* β.

The dual of this theorem applies for 1→0 transitions.

Before proving this theorem, let us look at a simple example that illustrates the conditions.

Example 3.2

Looking at the transition cube T[β, γ] in the Karnaugh map in Figure 3.6, we can see that if the order of the transitions is X↑→Z↓→Y↑, then the output will make a 0→1→0→1 transition independent of the implementation of the function, and that therefore a function hazard exists for that transition. The existence of a dynamic function hazard within that transition cube implies that some combination of gate delays can cause that function hazard to be exercised during the input burst, independently of the implementation.

Condition 2 can be seen by examining the transition cube T[α,γ] in the figure. If the inputs change in the order X↑→Z↓, then there can be some set of gate delays such that cubes *w′xz* and *xyz* turn on and off before cube *w′xy* turns on, thus yielding a hazardous transition on the output of $0 \longrightarrow 1 \longrightarrow 0 \longrightarrow 1$. (This particular hazard can only be eliminated by implementing the function with a single gate.) Note that if we pick the transition cube $T[\beta, \delta]$, then by Condition 2, a dynamic logic hazard does not exist, since there is no cube which intersects $T[\beta,\delta]$ that does not also intersect δ. More intuitively, the first gate which turns on during the transition will hold the output high while the rest of the gates are settling. ❑

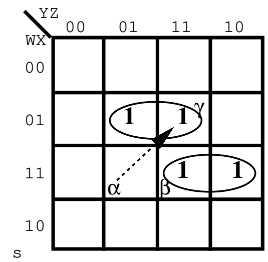
Proof of Theorem 3.3: Condition 1 follows from the definition of a dynamic logic hazard.

We only prove the theorem for the case where $\alpha = 0$ and $\beta = 1$. The opposite case follows by symmetry.

Condition 2: $\Rightarrow$ If a dynamic logic hazard exists, then the output must make a transition from $0 \longrightarrow 1 \longrightarrow 0 \longrightarrow 1$ during the transition between α and β. For this to happen, at least one cube must make the transition from $0 \longrightarrow 1 \longrightarrow 0$, and another cube must make the transition from $0 \longrightarrow 1$. Since the first cube has value 0 at the endpoint (β), it clearly cannot intersect β. And since that same cube makes a transition to 1 during the change, it must intersect $T[\alpha,\beta]$. Therefore condition 2 is satisfied.

$\Leftarrow$ If there is a cube that intersects the transition cube but does not intersect β, then there exists a path starting at β and ending somewhere at the border of the subspace where the intersecting cube makes a $1 \longrightarrow 0 \longrightarrow 1$ transition. This implies that a $1 \longrightarrow 0 \longrightarrow 1 \longrightarrow 0$ transition will occur during the transition between α and β. This transition will be visible at the output of the circuit if other gates are sufficiently slow. Therefore a dynamic logic hazard exists. ❑

Let Υ denote the set of transition cubes for a given function *f*. Let $\Upsilon_{min} \subseteq \Upsilon$ denote the set of minimal *function hazard-free* transition cubes for *f*, where a minimal transition cube is one that is not properly contained by any other transition cube in Υ for some $\alpha,\beta \in \{0,1\}^n$. We claim that we can detect dynamic hazards by focusing only on Υ_{min}. The following is an outline of the basic steps:



f = w'xz + wxy

Figure 3.7: M.i.c. hazard that is the result of a static 1-hazard.

1. Form the set of *minimal function hazard-free* transition cubes, Υ_{min}.
2. For each transition cube $T_{min}[\alpha,\beta] \in \Upsilon_{min}$, find the cubes in f that intersect the transition cube.
3. If any $T_{min}[\alpha,\beta] \in \Upsilon_{min}$ intersects a cube $c \in f$ that does not also intersect the transition cube at β, then a dynamic logic hazard exists for the transition defined by that transition cube.

Having observed that any dynamic logic hazard that results from a static logic 1-hazard is fully characterized by the static logic 1-hazard, we can just look for those dynamic logic hazards that are a result of intersecting cubes (i.e., we can ignore adjacent cubes). The next example illustrates this condition.

Example 3.3

In Figure 3.7, there is a multi-input change dynamic hazard when traversing from point α to point γ via point β. In that case, the gate corresponding to cube *wxy* can turn on and off before the output is eventually held high by the gate corresponding to cube *w′xz*. However, this dynamic logic hazard is fully characterized by the static logic 1-hazard which exists in the transition between *wxyz* and *w′xyz*. ❑

For step 1 of our procedure, rather than trying to restrict the Boolean function space to transition cubes that are function hazard-free, we can start with a given cube intersection and form the minimal function hazard-free transition cube directly from the cube intersection, identifying all dynamic hazards (if any) for that cube intersection. This yields a much more efficient procedure.

This leads to the following, more efficient procedure:

```
procedure findMicDynHaz2level(f) {
    hazards = ∅;
    I = {irredundant cube intersections of f};
    for all c ∈ I {                      /* complement one variable at a time in c */
        J_c = {cubes adjacent to c};
        α_c = ∅; β_c = ∅;
        for all d ∈ J_c {          /* Look at each cube adjacent to the cube intersection. */
            if (f(d) == 0)            /* If the value of the function at this point is 0, */
                α_c = α_c ∪ d                 /* then the cube belongs to the set α_c, */
            else
                β_c = β_c ∪ d                  /* otherwise it belongs to the set β_c. */
            }
                                            /* hazards are defined by all possible
                                               pairs of minterms from α_c and β_c */
        hazards = hazards ∪ {T[i,j] | i,j ∈ α_c × β_c};
    }
}
```

In the procedure above, α_c is the set of points adjacent to the cube intersection where the function is 0, and β_c is the set of points adjacent to the cube intersection where the function is 1. These points become the start and end points (respectively) of the minimal function-hazard free transition cubes. As we will prove shortly, if both α_c and β_c are non-null, then the function has at least one dynamic hazard, and those hazards are defined by the transition cubes determined by all possible pairings of elements from α_c with elements from β_c.

The procedure thus finds the remaining dynamic logic hazards that are necessary for complete characterization of the logic hazard behavior of the subnetwork. Note that it is sufficient to look for dynamic hazards only in the $0 \longrightarrow 1$ transitions, since they exist if and only if the reverse $1 \longrightarrow 0$ transitions also have them. We illustrate the procedure by the following example.

Example 3.4

In Figure 3.8, the only irredundant cube intersection is $c = w'xyz$. Taking the complement of each *care* variable in this cube we get the set $J_c = \{wxyz, w'x'yz, w'xy'z, w'xyz'\}$. Examining

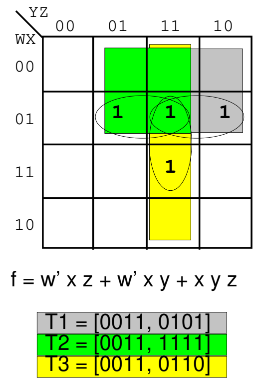


Figure 3.8: Illustration of procedure `findMicDynHaz2level`.

the value of f at each of these locations, we can see that we have one minterm in α_c ($w'x'yz$), and three minterms in β_c ($w'xy'z$, $wxyz$, and $w'xyz'$). The minimal function hazard-free transition cubes are then as shown by the shaded areas. Inspection reveals that each of these transition cubes has a dynamic logic hazard. ❑

We now must show that the set of dynamic logic hazards resulting from this procedure represents all possible dynamic logic hazards that are not the result of a static logic 1-hazard, and that the transition cubes generated by the procedure are function hazard-free. (Note that this second part is not as crucial, because although extra work would be required in the matching step if the procedure produced transition cubes with function hazards, the results would still be correct, since equivalent functions have the same function hazards.)

Theorem 3.4 *Procedure* `findMicDynHaz2level` *finds all possible m.i.c. dynamic logic hazards which are not the result of a static 1-hazard for the function f.*

Proof: Two different cubes must intersect the transition cube for a m.i.c. dynamic logic hazard to exist (Theorem 3.3). There are two ways that two cubes can intersect a transition cube and not introduce function hazards: two cubes can be adjacent, or they can intersect. In the case where the cubes are adjacent, either the adjacency represents a static logic hazard, which has already been characterized by other algorithms, or the adjacency is covered by another cube that intersects both cubes, in which case they are characterized by the procedure. If two cubes overlap, then the procedure will select them, so we will not miss a possible dynamic logic hazard. Picking minterms adjacent to the intersection ensures that we will satisfy condition 2 of Theorem 3.3—that is, the point β will not intersect the cube intersection. Picking points beyond the adjacencies of the cube intersections as possible candidates for α or β may result in selection of a Boolean space that contains function hazards. More importantly, these spaces contain any transition cube, and thus any dynamic hazards, which would be identified by our procedure. ❑

3.3.2.2 Multi-Input Change Dynamic Logic Hazard Analysis of Multi-Level Networks

In many cases the library elements and the subnetworks we select during the matching process will be multilevel. (See Figure 2.6 for an example in which this is a problem.) In this case, we can use the two-level procedure as a filtering process to narrow down the sets of transitions we need to examine for dynamic logic hazards in the multi-level expression, leading to the procedure below:

procedure `findMicDynHazMultiLevel`:

1. Transform the network into a two-level SOP expression by applying static hazard-preserving transformations.
2. Perform algorithm `findMicDynHaz2level` on the expression generated by step 1.
3. Examine the original multi-level network for dynamic logic hazards on those transitions produced by step 2. Throw away any hazards that aren't found in the multi-level network.

For step 3, we can label the paths as we do for single-input change dynamic hazards and use the transformed two-level expression together with the information we have on the potentially hazardous transitions to identify the real dynamic hazards. We can then eliminate from consideration any hazards that are found to be false hazards. Alternatively, ternary simulation can be used on the specific transitions to eliminate any false hazards.

3.3.2.3 Single-Input Change Dynamic Logic Hazard Analysis

In the previous subsection we ignored any dynamic logic hazards which are the result of vacuous terms in the two-level expression, as illustrated in Figure 3.4. We must now address this case.

Given a multi-level network, the network can be transformed into a sum-of-products expression suitable for dynamic hazard analysis by first relabeling the variables so that each distinct path the variable takes is identified, and then transforming the expression into an SOP form through hazard-preserving operations. A single-input change dynamic logic hazard will be present whenever a variable appears within a product term in both its complemented and uncomplemented forms, and the remaining variables in that product term are constant logic "1," while another product term containing that variable changes from 0 to 1 [83].

Figure 3.4b shows that there is a dynamic hazard when $w=0$, $x=1$, $z=1$ and y is changing. The transformed expression is $f = wx_2y_2 + wy_3'z + y_1'x_2y_2 + y_1'y_3'z + x_1'x_2y_2 + x_1'y_3'z$. With $w=0$, $x=z=1$ the expression reduces to $f = y_1'y_2 + y_1'y_3'$. With y changing, the first term in the reduced expression will make a $0 \rightarrow 1 \rightarrow 0$ transition, while the second term will make a monotonic change, which can result in a dynamic hazard.

The algorithm for detecting the presence of the hazard is straightforward. It involves transforming the literals in the expression to keep track of different paths for each literal, transforming the new expression to a two-level SOP form, and then statically analyzing the expression with simple bit operations to see if the conditions for a single-input change dynamic logic hazard apply. Once a variable is identified as a candidate for exciting a dynamic hazard, the rest of the expression can easily be examined to determine whether the dynamic hazard exists. As a by-product of the transformation into two-level SOP form, static 0-hazards are automatically identified by identifying those p-terms that contain pairs of complementary literals.

3.4 Results

The hazard analysis routines described in the previous section were incorporated into the CERES technology mapping program, and the modifications to the matching algorithms were implemented as described in the previous section.

The technology library is analyzed and annotated with hazard information only when the library is initially read in. Hazards in the subnetwork to be mapped are analyzed only when a hazardous library element is selected as a potential match.

Table 3.2 compares the run times of the library initialization for both the synchronous mapper and the asynchronous mapper, where the asynchronous mapper must also analyze the library elements for hazards during initialization. The mappers were run on three ASIC libraries and one custom library. The hazard analysis routines did not add appreciable run-time overhead for most libraries. The GDT library took longer to analyze mainly because of the complexity of its library elements. Even so, the time it takes to analyze a library is not significant, because the library analysis only needs to be done once, since the information can be stored permanently in the library.

Library	Sync	Async	# Elements
LSI	.6 sec	1.2 sec	86
Actel	.6 sec	1.1 sec	94
CMOS3	.2 sec	.4 sec	28
GDT	.6 sec	16.7 sec	72

Table 3.2 Hazard analysis run times for various libraries.
All times are user times reported in seconds.
All benchmarks were run on a DEC 5000.

In addition to some standard benchmark circuits, two asynchronous controllers taken from complete chip designs were mapped with the asynchronous mapper, and their results compare favorably against hand-mapped implementations (where available), as can be seen in Table 3.3. An asynchronous implementation of a SCSI controller was synthesized using the locally-clocked synthesis method [69] and then mapped with the asynchronous mapper. Since the logic equations were never mapped by hand, the hand-mapped entry is empty. The ABCS design is a part of the control logic for an asynchronous infrared communications chip currently under development at Stanford in collaboration with Hewlett-Packard. It was mapped from logic equations synthesized with the 3D synthesis method [89]. The automatically mapped version is approximately 13% smaller than the hand-mapped version, where in the table, each transistor in the pulldown network of a cell is assigned an area cost of 1. (The hand-mapped results do not include the cost for buffers which is included in the automatically mapped results.)

Design	Library	How Mapped	Cost (area)	Time (sec)
SCSI	LSI	async tmap	168	28.1
ABCS	GDT	hand-mapped	312	—
		async tmap	272	28.1

Table 3.3 A comparison of automatically-mapped and hand-mapped designs in terms of area (obtained by running CERES with option depth=5). Benchmarks were run on a DEC 5000.

Run times of both the synchronous and asynchronous mapping procedures are shown in Table 3.4 for mapping the two designs to four different libraries. The asynchronous mapper took from 50%-60% longer than the synchronous mapper in most cases. The overhead is very dependent upon the number of hazardous elements present in the library.

Design	Mapper	Library			
		Actel	LSI	CMOS3	GDT
SCSI	Synchronous	14.4	17.8	14.0	31.7
	Asynchronous	22.9	28.1	20.7	44.2
ABCS	Synchronous	6.3	8.7	5.7	22.9
	Asynchronous	10.2	13.5	9.0	28.1

Table 3.4 Comparison between the run-times of the synchronous and asynchronous mappers (depth=5).

Table 3.5 shows mapping results and mapping runtimes for some asynchronous benchmark circuits for two libraries. The hazard analysis of each library typically took a only fraction of a second. The area costs are relative to the particular library and will not compare between libraries.

Design	**Library**					
	CMOS3			**LSI9K**		
	CPU	**Delay**	**Area**	**CPU**	**Delay**	**Area**
chu-ad-opt	.6s	24ns	152	3.7s	2.1ns	9
dme-fast-opt	.8s	11.8ns	232	3.8s	1.7ns	13
dme-fast	.6s	11ns	136	3.7s	1.7ns	9
dme-opt	.7s	22.4ns	184	3.9s	2ns	10
dme	.6s	17.8ns	168	3.7s	1.9ns	10
oscsi-ctrl	10.6s	96.5s	3552	16.1s	7.3ns	172
pe-send-ifc	2.3s	47.2ns	864	5.3s	3.5ns	40
vanbek-opt	.6s	19.5ns	144	3.7s	2.6ns	9
dean-ctrl	33.6s	126ns	11320	43.5s	10.3ns	565
scsi	20.7s	95ns	6888	27.9s	7.2ns	330
abcs	9s	74.7ns	3288	13.3s	6.8ns	168

Table 3.5 Mapping results for the asynchronous mapper run on various examples (depth=5). Benchmarks were run on a DEC 5000/240.

3.5 Design Example: An Infrared Communications Receiver

The mapper was integrated into the STETSON toolkit for asynchronous design which represents joint work done with Alan Marshall and Bill Coates of Hewlett-Packard Laboratories. The toolkit was then used to design an asynchronous infrared communications IC which was fabricated, tested and worked first silicon. This section briefly discusses both the toolkit and the chip design. More details can be found in [49]. My contributions were primarily in the development of the design methodology and the toolkit, and not in the design of the actual chip.

3.5.1 STETSON Toolkit

The overall design methodology starts with a behavioral description of the design in VHDL. The design is partitioned manually into control and datapath components, much as in synchronous design. Structural VHDL is used to connect the high-level components together. The high-level tool flow is illustrated in Figure 3.9.

Once the partitioning into control and datapath blocks has been done, the designer creates behavioral asynchronous finite-state-machine (AFSM) descriptions for the control portions of the design. These descriptions are written in a simple language, and from them both standard-cell layouts and behavioral VHDL models can be automatically synthesized by the tools.

The datapath components are described initially as behavioral VHDL models, and then are manually refined into gate-level implementations composed of standard-cell components. The gate-level VHDL for each datapath component is then converted automatically to a standard-cell layout by the tools.

The design is simulated in VHDL at each step of refinement, to check both the control synthesis and the incremental refinement of the datapath. Once all components of the design have been synthesized and simulated, a block-level router is used to connect the standard-cell blocks together and to route external signals to the pads. A transistor-level netlist of the design is then extracted from the layout data and simulated, the results being compared against the results from the high-level VHDL simulation.

3.5.2 Behavioral Modelling Issues, Simulation and Timing Analysis

As mentioned earlier, the burst-mode asynchronous finite state machines which we have used in this design [89] are specified to accept only certain inputs when in a given state. If a burst-mode state machine is presented with an input event which is not specified for its current state, the behavior is undefined. To aid the designer in detecting such errors, the behavioral model must detect the occurrence of any unspecified input event, and either report the event or set the state machine's internal state and outputs to a specific "unknown" state. We found that behavioral

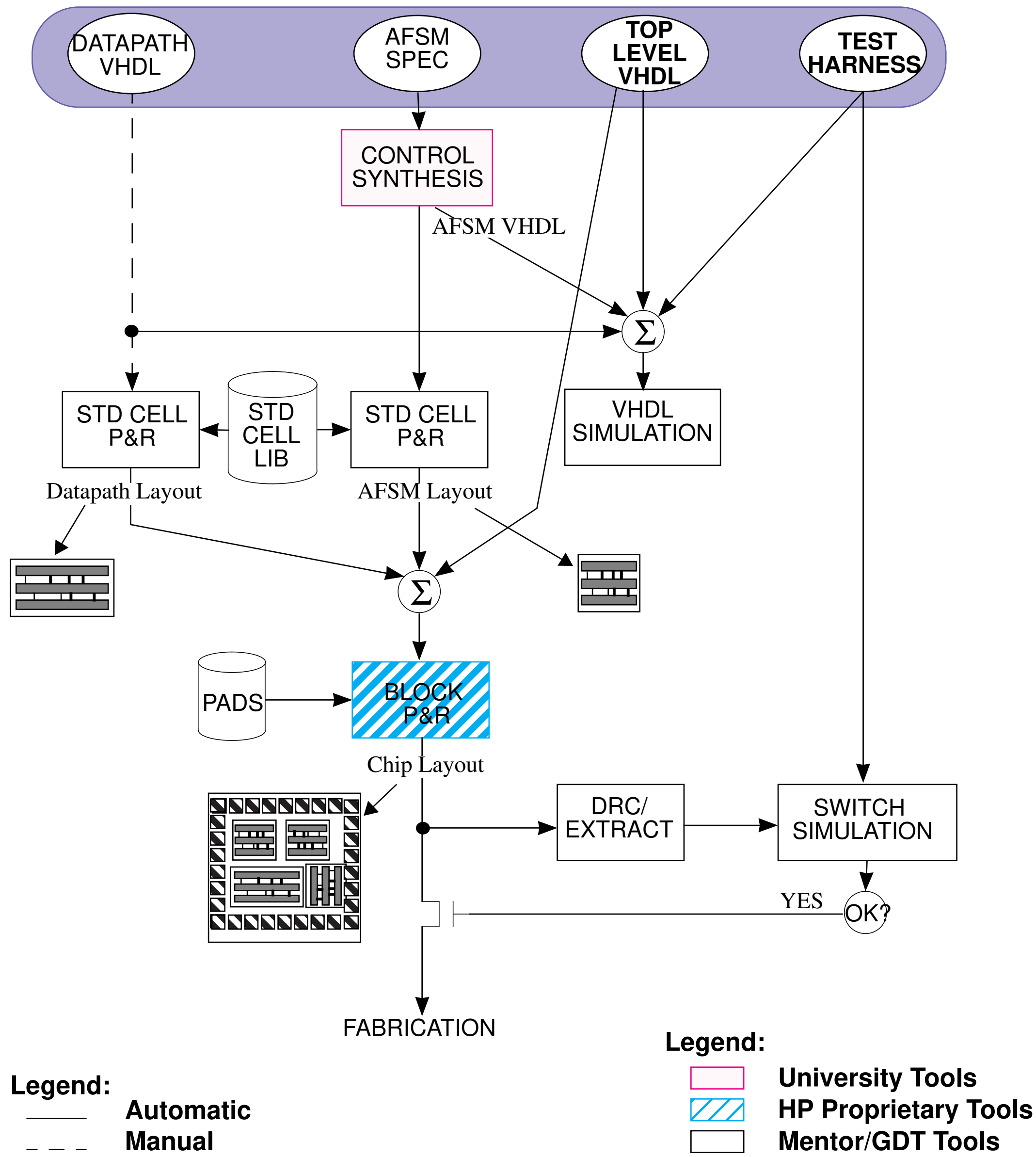


Figure 3.9: Overall design tool flow

VHDL models of burst-mode state machines which met these requirements could be very lengthy—over 1600 lines of VHDL for one of our state machines. Because of this, we wrote a tool to automatically generate these VHDL models from the AFSM descriptions.

