SYNTHESIS OF ASYNCHRONOUS CONTROLLERS FOR HETEROGENEOUS SYSTEMS

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

By
Kenneth Yi Yun
August 1994

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

David L. Dill
(Principal Advisor)

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

Teresa H.-Y. Meng

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

Stephen P. Boyd

Approved for the University Committee on Graduate Studies:

Abstract

There are two synchronization mechanisms used in digital systems: *synchronous* and *asynchronous*. Synchronous or asynchronous refers to whether the system events occur in lock-step based on a clock or not. Today's system components typically employ the synchronous paradigm primarily because of the availability of the rich set of design tools and algorithms and, perhaps, because of the designers' perception of "ease of design" and the lack of alternatives. Even so, the interfaces among the system components do not strictly adhere to the synchronous paradigm because of the cost benefit of mixing modules operating at different clock rates and modules with asynchronous interfaces. This thesis addresses the problem of how to synthesize controllers operating in heterogeneous systems — systems with components employing different synchronization mechanisms.

We introduce a new design style called *extended-burst-mode*. The extended-burst-mode design style covers a wide spectrum of sequential circuits ranging from delay-insensitive to synchronous. We can synthesize multiple-input change asynchronous finite state machines, and many circuits that fall in the gray area between synchronous and asynchronous which are difficult or impossible to synthesize automatically using existing methods. Our implementation of extended-burst-mode machines uses standard combinational logic, generates low-latency outputs and guarantees freedom from hazards at the gate level.

We present a complete set of automated sequential synthesis algorithms: hazard-free state assignment, hazard-free state minimization, and critical-race-free state encoding. We also describe two radically different hazard-free combinational synthesis methods: two-level sums-of-products implementation and multiplexor trees implementation. Existing theories for hazard-free combinational synthesis are extended to handle *non-monotonic* input changes. A set of requirements for freedom from logic hazards is presented for each combinational synthesis method. Experimental data from a large set of examples are presented and compared to competing methods, whenever possible.

To demonstrate the effectiveness of the design style and the synthesis tool, the design of a commercial-scale SCSI controller data path is presented. This design is functionally compatible with an existing high performance commercial chip and meets the ANSI SCSI-2 standard.

Acknowledgments

This work would not have been possible without the help of many people.

I would like to thank David Dill, my advisor, for his advice, support and encouragement for the past three years. He has taught me what I know about research in computer science and engineering. I am grateful for his patience in putting up with my inarticulate ramblings, his ability to sort out important ideas, and, most of all, his ability to instill confidence in me.

I would like to thank Teresa Meng, my associate advisor, for her help and guidance. She convinced me that research in this field is not about just proving theorems and developing algorithms but about building something real and interesting as well. I would like to thank late Professor Peterson for his support and guidance and Stephen Boyd, for rescuing me at the last minute, after Professor Peterson passed away.

I would like to thank the past and present members of the ASYNC group at Stanford for the insightful discussions and criticisms. They include Peter Beerel, Jerry Burch, Bill Coates, Al Davis, Mark Dean, Nanni De Micheli, Jeremy Gunawardena, Alan Marshall, Chris Myers, Steve Nowick and Polly Siegel.

Working with Bill, Alan, and Polly has been a real joy. I would like to thank them for putting up with many versions of the buggy prototype synthesis tool and actually using it to design and fabricate a working communications chip.

I would like to thank Al Davis and Nanni De Micheli for their extremely helpful guidance in launching my academic career.

I would especially like to thank Steve Nowick, who has been my mentor from day one and the source of many insights since then. I became fascinated with practical asynchronous circuits while working with Steve on the first SCSI example. The foundation of my research was laid by Steve's work on burst-mode machines. I am grateful for his patience in mentoring me and correcting all those ungrammatical sentences in our joint papers.

I would like to thank Bill Lin and Srinivas Devadas for their ideas on BDD-based hazard-free combinational synthesis. The application of this idea to the 3D sequential synthesis has complemented my earlier work very nicely.

I would like to thank the researchers in the asynchronous community world-wide for the in-depth discussions: Ivan Sutherland for the inspiring lecture on Counterflow Pipeline Processor and nice circuit examples, Luciano Lavagno for discussions on STGs, Peter Vanbekbergen for discussions on interfacing to synchronous environment, Ganesh Gopalakrishinan and Erik Brunvand for discussions on 3D synthesis and Tam-Anh Chu for discussions on my earlier work, just to name a few.

I would like to thank the users of the 3D synthesis tool, Prabhaker Kudva of Utah, Loc Nguyen of Intel and Forrest Brewer of UCSB for their patience and comments.

I would like to thank Mark Knecht and the entire SCSI group at AMD for helping me understand what it takes to build a real SCSI controller and for permitting me to discuss this example in this thesis.

I would like to thank Lilian Betters for making my life easier by taking care of so many administrative matters.

I would especially like to thank my parents for their love and support and for bringing me to the United States, which made all this possible. I would like to thank my brother, sisters and their significant ones for their friendship. I would like thank Kay and Andre for proofreading the final draft.

Finally, I am eternally grateful for all the love my wife, Minsup, has given me and for all the sacrifices she has made to support me all these years. Minsup and my children, Paul and Sara, have been the joy and inspiration in my life. Without them, this work would be meaningless.

The financial support for this research came from the Semiconductor Research Corporation, Contract nos. 91-DJ-205, 92-DJ-205 and 93-DJ-205, and from the Stanford Center for Integrated Systems, Research Thrust in Synthesis and Verification of Multi-Module Systems.

"The years of anxious searching in the dark, with their intense longing, their alternations of confidence and exhaustion and the final emergence into the light — only those who have experienced it can understand it."

Albert Einstein.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Asynchronous circuits are sequential circuits which do not require external clocks to coordinate their internal operations. Asynchronous circuits were used in the earliest computers, and the research in asynchronous circuits and systems blossomed in the 1960's. However, the interests in asynchronous circuits declined in the 1970's, as synchronous circuits which use external clocks to schedule all their operations became popular, and all but vanished by the early 1980's. The main reason for the decline was the difficulty of designing custom components and the amount of details with which designers had to cope. Synchronous designs offered simplicity: the only rule which designers had to be concerned about was that circuits be stable some prescribed time before and after each clock tick.

Nevertheless, as VLSI technology evolved, the limitations of synchronous circuits began to surface. Some of the notable problems are clock skew, power dissipation, and interfacing to the environment.

Synchronous designs have to cope with clock skew problems: the difference in arrival times of a clock signal at various parts of a chip or a system effectively reduces the amount of time alloted for useful computation. As the feature size of VLSI chips shrank, the transistors switched faster but the distance electrons had to travel to deliver clock edges remained constant. In addition, the width of wires shrank but not the vertical thickness, which meant it took even longer for electrons to travel the same distance. So the advances in VLSI technology actually aggravated clock skew problems. In order to minimize these effects, increasingly larger portions of the synchronous VLSI chips are devoted to clock distribution. It was reported that DEC's new RISC chip Alpha 21064 [19, 21] uses one third of its chip area for clock distribution.

Furthermore, as more and more transistors were packed into a single chip, designers began to face real power dissipation problems. In synchronous chips with global clocking, even the inactive parts of the chip, including the clocks to those parts, dissipate power.

Finally, most VLSI chips and virtually all digital systems have to interface to signals that are asynchronous. It is absurd to even try to imagine two computer systems connected via a network synchronized with a common global clock. A synchronous chip or system that interfaces to an environment which does not share a common clock must synchronize asynchronous external inputs to its own internal clock. This introduces an inherent risk of synchronization failure [12, 44]. When the digital circuits operated at relatively low speed, the probability of synchronization failure was extremely low; however, when the computer systems started using a clock rate in access of 75MHz, the probability of failure became significant enough to warrant a search for a new solution.

1.1.1 Justifications for Asynchronous Circuits

There are many benefits asynchronous circuits can bring to system designs. We consider the necessity of asynchronous circuits in the systems context:

- Many interface signaling *protocols*, such as the SCSI bus data transfer protocol, *are asynchronous*. Synchronous controller design would require synchronizing asynchronous handshaking signals to a high speed internal clock, complicating the design and potentially sacrificing performance.

- Asynchronous circuits are ideal for building *modular* components. Modularity is an attractive feature in any system, because it makes global timing verification unnecessary. Asynchronous circuits designed for high performance applications work just as well when the system speed is lowered. For example, an asynchronous module designed for a desktop workstation works equally well when placed in a laptop computer with the same architecture operating at much lower speed.

- Asynchronous circuits tend to *dissipate less power* than synchronous counterparts for certain applications using CMOS technology. Asynchronous circuits have no power dissipation due to clock transitions and glitches, because CMOS circuits dissipate power only while switching. Furthermore, the data-driven nature of asynchronous circuits is ideal for low power applications that require quick transitions from standby to active, because inactive parts of the asynchronous circuits are always in “hot” standby mode without dissipating power. With the growing emphasis on portable electronics and wireless applications, low power may become a driving requirement for many new VLSI chip designs.

- Asynchronous circuits have *low latency output* because outputs are generated immediately upon receipt of enabling inputs, without having to wait for the next clock tick. Low latency outputs are useful in memory controllers and high speed switching networks.