Because timing information is not available until the design has been implemented, estimated timing is used in the behavioral VHDL descriptions. Once the design is fully implemented, accurate timing information is extracted from the layout and used in a switch-level simulation of the complete design. The lack of accurate timing information in the early design stages meant that some timing problems were revealed by switch-level simulation that weren't evident in the behavioral simulations. Because our design path is highly automated, these problems were easily corrected by changing the VHDL and state machine descriptions, and then re-synthesizing the design.

Since our toolset lacks a timing analysis tool and our state machine synthesis tools do not give us any control over the timing specifications of the state machines, we had to check whether the burst-mode timing constraints were satisfied through manual analysis of data from the switch-level simulation. We found a few violations where relatively slow state machines interacted with faster datapath components or state machines, and we added delays (inverter chains) manually in a few places to ensure safe timing margins.

3.5.3 Datapath Generation

For our design, we generated the datapath components through successive simulation and manual refinement of VHDL from the behavioral down to the gate-level. We could have used existing commercial datapath generation tools to generate a gate-level datapath description from the behavior, but the cost of library customization and software would have been more than the cost of the labor for manual refinement of this one design. However, automatic datapath generation software can be incorporated into the toolset easily, and should be for future designs.

Once a datapath component has been described down to the gate level, its structural VHDL is converted automatically to the format used by the Mentor GDT AUTOCELLS standard-cell place-and-route tool, and the datapath component is implemented as a standard-cell block. This block is then

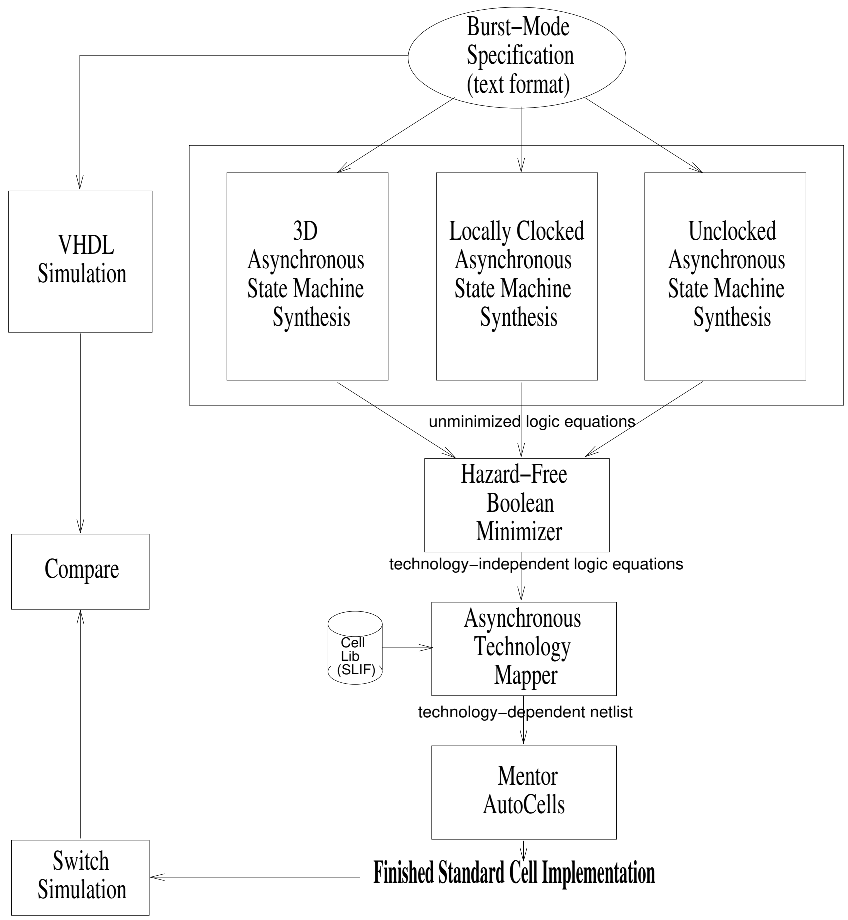


Figure 3.10: Control synthesis tool flow

connected into the circuit by an HP proprietary block place-and-route (BPR) tool (although any BPR tool could be used).

3.5.4 Control Synthesis

The control synthesis process is by far the most interesting part of the tool flow, and will be described in some detail. The control synthesis path uses a combination of university tools, custom scripts, and commercial tools to take a high-level AFSM specification and generate a standard-cell block implementing the state machine. The resulting implementation is then connected to the rest of the circuit with the block place-and-route (BPR) tool.

Figure 3.10 outlines the overall flow of the control synthesis process. The control portion of the design is entered as a simple textual description of the state machine. This description is then translated automatically into VHDL for simulation. The description can also be fed into one of three state machine synthesis programs, each of which synthesizes an AFSM in a specific implementation style. The output of the state machine synthesis step is a set of hazard-free equations which implement the specified state machine behavior. The equations are then minimized using a hazard-free asynchronous logic minimizer and the resulting equations can be mapped automatically to a standard-cell or gate-array implementation using the asynchronous technology mapper described in this chapter. Each state machine is implemented as a separate standard-cell block, to maintain close control over the wire lengths and hence the delays of the internal signals.

3.5.4.1 State Machine Specification

A burst-mode AFSM is specified by a state diagram consisting of a finite number of states, joined by directed arcs representing state transitions. Each arc is labeled with a set of input signal transitions, called an *input burst*, and a (possibly empty) accompanying set of output transitions, called an *output burst*. In a given state, once all input transitions in an input burst have occurred, the state machine moves to the new state and generates the corresponding output burst. Figure 3.11 shows an example of a burst-mode state machine specification.

There are several restrictions on the AFSM specification which must be taken into account during the design process:

1. To ensure deterministic behavior, any two arcs leaving a single state must be mutually exclusive.
2. Any input or output transitions not explicitly specified are forbidden.
3. A given state must always be entered with the same combination of input and output values.

Some constraints, such as restriction 3, are checked automatically, while others impose requirements on the external environment. In some cases, restriction 1 requires that special asynchronous blocks such as mutual exclusion elements or arbiters be used at the inputs to the AFSM.

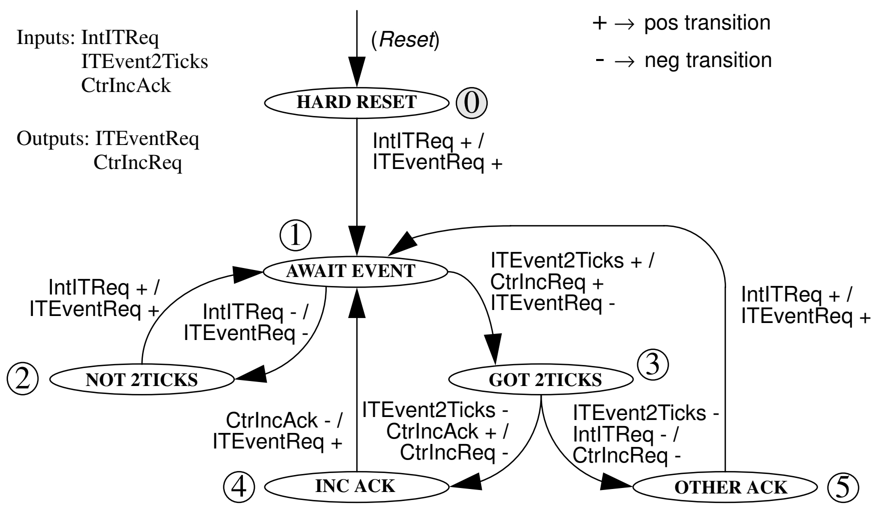


Figure 3.11: Burst-mode AFSM specification

Restriction 2 may be the most difficult one for a synchronous designer to become accustomed to. Since all transitions on input signals can potentially affect the circuit, care must be taken to ensure that no “unexpected” transitions occur. This requires a very careful analysis of the environment in which the AFSM is to operate, and is one of the reasons why asynchronous signaling events typically obey handshake protocols.

3.5.4.2 State Machine Synthesis and Implementation

Recall from Chapter 1 that there are several burst-mode design styles available which map into the basic AFSM architecture ([67], [89], [20]). In some cases, the state variables are fed back directly to the combinational logic. In other cases, a set of latches isolates the feedback path. Since there is no global clock, the AFSM is sensitive at all times to the level values on the inputs. This implies that outputs must make clean, monotonic transitions, and must remain stable until the next desired transition; i.e., they must be *hazard-free*.

Three synthesis methods, all developed at Stanford and based on burst-mode synthesis tools originally developed at HP Labs [20], are available to synthesize the control logic from a high-level

```
;start specification
;input is free-format, comments start with a semicolon

;signal names section:
;type   signame    initial value

input   IntITReq       0
input   ITEvent2Ticks  0
input   CtrIncAck      0
output  ITEventReq     0
output  CtrIncReq      0

;behavioral description section:
;from   to      input burst         output burst

0       1       IntITReq+         | ITEventReq+
1       2       IntITReq-         | ITEventReq-
2       1       IntITReq+         | ITEventReq+
1       3       ITEvent2Ticks+    | CtrIncReq+
                                    ITEventReq-
3       4       ITEvent2Ticks-
                CtrIncAck+        | CtrIncReq-
3       5       ITEvent2Ticks-
                IntITReq-         | CtrIncReq-
4       1       CtrIncAck-        | ITEventReq+
5       1       IntITReq+         | ITEventReq+

;end specification
```

Figure 3.12: Textual state machine specification

textual description. At present, however, only the 3D synthesis method [89] is fully integrated into our toolset. All methods accept a common input format, which is a simple textual description of the desired AFSM behavior, and produce a hazard-free implementation of a burst-mode specification, differing in the details of the circuitry produced. An example of an input file, corresponding to the state diagram in Figure 3.11, is shown in Figure 3.12.

The operation of a 3D state machine is similar to the operation of a Huffman machine [83]. In response to an input burst, the outputs and state variables change as specified. The fundamental-mode assumption must hold, meaning that the next input burst must not occur until the state variables and outputs have settled at their final values. The feedback path must meet some one-sided timing constraints to ensure proper state machine operation.

The output of the state machine synthesis step is a set of hazard-free logic equations which implement the state machine behavior. These equations are not necessarily minimal, however, so they are processed by an asynchronous logic minimizer before being passed to the technology mapper. Each input burst allowed by the specification must be examined to ensure that no hazards will exist in the implementation.

The asynchronous logic minimizer [68] uses a constrained version of the familiar Quine-McCluskey algorithm [54], and takes hazards into account during the minimization process. The logic produced by the minimizer is typically larger than that produced by an equivalent synchronous minimizer, but in the cases we have examined the overhead (with respect to the number of product terms) has been no more than 6%.

The hazard-free logic produced by the minimizer is then passed to the technology mapper described in this chapter. The output of this step is a netlist of elements from the library which represents a hazard-free implementation of the state machine in the given technology.

Netlists generated by the technology mapper are automatically translated into a format used by the Mentor GDT suite of tools. Place-and-route of the basic state machine is then done automatically using the GDT AUTOCELLS place and route tool. The resulting state machine is thus implemented as a single standard-cell block.

Once all control and datapath blocks have been generated, a block place-and-route tool is used to wire the chip together from the top-level VHDL netlist description. After routing, the design is extracted with Mentor's CHECKMATE and simulated with LSIM, using a test harness equivalent to the one used for VHDL simulation. The VHDL and LSIM simulation results are compared to check for any errors introduced by the synthesis process, and for those timing errors which can only be detected after synthesis.

3.5.5 ABCS Receiver Chip Design

We used the STETSON toolkit to design a receiver chip for an infrared communications protocol, called ABCS. This application was chosen as one that would demonstrate the power-saving poten-

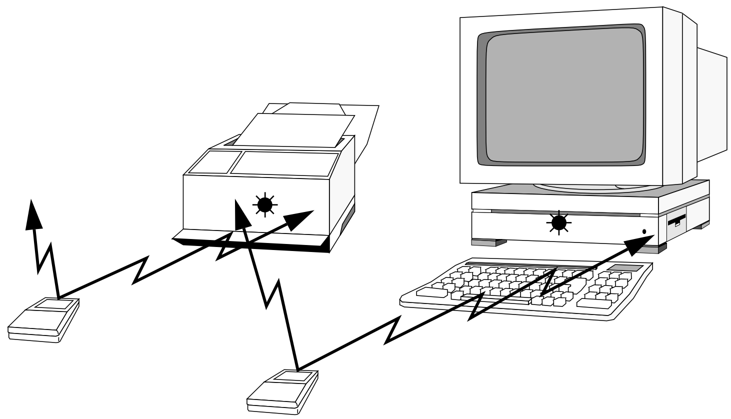


Figure 3.13: Typical ABCS application

tial of asynchronous design. The goal was to design a receiver which will draw only leakage current while waiting for incoming data, but can start up as soon as a signal arrives, so that it loses no data. A conventional synchronous implementation of such a design would necessarily incur a power penalty while waiting for incoming data, because it can only be sensitive to incoming signals by repeated polling or through the use of active circuitry. On the other hand, an asynchronous implementation will draw no power in its quiescent state.

The receiver handles an experimental multi-party communication protocol, called ABCS, which operates over an 800Kbit/sec infrared (IR) link. The protocol can be used for line-of-sight communications between portable computers and peripherals. Figure 3.13 shows a typical use of this protocol, where peripherals can be shared over an IR link. Other examples include file exchange over an IR link by a group of portable computer users in a conference room, and shared access to the wired LAN via an IR gateway.

The communication receiver chip is a subsystem of the full transceiver circuit that would be required for each station to communicate via the protocol, as shown in Figure 3.14. Because the environment in which the IR transceivers operate is not well controlled, the receiver must deal with noisy inputs from the IR “link”, which could be caused by interference from simultaneous transmissions by several transmitters, reflections from its own transmissions, or stray signals from incompatible IR devices and room lighting.

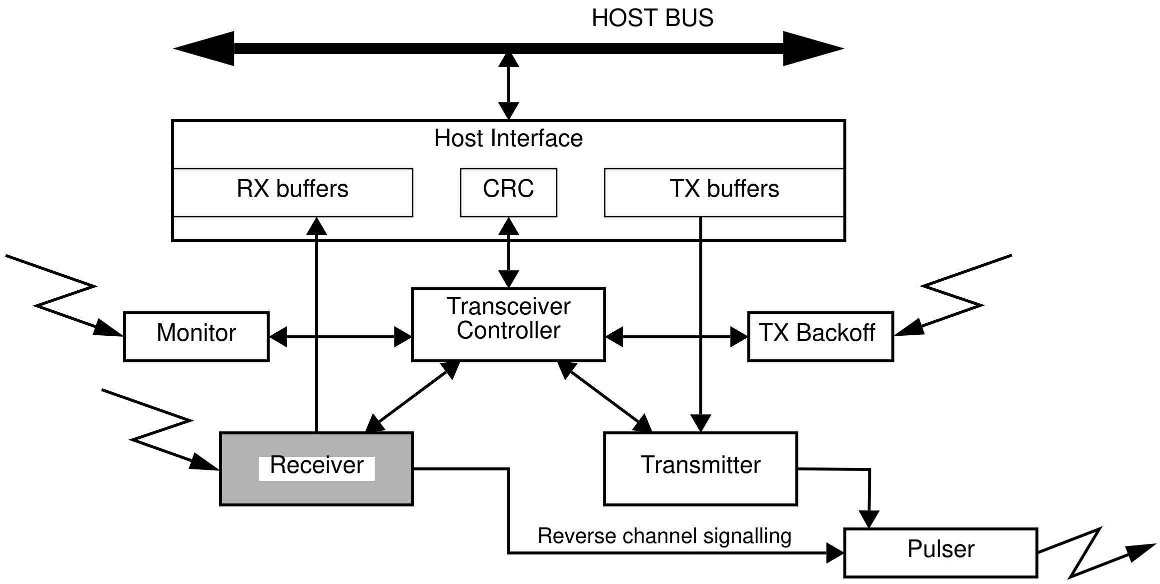


Figure 3.14: Simplified block diagram of an ABCS transceiver

Because asynchronous circuits are susceptible to glitches on their inputs, they are usually designed to operate in an environment which provides glitch-free inputs, typically conforming to a handshake protocol. We had to develop extensions to traditional asynchronous design to deal with the non-ideal environment in which the transceiver will operate.

The structure of the receiver, shown in Figure 3.15, reflects the structure of the ABCS protocol, so that the circuitry corresponding to each protocol level is only active and power consuming when an event at that level occurs.

The receiver is composed of:

- Two Interval Timers that measure the IR pulse-to-pulse intervals in terms. They are used alternately, allowing each to "recover" between measurements.
- A Symbol Counter, which sequences the Interval Timers and converts the pulse-to-pulse measurements into a representation of the position of each IR pulse within its symbol period.

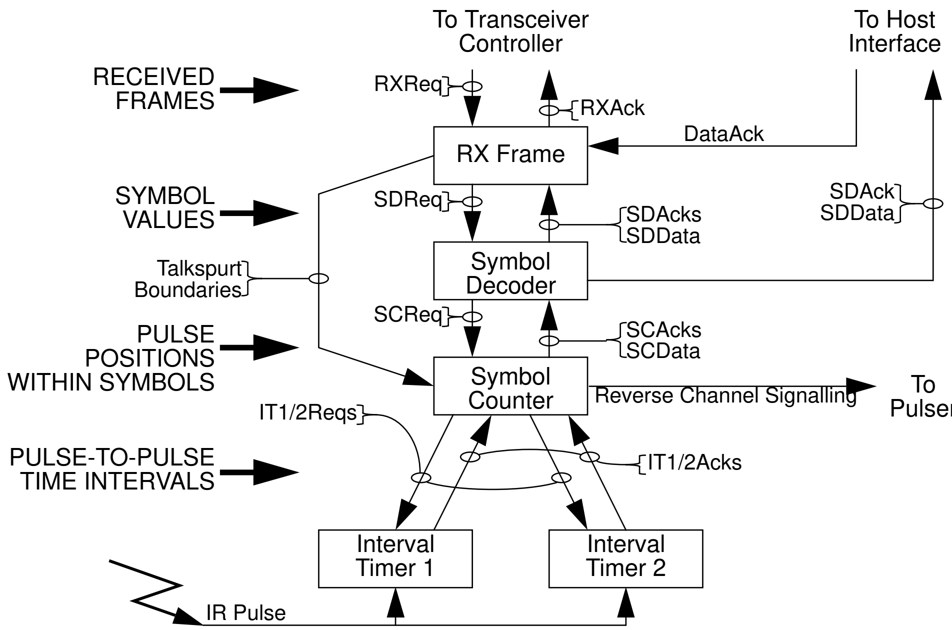


Figure 3.15: Simplified block diagram of an ABCS receiver

- A Symbol Decoder, which decodes the IR pulse positions into logical symbols and checks for line coding errors.
- The RXFrame block which, in addition to a few other functions, decodes the incoming frame structure, checks for framing errors, and decides whether the frame is addressed to this station.

3.5.5.1 Receiver Implementation

The receiver chip was manually decomposed into datapath and control blocks, paralleling synchronous chip design methodologies. The control blocks provide localized control of the supporting datapath blocks and handle exception conditions. However, the receiver is designed so that the control blocks do not handle individual data samples directly, reducing power consumption both by reducing the complexity of the control circuitry and by allowing the controllers to be inactive while "routine" data samples are handled by the datapath. The datapath circuitry (mainly registers, counters, comparators and adders) was specifically designed to handle the individual data samples, and does so with less power than a state machine implementation of the same function. Overall, separating the datapath from the control saved power in this design.

3.5.5.2 Inter-Block Communication

Since asynchronous designs do not use a global clock, the system must be coordinated through some handshaking protocol between communicating blocks. The handshake protocol forces each participating block to wait until the other is ready to proceed.

The two most common handshaking protocols in use for asynchronous design are *2-phase* and *4-phase* [76]. Both handshaking protocols connect two blocks through two wires, Request and Acknowledge. In a 2-phase handshake a request is made by a transition of the Request wire from its current state, which can be either high or low, whereas a 4-phase handshake obeys a more complicated protocol that, when complete, leaves the Request and Acknowledge wires in the same state in which they started.