1.1.2 Current State of Asynchronous Controller Design

Although we made strong claims for the necessity of asynchronous circuits in the last subsection, there are several problems with asynchronous design in its current state.

Asynchronous circuits are difficult to design manually, mainly because of the phenomenon called *hazard* — potential glitches, or undesired pulses, in the circuit. In synchronous circuits, glitches in combinational circuits do not cause problems as long as the combinational circuits stabilize before the next clock tick. However, in asynchronous circuits, no amount of glitches can be tolerated because the circuits are sensitive to every input change. Therefore, designers need to pay close attention to whether each synthesis step introduces hazards in the final design.

An even greater hurdle for asynchronous design is that most existing design methodologies are either not powerful enough or too difficult to use, and, in some cases, may not even produce correct results. The machines that were popular in the 1960's, such as single-input change Huffman-mode machines [30, 69], are inadequate for today's design requirements. Describing asynchronous circuits using system modeling languages, such as Petri nets [56], is too complex for large machines. Because these languages were designed primarily for modeling, not every specifiable machine is implementable.

Most importantly, for the foreseeable future, practically every asynchronous design must interface to existing synchronous designs. Current asynchronous design methodologies are not adequate for designing controllers that are to be used in heterogeneous systems — systems which consist of both synchronous and asynchronous components.

This thesis addresses all of these issues: *automation*, *correct design methodology*, *simple user-level specification formalism*, and *capability to interface to synchronous designs*.

1.2 Models

Results produced by synthesis methods are only as good as the models used to approximate the physical circuits. If the circuit model used by a synthesis method is too *optimistic*, then the synthesized circuits may be incorrect, although the circuits function correctly according to the model. On the other hand, if the circuit model is too *pessimistic*, then the synthesized circuits may be very robust but suboptimal. In this thesis, we attempt to strike a balance between optimistic and pessimistic models so that our models represent the current generation of technology accurately. However, in some cases, we use a pessimistic model to make the analyses to guarantee the correct results more manageable.

1.2.1 Circuit Models

A *gate* is a circuit component which computes the value of a logic function instantaneously, then asserts the computed value on its output after some delay. For example, an AND gate asserts the boolean product of its inputs after some delay. A gate output is connected to gate inputs by *wires*.[1] A *logic circuit* (see figure 1.1) consists only of gates and wires; it is connected to its environment via wires. A wire transports the value of a gate output to an input of a gate or to the environment of the circuit after some delay.

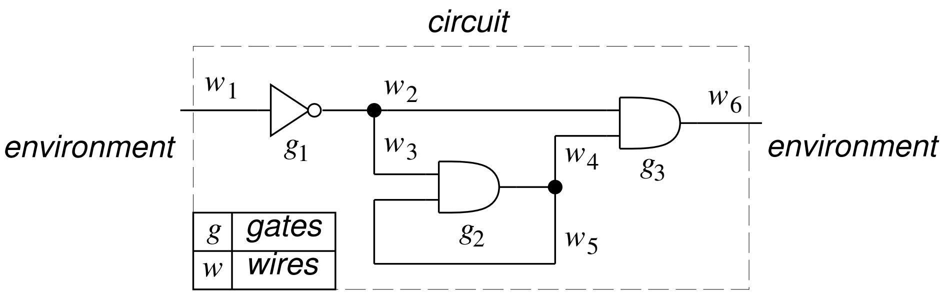


Figure 1.1: Circuit model.

A change of a gate input is said to *enable* the gate output if the input change alters the value of the associated logic function. A gate is said to be *stable* if the value of the associated logic function is the same as the asserted output value. A wire is said to be *stable* if the value of the gate input it is connected to is the same as that of the gate output to which it is connected. A logic circuit is said to be *stable* if every gate and every wire in the circuit is stable.

1.2.2 Environment Models

The environment of a circuit is said to operate in *fundamental mode* if it changes inputs to the circuit only after the circuit is stable. The environment of a circuit is said to operate in *multiple-input change fundamental mode* if it changes a specified

[1]We assume that gate outputs cannot be connected other gate outputs to simplify our circuit model, although, in some data paths, tri-state outputs and open-drain outputs can be connected to other tri-state outputs and open-drain outputs.

set of inputs to the circuit and waits for a specified set of outputs to change and for the circuit to stabilize before it changes the next set of inputs. The synthesis method presented in this thesis assumes a *variation* of the multiple-input change fundamental mode operation: The environment is allowed to *change certain inputs* that do not cause the outputs of the circuit to change *before the circuit has stabilized.* The discussion on this variation is presented in detail in section 2.1.

1.3 Background and Related Work

A conventional way of classifying asynchronous circuits is by the robustness of each class of circuits with respect to delay variations of physical components in the system. Each class of circuits makes certain assumptions about delays. These assumptions directly affect the validity of synthesis methods. For example, a circuit model that assumes that gates have arbitrary delays would not be suitable for synchronous circuit synthesis, because in that model there is no guarantee that gate outputs enabled to change will stabilize within a clock period.

A *delay-insensitive* (DI) circuit is an asynchronous circuit which operates correctly regardless of wire delay variations. In other words, the DI circuit model assumes the *unbounded wire delays*, that is, both gates and wires are assumed to have arbitrary finite delays. A delay-insensitive circuit consists of *macromodules* and the wires that connect them. Each macromodule must be designed so that delay variations on the wires do not cause malfunction of the circuit. Early work on macromodules was done by Molnar et al [46]. Since the internal design of each module assumes certain timing bounds, the internals of the circuits are not really delay-insensitive, but the communications among modules. There is a large body of theoretical work in this area, beginning with Udding and others [68, 22, 62].

A *speed-independent* (SI) circuit is an asynchronous circuit which operates correctly regardless of gate delay variations. *Wire forks*, or points of multiple fanouts, such as the junction of w_2 and w_3 in figure 1.1, are assumed to be *isochronic.* That is, changes of a gate output with multiple fanouts are assumed to arrive at the gate inputs it drives at the same time. In effect, wire delays can be lumped into gate delays, that is, gates are assumed to have arbitrary finite delays but wires have zero delays — the *unbounded gate delay assumption.* The speed-independent assumption was viable during the early days of VLSI, when the wires within a chip had negligible delays compared to gate delays. However, in the current generation of VLSI technology, in which wire delays are no longer insignificant, circuits designed with the SI assumption require careful layout and routing to ensure that the isochronic fork assumption is not violated. Many synthesis methods use SI circuits as target implementations [13, 45, 2, 71]. There have been some practical designs using SI circuits as well [41, 42].

There is a wide spectrum of circuits that are not SI circuits. Clearly, synchronous circuits are not speed-independent. All variations of Huffman-mode machines [30, 69] fall outside of the SI category. In fact, we claim that the robustness of the circuit really depends on how localized the delay variations are. There is no need to make a sweeping assumption that every gate can have a delay between 0 and ∞. Nor is it safe to assume that a wire that stretches from one corner of the chip to the other has a zero delay. The synthesis method in this thesis assumes the unbounded wire delays in combinational circuits used as building blocks of sequential circuits and the bounded wire delays in sequential circuits constructed by feeding back outputs of the combinational circuits.

Below, we will briefly survey current thoughts on asynchronous data path design and various asynchronous controller design styles. Although this thesis is about designing asynchronous controllers, we discuss the asynchronous data paths here briefly, partly to show just what kind of mechanisms the asynchronous controllers we design actually controls and partly to provide some background for the design example in chapter 5.

1.3.1 Asynchronous Data Paths

Although it is possible to construct a chip or a module which performs nothing but control tasks, it is difficult to conceive a system that do not contain some sort of data path. The purpose of a data path is to move the data both physically and logically, either as is or after performing some operations on it. A data path is controlled by a controller or a group of controllers.

Conventional synchronous data paths are pipelines. Each stage of a pipeline consists of some combinational logic or wires followed by a register to store the data. A pipeline is normally controlled by a clock. In contrast, there are two radically different ways to design asynchronous data paths.

The first method, called *micropipeline* [65], was introduced by Ivan Sutherland. A micropipeline without its control circuits is identical to a synchronous pipeline with all clocks removed. Each stage of a micropipeline uses a *request* wire, instead of a clock edge, to signal the next stage that the data is valid. The next stage, when it receives the data, uses an *acknowledge* wire to signal the sender that it has received the data. The sender is not to remove the data until it receives an acknowledge signal from the receiver. This method is attractive because of its simplicity, albeit some concerns about its robustness. Specifically, designers must take steps to assure that valid data arrive at the receiver before the accompanying request signals, which generally implies that carefully measured delays have to be added to the request wires. The SCSI controller example in chapter 5 uses a variation of the micropipeline for its data path.

The second method is based on generating a completion signal in a truly *self-timed* way. Each data bit is encoded in 2 wires, 01 for logic 0, 10 for for logic 1, for example. Before the computation, both wires are 0. When the computation is complete, one of the wires becomes 1. When one wire of every data bit changes to 1, the sender generates a completion signal to the receiver. When the receiver acknowledges the receipt of data, the sender resets all data bits to 00. Disadvantages of this method are the large overhead in area for the *dual-rail* logic and the extra energy cost associated with toggling wires of the data bits whose logic values do not change.