Although a 2-phase handshake is seemingly simpler, a state machine using a 2-phase interface needs extra states to cope with each of the two possible initial states of the interface, potentially doubling the number of states compared to those needed for an equivalent state machine using a 4-phase interface (and re-doubling the number of states for each additional independent 2-phase interface connected to the state machine). The larger size of the 2-phase state machines outweighed the power savings from elimination of the reset transitions over the interface wires, so we chose to use a 4-phase protocol.

The basic 4-phase handshake can be expanded to include multiple Request and/or Acknowledge wires when one of several actions or results needs to be communicated. Also, one or more data wires can be controlled by a single 4-phase handshake interface to control the data transfer. This approach is described in [76], and has since become known as *bundled data*. The major inter-block connections shown in Figure 3.15 use all of these techniques.

Design Issues for Communicating AFSMs

Handshaking alone cannot always ensure that control blocks can communicate without violating the constraints imposed by the state machine design style we use. Problems can arise when the control block is connected to an environmental (off-chip) signal and is exposed to unexpected inputs (like noise), and when the control block is controlling multiple blocks. Furthermore, timing requirements on feedback paths must be met for the control blocks to work properly.

There are three cases in which state-machine inputs must be preprocessed to meet burst-mode constraints:

1. When there is a fast feedback path from a state machine output to its inputs, resulting in a change of the state machine's inputs before the state machine's internal state had settled. This can occur when a slow state machine interacts with a much faster environment.
2. When a state machine must process a sequence of external signals, where only a few of the possible sequences occur in normal operation. This can occur when a signal comes from the environment and numerous erroneous sequences are possible.
3. When the relative timing of some input signals determines the next state. This can occur when an AFSM is controlling multiple blocks and wants to act on the first acknowledgment from any of the controlled blocks, or when a signal comes from the environment and is thus not synchronized with any internal signal.

Case 1 can be corrected by adding a delay in the feedback path, although this might reduce the operating speed of the system. We added delays built out of inverter chains at a few places in this design to handle problems of this type.

Case 2 is a problem because a burst-mode AFSM is only guaranteed to work correctly if an input burst falls within the set of allowable bursts specified at design time. Thus, it can only handle erroneous inputs if all possible input sequences are included in the specification, which can lead to a complex state machine. In our design, the Symbol Decoder had this problem. Rather than adding all possible sequences of input values to the specification of the Symbol Decoder controller, we explicitly preprocess the inputs of the AFSM so that only valid inputs are passed to the AFSM, and invalid inputs are flagged as errors.

Case 3 can result in the AFSM entering a metastable state, or becoming locked in some unspecified state until it is reset. To handle these situations, the receiver design uses several mutual-exclusion elements (MEs), derived from the design described in [76]. Arbiters [76] provide a more general mechanism to handle such situations, but MEs worked fine for the cases we encountered.

3.5.6 Results

The receiver IC was implemented using the MOSIS 1.2μ CMOS process, and contained approximately 14,000 transistors, of which 1,200 were used to improve the observability of internal nodes for testing. The current consumption of the receiver core (i.e. without pads) was measured at less than 1.0mA @ 5V when the receiver is actually receiving data. The receiver draws less than 1 μA while waiting for data. The final VHDL description of the receiver is about 13,000 lines long, of which 6,000 lines are structural VHDL representing the netlist of the circuit, 4,000 lines are (generated) behavioral VHDL representing the asynchronous state machines, and 3,000 lines are behavioral VHDL representing the standard-cell gates in our library.

Power measurements of the chip confirm that asynchronous design techniques allow designs to operate in a data-driven fashion, allowing individual modules to be electrically active only when they have processing to do. In our design this means that most of the circuitry operates well below the maximum frequency of operation. However, this less-active circuitry is not constrained to operate with large latencies and low potential throughput as would be the case if it was simply clocked more slowly. The effects of this are illustrated in Table 3.6, which gives the average current consumption during data reception for each of the receiver's major blocks. The table also shows the current consumption on a current/FET basis as a measure of the level of circuit activity. All of the measured current consumption values were within 20% of the values obtained from simulation with the Mentor LSIM simulator, using the ADEPT circuit simulation mode.

BLOCK		# FETs	I_{supply} (µA typ.) @ 5V	I_{supply} / FET (nA typ.)
Interval Timer	- Control	242	6	25
	- Oscillator	534	151	283
	- Remainder	2,296	180	78
Symbol Counter	- Control	794	45	57
	- Datapath	672	67	100
Symbol Decoder	- Control	816	52	64
	- Datapath	306	17	56
RX Frame	- Control	218	1	5
	- Datapath	2,452	14	6
TOTAL (with 2 Interval Timers)		11,402	915	80

Table 3.6 Average current consumption of major receiver blocks during data reception

The ring oscillators in the Interval Timers are the most active blocks, having the highest current consumption/FET in spite of having very short internal interconnections, and accounting for about a third of the total current consumption. The controller in the Interval Timer is small and makes only a few transitions per received symbol, resulting in a surprisingly low current consumption/FET for a block at the lower levels of the receiver. The remainder of the Interval Timer circuitry operates at a wide range of frequencies, and has an average current consumption/FET. The Symbol Counter and Symbol Decoder blocks are both active only once per received symbol (every 2.25 µs), and have lower values of current consumption/FET than the Interval Timer. The RX Frame block is active mainly at talkspurt and frame boundaries, resulting in very low current consumption/FET figures.

The toolkit was implemented as a mixture of university tools, including the technology mapper described in this chapter, commercial tools from the Mentor GDT toolset, and some custom code for netlist translation, constraint checking and tool sequencing. All totalled, the custom code consisted of approximately 5,000 lines of LISP code, 9,000 lines of C code and 2,000 lines of PERL, SHELL and NAWK code. All tools were run on an HP 9000/7xx platform, but are portable to other platforms. Synthesis of AFSMs from specification through netlist generation was very fast, with the bulk of the time being spent in layout of the standard-cell netlist.

3.6 Summary

In this chapter we showed how technology mapping algorithms for synchronous designs can be adapted to work for generalized fundamental-mode asynchronous designs by detecting the hazards in the library elements and using this information to ensure that no new hazards are introduced during the mapping process. The resulting mapped network will then have no new hazards. We proved that in characterizing the hazard behavior of the mapped and unmapped network, only logic hazards need to be taken into account. Based on that information we have created efficient algorithms for hazard detection and incorporated these algorithms into an existing technology mapper to produce an asynchronous technology mapper that works well for generalized fundamental-mode asynchronous designs. A crucial part of the algorithm assumes that the library is properly characterized, and that its hazard behavior is properly expressed through the structure of a Boolean factored form. We integrated this mapper into the STETSON toolkit for asynchronous design and used the toolkit to implement an asynchronous communications IC receiver chip which was fabricated and worked at first silicon.

Chapter 4

Technology Mapping for Speed–Independent Designs

4.1 Introduction

In this chapter, we address the problem of automatic technology mapping for speed-independent asynchronous designs. The asynchronous speed-independent design style is characterized by delay assumptions of zero wire delay and (potentially) unbounded pure gate delay. Several automatic synthesis methods exist that take a high-level description of a design and automatically generate a correct speed-independent implementation composed of low-level logic primitives ([19], [4], [46], [42]). However, there is no general procedure for mapping those primitives to an existing library of standard-cell or gate-array components. As with burst-mode asynchronous designs, the main difficulty lies in coping with the hazard properties of transformations. In this chapter, we explore these issues in detail and propose algorithms for library binding of speed-independent circuits.

We first introduce background relevant to speed-independent circuits, some of which was initially presented in Chapter 1. We then show how incorrect decomposition of gates can lead to hazards in speed-independent designs to illustrate the difficulties in library binding for this design style. Starting with the basic technology-mapping procedure presented in the previous chapter, we show that the decomposition step must be modified for hazard-free mapping, and we develop theory and algorithms for hazard-free decomposition of basic gates. Although the matching/covering algorithms of the previous chapter can be used without modification for speed-independent designs, we present a covering algorithm which takes advantage of properties of this design style to produce better quality mapped circuits. We implemented these algorithms in a new mapper, and we conclude the chapter by presenting results from its application to standard benchmarks.

4.2 Background and Definitions

In this section we review the delay model, the specification style and properties of the initial speed-independent implementation that we introduced in Chapter 1.

In addition to assuming that wires have zero delay and gates have unbounded, but finite, delay, each delay element is assumed to exhibit *pure* delay properties [45], implying that each enabling event on a gate's input will be transmitted to its output. Thus, glitches will not be filtered out by inertial delays, and any glitch can be propagated to the output of a gate.

Speed-independent designs operate in *input-output* mode. In this mode, the environment can apply changes to the inputs of the circuit as soon as it detects changes on the circuit's outputs. (This is in contrast to fundamental mode, where the circuit must settle before the next set of input changes is applied.)

Each gate in a speed-independent design is assumed to be *atomic,* and can be viewed as an instantaneous Boolean function of its inputs with a single pure delay on the output. The atomic gate assumption implies that each gate in the circuit either already exists as a single CMOS gate in a library or that a custom transistor-level implementation will be created for it. If a function is implemented as an atomic gate, then given any enabling input change, the gate will respond with a corresponding output change after some (finite, but unspecified) delay. Because of the atomic gate assumption, the relative ordering of signal transitions on the inputs to a gate will be preserved irrespective of the actual gate delays.

An initial speed-independent implementation composed of simple combinational logic and C-elements is used as input to the technology mapping step. This implementation may have been automatically synthesized from a speed-independent state transition graph (STG) or state graph (SG), or may have been derived by some other means ([19], [56], [4], [46], [42]). Although the specifica-

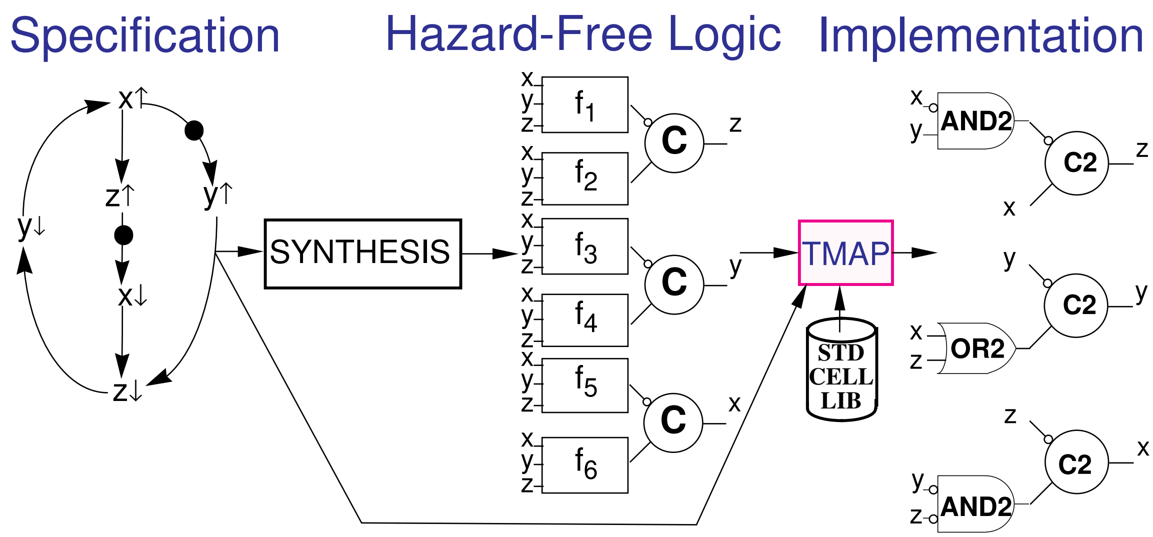


Figure 4.1: Illustration of synthesis steps and their inputs and outputs

tion style is not important, we will need to extract environmental information from the specification to do the technology mapping. Figure 4.1 shows a simple STG specification and the general synthesis flow leading to a hazard-free technology-specific implementation in the implementation style we initially consider.

There are several interesting points about this specification and implementation style that impact the technology mapping algorithm. First, since the gates in the initial implementation can have arbitrarily large fan-in, some of the gates produced by the synthesis method may not exist in the given gate-array or standard-cell library. Therefore, the technology mapping algorithm must be able to decompose these large gates into an interconnection of smaller gates while preserving the speed-independence of the design, a non-trivial task. Second, because of the delay assumptions inherent in the speed-independent design style, the relative signal ordering through each gate in the combinational logic must be preserved in the final implementation or hazards can result. This implies that technology mapping must be done in the context of the circuit's environment. The signal ordering imposed by the environment also has a large impact on how large gates can be broken into smaller ones, as we will show later.

4.3 Hazards and Gate Decomposition

This section uses hazard terminology that was introduced in Chapter 2. We briefly review and expand the terminology as relevant for our purposes.

As with burst-mode designs, the initial speed-independent design (the input to the technology mapping step) is assumed to be hazard-free within the context of the environment in which it operates. For the speed-independent design style that we consider in this chapter, both combinational and *sequential* hazards are of interest.

Recall that we separated the classes of combinational hazards into *function* and *logic* hazards. A circuit that has a function hazard for a given multi-input-change transition cannot also have a logic hazard for that transition. Logic hazards only apply to function-hazard-free transition cubes. Logic hazards can cause speed-independent designs to malfunction, just as for burst-mode designs. Therefore, they must not be introduced during the technology-mapping phase.

With the speed-independent design style, we must additionally consider function hazards in our analysis. Function hazards in basic gates are avoided by the sequencing imposed by the environment and the sequential behavior of the circuit. However, this sequencing can be destroyed if the gate is improperly decomposed—logic transformations that alter the order of signal propagation through the network can cause function hazards to be exercised. With burst-mode designs, function hazards could be ignored because all input bursts were monotonic and happened within the context of function-hazard-free transition cubes.

We can now formalize these notions and further classify the hazards that are of interest for the speed-independent design style. Armstrong [2] classified all non-logic hazards within speed-independent designs as *delay hazards*:

Definition 4.1 *A logic-hazard-free circuit operating in input-output mode realizing a combinational function f contains a* static delay hazard *if for some input sequence* (I_1, I_2, I_3)*, where* $f(I_2) = f(I_3)$*, the output sequence* $f(I_1), f(I_2), f(I_2)', f(I_3)$ *can be produced.*

In this definition, I_1, I_2, and I_3 are all single-input transitions on one or more primary inputs.

Definition 4.2 *A logic-hazard-free circuit operating in input-output mode realizing a combinational function f contains a* dynamic delay hazard *if for some input sequence* (I_1, I_2, I_3), *where* $f(I_2) \neq f(I_3)$, *the output sequence* $f(I_1), f(I_2), f(I_3), f(I_2), f(I_3)$ *can be produced.*

We can further classify delay hazards as being either *sequencing* hazards or *sequential* hazards, where sequencing hazards are due to the modification of the ordering of signal transitions within a portion of the circuit due to the specific implementation, and sequential hazards occur as a result of the feedback in the circuit. (When Armstrong introduced the notion of delay hazards, he only recognized the significance of sequential hazards.) These hazards are all the result of multi-input-change transitions.

The notion of *acknowledgment* is key to the definition of a sequential delay hazard.

Definition 4.3 *An output transition on a gate is* acknowledged *if one of the gates connected to its fanout cannot change value until that output transition reaches the input of the latter gate.* [4]

Definition 4.4 *A* sequential delay hazard *is a delay hazard that results from an unacknowledged transition on a gate output.*

Because the circuit is operating in input-output mode in this design style, any change in a state variable can be immediately fed back to the next-state computation circuitry without having to wait for the rest of the circuit to settle. With burst-mode designs, the fundamental-mode assumption imposed constraints on the feedback circuitry that ensured that the internal state logic would settle before the next set of transitions was applied. As a result, we were able to ignore any sequential hazards (with the caveat that we had to satisfy timing constraints relative to the delays in the feedback path).

An example of a sequential hazard is given later in Section 4.4.2.

Sequencing delay hazards are defined as follows:

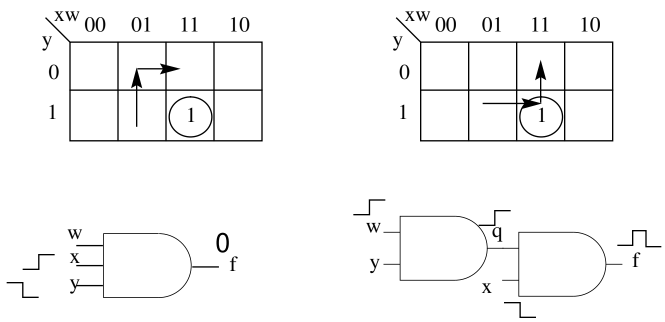


Figure 4.2: Example illustrating a sequencing delay hazard

Definition 4.5 *Given a block of combinational logic in a speed-independent implementation, and a set of ordered input transitions on its inputs that avoid function hazards in the function represented by the combinational logic, then a* sequencing delay hazard *is a hazard that occurs when the implementation does not maintain the order of propagation through the block of combinational logic, thus exercising the function hazard.*

Given the delay assumptions inherent in speed-independent design, if a section of combinational logic has a sequencing delay hazard for some set of input changes, then it will still occur for the those input changes even if the same set of input changes are applied to the combinational logic block after it has been removed from the circuit (i.e., if the feedback is cut for that portion of the circuit). Later in this chapter we give examples of specific instances where these hazards can be exercised due to improper decomposition.

Figure 4.2 illustrates a sequencing hazard caused by improper decomposition (we will return to this example in more detail later. The decomposition into a set of gates allows the relative ordering between *q* and *x* to be disturbed, thus exercising a function hazard in the final gate.

The partitioning of the class of applicable hazard behaviors can be seen in Figure 4.3.

For the remainder of this thesis, we will use the terms *sequencing hazard* and *sequential hazard* to refer to sequencing delay hazards and sequential delay hazards respectively.

4.4 Technology Mapping for Speed-Independent Designs

To tackle the library binding problem, we started with the mapping procedure described in the previous chapter and examined each step to see its effect on the hazard-behavior of the design. We found the decomposition step to be the most troublesome, and we focus on this step after briefly describing the overall technology mapping procedure. First we must properly characterize the initial design. We then show how improper decomposition can lead to hazards, motivating the need to characterize the circuit environment (a step that is not necessary for library binding of burst-mode and synchronous designs). We then present theorems supporting a simple hazard-free decomposition algorithm that works for some gates. We extend the simple procedure to handle cases where the simple procedure yielded a hazardous decomposition. Finally, we present a modified covering algorithm that takes advantage of the relatively small size of these designs to produce a better cover than that produced by the algorithm used in the previous chapter.

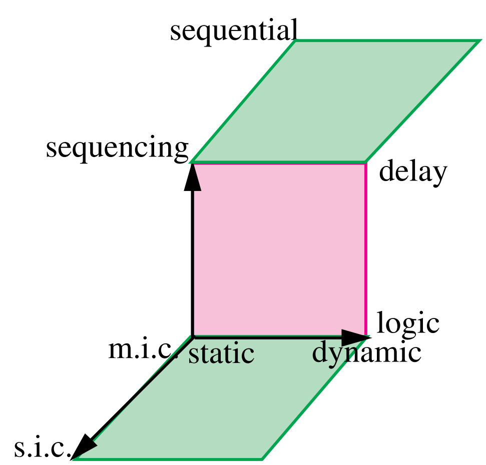


Figure 4.3: Hazard space for speed-independent circuits

We start with our previous technology mapping procedure and modify it to work for the delay assumptions of the speed-independent design style. Recall the burst-mode technology mapping procedure:

```
procedure bm_tmap(network, library) {
    decomposed_network = tech_decomp(network);
    cones = partition(decomposed_network);
    foreach output in cones {
        find_best_async_cover(output, library);
    }
}
```

The input to the procedure is a technology-independent combinational netlist. From there, each gate within the network is broken up into fine-granularity basic gates, e.g. two-input NAND gates. The network is then partitioned into single-output cones of logic, and each cone is then covered with the best set of matching elements from the cell library. In the burst-mode case we showed that the decomposition step of a synchronous mapping algorithm could easily be modified to be hazard-preserving. We also showed that the partition step of a synchronous mapping algorithm was hazard-preserving and could be used without modification in an asynchronous mapper. The matching and covering step of the synchronous mapper was modified to ensure that no new logic hazards were introduced during that step.

For the speed-independent case we start with an initial speed-independent sequential implementation consisting of two-input C-elements and unlimited-fanin ANDs and ORs. We then partition the circuit by breaking the circuit at the outputs of the C-elements. This induces a partition into blocks of circuitry that implement each output. For each block, we extract cones of combinational logic, starting with a specific C-element input and working backwards to include all gates between the primary inputs and the particular C-element input. The gates within each cone of logic are then decomposed into smaller gates. We assume that the basic two-input C-element exists in the library.

As with burst-mode designs, we must also take the hazard-behavior of the implementation into account during mapping. However, with the delay assumptions of the speed-independent designs, we can no longer assume that the outputs from a logic block will settle before the next set of input changes is applied. As a result, we must consider hazards due to interactions from the feedback path (i.e. the sequential hazards). Additionally, because of our assumptions of arbitrary gate delay and zero wire delay, we also must ensure that our implementation does not change the sequencing of any of the input signal transitions. If the signal ordering changes, then sequencing hazards in the combinational logic may be exercised.

The decomposition step is now the biggest problem, and we must find a way to decompose the circuit such that no hazards are introduced or exercised. We still must use the modifications that were made to the matching and covering step in the burst-mode mapper, since introduction of logic hazards into a speed-independent design may also cause the design to malfunction.

Our initial speed-independent technology mapping procedure is thus as follows:

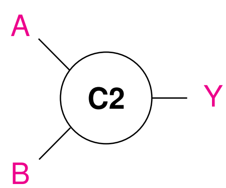
```
procedure si_tmap(network, library) {
    cones = partition(network);
    foreach subnetwork in cones {
        decomposed_subnetwork = si_tech_decomp(subnetwork);
        find_best_cover(decomposed_subnetwork, library);
    }
}
```

4.4.1 Characterization of the Initial Implementation

We now define the initial speed-independent implementation that is used as the input to the technology mapping step.

Definition 4.6 *A two-input* C-element *is a sequential element with no metastable states that implements the equation* $C = AB + (A+B)\,C$.

Figure 4.4 shows the C-element's schematic symbol along with its next-state table. Unlike with an SR-latch, all states are well-defined.



A	B	C
0	0	0
0	1	hold
1	0	hold
1	1	1

Figure 4.4: C-element symbol and its next-state table

Definition 4.7 *An* initial implementation *of a speed-independent circuit consists of two-input C-elements which implement the outputs of the design, and blocks of combinational logic connected to each C-element input.*

Some of the combinational logic blocks may be cones of combinational logic (i.e., they have no internal wires fanning out to other combinational logic blocks).

Definition 4.8 *An* output logic block *consists of a two-input C-element with one inverted input, along with the combinational logic connected to each of the C-element's inputs. Each output logic block has a single output, and the inputs to the block are either primary inputs (i.e., signals from the environment) or outputs of other output logic blocks (i.e., primary outputs).*

In the design style we consider in this thesis, because the C-elements have one input inverted they operate in a push-pull fashion, requiring an opposite change on each input before the output can change.

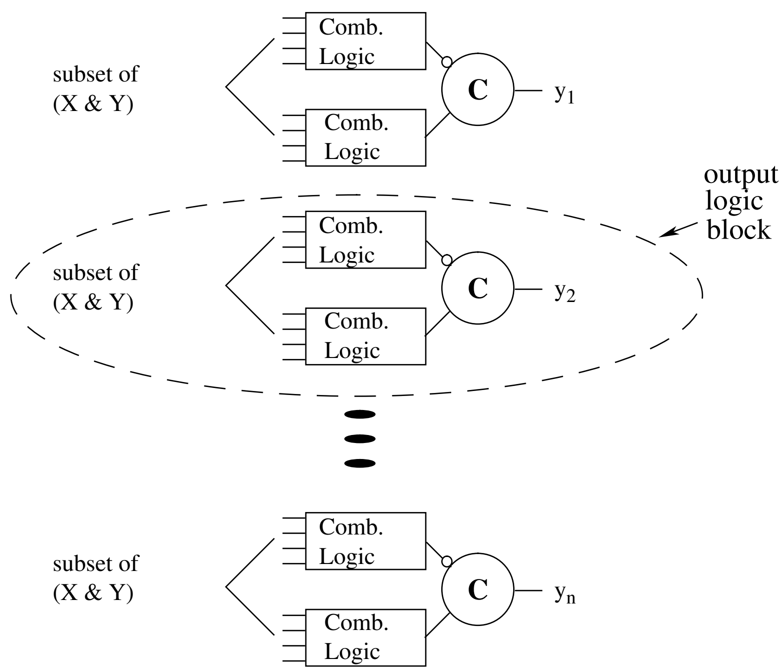


Figure 4.5: Initial implementation

Figure 4.5 shows an example of the initial implementation structure composed of *n* output logic blocks. The inputs to each logic block are a subset of the primary inputs and outputs, and each output logic block implements a single primary output. For the purposes of decomposition, each output logic block can be treated independently.

4.4.2 How Hazards Can Occur During Decomposition

Before characterizing the behavior of the circuit's environment, we show through a few examples how knowledge of the environment is necessary to do proper decomposition. We illustrate where hazards can occur in an improperly decomposed circuit, focusing on the combinational portions of the design.

The following example illustrates how a sequencing static 0-hazard can occur due to improper decomposition. Figure 4.6a shows a simple AND gate which implements the function $f = wxy$. We want to implement this function as a cascaded connection of two-input AND gates. Furthermore,

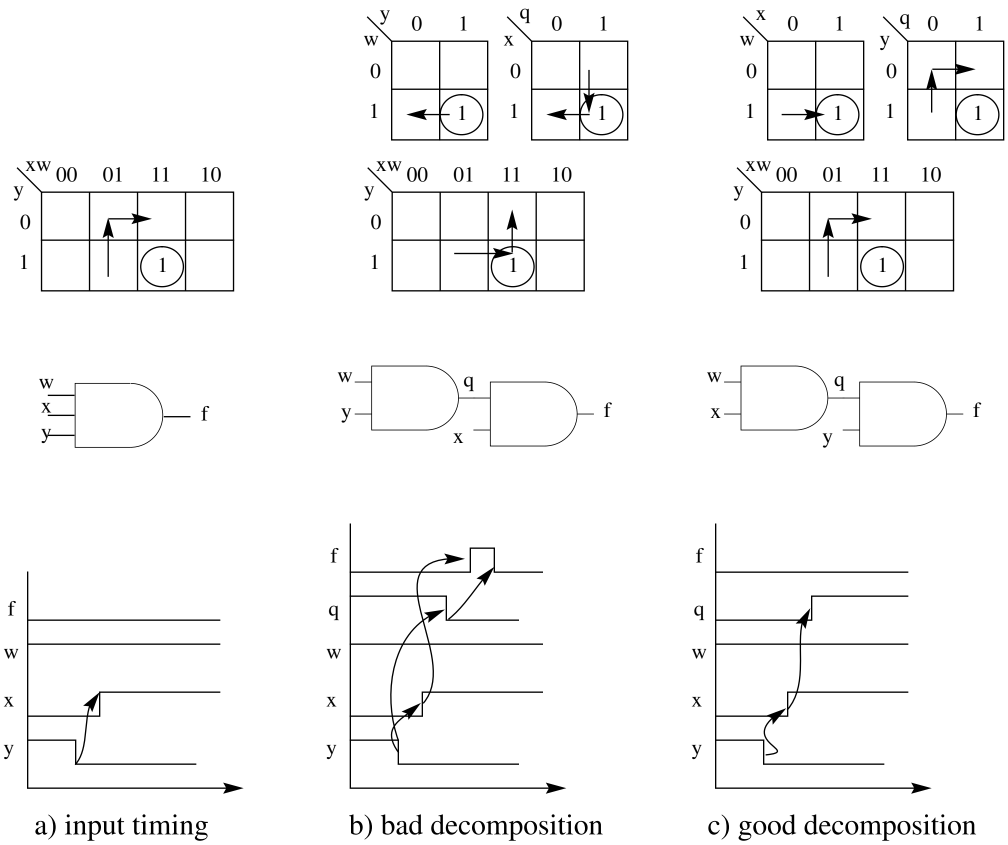


Figure 4.6: Example of sequencing delay hazard due to incorrect decomposition

suppose that the input timing (derived from the STG) is such that $y\downarrow$ is followed by $x\uparrow$, as shown, and w remains constant at 1. Because of the input signal ordering, the inputs change along the trajectory shown in the Karnaugh map, and the function hazard inherent in the original function is avoided. As a result, the output of the gate remains at 0 during the transitions on x and y.

Now, suppose the gate is decomposed, with w and y being placed together as inputs to an intermediate AND gate, as shown in Figure 4.6b. Then, when y changes, the intermediate AND gate will go high some (indefinite) time later. Prior to the change propagating to q, x can go high resulting in a glitch on the output of the final gate. This glitch can propagate elsewhere in the circuit, causing it to malfunction.

A hazard-free decomposition is shown in Figure 4.6c, where *w* and *x* are connected to the same gate, and *y* is connected to the AND gate closest to the output. In this case, the change on *y* is immediately seen by the final gate. The intermediate AND gate will change some time after *x* changes, but the output of the network will not change, since *y* is guaranteed to be low by the time the change in the intermediate AND gate propagates to *q*, keeping the final AND gate low.

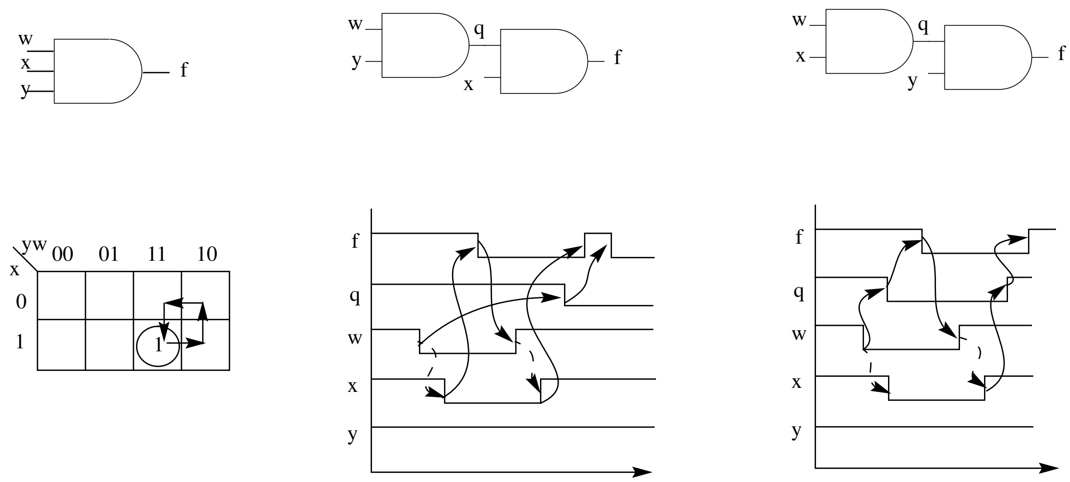
In this case, there is a multi-input change on the inputs to the AND gate, and proper signal sequencing is required to avoid the static function 0-hazard inherent in the original function. In the decomposition of Figure 4.6b, the relative order between the two signals is not maintained as it propagates through the combinational logic, exercising the sequencing hazard as shown. In the decomposition of Figure 4.6c, the critical signal ordering is maintained by placing the first arriving signal closest to the final gate's output, maintaining the signal ordering necessary to avoid the function hazard. (Note that this "good" decomposition has a problem when we introduce feedback into the circuit, but we will ignore that for now.) Although the good decomposition is sequencing-hazard free, it is not sequential-hazard free.

Note that sequencing hazards were not an issue for burst-mode designs because burst mode never exploits ordering assumptions on inputs or internal signals. Because burst-mode assumes arbitrary wire-delays, the ordering can never be depended upon. Additionally, bursts are guaranteed to occur within a monotonic transition space. As a result sequencing hazards are never an issue.

This example suggests that circuit transformations must preserve the ordering of certain signals through the internal combinational logic of any decomposed gates. Furthermore, these transformations are dependent upon the specification, which defines the signal ordering.

We just illustrated a sequencing static 0-hazard, where the output of the AND gate was supposed to remain low during the changes in the inputs. Let us now take a look at the case where the AND gate's output falls. In this case, the first signal that arrives causes the AND gate to change. Let us examine how incorrect decomposition can lead to a sequential hazard.

$w\downarrow \rightarrow x\downarrow \rightarrow f\downarrow \rightarrow w\uparrow \rightarrow x\uparrow$



a) initial circuit b) bad decomposition c) good decomposition

Figure 4.7: Illustration of a dynamic hazard on the falling transition of an AND gate.

Figure 4.7 shows a simple 3-input AND function, along with a sequence of transitions on the inputs to the function. Initially, the output of the function is a logic “1”. As soon as *w* transitions low, the output is enabled to go low. Finally, once both *x* and *w* have gone high, the output goes back to a logic “1”. This defines the correct operation of this particular function under the given set of input transitions.

In the circuit of Figure 4.7b, as soon as *x* changes, *f* can go low, prior to the change on *w* reaching the input to the final AND gate. If *w* then changes back to “1” in the next input sequence, it is possible that it does so before the “0” has propagated to the input to the final AND gate. In the meantime, *x* may have changed back to 1, at which point the “1” on the final AND gate is acknowledged. *f* then changes to “1”, but the glitch on the internal gate appears on the output. In the decomposition of Figure 4.7c, the two signals which are changing maintain their temporal ordering, and thus no glitch appears. More simply, there is a single-input change on the inputs to the final AND gate, causing the output to make a monotonic change in response to the input change.

This example also illustrates another more subtle point. An unacknowledged signal in the presence of feedback can cause hazards. In the circuit of Figure 4.7b, the change on the intermediate gate remains *unacknowledged* by the final AND gate. That is, the final AND gate is allowed to change before it sees the change on the output of the intermediate AND gate. In fact, depending upon the delays in the circuit, the change on the intermediate AND may take a long time to propagate, well after *f* has reached its final value. By the time the final AND gate sees the change it may no longer be a valid transition, resulting in a hazard. In the circuit of Figure 4.7c, on the other hand, it is the change on that intermediate node that causes the final AND gate to change, and thus the transition is properly acknowledged.

So, in addition to ensuring that a decomposed circuit is logic-hazard free, we must also ensure that it is free from both sequential and sequencing delay hazards.

4.4.3 Characterization of the Circuit's Environment

As we have just illustrated, knowledge of signal transition ordering is essential to be able to decompose gate-level speed-independent circuits in a hazard-free way. Therefore, we cannot do the decomposition without knowledge of the signal ordering—we must either extract all possible sequences from the initial implementation or we must extract information from the specification (e.g. the STG or the SG). Our initial attack at the problem will extract information from the specification, since we may not be able to infer all degrees of freedom from examination of the initial implementation. We then use the extracted signal ordering to drive the decomposition.

Definition 4.9 *An* input burst *is an unordered set of input transitions that can appear at the inputs to an output logic block.*

Input bursts are denoted as signal transitions separated by vertical bars enclosed within curly braces, for example, $\{x_i\downarrow \parallel x_j\uparrow \parallel x_k\uparrow\}$. They represent a concurrent portion of a state graph, where the transitions may occur in any order. A degenerate input burst consists of a single signal transition.

Definition 4.10 *An* input sequence *is an ordered set of input bursts.*

Input sequences are denoted as input bursts separated by commas enclosed in parentheses. An example of an input sequence is (x_i↑, x_j↓, x_k↑), where the signals are all inputs to the output logic block. ({x_i↓ || x_j↑}, x_k↑, x_i↑) is an example of an input sequence having non-degenerate input bursts. In this case, x_i goes low in parallel with x_j going high, followed by x_k going high, followed by x_i going low.

Definition 4.11 *A* context signal *is a signal that remains constant during a given input sequence and is one of the inputs to an output logic block. Furthermore, the* context *for a given sequence is the set of context signal values for that input sequence.*

We can now describe the input-output behavior of an output logic block.

Proposition 4.1 *The input-output behavior of an output logic block for output y_i is completely defined by the set of possible (input sequence; context) tuples that are extracted from the state transition graph for each instance of y_i↑[y_i↓], where the starting point of each input sequence is the state immediately following the preceding downward [upward] transition on y_i, and the context for that input sequence is the set of input variables and their values that remain constant during that sequence.*

Note that if we were to extract the input-output behavior of the circuit from the initial implementation, we would not have as many degrees of freedom as we do by taking the information from the specification, since we know nothing about the behavior of environmental signals. Because some of the signals are external to the circuit, we cannot know when they occur just from the circuit description itself.

4.4.4 Decomposition

Having characterized the initial implementation and the circuit's environment, we are ready to tackle the decomposition problem. The rough outline of our approach is as follows. We first partition the initial design by separating the C-elements from the combinational portions of the circuit. Next, after proving that each cone of logic can be treated independently, and furthermore, that each

AND gate within a cone of logic also operates independently (i.e. that the inputs to the OR gate in the implementation of an SOP expression operate disjointly), we formulate legal decompositions of each gate for each type of input-output behavior the gate may experience (inputs change⇒output rises; inputs change⇒output falls; inputs change⇒output does not change). Next, we use these conditions to determine the set of legal decompositions for a gate. We then propose a decomposition algorithm based on the theory presented, and give results showing where the algorithm succeeds and fails. Finally, we extend this algorithm to insert wire forks to eliminate sequential hazards, and show results of applying this algorithm to the cases that fail with the simple algorithm.

Theorem 4.2 *Given the implementation of an output logic block as in Definition 4.8, the inputs to the C-element (i.e. the outputs of each combinational logic block) each make a single monotonic transition before the C-element's output changes. Furthermore, the inputs to the C-element can change in any order, and therefore each combinational logic block that feeds into the C-element can be treated independently.*

Proof: Because the initial implementation is speed-independent, one logic block can take a much longer time than the other to propagate to the input of the C-element, destroying any initial ordering imposed by the input signals to the combinational logic block. If an input to the C-element were to change multiple times before the output changed, then this would imply a possible hazard on the output of the C-element. However, the initial implementation is guaranteed to be hazard-free. Therefore, each input to the C-element changes monotonically. Because the inputs to the C-element have no ordering implied, we can treat each combinational logic block separately, since the behavior of one C-element input is independent of the other C-element input. ❑

Recalling that by definition an input sequence causes a C-element to change state, we can present the following corollary:

Corollary 4.2.1 *During an input sequence, each input to the C-element in an output logic block will change exactly once. Additionally, the inputs will change in opposite directions.*

The input-output behavior of an individual combinational logic block within a given output logic block can easily be extracted from the output logic block's characterization, by filtering out those signals which do not appear as input variables to the combinational logic block, and by inverting the output of the combinational logic block when the output is connected to a complemented C-element input.

Now we can examine the hazard-behavior of an individual combinational logic block. We can concentrate on the combinational logic block connected to a C-element input.

Theorem 4.3 *Any logic hazards in the cone of logic for the transitions of interest (as defined by its input-output behavior), may cause the circuit to perform incorrectly.*

Proof: If the logic hazard is exercised by one of the valid input bursts or sequences, then the output of the combinational logic block will cause a non-monotonic transition to occur on the input of the C-element which may result in the circuit making a transition to an improper state. ❑

Note that we must look within the set of possible transitions to see if that hazard will be exercised.

Corollary 4.3.1 *Given a cone of logic, the decomposition must be done in a logic-hazard-preserving way.*

Note that Corollary 4.3.1 is a necessary, but not sufficient, condition for hazard-free decomposition, because even if a decomposition is logic hazard free, it can still have sequencing hazards.

We now consider how to decompose a circuit for the special case where the input sequences are simply single input bursts.

Theorem 4.4 *Given a cone of combinational logic where the input-output behavior has only a single input burst in each (input sequence; context) tuple, then the circuit can be decomposed as for the burst-mode case.*

Proof: A single input burst represents a set of unordered monotonic input changes on one or more inputs. The transition cube over the input space as defined by the starting point and ending point of those changes must be function-hazard free or a hazard would result in the original implementation. Given a logic-hazard-free decomposition, the decomposition will only change the ordering of the signal transitions as they propagate through the logic. However, because the operation is over a function-hazard-free transition cube, changing the order of signal propagation cannot cause a hazard. Since the decomposition is by definition logic-hazard free, no logic hazards are introduced and thus the decomposed circuit is still hazard-free. ❑

Now we can consider the more general case where we have a combination of sequential input changes and input bursts in an input sequence. We restrict possible decompositions to those that are logic-hazard-preserving.

Without loss of generality, we initially assume that the logic inside a combinational logic block consists of an AND-OR implementation of a sum-of-products (SOP) logic expression.

Theorem 4.5 *Given a block of logic that consists of an AND-OR implementation of a sum-of-products expression, the OR gate can be decomposed according to the associative law, without regard to signal ordering, and the resulting decomposed network will still be hazard-free.*

Proof: If we can show that for any (input sequence, context) tuple, the OR gate will only see a single-input change, then we know that we can decompose the gate by any application of the associative law and we will not affect the hazard-behavior of the circuit. So it just remains to be shown that the OR gate will only see single-input changes. Suppose the OR gate saw a multi-input change. Then, there are three possibilities: one input makes a $0 \rightarrow 1$ transition while the other makes a $1 \rightarrow 0$ transition; both inputs make $0 \rightarrow 1$ transitions; both inputs make $1 \rightarrow 0$ transitions. In the case of $0 \rightarrow 1$, $1 \rightarrow 0$ transitions on the inputs, we know that any temporal ordering of those transitions are lost, and therefore it is possible that these transitions will exercise a static 0-hazard (sequencing hazard) on the OR gate, resulting in a hazard in the circuit. But the initial implementation is hazard-free, so this is impossible. Suppose, instead, both inputs make $0 \rightarrow 1$ transitions. Then, the second input will never be acknowledged, leading to a potential hazard in the design. Again, we have assumed that the circuit is hazard-free, so this is impossible. It is also impossible for one input to remain at 1 while the other one transitions to 1, since that transition will also not be acknowledged.