1.3.2 Asynchronous Controllers

This thesis is concerned with designing correct and efficient asynchronous controllers. There are three factors that affect the quality of the final design and the scope of its applications: *specification method*, *target implementation*, and *synthesis method.* The *design style* is a general term which refers to a combination of all of these.

Given a design style for controllers, there are two questions one must ask: "Can the signaling and timing constraints of most controllers be described in this design style?" and "Can every legal specification in this design style be implemented efficiently and correctly by automatic synthesis tools?" Clearly, maximizing expressiveness and maximizing implementability are conflicting goals.

Answers to the first question can only be found empirically, by examining the design examples in the target application area. For example, the specification formalism in our design style was extended from earlier ones after consideration of the HP Post Office packet routing chip [17], and industry-standard backplane and local bus controllers. It is the author's view, after having designed and analyzed a large number of controllers, that mechanisms to handle a moderate degree of concurrency and mechanisms to select alternative responses based on conditional signal levels as well as signal edges are essential. Fine-grained concurrency is not necessary in most applications.

The correctness part of the answer to the second question can only be demonstrated reasonably by *proving* that the synthesis procedure guarantees that the resulting circuits observe the constraints on behavior described in the specifications. The efficiency part of the answer can only be found by performing a large number of experiments and examining the results.

There have been a number of asynchronous controller design styles in the past 30 years. Many are based on high-level languages of concurrency [40, 7, 11, 22, 1] or Petri nets [13, 45, 34, 47, 71, 2, 48, 73]. These methods have proved successful for small, highly concurrent designs. However, they often lack flexibility in minimization and encoding of state and in the implementation of logic functions, and thus have difficulty taking advantage of global optimization techniques. Target implementations for almost all of these methods are either DI or SI circuits. Other methods based on asynchronous state machine specifications make more realistic delay assumptions and have been implemented in numerous styles [49, 33, 59, 69]. However, these methods often allow only single-input changes or restricted multiple-input changes which impose timing constraints on inputs [69, 14, 28]. None of the currently known methods are capable of correctly implementing synchronous features.

We will briefly survey existing design styles and zero in on a design style called *burst-mode*, which has a reasonable combination of expressiveness and implementability, and present a design style that extends burst-mode by allowing more concurrency and by adding a synchronous-like feature. We call this style, *extended-burst-mode.*

Compilation from High-Level Languages of Concurrency

Several design styles use high-level languages of concurrency as their user-level specification formalisms. The synthesis procedures typically consist of a series of transformations to generate either DI or SI circuits.

Martin's method [39] uses Hoare's *CSP* (Communicating Sequential Processes) [29] as the specification formalism. A specification describes a set of concurrent processes that communicate via commands on channels. The specification is translated into a set of operators called *production rules*, which are mapped to hardware components. The resulting circuit is speed-independent.[2] Burns [10] developed an automatic synthesis and performance analysis method for this design style. This method has been applied to a number of practical examples [38, 41, 42, 66].

Ebergen [22] uses a language derived from *trace theory* [58], developed by Rem, Snepscheut, and Udding. A specification in this language describes a sequence of events. The specification is *decomposed* into a network of macromodules; the resulting circuit is delay-insensitive.

Brunvand's method [7] is based on a CSP-like language called *Occam*. A specification described in *Occam* is compiled into an unoptimized delay-insensitive circuit, which is later enhanced by a technique called *peephole optimization.*

An advantage of the compilation methods over other methods is that complex concurrent systems can be described elegantly and concisely in high level constructs without low-level timing concerns, which makes it easier to modify and verify the system behavior. However, because it is difficult to utilize global optimization techniques

[2]Martin uses a term, *quasi-delay-insensitivity*, which is equivalent to our definition of speed-independence.

during the translation process, the automated synthesis often produces inefficient results. In general, the circuits generated using the compilation methods tend to incur considerably more area than those synthesized by other methods.

Recently, some efforts have been made to address the optimization problems. Gopalakrishnan et al [26] introduced a new way of doing peephole optimization by translating a DI module or a group of DI modules into *burst-mode* specifications and resynthesizing using the *3D* tool described in this thesis.

Also, there has been a concerted effort to synthesize commercial-scale circuits to demonstrate the practicality of these methods, such as the error detection and correction chips developed at Philips Research Laboratories by van Berkel et al [70] and Kessels et al [32] using a synthesis tool called *Tangram* compiler. A notable design problem with these chips is that interfacing to the environment is difficult because the internal circuits are designed to be delay-insensitive but the environment is synchronous.

Graph-Based Methods

Almost all of the graph-based methods use the Petri net or a restricted form of the Petri net as the specification formalism. A *Petri net* is a graph model used for describing concurrent systems.