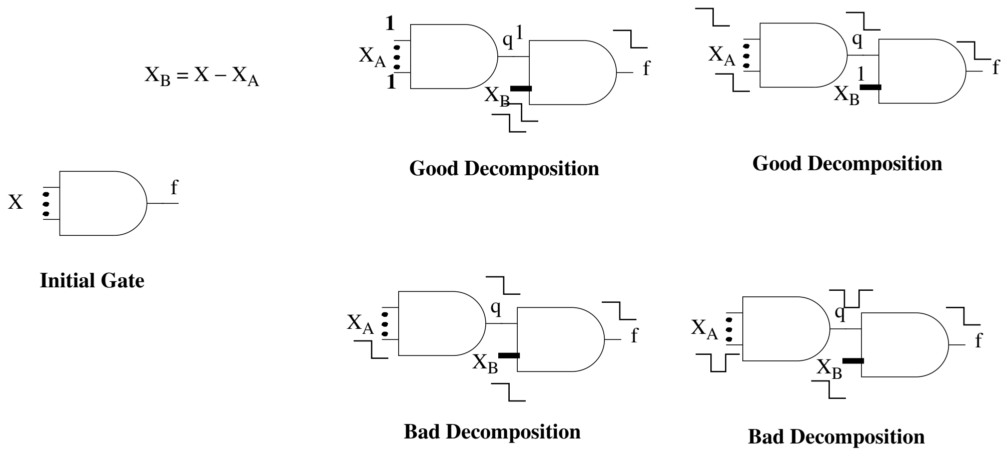


Figure 4.8: Illustration of Theorem 4.6

Therefore, it is impossible for both inputs to simultaneously be 1, which makes it impossible for both inputs to make 1→0 transitions. ❑

Corollary 4.5.1 *Given a block of logic that consists of an AND-OR implementation of a sum-of-products expression, and a set of (input sequence, context) tuples that completely describe the behavior of the block of logic, the OR gate will only see single-input changes.*

Corollary 4.5.1, coupled with Theorem 4.5, allows the AND gates to be treated independently from the other gates in the circuit.

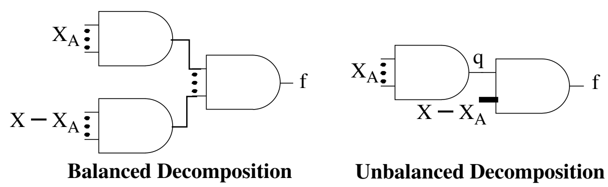
Let us explore the AND gates in more detail. We first look at conditions governing how we can split up the AND gate for a falling transition on its output. Next, we look at how we can split up the AND gate for rising transitions on its output. Finally, we look at the case where there are input changes, but the AND gate is to remain stable low. For all cases we only consider disjoint decompositions (i.e. where none of the input variables are replicated.)

Theorem 4.6 (Falling Transition of an AND gate) *Given an atomic AND gate with inputs X, a set of (input sequence; context) tuples resulting in a falling transition on its output, and a decomposition of the gate into two cascaded AND gates, where the inputs to the intermediate AND gate are $X_A \subset X$, the output of the intermediate AND gate is a new signal q, and the inputs to the final AND gate are $(X - X_A) \cup q$, then this decomposition is hazard-preserving for the set of (input sequence; context tuples) if and only if a transition on q for a given input sequence implies that no transition can occur on the inputs in $X - X_A$. Correspondingly, a transition on one or more inputs in $X - X_A$ for a given sequence implies that no transition can occur on q.*

Proof: $\Rightarrow$ By contradiction, suppose we start with a hazard-preserving decomposition, and suppose we have a transition on q and on some subset of inputs in $X - X_A$ for some input sequence that results in a falling transition on the final gate. Since the output is falling, all inputs to both AND gates must have been 1 at the start of the input sequence. If we have a transition on q then we know that one of the inputs in X_A must have fallen, causing q to fall at some time later. However, since at least one input $x_i \in (X - X_A)$ also falls, x_i can cause the output to go low before the transition on q is ever seen by the final AND gate, resulting in an unacknowledged transition, which eventually leads to a hazard (i.e. the transition on the internal node is never acknowledged). But our decomposition was assumed to be hazard-free.

$\Leftarrow$ Suppose we have a single-input change either on q or on some $x_i \in (X - X_A)$. Then, only one path is causing the circuit to transition to 0. Therefore, the transition will be acknowledged. Now, suppose we have several inputs in $X - X_A$ changing, while q remains stationary. Still no hazard will result, since a hazard in response to multiple-input changes would imply a hazard in the original circuit, and we assumed that the original circuit was hazard-free. ❑

This theorem establishes a rule that allows the gate to be recursively decomposed in a hazard-free way for the falling transitions. An initial decomposition of an AND gate according to the associative law induces a partition on the inputs of the design, splitting the single gate into two interconnected AND gates. One set of inputs goes to the intermediate AND gate, and the other set of inputs is connected directly to the decomposed gate together with the output of the intermediate AND gate. If both the output of the intermediate gate and the other inputs to the final gate can change during an input sequence, resulting in a falling transition on the final output, then a hazard can result. This theorem also implies that if a feasible partition is found that satisfies Theorem 4.6 for falling transitions, the gate can either be decomposed as a balanced tree, or as an unbalanced tree with one set of inputs connected directly to the final AND gate, and the other set connected to the intermediate AND gate, as illustrated below.



Let us now apply Theorem 4.6 to one of our previous examples. In Figure 4.9, there is a single (input sequence; context) tuple for *f*↓*: (w*↓*, y*↓*; x=1)*. If the gate is partitioned such that X_A={*w, x*} and $X - X_A = \{y\}$, then the transition on *w* causes both a transition on *y* and a transition on *q*, where *q* is the output of the intermediate AND gate, and therefore the decomposition is invalid. For the decomposition with X_A={*w*, *y*}, and $X - X_A = \{x\}$, because *x* is a context signal and remains high during the input sequence, the conditions of the theorem are satisfied and the decomposition is valid for the sequence.

f↓: (w↓, y↓; x)

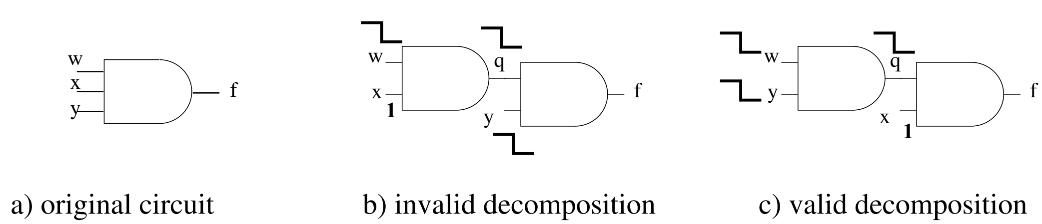


Figure 4.9: Illustration of Theorem 4.6: decompositions for a falling transition.

Corollary 4.6.1 *Given a hazard-free decomposition of an AND gate as defined by Theorem 4.6 and a set of (input sequence; context) tuples that result in falling transitions of the final AND gate, then the only valid decompositions for a given (input sequence; context) tuple in that set have the property that all signals that change during that input sequence are connected to a single gate.*

Proof: Suppose there is a valid decomposition where one of the signals in the input sequence is used as the input to a different gate than the others. Then, both the internal node and one or more inputs connected directly to the last AND gate are changing. But, this will result in a hazard contradicting our original assumption. ❑

Applying the corollary to the example in Figure 4.9, we can see that for the decomposition in Figure 4.9c, $X - \{context\ signals\} = \{w, x, y\} - \{x\} = \{w,y\}$, which are connected to a single AND gate, thus satisfying the conditions of the theorem.

Corollary 4.6.1 suggests that a simple filter can be created to quickly determine whether a gate can be decomposed. By taking the transitive closure of the partitions induced by each input sequence for the gate we can get decompositions which are acceptable for the falling sequences. If we cannot find any acceptable decompositions for the downgoing sequences, then the gate cannot be decomposed at all. For example, if {a, c} are in one partition, and {b, c} are in another, then {a, b, c} must be in the same partition for there not to be hazards on the downgoing edge of the AND gate. Thus no decomposition is possible in this case.

We can construct an algorithm to produce a decomposition which is valid for the downgoing transitions of an AND gate as follows:

```
procedure decompose_falling(falling_transitions){
    i = 0;
    foreach inputSequence in falling_transitions {
        partitions[++i] = vars_that_change(inputSequence);
    }
    num_partitions = i;
    while (changing(num_partitions)) {
        if (variables_overlap(partitions[i], partitions[j])) {
        combine_partitions(i, j);
        delete_partition(j);
        num_partitions--;
        }
    }
    return (partitions);
}
```

Now let us look at the conditions for the rising transition of an AND gate. There are several important points to consider for logic-hazard-free decompositions into cascaded AND gates. First, each AND gate will change only when its last arriving signal goes to “1”. This means that all inputs to an AND gate need to be “1” before it gate can change, and therefore any internal signals are automatically acknowledged by subsequent AND gates. Secondly, a subset of signals changing during a given input sequence can cause an intermediate AND gate to experience a dynamic hazard. This second point is the critical point for the decomposition, since otherwise we could decompose the original AND gate in any logic hazard-free way. In other words, the original gate must be decomposed such that the outputs of all intermediate gates change monotonically.

Theorem 4.7 (Rising Transition of an AND gate) *A decomposition of an atomic AND gate into two cascaded AND gates is hazard-free with respect to its set of rising (input sequence; context) tuples if and only if for each (input sequence, context) tuple in that set the output of the intermediate AND gate of the decomposed circuit makes at most a single transition.*

Proof: $\Rightarrow$ (By contradiction.) Suppose we have a hazard-free decomposition for the rising transitions of the original AND gate. Further suppose that the intermediate gate makes a non-monotonic transition for one of the input sequences for which the AND gate is to go high. Then, we know that the intermediate AND gate had to make a 1→0→1 transition, since it must be 1 at the end of the sequence for the final AND gate to reach its desired value. This non-monotonic transition represents a hazard on the intermediate AND gate which we know can propagate to the output. But we assumed that the decomposition was hazard-free, and thus our assumption is contradicted.

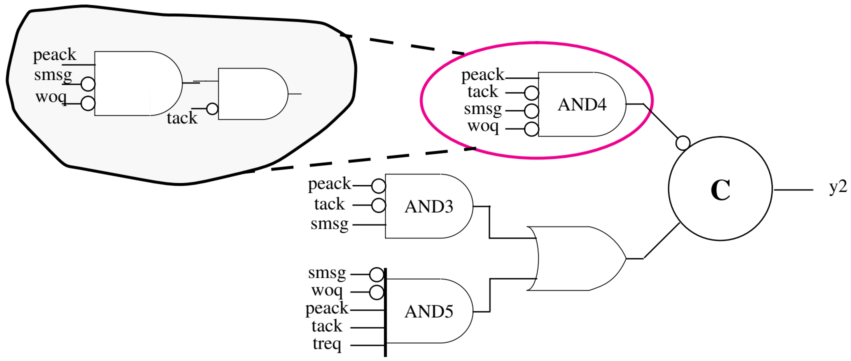
⇐ For an input sequence where the AND gate makes a rising transition, if the intermediate AND gate makes a single monotonic transition, then it must transition from 0→1 before the final AND gate can go high. Since the final AND gate cannot change until all its inputs are high, the monotonic transition is acknowledged by the final AND gate, and thus the decomposition is hazard-free. ❑

Because we know that all changes are automatically acknowledged by the gates in the fanout for the rising transition of an AND gate, the only way for there to be a hazard is if signals don't change monotonically. This observation leads to the following corollary.

Corollary 4.7.1 *Given the assumptions as in Theorem 4.7, if the transitions on the inputs of all intermediate AND gates are monotonic then the decomposition is hazard-free for the rising transitions of the AND gate.*

Given a proposed decomposition for rising sequences, we must examine the signals within the input sequence to determine whether the decomposition is valid. If each signal that is connected to the intermediate AND gate makes a single monotonic transition, then the intermediate AND gate will in turn make a single monotonic transition and the conditions of the theorem will be satisfied. However, if one of the signals connected to an intermediate AND gate changes non-monotonically, then there may be a problem, since the intermediate gate can transition high and then low before settling at its final value.

The following example illustrates the point. Referring to the decomposition in the inset of Figure 4.11, for sequence 1, the output of the intermediate AND gate in the decomposition is high at the beginning of the sequence, and then transitions to low when *peack* goes low. Finally, it goes high again when *smsg* goes low. Therefore, this decomposition is invalid since the sequence exercises a



(Input sequence; context signal) tuples:

y2↓: (AND4 rises)

1. (peack↓, smsg↑, {treq↓ || peack↑}, {tack↓ || smsg↓}; woq’) (AND5 falls, AND3 stays 0)

2. (peack↓, woq↑, treq↓, tack↓, peack↑, woq↓; smsg’) (AND5 falls, AND3 stays 0)

3. (peack↑, smsg↓; woq’, treq’, tack’) (AND3 falls, AND5 stays 0)

Figure 4.10: Hazardous decomposition of AND4 for rising transitions of the output

hazard on the intermediate AND gate. In this particular situation both *peack* and *smsg* changed non-monotonically, resulting in the hazard.

This example suggests the following corollary:

Corollary 4.7.2 *Given the assumptions as in Theorem 4.7, any decomposition which has all non-context signals connected to a single gate is hazard-free for the rising transition of the AND gate.*

Essentially Corollary 4.7.2 simply ensures that all gates which are changing are placed in a single gate. Since all context signals will be at a stable value enabling the function to go high, any hazards present in the decomposed circuit must also be in the original circuit.

Theorem 4.7 and its corollaries lead to the following algorithm for determining valid decompositions of the AND gate for rising transitions. We start with a set of decompositions that were extracted from the procedure following Theorem 4.6.

```
procedure decompose_rising(rising_transitions, decompositions) {
    foreach inputTuple in rising_transitions {
        foreach intermediate_gate in decompositions {
            if (nonmonotonic_change(intermediate_gate, inputTuple)) {
                reject_decomp(intermediate_gate, decompositions);
            }
        }
    }
    return(decompositions);
}
```

Since the procedure starts with the set of feasible decompositions determined by the falling transitions of the AND gate, we typically have only a few decompositions to examine.

The final situation we must examine is when there are a set of transitions on the inputs to an AND gate, but the AND gate remains low. Such a sequence will be referred to as a *stationary* sequence. In this situation, incorrectly decomposing the gate may lead to a static 0-hazard on the output of an intermediate gate.

Theorem 4.8 (AND gate remains stable) *A decomposition of an atomic AND gate into two cascaded AND gates is hazard-free with respect to the set of (input sequence; context) tuples where the AND gate output is to remain stable if and only if for each such (input sequence; context) tuple the output of the intermediate AND gate also does not change.*

Proof: $\Rightarrow$ Suppose we have a valid decomposition for an AND gate and a sequence of input changes during which the output of the AND gate must remain low. Suppose we have a transition on the output of an intermediate AND gate. Then it will not be acknowledged and a glitch can result from a later input sequence. But we said that the decomposition was valid, and this contradicts our assumption.

$\Leftarrow$ Suppose the output of the intermediate AND gate does not change for all (input sequence; context) tuples where the original AND gate is to remain low. Then it is clear that the final AND gate cannot change, and that we will observe the desired behavior. Thus, it is a valid decomposition. ❑

By this theorem, then, we can have valid decompositions in several scenarios. If the intermediate AND gate has as its input a context signal for each sequence where it is to remain low, the AND gate cannot transition, and the decomposition will be valid. However, even if all inputs to the intermediate AND gate change, the decomposition can still be valid, depending on the ordering of the signals.

The algorithm below is used to determine the valid decompositions for the stationary sequences of an AND gate.

```
procedure decompose_nochange(stationary_transitions, decompositions) {
    foreach inputTuple in stationary_transitions {
        foreach intermediate_gate in decompositions {
            if (changes(intermediate_gate, inputTuple)) {
                reject_decomp(intermediate_gate, decompositions);
            }
        }
    }
    return(decompositions);
}
```

Having covered all cases for the AND gate, we can state the overall decomposition theorem.

Theorem 4.9 *The circuit produced by decomposing an AND gate in accordance with the constraints imposed by Corollary 4.6.1 and Theorems 4.7 and 4.8 is hazard-free.*

Proof: Follows from proofs of the theorems. ❑

Having established the key theorems for decomposition of AND gates, we can now analyze the set of (input sequence; context) tuples for a given output to determine whether the AND gate can be decomposed. Any conflicts between the constraints imposed by the individual theorems implies that the gate cannot be decomposed.

We are now ready to review the overall decomposition algorithm.

4.4.4.1 Basic Decomposition Algorithm for Speed-Independent Design

The basic decomposition algorithm works as follows:

```
procedure decompose(input sequences, gate-level implementation):
foreach output {
    foreach block of combinational logic {
        foreach gate in block{
            if (gate == AND) {                                          /* AND gate */
                (falling, rising, nochange) = partition(input sequences, inputs(gate));
                falling_decomps =decomp_falling(gate, falling);
                rising_decomps = decomp_rising(gate, rising);
                stationary_decomps = decomp_nochange(gate, nochange);

                decompositions = intersection_of(falling_decomps, rising_decomps,
                    stationary_decomps);
            }
            else {                                                      /* OR gate */
                decompositions = fundmodedecomp (gate);
            }
        }
    }
}
```

The input sequences which are one of the arguments to the decomposition procedure are derived from a state graph which has been extracted from the STG. Although there are a finite number of distinct input sequences for any STG, the number of input sequences can be quite large for an entire design. However, since the number of distinct input sequences for any given logic block or gate is small, by performing a filtering operation before extracting the sequences, we can substantially reduce the number of sequences that must be handled. As a result, the procedure is quite efficient.

Results from Simple Decomposition Procedure

Design	# of gates	# of AND gates (>= 3 inputs) able to decompose	# of AND gates (>= 3 inputs) unable to decompose
a_to_d_controller	6	2 AND3	0
chu133	6	2 AND3	0
chu150	7	0	2 AND3
converta	13	0	2 AND3
dff	6	2 AND3	0
ebergen	9	0	2 AND3
half	6	0	1 AND3
hazard	6	2 AND3	0
mp-forward-pkt	10	2 AND4	0
nak-pa	10	0	1 AND3
nowick	11	2 AND3	0
pe-rcv-ifc	36	5 AND3, 2 AND4	4 AND3, 1 AND4, 2 AND5
qr42	9	0	2 AND3
ram-read-sbuf	11	2 AND3	0
rcv-setup	3	1 AND4	0
rpdft	4	1 AND3, 3 AND4	0
sbuf-ram-write	11	3 AND3	1 AND3
sbuf-send-pkt2	13	2 AND3	1 AND3
two_phase_fifo	7	0	2 AND3
vbe5b	6	0	1 AND3
vbe5c	4	0	1 AND3

Table 4.1 Results from simple decomposition procedure

We ran the simple decomposition procedure on a number of examples from the asynchronous community, as shown in Table 4.1. Examining the results, we see that more than half of the AND3 gates in the designs were decomposable, as were all but one of the AND4 gates in the designs. This means that by using a library containing AND gates with three or fewer inputs, we were able to successfully decompose over 95% of the designs. For about half the designs we were also able to decompose all AND3 gates. Although successful decomposition of AND3s does not affect the implementability of the designs, it will have a positive impact on the quality of the eventual mapped circuit. By decomposing the gates into finer granularity functions it is possible that a better quality cover of the design can be found during the matching/covering step.

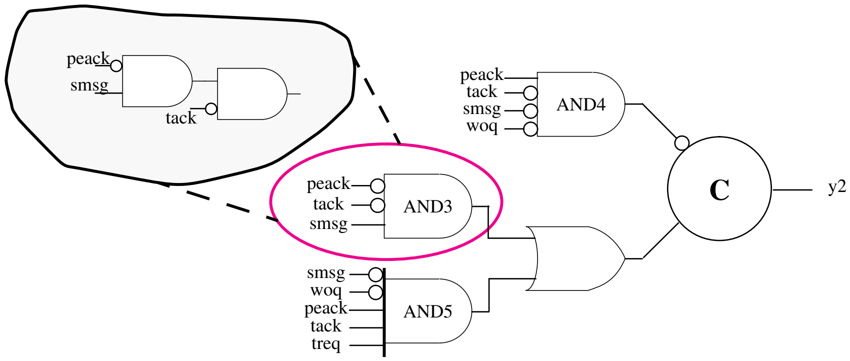
The only design that was not decomposable for gates of size four or greater was *pe-rcv-ifc*. The next section examines the source of the failures, and proposes a method to successfully decompose these gates.

4.4.4.2 Extensions to Decomposition to Enforce Sequencing

As we saw from the results, there were many gates we could not decompose. Typically, this only has an impact on the quality of the eventual cover. It only affects feasibility when the gate to be decomposed is not in the library. In the `pe-rcv-ifc` example, we saw that there was one gate that could not be decomposed that was unlikely to exist in a library. Before solving the problem, we will examine the source of the decomposition failures by trying to decompose the gates in a portion of the `pe-rcv-ifc` circuit, shown in Figure 4.11. This example is taken from the circuit [20].

Let us try to decompose the instance AND5, since it is the most difficult. For y2↓, AND5 will stay low during sequence 3, and will transition low for sequences 1 and 2. By Corollary 4.6.1, we know from sequence 2 that {*peack, woq, treq, tack*} must be used as inputs to the same gate, and from sequence 1 that {*peack, smsg, treq, tack*} must be put in the same gate. These two partitions overlap, and therefore we must include the signals {*peack, woq, treq, tack, smsg*} in the same gate, which implies that the gate cannot be decomposed. We do not need to look at any other input sequences at this point.

Instance AND3 may be easier to decompose. Instance AND3 falls during sequence 3. From Corollary 4.6.1, only {*peack, smsg*} must be used as inputs to the same gate. This leads to the one possible decomposition (modulo permutations of the inputs), which is shown in the inset in Figure 4.11. Instance AND3 will rise during input sequence 5. Since the changes on the inputs are monotonic during that sequence, any decomposition will do (by Theorem 4.7), so the previous decomposition also works for the rising transition. AND3 will stay low during input sequences 1, 2, 4 and 6.



(Input sequence; context signal) tuples:

y2↓: (AND4 rises)

1. (peack↓, smsg↑, {treq↓ || peack↑}, {tack↓ || smsg↓}; woq') (AND5 falls, AND3 stays 0)

2. (peack↓, woq↑, treq↓, tack↓, peack↑, woq↓; smsg') (AND5 falls, AND3 stays 0)

3. (peack↑, smsg↓; woq', treq', tack') (AND3 falls, AND5 stays 0)

y2↑: (AND4 falls)

4. (peack↓, woq↑, treq↑, tack↑, peack↑, woq↓; smsg') (AND5 rises, AND3 stays 0)

5. (peack↓, smsg↑; woq', tack', treq') (AND3 rises, AND5 stays 0)

6. ({peack↓ || treq↑}, tack↑, treq↓, tack↓, woq↑, treq↑, tack↑, peack↑, woq↓; smsg') (AND5 rises, AND3 stays 0)

Figure 4.11: Example of decomposition process

Unfortunately, if we look at input sequence 1, we see that the intermediate AND gate will undergo a non-monotonic transition, so we cannot decompose AND3 at all.

Finally, let us take a look at instance AND4. For this gate, when y2↑, AND4 falls. By Theorem 4.6, we see that from sequence 4, {*peack, woq, tack*} must be placed in the same gate. From sequence 5, we must include {*peack, smsg*} in the same gate, so we know that the gate cannot be decomposed further.

From this example we see that none of the gates were decomposable by the conditions of the theorems. This motivates us to look for ways in which we can eliminate some of the hazards introduced by the decompositions.

There are two fundamental causes for decomposition failures. One is where the decomposition no longer preserves the ordering of the signals, resulting in a sequencing hazard in the combinational logic. The second situation is where an unacknowledged transition leads to a sequential hazard.

Suppose, for example, that an intermediate AND gate has a sequencing static 0-hazard for some sequence. This situation can be prevented by adding a signal as an input to the AND gate to keep the gate low during transitions in the other signals that are connected to it. This must be done without otherwise disturbing the functionality of the gate. (This is referred to as a *term takeover* by several researchers [4], [84].) However, we are attempting to decrease the fanin of a gate by *decomposing* it into an interconnection of smaller gates, so adding inputs to a decomposed gate will make it difficult for the decomposition process to converge.

Suppose instead that we have a monotonic transition on the intermediate gate that is not acknowledged by a gate in the output logic block. If we can ensure that it is acknowledged somewhere else in the circuit *before* the block's output changes, then we can enforce the ordering requirement we had previously. Beerel refers to such connections as *acknowledgment wire forks* [4], and proposed their use during decomposition in [5].

Returning to the hazardous decompositions of the example in Figure 4.11, we can trace the failures on the falling transition of an AND gate to sequential hazards. The failures on the rising transition of the AND gate and on the stationary sequences of the AND gate were sequencing hazards. We want to use the data we have collected about the restrictiveness of the various requirements to drive our search for a suitable fix.

We want to modify our decomposition procedure for those gates which we cannot decompose further and for which we cannot find a suitable gate replacement in our library. In such cases, we will need to insure that we pick a decomposition such that the intermediate node is acknowledged somewhere in the circuit before the first change in the next input sequence to that block is applied. Thus, we are sure that there will be no hazard resulting from the intermediate signal changing too late.

a) Decomposition valid for sequence 2 **b)** Decomposition valid for sequence 1

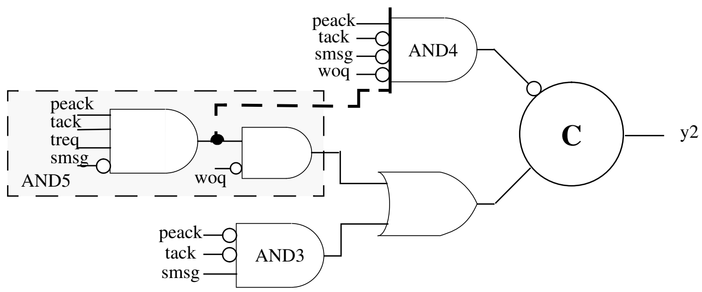
Figure 4.12: Decompositions for AND5 with sequential hazards

Recall that in the example presented earlier (Figure 4.11 on page 124) we were unable to decompose the two larger gates because of the requirements of Theorem 4.6 for the falling transition of an AND gate. In these cases, we encountered sequential hazards which prevented us from decomposing the gates. Let us now try to remove the sequential hazard in the largest AND gate by searching for an appropriate place to add an acknowledgment wire fork.

Let us try to decompose AND5. There were two possible decompositions of this gate where a sequential hazard arose in only a single sequence. These decompositions are shown in Figure 4.12. When we looked at these decompositions previously, because we could not decompose the gate without introducing hazards for the falling sequences, we did not try to decompose the gate for the remaining sequences. We will start with these decompositions and check the conditions for the rising and stationary sequences. If the decomposition is hazard-free for those sequences then we will try to add a wire fork to remove the sequential hazard.

For the decomposition shown in Figure 4.12a, examining sequence 1 we find that there is a function hazard on the intermediate AND gate caused by *peack* changing twice while the other inputs to the gate are still high. Therefore, this decomposition is unacceptable. On the other hand, the decomposition shown in Figure 4.12b is valid for the conditions imposed by Theorems 4.7 and 4.8, since for all input sequences, the intermediate gate makes at most a single monotonic change.

Having selected a decomposition of AND5 which has a single sequential hazard (see Figure 4.13), we must find a place to acknowledge the transition on the intermediate AND gate during sequence 1 to remove the hazard. Noting that whenever the intermediate AND gate falls, AND4 rises (due to



(Input sequence; context signal) tuples:

y2↓: (AND4 rises)

1. (peack↓, smsg↑, {treq↓ || peack↑}, {tack↓ || smsg↓}; woq') (AND5 falls, AND3 stays 0)

2. (peack↓, woq↑, treq↓, tack↓, peack↑, woq↓; smsg') (AND5 falls, AND3 stays 0)

3. (peack↑, smsg↓; woq', treq', tack') (AND3 falls, AND5 stays 0)

y2↑: (AND4 falls)

4. (peack↓, woq↑, treq↑, tack↑, peack↑, woq↓; smsg') (AND5 rises, AND3 stays 0)

5. (peack↓, smsg↑; woq', tack', treq') (AND3 rises, AND5 stays 0)

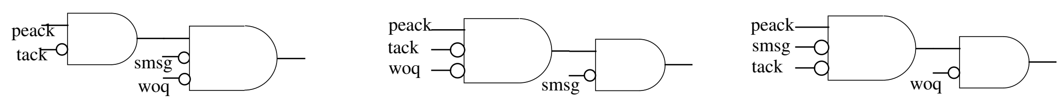
6. ({peack↓ || treq↑}, tack↑, treq↓, tack↓, woq↑, treq↑, tack↑, peack↑, woq↓; smsg') (AND5 rises, AND3 stays 0)

Figure 4.13: Decomposition for AND5 with sequential hazard

the push-pull nature of an output logic block), we can use the AND4 to acknowledge the signal. Adding the wire fork as another input to AND4 (as indicated by the dashed line) creates a new five-input AND gate which we must now try to decompose by the same procedure.

Examining the original AND4, we see that there are three decompositions of AND4 that have no non-monotonic changes on the intermediate gate, as shown in Figure 4.14. Decomposition 1 has a sequential hazard for input sequences 4, 5 and 6. Decomposition 2 has a sequential hazard for input sequence 5. Decomposition 3 has a sequential hazard for input sequences 4 and 6. Given that information, we must look for sequences that will acknowledge the changes on the intermediate AND gates locally to remove the sequential hazard.

The final AND gate in the decomposed AND5 will acknowledge the change for sequences 4 and 6, since it rises during those sequences. However, since AND5 does not change during sequence 5,

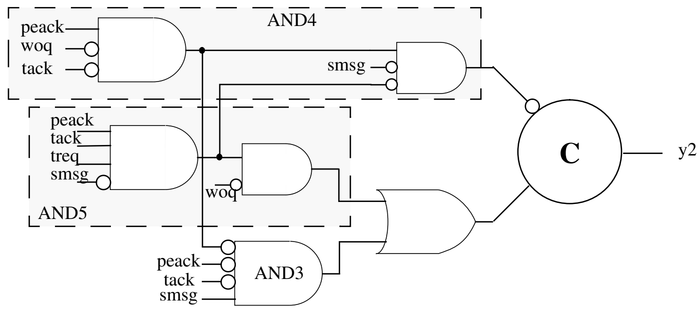


a) decomposition 1 **b)** decomposition 2 **c)** decomposition 3

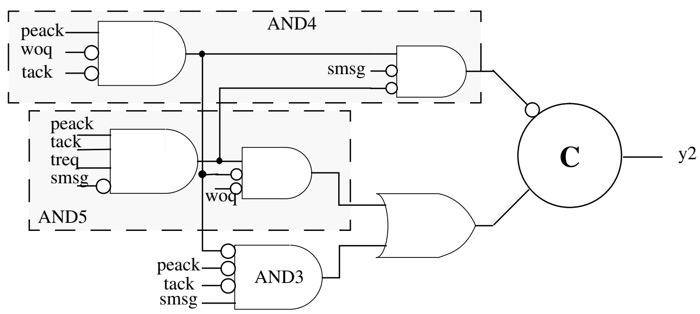
Figure 4.14: Possible function-hazard-free decompositions for AND4 (with one sequential hazard)

we must use AND3 to acknowledge any changes during that sequence. Thus, if we were to select decomposition 1, we would need to connect the intermediate AND gate of the decomposition for AND4 to both AND3 and the decomposed AND5 to properly acknowledge the change. For decomposition 2, AND3 will acknowledge the transition during sequence 5. For decomposition 3, AND5 will acknowledge the transitions during sequences 4 and 6. Figure 4.15 shows the two simpler hazard-free decompositions.

Let us verify whether either of these decompositions are hazard-free. Theorems 4.7 and 4.8 and satisfied since we chose only decompositions where the intermediate AND gate makes at most a monotonic change during each input sequence. Furthermore, we know that the intermediate gate for the decomposition of AND4 in Figure 4.15a is properly acknowledged because AND3 rises during sequence 5 when the intermediate gate in the decomposed AND4 falls. Since the change is monotonic, it cannot introduce a new hazard into the circuit at AND3. Unfortunately, we now have changed the conditions of Theorem 4.6 for the decomposed AND4 by adding in the new input to acknowledge the decomposed AND5, and we must reexamine whether condition 1 of the theorem still holds. In sequences 4 and 6, the intermediate AND gate in AND4 also falls, but since AND3 does not change, we have a problem because the intermediate change is no longer acknowledged. We must therefore add an additional wire fork to the final AND gate of the AND5 decomposition to acknowledge the downgoing transition for those sequences. Because we have done this, we must check it to see if the conditions of Theorem 4.6 still hold. Since AND4 rises in those sequences where AND5 falls (sequences 1 and 2), the changes are acknowledged. A correct decomposition is thus shown in Figure 4.15b. However, a simpler decomposition exists.



a) Initial decomposition using AND3 to acknowledge the decomposed AND4



b) Hazard-free decomposition using AND3 to acknowledge the decomposed AND4

(Input sequence; context signal) tuples:

y2↓: (AND4 rises)

1. (peack↓, smsg↑, {treq↓ || peack↑}, {tack↓ || smsg↓}; woq’)
(AND5 falls, AND3 stays 0)

2. (peack↓, woq↑, treq↓, tack↓, peack↑, woq↓; smsg’)
(AND5 falls, AND3 stays 0)

3. (peack↑, smsg↓; woq’, treq’, tack’)
(AND3 falls, AND5 stays 0)

y2↑: (AND4 falls)

4. (peack↓, woq↑, treq↑, tack↑, peack↑, woq↓; smsg’)
(AND5 rises, AND3 stays 0)

5. (peack↓, smsg↑; woq’, tack’, treq’)
(AND3 rises, AND5 stays 0)

6. ({peack↓ || treq↑}, tack↑, treq↓, tack↓, woq↑, treq↑, tack↑, peack↑, woq↓; smsg’) (AND5 rises, AND3 stays 0)

Figure 4.15: Decomposition using AND3 to acknowledge AND4

Looking in Figure 4.16 we see that instead of using AND3 to acknowledge the change, we can use AND5, where we have mutual acknowledgment. This is the decomposition we will choose.

We can now develop the theory to go with this example. Our general decomposition procedure must be modified to accommodate these changes.

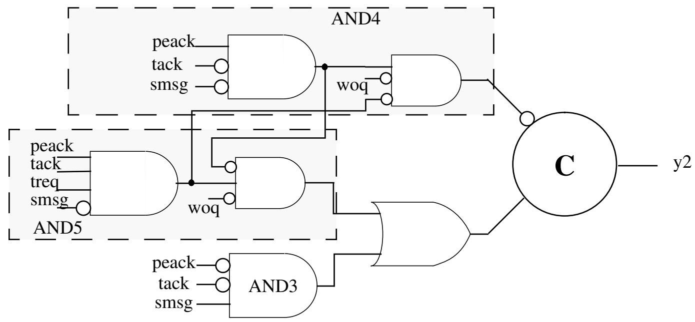


Figure 4.16: Decomposition using AND5 to acknowledge the decomposed AND4

Recall that an output logic block has a two-input C-element driven by separate blocks of combinational logic. During any input sequence, we have one block of combinational logic that is rising and one that is falling.

Theorem 4.10 *Given a decomposed AND gate within the combinational portion of an output logic block that satisfies the conditions of Theorems 4.7 and 4.8, the falling transitions of the intermediate AND gate can be acknowledged by adding an inverted input to one or more AND gates in the combinational logic connected to the other input of the C-element and the resulting circuit will be hazard-free.*

Proof: We need to prove that the connection we added correctly acknowledges the falling sequence, that the added connection does not change the circuit's function and that no hazards are introduced.

The connection correctly acknowledges the falling edge of the intermediate AND gate, since when it is falling, one of the AND gates to which it is connected in the complementary logic block is rising (by definition of an output logic block). Since the AND gate in the complementary logic block cannot transition to 1 until all its inputs are 1, then it must wait for the transition from the intermediate AND gate to change before it change, thus acknowledging that transition.

We now must show that adding the connection to the complementary AND gate does not change the function of the circuit. We know that when the output of the intermediate AND gate is 1, then any AND gate in the complementary circuit must be 0 (by definition of the output logic block). Therefore, connecting the output of the intermediate AND gate to the input of the complementary AND gate does not change the function for that case. When the output of the intermediate AND gate is 0, then the complementary AND gate can be either 0 or 1. In the case where the complementary AND gate's output is to be 1, the input from the intermediate AND gate is also 1, so there is no change in function. In the case where the complementary gate's output is to be 0, the only way we can get an erroneous output is if all the other inputs of the complementary AND gate were 1 as well. But if that were the case, then the complementary AND gate's output would be 1 and not 0 as we supposed. Therefore there is no functional change by adding the connection to the intermediate AND gate.

Finally, we must show that we introduce no new hazards by adding the connection. To have a static 0-hazard on the complementary gate, we would need all inputs to be 1 momentarily during an input sequence. Adding an input to a hazard-free gate cannot add a static 0-hazard, since that would imply that a static 0-hazard existed in the original gate, contradicting our hazard-free assumption. To add a static 1-hazard would imply that the gate would momentarily transition to 0 before going back to 1. Since the gate was initially hazard-free, this means that once all the initial inputs change to 1, the gate stays at 1. The addition of the new connection cannot change the hazard-behavior, since its value is guaranteed to change monotonically. Therefore adding the connection will not change the hazard-behavior of the circuit. ❑

Extended Decomposition Procedure

Now that we have demonstrated where a connection can be added, we can modify our decomposition procedure to take advantage of the situations where we cannot decompose a gate by the simpler method.

```
if (decompositions == NULL) {
    gates_not_in_library = undecomposable gate;
    while (gates_not_in_library) {
        gates_not_in_library = extended_decomp(gates_not_in_library, decompositions);
    }
}
procedure extended_decomp(gates_not_in_library, decompositions, sequences) {
    gate_decomps = find_all_fhf_decomps(gate);
    foreach intermediate_gate in (gate_decomps) {
        hazardous_sequences = find_sequential_hazards (intermediate_gate,
            falling_transitions);
        acknowledging_gates = find_rising_gates(hazardous_sequences);
        connect(intermediate_gate, acknowledging_gates);
                                        /* Must check to see if we have exceeded the
                                         fanin of rising gates by adding connection */
        foreach (acknowledging_gates ⊄ library) {
            gates_not_in_library += acknowledging_gates;
        }
    }
}
```

With this extended decomposition procedure the AND5 gate can be decomposed in the design at the penalty of adding two wire interconnections to intermediate gates. The final decomposed `pe-rcv-ifc` design has 14 AND3s, two AND4s and a number of AND2s.

The full decomposition procedure is shown in Figure 4.17.

The algorithms we described were implemented in approximately 5000 lines of C code, using Tcl and Tk as the user-interface [70]. All examples ran very quickly on an HP 9000/730.

4.4.4.3 Failure to Decompose Under the Extended Decomposition Procedure

We cannot guarantee that the extended decomposition procedure will always work. As mentioned earlier, we only invoke the extended decomposition procedure if the gate does not exist in the library, so the number of cases in which we will need to invoke the procedure is already very small. Should we fail to decompose, however, we can look more globally within the design for a

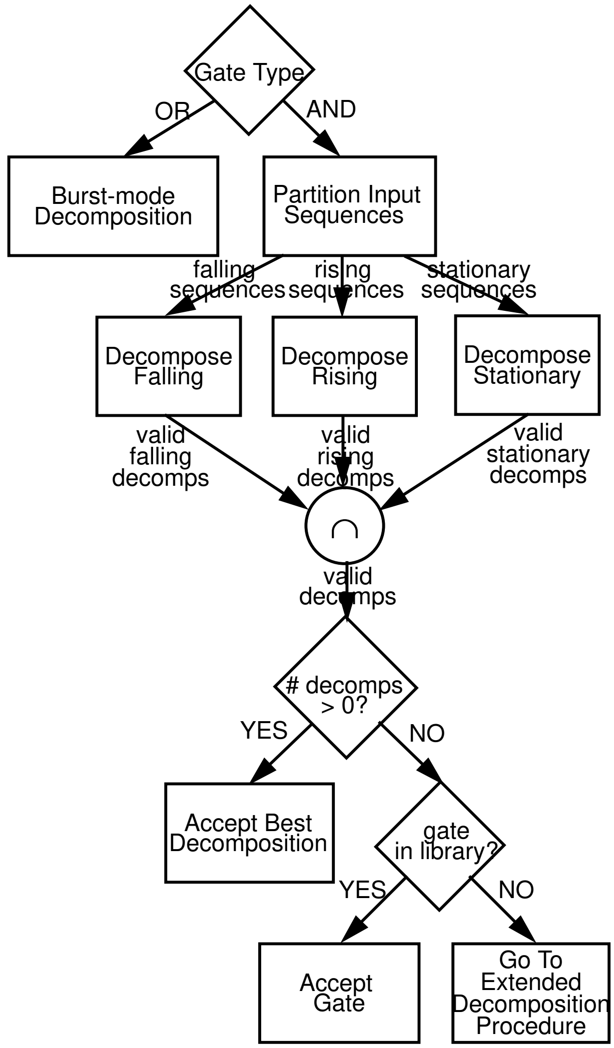


Figure 4.17: Overall decomposition algorithm

place to add an acknowledgment wire fork, as Beerel does [5], at the expense of reducing concurrency and adding large points of multifanout. Alternatively, we can modify the original specification and add state variables as necessary to split up the problem, as Myers does in [63]. Although our algorithm does not currently use either of these methods, it can be easily modified to do so.