Chu [13] introduced a restricted form of the Petri net called *STG* (Signal Transitions Graph) to specify asynchronous circuits. An STG in Chu's initial definition is an *interpreted free-choice Petri net*, which is capable of describing concurrent signal transitions but only a limited form of *choice*, the mechanism to select alternative responses of the circuit. Chu developed a synthesis method to implement SI circuits from STG descriptions.

Meng [45] extended Chu's work and developed an automatic synthesis tool which optimizes concurrency by exploiting known delays of the environment. Meng also designed and implemented a digital signal processor chip using this method.

The synthesis methods of Chu and Meng assume that the building blocks that comprise the resulting circuits are *complex gates*, the networks of basic gates, such as ANDs and ORs. Because these complex gates may not be free of internal hazards, they rely on automatic verifiers, such as Dill's AVER [20], to guarantee correctness of the resulting circuits.

More recent works include the synthesis methods of Lavagno [34], Moon [47], Vanbekbergen [71], and Ykman-Couvreur [73]. Lavagno's method inserts fixed delay elements to avoid hazards. In order to compute the size of delays, this method uses the *bounded wire delay model*, a circuit model in which the delay of each gate and wire has a lower and an upper bound. However, this method assumes, a bit too optimistically, that certain types of glitches are filtered out by a gate if it is narrower than the gate delay. Moon's method uses an automatic verifier to flag whether the synthesized circuits are hazard-free or not. If the resulting circuit is not hazard-free, the designer must modify the specification or abort the synthesis. The methods of Vanbekbergen and Ykman-Couvreur are based on the notions of *lock class* and *extended lock class*, the ordering constraints on signals. These methods are limited to a subclass of STGs called *Marked Graphs*, which cannot describe input choices.

In general, the strong suit of the STG is its ability to express concurrency. Its main weakness is the awkwardness in specifying input choices. That is, the mechanisms to guide the responses of the machine is limited. In a free-choice STG specification, the machine selects the course of its future behavior solely based on input transitions. The machine cannot handle choices based on input levels. Chu described a syntactic extension of STG in [13] to allow more flexible input choices. Moon et al [47] proposed more comprehensive extensions. However, no synthesis method in existence today can guarantee to synthesize correct circuits from specifications with these extensions. Vanbekbergen [71] introduced the *generalized* STG, that includes extensions proposed by Chu and Moon and also adds synchronous-like features. However, the synthesis method cannot guarantee to generate hazard-free circuits.

Some recent graph-based methods, not strictly Petri net derived, include Beerel's [2] and Myers's [48]. Beerel's method is based on an earlier work by Varshavsky [72]. This method automatically synthesizes gate-level SI circuits from a *state graph* description, which is typically generated by enumerating all possible signal transition orderings of a higher level specification, such as an STG. Because the synthesis begins at a lower level, this method must rely on higher level synthesis tools to generate *race-free* state graphs, the state graphs with no duplicate state codes. Myers's method uses an STG-like specification formalism called *Event-Rule Systems* [11]. This method synthesizes very compact area-efficient circuits by exploiting all known delays, both internal and external; however, it is difficult to construct large circuits because the basic building blocks used are complex gates called *generalized C-elements*.

Asynchronous Finite State Machines

AFSMs (Asynchronous Finite State Machines) have been around for the past 30 years. The work on AFSMs was pioneered by Huffman and others. Early AFSMs [30, 69] assumed that the environment operates in fundamental mode, that is, the environment generates a single input change and waits for the machine to stabilize before it generates the next input change. Recent work in AFSMs allows the multiple-input change fundamental mode operations. As shown in figure 1.2, some machines have Huffman-mode machine structure, combinational logic with some outputs fed back, and others have self-synchronizing structure, similar to synchronous state machines but with a locally generated clock.

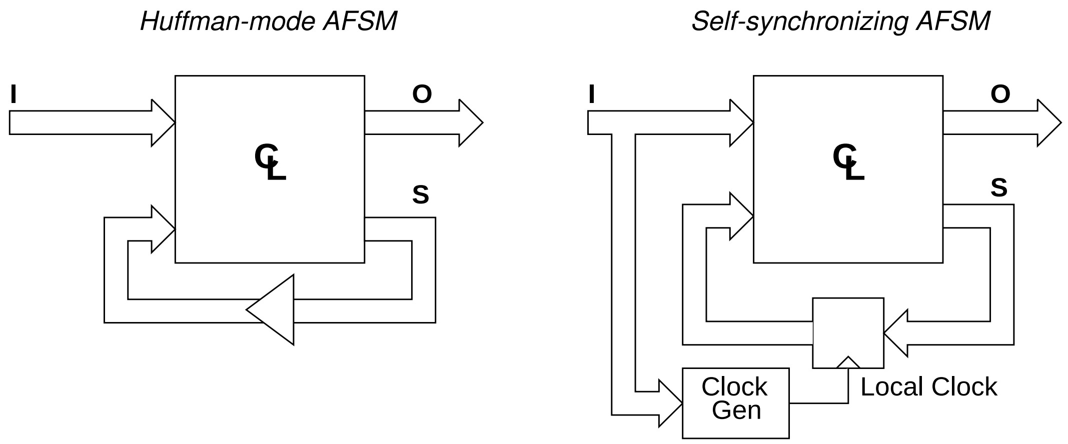


Figure 1.2: AFSM implementations.