4.4.5 Matching/Covering

The remaining step in the technology-mapping procedure is the matching/covering step. First, we must establish whether or not the burst-mode matching/covering procedure will also work for speed-independent designs.

Proposition 4.11 *Given a decomposed network, we can replace portions of the network by larger hazard-free atomic gates without introducing hazards into the network.*

Proposition 4.12 *Introduction of logic hazards during the covering portion of the technology mapping process will result in hazards on the primary outputs.*

These propositions indicate that we may be able to use existing synchronous matching/covering algorithms modified as for the burst-mode mapper in the mapper for speed-independent designs. The only qualification is that the covering routine cannot modify the ordering of signals through certain portions of the circuit. This implies a heuristic like the *inverter-pair* heuristic[1] [73] cannot be used on the primary inputs (although it can within the circuit) because its application can affect the ordering of signal propagation through the network.

We would like to take advantage of the relatively small size of the speed-independent designs to find a better quality cover, even though by modifying the inverter-pair heuristic within the burst-mode mapper to disallow addition of inverter pairs on the inputs, we can use the burst-mode mapper's matching and covering routines to map speed-independent circuits. Because the decomposition procedure described in the previous section will not, in general, break the initial set of gates into two-input one-output base functions, we will have fewer matches to work with during the covering algorithm. We can address the covering problem more globally by allowing sharing

1. The inverter-pair heuristic is a heuristic used by some mappers to improve the quality of the mapped circuit by introducing pairs of inverters at each gate output. This does not affect the behavior of the network, and allows further optimization by making both phases of a signal available, at a slight penalty in computation cost. For a speed-independent design, because inverters are single-input single-output gates, adding them does not affect the relative signal ordering within the internals of a logic block. However, if inverter pairs are introduced at the inputs to the logic block, input signal ordering will no longer be maintained and hazards could result, so this must be avoided.

across cones of logic. The small size of the circuits coupled with the smaller number of matches suggests that a global covering algorithm can be employed to take advantage of any sharing between gates across output logic blocks.

We will start with a covering formulation similar to that used by other researchers [73, 47]. In particular, we formulate the covering problem as a *binate equation* whose minimum-cost satisfaction represents the optimal cover of the network by elements from the library. Because the binate-covering problem is NP-complete, an exact solution for mapping of an entire synchronous design is, in general, intractable [26]. As a result, previous mappers have focused on solving the problem for individual subnetworks of the complete design, yielding an eventual solution that is suboptimal. Because we will have far fewer clauses in our equation, and fewer variables in the clauses, we will formulate the problem *globally* and solve the problem exactly over the global design to result in an optimal solution (given the particular decomposition).

In our problem formulation, we represent each successful match of a subnetwork by an element from the library as a binary variable with an associated cost. For a given node in our decomposed design we can then represent the set of possible matches by a clause whose satisfaction is assured by setting at least one of the variables to 1 (i.e., binding a library element to that node as part of the solution). In addition to ensuring that each node in the subnetwork is covered by an element, we must ensure that if an element is selected to cover a particular node or set of nodes, that the inputs to the nodes will also be generated. As a result, we must formulate a set of binate clauses representing the implications that will cause the inputs for a particular match to be generated (assuming they are not primary inputs). These two sets of clauses are then combined into a product-of-sums expression whose satisfaction represents successful binding of library elements to the nodes in the unmapped network to implement the network. Our formulation differs from prior formulations in that we recognize shared logic as we compose the clauses, since we are doing the binding globally. An example best illustrates the formulation.

Example 4.1

Figure 4.18 shows a portion of the `pe-rcv-ifc` circuit that has been decomposed by the `si_tech_decomp` procedure. Each set of gates is annotated with the set of possible

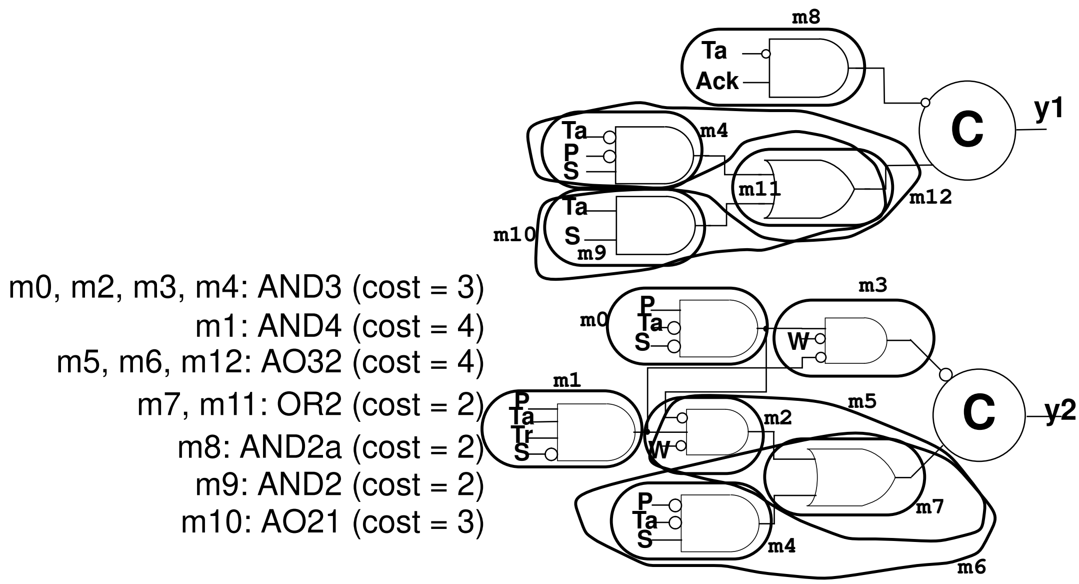


Figure 4.18: Illustration of covering algorithm (section of pe-rcv-ifc circuit).

matches that represent its functionality. (In the library, AO21 and AO32 are two types of and-or-invert gates.) We label each match with a variable to indicate the match, and then form the covering clauses,

```
(m0)(m2 + m5)(m3)(m1)(m7+ m5 + m6)(m4 + m6)(m8)(m11 + m10 + m12)
        (m12 + m4)(m9 + m10)
```

and the implications,

```
(m7′ + m4)(m7′ + m2)(m11′ + m4)(m11′ + m9)
```

to ensure that if a match is chosen that requires an internal input to be available, then that internal input will be generated. In this formulation we recognize that the gate represented by `m4` is shared across cones of logic, so we only assign a single variable to that gate.

With these two sets of clauses, we can formulate the following binate equation:

```
(m0)(m2 + m5)(m3)(m1)(m7 + m5 + m6)(m4 + m6)(m8)(m11 + m10 + m12)
(m12 + m4)(m9 + m10)(m7′ + m4)(m7′ + m2)(m11′ + m4)(m11′ + m9) = 1
```

Note that each variable in the equation has a cost associated with it, and this cost will be taken into account during solution of the equation. ❑

Our formulation then can be represented in pseudocode as follows:

```
procedure form_covering_equation(decomposed_network) {
    foreach cone in decomposed_network {
                    /* generate all matches for the cone of logic noting that the matching routine will
                       hash the matches; ensure that redundant matches are avoided              */
        clauses += async_matching_routine(cone, library) {
    }
    foreach match in clauses {
        implications += create_implications(match);
    }
                    /* min-cost satisfaction of covering_equation is optimal cover */
    covering_equation = conjunction_of(clauses, implications);
    return(covering_equation);
}
```

To formulate the covering equation, we will still need to use the hazard-non-increasing matching algorithms presented in the previous chapter to avoid introduction of logic hazards. The procedure above thus works as follows:

1. Generate all possible hazard-free matches for combinational elements in the design. Store each match in a table indexed by its inputs and function.
2. Generate covering clauses (1 per gate) and implication clauses (to ensure that inputs to sub-networks are generated).

As we mentioned earlier, although the binate covering problem we have formulated is NP-complete, we can still obtain an exact solution for it. Because the granularity of the decomposition is coarse, and the cones of logic are small, the exact solution will be tractable. As the size of speed-independent designs increase, heuristics may be necessary to speed the solution. Assuming we were able to successfully decompose the design (by the procedure in the previous section), we are guaranteed the existence of an initial solution, because the decomposed network consists of elements which are in the library. So the initial solution is found from the one-to-one mapping between each element in the decomposed design and gates from the library.

Example 4.2

In the circuit of Figure 4.18 the initial solution is found by taking a one-to-one mapping between the decomposed circuit elements and gates from the library. In this particular example, the circuit can be implemented by two 2-input ANDs, five 3-input ANDs, 1 4-input AND, and two 2-input ORs for a total cost of 27. This is the initial solution we will start with. ❑

We use a branch-and-bound technique [22] to solve the binate covering problem. This works well because there is little overlap between the clauses in the binate equation. Variable selection at each branching point is done iteratively as follows:

1. Choose variables that appear in essential clauses (clauses with one variable).
2. Choose the binate variable that appears in the largest number of clauses.
3. Choose the unate variable that appears in the largest number of clauses.

The essential clauses are those clauses with cardinality one; i.e., these are the variables that must be chosen to satisfy the equation. Recalling the equation formulated in our earlier example:

```
(m0)(m2 + m5)(m3)(m1)(m7 + m5 + m6)(m4 + m6)(m8)(m11 + m10 + m12)
(m12 + m4)(m9 + m10)(m7′ + m4)(m7′ + m2)(m11′ + m4)(m11′ + m9) = 1
```

we can see that `m0`, `m1`, `m3` and `m8` are all essential variables. Although in this case (our initial equation) selection of these variables does not simplify the remaining clauses, in many cases it does, either by reducing the number of terms in a particular clause or by eliminating the clause through its successful satisfaction. In many cases, selection of the essential variables will result in the reduction of clauses to size one, creating new essential variables which can in turn be used to satisfy more clauses.

We next choose a binate variable because of its potential for satisfying a large number of clauses. For example, in the equation above, choosing `m7` leads to `m2` and `m4` becoming essential variables, which in turn eliminates other clauses. Conversely, choosing `m7'` (i.e., not selecting the match), eliminates two clauses and reduces the size of one of the clauses.

Finally, once all binate variables have been selected, the algorithm chooses the unate variable with the largest number of instances, again with the intent of reducing the expression as quickly as possible. Note that this variable selection procedure is called iteratively after each variable is selected, since at any stage the equation may include new essential variables.

We can now present the complete covering algorithm.

```
procedure find_best_si_cover(decomposed_network, library) {
    clauses = form_covering_equation(decomposed_network) {
    MinCost = cost_of(decomposed_network);                /* initialize best solution found so far */
    best_solution = branch_n_bound(clauses, 0);
}

procedure branch_n_bound(clauses, incremental_cost)
    if (incremental_cost >= MinCost)                                          /* bounding function */
        return;
    if (number_of(Variables) > 0) {
        branch_variable = find_best_variable(clauses);
        original_clauses = clauses;
        original_cost = incremental_cost;
        Variables -= branch_variable;
                                  /* satisfy reduces clauses by the clause(s) that is (are) satisfied */
        incremental_cost += satisfy(clauses, branch_variable);                  /* choose the match */
        branch_n_bound(clauses, incremental_cost);
                                  /* now take the complement (i.e., choose not to take the match) */
        satisfy(original_clauses, not(branch_variable));                       /* cost doesn't change */
        branch_n_bound(modified_clauses, original_cost);
    }
    else {
        record_solution(incremental_cost);
        MinCost = incremental_cost;                  /* update the cost to reflect the new solution */
    }
}
```

The procedure `find_best_si_cover` is called by the mapper. First, the binate equation is formulated as described earlier. The variable `MinCost` is initialized to the cost of the solution found by taking a one-to-one mapping of elements of the decomposed network to single gates from the library. The cost of the current solution is initialized at zero. Then, the branch-and-bound algorithm is called to satisfy the equation. In the branch-and-bound procedure, `find_best_variable` is called to select the next variable to choose, with the variable selection proceeding as described earlier. Then, `satisfy` is called to simplify the equation by reflecting the choice of variable. As variables are selected, the incremental cost is updated. If at any point the incremental cost exceeds the cost of an already chosen solution, that path in the virtual solution tree is abandoned.

In our algorithm, we first try the solution found by selecting the match, and then try the solution found by not selecting the match. (We chose to select a match first because it generally leads to reduction of more clauses then choosing to not select a match.)

This algorithm is best illustrated by continuing with the example we used earlier.

Example 4.3

We will use the branch-and-bound algorithm to solve the earlier equation. Figure 4.19 illustrates the traversal taken by the branch-and-bound algorithm during its operation. After selecting the four essential variables, the initial cost of our solution is 15. (We will only consider the cost of the non-essential variables for the remainder of this example, since we must include the essential variables in all solutions.) We next choose the match represented by variable `m7`. This reduces the clauses so that `m2` and `m4` are now essential, so we select those next. Selecting `m11` then reduces the equation to make `m9` essential, and we have our first solution at a cost of 12 plus the original essential matches. Returning, the procedure next decides to not choose `m11`, following which the choice of `m10` is taken leading to a solution whose cost is less than the first choice, at a cost of 11. We select this as our best cover so far. The procedure continues as depicted in the tree in Figure 4.19.

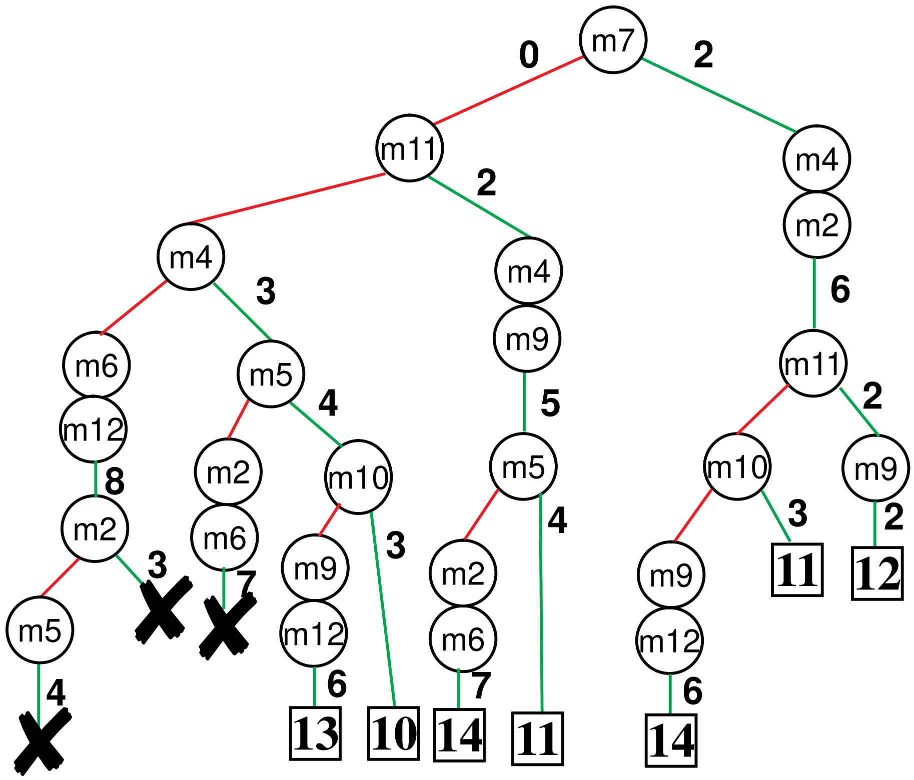


Figure 4.19: Branch-and-bound solution to the binate covering problem.

Whenever the minimum cost is exceeded by selecting a match, the search along that branch terminates without further exploration. This can be seen in Figure 4.19 by observing those branches that have X's as their leaves.

The optimal solution found is illustrated in Figure 4.20. This solution resulted in sharing one of the matches across the two groups of combinational logic (as seen by the shaded gate in the figure). ❑

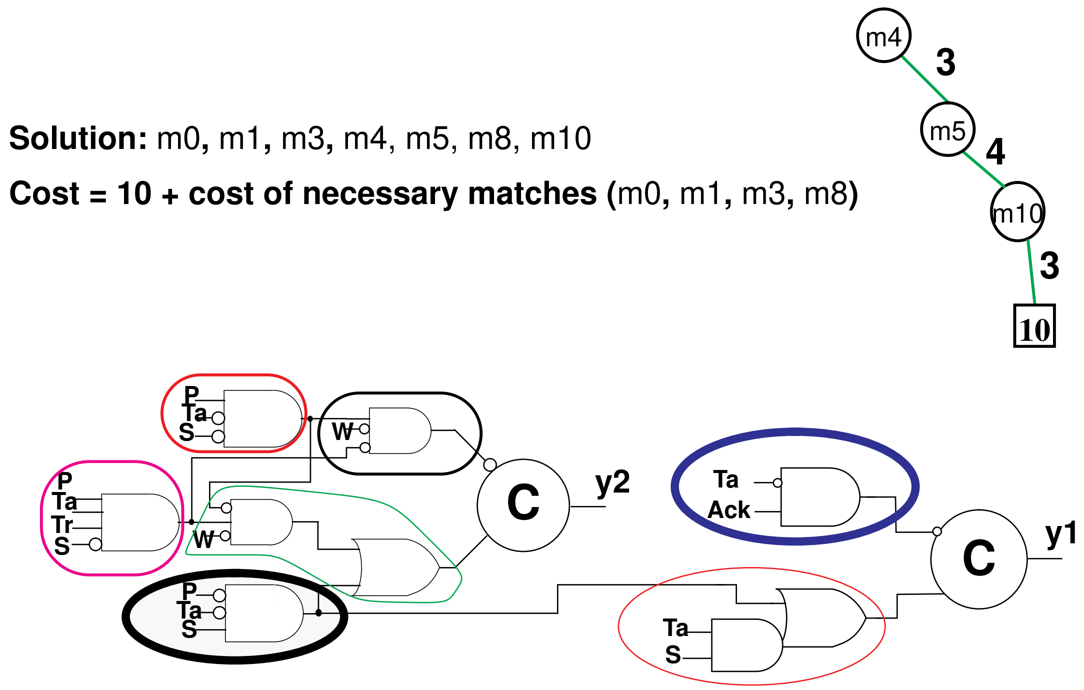


Figure 4.20: Solution to binate covering problem

Our overall technology mapping procedure, including the new covering algorithm, can now be stated:

```
procedure si_tmap2(network, library) {
    cones = partition(network);
    foreach subnetwork in cones {
        decomposed_network += si_tech_decomp(subnetwork);
    }
    find_best_si_cover(decomposed_network, library);
}
```

After breaking the circuit into cones of logic (for the purpose of decomposition), we invoke the hazard-free decomposition procedure described in Section 4.4.4 for each cone of logic. We then use the new matching and covering routine just described to find the best cover for the entire decomposed network.

4.4.6 Results

We applied the algorithms discussed in the previous section on the examples used earlier. Using the hazard-non-increasing matching routines described in the previous chapter, we found covers for the benchmark designs, resulting in the following results:

Design	# of gates	Initial mapping cost (1-1 mapping with library elements)	Final mapping cost	# of shared gates
a_to_d_controller	6	12	10	0
chu133	6	12	10	0
chu150	7	13	13	0
converta	13	22	18	0
dff	6	12	10	0
ebergen	9	16	16	0
half	6	6	6	0
hazard	6	12	10	0
mp-forward-pkt	10	16	14	0
nak-pa	10	13	13	0
nowick	11	20	18	0
pe-rcv-ifc	36	97	68	5
qr42	9	16	16	0
ram-read-sbuf	11	17	14	1
rcv-setup	3	7	6	0
sbuf-ram-write	11	21	18	0
sbuf-send-pkt2	13	25	21	0
two_phase_fifo	7	13	12	0
vbe5b	6	9	9	0
vbe5c	4	3	3	0

Table 4.2 Results from applying covering algorithm to benchmark circuits

The initial mapping cost in the table above represents the area cost of mapping the *decomposed* network one-to-one with elements from the library. Only combinational circuitry is counted in the cost equations, since the cost of the sequential elements is a constant for each design. In this particular case we used a library containing all simple gates (ANDs or ORs) of up to four inputs and some simple AND-OR and OR-AND gates of up to four inputs. The final mapping cost represents the best cost found after applying the modified matching and covering algorithm on the entire network.

The astute reader will note that in many of the cases, the optimal solution for most of the designs could have been easily found by taking a one-to-one mapping between the starting design and the library elements. This is understandable, since most of the designs are very small. However, as the designs get larger, the mapping algorithms give better results.

The only moderately large sized design we tried, pe-rcv-ifc, showed a 30% area savings over the initial decomposed circuit. Some of this was achieved through sharing of gates across logic cones, and some through application of the standard matching and covering techniques.

The covering equation for this design had 64 variables in 97 clauses, and took 4 minutes, 11 seconds to solve on an HP 9000/755 (there were eight equal-cost solutions). By contrast, solving the equation with a binate covering solver that takes variables in sequential order (after accounting for essential variables) took over 3 hours on the same machine.