We omit discussing earlier works on AFSMs here, because there is a large body of literature on this topic. Instead, we will focus on a recently introduced multiple-input change machine called *burst-mode* machine. Burst-mode asynchronous finite state machines were first introduced by Davis et al [18, 17] and formalized by Nowick and Dill [52, 49]. Burst-mode machines have been implemented using a method developed at HP Laboratories called *MEAT* [16], the *locally clocked* method [52, 49], the *3D* method [76, 74], and the *UCLOCK* method [50].

A *burst-mode* specification is a variation of a Mealy machine that allows multiple-input changes in a burst fashion — in a given state, when all of a specified set of input edges appear, the machine generates a set of output changes and moves to a new state. The specified input edges can appear in arbitrary order, thus allowing input concurrency, and the outputs are generated concurrently. The advantages of a burst-mode specification over STG specifications are that it is similar to the synchronous Mealy machine designers are familiar with, that the input choice is more flexible than that of the STG, and that the state encoding is more flexible in the implementations. Burst-mode specifications have been very useful in specifying large, practical controllers, such as a SCSI data transfer protocol controller [54], an asynchronous high performance cache controller [51], and asynchronous communications controllers [37].

Its main practical disadvantage is that it does not allow input changes to be concurrent with output changes. The input choice mechanism is more flexible than the STG but still primitive. For example, it cannot handle choices between two sets of concurrent events if one set is a subset of the other.

Extended-Burst-Mode Machine

The extended-burst-mode design style described in this thesis is a superset of burst-mode with two new features: *directed-don't-cares* and *conditionals*. Directed don't cares allow an input signal to change concurrently with output signals, and conditionals allow control flow to depend on the input signal levels, in the same way synchronous state machines regulate control flow. Thus this design style not only supports burst-mode multiple-input change asynchronous designs with added input/output concurrency, also allows the automatic synthesis of *any synchronous*

Moore machine, in which the synchronous inputs are represented as conditional signals, and the clock is the only non-conditional signal. Moreover, this design style covers a wide range of circuits between burst-mode and fully synchronous. We can easily specify and synthesize sequential circuits which change state on both rising and falling clock edges, have multiple-phase clocks, etc., subject only to setup and hold-time constraints.

1.4 Contributions of the Thesis

The contributions of this thesis cover all aspects of an asynchronous controller synthesis: user-level specification formalism, hazard-free combinational synthesis theory and its application to sequential synthesis, automated sequential synthesis algorithms, and a large commercial-scale design example. The main contributions are summarized below:

Extended-burst-mode design style: This thesis introduces a new design style called extended-burst-mode. The extended-burst-mode design style covers a wide spectrum of sequential circuits, which ranges from delay-insensitive to synchronous. It is significant *theoretically* because it is the first asynchronous design style that subsumes fully synchronous designs. It is significant *practically* because a wide range of practical circuits can be specified in a common specification language and synthesized using a single synthesis tool. For example, we can synthesize multiple-input change asynchronous finite state machines, including all burst-mode machines, and many circuits that fall in the gray area between synchronous and asynchronous which are difficult or impossible to synthesize automatically. These include circuits that require clocking with multiple clocks, circuits that require clocking on both edges of a clock signal, and circuits that require selective clocking.

Hazard-free combinational synthesis requirements: This thesis presents two radically different hazard-free combinational synthesis methods: hazard-free two-level sums-of-products implementation and hazard-free multiplexor trees implementation. Existing theories for hazard-free combinational synthesis are extended to handle *non-monotonic* input changes.

3D automatic synthesis algorithm: This thesis presents a complete set of automated sequential synthesis algorithms: hazard-free state assignment, hazard-free state minimization, and critical-race-free state encoding. The automated synthesis tool uses heuristics that find near-optimal solutions in polynomial time, whenever appropriate. Experimental data from a large set of examples are presented and compared to competing methods, whenever possible.

Commercial-scale design example: This thesis reports on the design and implementation of a commercial-scale SCSI controller data path, which demonstrates that the extended-burst-mode design style and the 3D synthesis tool are feasible for real-world designs.