Additional experimentation showed promise of an additional two orders of magnitude improvement in performance. We augmented our variable selection mechanism by partitioning the equation into disjoint smaller equations before solving. After application of this simple partitioning scheme at the top level, the equation was solved in less than 10 seconds on the same machine. As synthesis methods improve, resulting in larger circuits to be mapped, this performance improvement will be essential to solve the resulting covering equations.

4.5 Summary

In this chapter we presented technology mapping techniques for speed-independent circuits. We analyzed the burst-mode mapping procedure for its suitability for speed-independent designs. We found that the decomposition step was difficult, and developed new theory and algorithms for hazard-free decomposition of speed-independent networks. Unlike with burst-mode designs, information about the environment was necessary to drive the decomposition. Although we developed theory and algorithms for decomposition of sum-of-products circuits, these are easily extended to multi-level AND-OR networks. After implementing the decomposition algorithms and running them on some standard speed-independent benchmarks, we found that the resulting decomposed circuits were typically composed of functions larger than the 2-input 1-output base functions used by the burst-mode mapper.

The coarse granularity of decomposition allowed us to propose an exact covering algorithm that takes advantage of global sharing within the design. We applied these algorithms to the standard benchmark circuits to explore the amount of sharing the algorithm gives us.

The covering procedure described in this chapter can be extended to apply to synchronous mapping. However, larger designs may require additional heuristics to cope with the complexity of the resulting binate equation. The application of a variable partitioning scheme within the binate covering solver shows great promise for future performance improvements.

Chapter 5

Conclusions and Future Work

5.1 Contributions

We have presented theory and algorithms for technology mapping of two popular asynchronous design styles: burst-mode and speed-independent.

For burst-mode designs, we have analyzed synchronous mapping algorithms for their impact on the hazard behavior of the resulting designs and have proven the following:

1. The decomposition step of technology mapping must be restricted to hazard-preserving transformations. In particular, we showed that De Morgan's Law and the associative law can be used to decompose circuits.
2. Partitioning techniques similar to those used by synchronous mappers can be used without modification because they do not affect the hazard behavior of the resultant mapped circuit.
3. The matching step must restrict the matches to those functionally equivalent library elements that have a subset of the hazards present in the matched subnetwork. Furthermore, we proved that because matching networks have matching function hazards, we can ignore function hazards during this process.

We used this theory to build a technology mapper for burst-mode designs. The technology mapper takes a hazard-free burst-mode logic description and generates a hazard-free implementation composed of parts from a standard-cell or gate-array library. To complete the implementation of this mapper we also had to develop efficient algorithms to analyze the hazard-behavior of the library elements and of the subnetworks. As part of joint work with Alan Marshall and Bill Coates of HP

Labs, we incorporated this technology mapper into the STETSON toolkit for asynchronous design, and it was used in the design of an asynchronous communications IC receiver chip [49].

We used the burst-mode technology mapper as the starting point for our investigation into technology mapping for speed-independent designs. After analyzing the basic steps we proved the following:

1. Gates cannot be decomposed without consideration of the environment. In particular, we must restrict the decompositions to those that preserve the sequencing of signals from the environment to ensure that the implementation remains speed-independent.
2. The matching step must restrict the acceptable matches to those without logic hazards. Since the initial design is hazard-free, we showed that we can use the matching routines of the burst-mode mapper to ensure that hazards are not introduced during this step.
3. The covering routines can be extended to take advantage of logic sharing within the design.

In addition to this, we developed theory and accompanying algorithms to establish the set of acceptable decompositions. We also developed a new covering formulation that operates globally over all logic in the design. Taking advantage of the small size of the designs, we came up with an exact solution to this covering problem.

We applied these ideas to the implementation of some standard benchmark circuits, and characterized the amount of sharing found across the cones of logic.

The contributions of this thesis allow completion of the synthesis path for two of the most important asynchronous design styles, thus removing a key obstacle to acceptance of asynchronous design. With the technology mapping techniques described, designers can now experiment with using asynchronous techniques in place of synchronous techniques to determine whether asynchronous design can pay off in terms of power savings and performance for their applications.

5.2 Future Work

5.2.1 Application to Synchronous Systems

Because asynchronous designs are sensitive to glitches at all times, they must be hazard-free. Some of the low-power benefits of asynchronous design come as a result of this glitch-free logic. If we look at *synchronous* designs, we find that glitches are responsible for up 15-20% of the total average power consumption (measured on a number of standard benchmarks) [7]. This suggests that we might extend our hazard analysis algorithms into hazard removal algorithms and investigate whether this results in a net power savings for synchronous designs.

Technology mapping for low power has already been tackled from several angles. In [82], the authors propose a power-efficient decomposition method, and combined that with a covering algorithm that trades off power and delay to give an average of 21% improvement in performance on the benchmark set they ran. A different approach was taken in [81], with a covering algorithm that attempts to hide the high transition-count nodes by covering those nodes with a single gate, in an attempt to minimize the capacitance driven by a gate. They report average power savings of 24% on their benchmark set.

We propose to investigate whether any power savings is to be had from eliminating the hazards in the design. For asynchronous designs we had to make sure that the circuits were hazard-free; however, with synchronous designs we can selectively remove those hazards that are responsible for the most power consumption. The hazard detection algorithms presented in this thesis can be easily extended to remove hazards in the design. Because some hazards are removed by adding redundant gates, there is an obvious tradeoff between the power consumed by the additional gate and the power saved by removing the glitch. Because some logic hazards can be eliminated through factoring into a multilevel expression, some hazards can be removed by appropriate factoring of the function to be mapped, at less cost than adding a redundancy. To accurately measure the power savings a simulation-based approach is preferred over a probabilistic one, although knowledge of the circuit's environment will be necessary to do this.

5.2.2 Extensions to Technology Mapping for Burst-Mode Designs

Several extensions to the technology mapper for burst-mode designs automatically come to mind. First, more libraries need to be examined for their hazard behavior, and more designs need to be mapped to formulate the average penalty that is paid for rejecting hazardous elements (in contrast to performing the mapping with a synchronous mapper using a library composed of hazard-free elements). In the experiments we tried, there were very few hazardous elements in any of the libraries, so many of the experiments were not that enlightening. If the penalty for rejecting hazardous elements turns out to be large, then knowledge of the circuit's environment can be used to formulate a set of *hazard don't cares.* These *hazard don't cares* represent the transitions which never occur in normal operation of the state machine, and must be extracted from the state machine specification. Using these *don't care* conditions will allow more mappings to be made to hazardous elements, because the hazards they exercise can be ignored, since these transitions never occur during operation of the circuit.

The results of the hazard analysis of FPGA elements suggest that FPGA designers can create libraries that are more suitable for asynchronous design by limiting the set of cell personalizations to those that are hazard-free whenever possible. This may have a negative impact on routing, since undoubtedly the personalizations were included for ease of routing.

5.2.3 Extensions to Technology Mapping for Speed-Independent Designs

Our research only scratched the surface of the speed-independent design styles. The first obvious extension is to apply the results of this work to other speed-independent implementation styles. With automation of the technology-mapping step, larger designs are likely to be created which may require new and more efficient algorithms for library binding. In conjunction with this, further work is required to incorporate other mechanisms for arriving at an initial decomposition in case both the simple decomposition procedure and the extended decomposition procedure fail, and to more precisely identify conditions for the existence of a successful decomposition.

Up to this point, we assumed that the isochronic-fork assumption holds. However, as we have seen from the many examples we have done, it is very unlikely that the isochronic-fork assumption will hold with so many points of large fanout in the designs. Future work must address whether the isochronic-fork assumption still holds, and devise a mechanism for correctly breaking up the points of multi-fanout while ensuring correct circuit operation under the given delay assumptions.

Bibliography

[1] V. Akella and G. Gopalakrishnan, "SHILPA: A high-level synthesis system for self-timed circuits," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pages 587–591, Nov. 1992.

[2] D. Armstrong, A. D. Friedman, and P. R. Menon, "Design of asynchronous circuits assuming unbounded gate delays," *IEEE Transactions on Computers*, C-18(12):1110–1120, Dec. 1969.

[3] P. Beerel, J. Burch, and T. H.-Y. Meng, "Efficient verification of speed-independent circuits," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, Nov. 1993.

[4] P. Beerel and T. H.-Y. Meng, "Automatic gate-level synthesis of speed-independent circuits," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pages 581–586, Nov. 1992.

[5] P. Beerel and T. H.-Y. Meng, "Logic transformations and observability don't cares in speed-independent circuits," in *TAU*, Aug. 1993.

[6] J. Beister, "A unified approach to combinational hazards," *IEEE Transactions on Computers*, 23(6):566–575, June 1974.

[7] L. Benini, M. Favalli, and B. Ricco, "Analysis of hazard contributions to power dissipation in CMOS ICs," in *Napa Workshop on Low Power Design*, May 1994.

[8] G. Borriello and R. H. Katz, "Synthesis and optimization of interface transducer logic," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pages 274–277, Nov. 1987.

[9] R. K. Brayton, R. Camposano, G. De Micheli, R. H. J. M. Otten, and J. van Eijndhoven, "The Yorktown Silicon Compiler System," in D. Gajski, editor, *Silicon Compilation*, Addison Wesley, 1988.

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

[11] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A multiple-level logic optimization system," *IEEE Transactions on CAD/ICAS*, 6(6):1062–1081, Nov. 1987.

[12] J. Bredeson, "Synthesis of multiple input change hazard-free combinational switching circuits without feedback," *International Journal of Electronics*, 39(6):615–624, Dec. 1975.

[13] J. Bredeson and P. Hulina, "Elimination of static and dynamic hazards for multiple input changes in combinational switching circuits," *Information and Control*, 20(2):114–124, Mar. 1972.

[14] E. Brunvand, "Using FPGAs to implement self-timed systems," *Journal of VLSI Signal Processing*, 6(2):173–190, Aug. 1993.

[15] E. Brunvand and R. F. Sproull, "Translating concurrent programs into delay-insensitive circuits," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, IEEE Computer Society Press, 1989.

[16] R. E. Bryant, "Race detection in MOS circuits by ternary simulation," in *VLSI '83*, pages 85–95, Elsevier Science Publishers B.V. (North Holland), 1983.

[17] J. A. Brzozowski and M. Yoeli, "Combinational static CMOS networks," in *VLSI Algorithms and Architectures. Agean Workshop on Computing Proceedings*, 1986.

[18] S. M. Burns, "Automated compilation of concurrent programs into self-timed circuits," Technical Report Caltech-CS-TR-88-2, California Institute of Technology, 1987.

[19] T.-A. Chu, "Synthesis of self-timed VLSI circuits from graph-theoretic specifications," Technical Report MIT-LCS-TR-393, MIT, 1987.

[20] W. S. Coates, A. L. Davis, and K. S. Stevens, “Automatic synthesis of fast compact self-timed control circuits,” in *IFIP Workshop on Asynchronous Circuits*, Manchester, UK, 1993.

[21] W. S. Coates, A. L. Davis, and K. S. Stevens, “The post office experience: Designing a large asynchronous chip,” *INTEGRATION, the VLSI journal*, 15(4):341–366, 1993.

[22] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1990.

[23] A. Corporation, *ACT Family Field Programmable Gate Array Databook*, Actel, 1991.

[24] O. Coudert and J. Madre, “Implicit and incremental computation of primes and essential primes of Boolean functions,” in *DAC, Proceedings of the Design Automation Conference*, pages 36–39, June 1992.

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

[26] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[27] G. De Micheli, D. Ku, F. Mailhot, and T. Truong, “The Olympus Synthesis System,” *IEEE Design & Test of Computers*, pages 37–53, Oct. 1990.

[28] M. E. Dean, D. L. Dill, and M. Horowitz, “Self-timed logic using current-sensing completion detection (CSCD),” in *ICCD, Proceedings of the International Conference on Computer Design*, pages 187–191, IEEE Computer Society Press, Oct. 1991.

[29] E. Detjens, G. Gannot, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, “Technology mapping in MIS,” in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pages 116–119, Nov. 1987.

[30] J. C. Ebergen, *Translating Programs into Delay-Insensitive Circuits*, Amsterdam: Centrum voor Wiskunde en Informatica, CWI Tract 56, 1989.

[31] J. C. Ebergen, “A formal approach to designing delay-insensitive circuits,” *Distributed Computing*, 5(3):107–119, 1991.

[32] E. B. Eichelberger, “Hazard detection in combinational and sequential switching circuits,” *IBM Journal*, Mar. 1965.

[33] A. El Gamal, J. Green, J. Reyneri, E. Rogoyski, K. A. El-Ayat, and A. Mohsen, “An architecture for electrically configurable gate arrays,” *IEEE Journal of Solid-State Circuits*, 24(2):394–398, Apr. 1989.

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

[35] 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*, pages 79–85, 1986.

[36] S. Hauck, “Asynchronous design methodologies: An overview,” Technical report, Department of Computer Science and Engineering, University of Washington, May 1993, Technical Report No. UW-CSE-TR-93-05-07.

[37] S. Hauck, S. Burns, G. Borriello, and C. Ebeling, “An FPGA for implementing asynchronous circuits,” *IEEE Design & Test of Computers*, Sept. 1994.

[38] D. V. Heinbuch, *CMOS3 Cell Library*, Addison-Wesley, 1987.

[39] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice Hall International, UK LTD., Englewood Cliffs, New Jersey, 1985.

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

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

[42] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev, “Basic gate implementation of speed-independent circuits,” in *DAC, Proceedings of the Design Automation Conference*, 1994.

[43] A. Kondratyev, M. Kishinevsky, and A. Yakovlev, “On hazard-free implementation of speed-independent circuits,” in *Submitted to Async 94*, 1994.

[44] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli, “Algorithms for synthesis of hazard-free asynchronous circuits,” in *DAC, Proceedings of the Design Automation Conference*, pages 302–308, IEEE/ACM, 1991.

[45] L. Lavagno and A. Sangiovanni-Vincentelli, *Algorithms for Synthesis and Testing of Asynchronous Circuits*, Kluwer Academic Publishers, 1993.

[46] K.-J. Lin, J.-W. Kuo, and C.-S. Lin, “Direct synthesis of hazard-free asynchronous circuits from STGs based on lock relation and MG-decomposition approach,” in *Proceedings of the European Design and Test Conference EDAC-ETC-EUROASIC*, pages 178–183, Mar. 1994.

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

[48] F. Mailhot and G. De Micheli, “Algorithms for technology mapping based on binary decision diagrams and on boolean operations,” *IEEE Transactions on CAD/ICAS*, pages 599–620, May 1993.

[49] A. Marshall, W. Coates, and P. Siegel, “Designing an asynchronous communications chip,” *IEEE Design & Test of Computers*, pages 8–21, Summer 1994.

[50] A. J. Martin, “Compiling communicating processes into delay-insensitive VLSI circuits,” *Distributed Computing*, 1:226–234, 1986.

[51] A. J. Martin, “The limitations to delay-insensitivity in asynchronous circuits,” in *Proceedings of the Sixth MIT Conference in Advanced Research in VLSI*, pages 263–278, 1990.

[52] A. J. Martin, “Asynchronous datapaths and the design of an asynchronous adder,” *Formal Methods in System Design*, 1(1):117–137, July 1992.

[53] A. J. Martin, S. M. Burns, T. Lee, D. Borkovi’c, and P. Hazewindus, “The design of an asynchronous microprocessor,” in *Decennial Caltech Conference on VLSI*, pages 226–234, 1989.

[54] E. J. McCluskey, “Minimization of Boolean functions,” *Bell Syst. tech J.*, 35(5):1417–1444, Nov. 1956.

[55] E. J. McCluskey, *Logic Design Principles With Emphasis on Testable Semicustom Circuits*, Prentice-Hall, 1986.

[56] T. H. Meng, *Synchronization Design for Digital Systems*, Kluwer Academic, 1990.

[57] T. H.-Y. Meng, R. W. Brodersen, and D. G. Messerschmitt, “Automatic synthesis of asynchronous circuits from high-level specifications,” *IEEE Transactions on CAD/ICAS*, 8(11):1185–1205, Nov. 1989.

[58] C. E. Molnar, T.-P. Fang, and F. U. Rosenberger, “Synthesis of delay-insensitive modules,” in H. Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 67–86, Computer Science Press, Inc., 1985.

[59] C. W. Moon, *Synthesis and Verification of Asynchronous Circuits from Graph Specifications*, PhD thesis, U. C. Berkeley, 1992.

[60] D. E. Muller and W. S. Bartky, “A theory of asynchronous circuits,” in *Proceedings of an International Symposium of the Theory of Switching*, pages 204–243, 1959.

[61] T. Murata, “Petri nets: Properties, analysis, applications,” *Proceedings of the IEEE*, 77(4):541–580, Apr. 1989.

[62] C. Myers and T. H.-Y. Meng, “Synthesis of timed asynchronous circuits,” *IEEE Transactions on VLSI Systems*, 1(2):106–119, June 1993.

[63] C. J. Myers and T. H.-Y. Meng, “Automatic hazard-free decomposition of high-fanin gates in asynchronous circuit synthesis",” 1994, forthcoming paper.

[64] S. Nowick and D. Dill, “Asynchronous state machine synthesis using a local clock,” in *International Workshop on Logic Synthesis*, MCNC, 1991.

[65] S. M. Nowick, M. E. Dean, D. L. Dill, and M. Horowitz, “The design of a high-performance cache controller: a case study in asynchronous synthesis,” *Integration, The VLSI Journal*, 15(3):241–262, Oct. 1993.

[66] S. M. Nowick and D. L. Dill, "Automatic synthesis of locally-clocked asynchronous state machines," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pages 318–321, 1991.

[67] S. M. Nowick and D. L. Dill, "Synthesis of asynchronous state machines using a local clock," in *ICCD, Proceedings of the International Conference on Computer Design*, pages 192–197, IEEE Computer Society Press, 1991.

[68] S. M. Nowick and D. L. Dill, "Exact two-level minimization of hazard-free logic with multiple-input changes," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pages 626–630, 1992.

[69] S. M. Nowick, K. Yun, and D. L. Dill, "Practical asynchronous controller design," in *ICCD, Proceedings of the International Conference on Computer Design*, pages 341–345, IEEE Computer Society Press, 1992.

[70] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison Wesley, 1993.

[71] C. Pedron and A. Stauffer, "Analysis and synthesis of combinational pass transistor circuits," *IEEE Transactions on CAD/ICAS*, 7(7):775–785, July 1988.

[72] D. Radhakrishnan, S. Whitaker, and G. Maki, "Formal design procedures for pass transistor switching circuits," *IEEE Journal of Solid-State Circuits*, 20(2):531–536, Apr. 1985.

[73] R. Rudell, *Logic Synthesis for VLSI Design*, PhD thesis, U. C. Berkeley, Apr. 1989, Memorandum UCB/ERL M89/49.

[74] R. Rudell and A. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization," *IEEE Transactions on CAD/ICAS*, 6(5):727–750, Sept. 1987.

[75] S. Sasi and D. Radhakrishnan, "Hazards in CMOS circuits," *International Journal of Electronics*, 68(6):967–990, June 1990.

[76] C. L. Seitz, "System timing," in *Mead and Conway, Introduction to VLSI Systems–Ch 7.*, pages 218–262, Addison-Wesley, 1980.

[77] E. M. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," Technical report, Electronics Research Laboratory, Department of Electrical Engineering and Computer Science, University of California at Berkeley, May 1992, Memorandum No. UCB/ERL M92/41.

[78] D. A. S. Subcommittee, "IEEE standard VHDL language reference manual," Technical Report IEEE Std 1076-1987, IEEE, Mar. 1988.

[79] I. E. Sutherland, "Micropipelines," *CACM*, 32(6):720–738, First Quarter 1989.

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

[81] V. Tiwari, P. Ashar, and S. Malik, "Technology mapping for low power," in *DAC, Proceedings of the Design Automation Conference*, pages 74–79, 1993.

[82] C. Tsui, M. Pedram, and A. Despain, "Technology decomposition and mapping targeting low power dissipation," in *DAC, Proceedings of the Design Automation Conference*, pages 68–73, 1993.

[83] S. H. Unger, *Asynchronous Sequential Switching Circuits*, New York: Wiley-Interscience, 1969.

[84] V. I. Varshavsky, editor, *Self-Timed Control of Concurrent Processes*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.

[85] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design (Second Edition)*, Addison-Wesley, 1992.

[86] T. E. Williams, "Performance of iterative computation in self-timed rings," *Journal of VLSI Signal Processing*, 7(1-2):17–31, Feb. 1994.

[87] M. Yoeli and S. Rinon, "Application of ternary algebra to the study of static hazards," *ACM Journal*, 11(1):84–97, Jan. 1964.

[88] K. Yun, *Automatic Synthesis of Asynchronous Controllers*, PhD thesis, Stanford University, 1994.

[89] K. Yun and D. Dill, “Automatic synthesis of 3D asynchronous finite-state machines,” in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pages 576–580, Nov. 1992.

[90] K. Yun, D. Dill, and S. M. Nowick, “Practical generalizations of asynchronous state machines,” in *EDAC, Proceedings of the European Design Automation Conference*, pages 525–530, Feb. 1993.