1.5 Overview of the Thesis

This thesis is organized as follows:

After the introduction in this chapter, chapter 2 describes the extended-burst-mode specification and the 3D asynchronous state machine implementation. The extended-burst-mode specification is described first informally using an example and then formally. An overview of the synthesis procedure using a simple example is presented at the end.

Chapter 3 precisely characterizes every possible hazard that can arise in the 3D implementation of extended-burst-mode machines. For every type of hazard, a necessary and sufficient condition or at least a sufficient condition for freedom from hazards is stated and proved. This chapter presents the notion of generalized transition which is used for functional synthesis and for analysis of function hazards. This chapter also presents two different combinational synthesis methods: two-level sums-of-products implementation and multiplexor trees implementation.

Chapter 4 presents automatic synthesis procedure and algorithms. The first section describes the hazard-free state assignment algorithm and proves the existence of a hazard-free implementation for every legal extended-burst-mode specification. The second section presents a state minimization heuristic and compares it to a classical method. The third section describes a critical-race-free state encoding algorithm.

The last section reports the experimental results.

Chapter 5 describes the design and implementation of a SCSI controller data path. Each section describes one state machine in detail from each area of the data path, Bus Interface Unit, FIFO, and SCSI Bus Interface, and the rationale for key design decisions. The last section discusses important parameters of the implementation qualitatively.

Chapter 6 presents concluding remarks and some related open problems to solve.

Chapter 2

Specification and Implementation

2.1 Introduction

This chapter describes a user-level specification formalism, called *extended-burst-mode*, a target implementation style, called *3D*, and an overview of how an asynchronous controller specified in extended-burst-mode is transformed into a correct implementation.

The extended-burst-mode is a powerful user interface for specifying a large class of controllers. It is intended for designing asynchronous state machines and the machines that fall in the gray area between asynchronous and synchronous, although it is theoretically possible to specify any synchronous Moore machine [43] and practically feasible to design small to medium size synchronous Moore machines. It is particularly useful for designing controllers that operate in heterogeneous systems — systems with a mixture of synchronous and asynchronous components. It is important to note that every legal extended-burst-mode specification is implementable. In other words, the extended-burst-mode is a design tool, not an abstract modeling tool.

There are many implementation styles that can be used to synthesize asynchronous controllers — each has advantages and disadvantages. This thesis describes one particular implementation style called 3D, which is suitable for implementing extended-burst-mode machines. It is similar to Huffman-mode machines [30, 69] in structure and similar to Mealy machines [43] in functionality.

This chapter is organized as follows: Section 2.2 describes the extended-burst-mode specification formalism. Section 2.3 describes hardware implementations of extended-burst-mode machines and an overview of the synthesis using a simple example. Finally, section 2.4 describes three different types of machine cycles in which 3D machines may operate.

2.2 Controller Specification

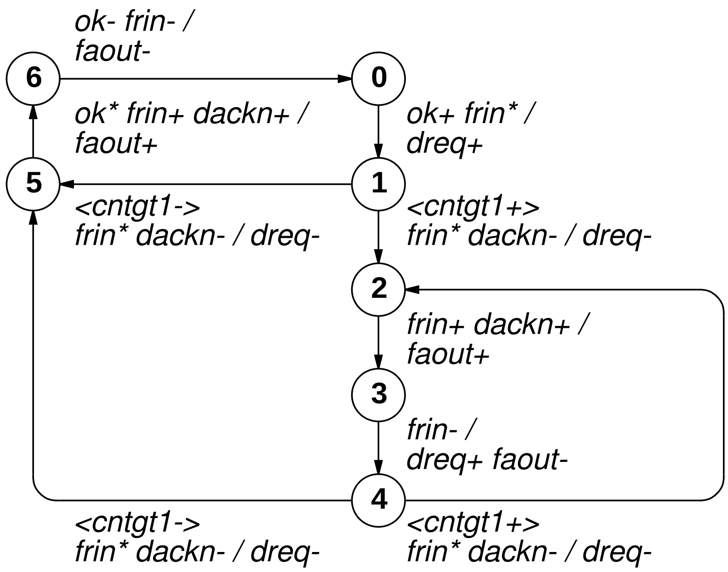


Figure 2.1: Extended-burst-mode specification.

Figure 2.1 shows an example of the extended burst-mode specification. Signals not enclosed in angle brackets and ending with + or − are *terminating signals*. These are edge signals. The signals enclosed in angle brackets are *conditionals*, which are level signals whose values are sampled when all of the terminating edges associated with them have occurred. A conditional $\langle a^{+} \rangle$ can be read "if a is high" and $\langle a^{-} \rangle$ can be read "if a is low." A state transition occurs only if all of the conditions are met and all the terminating edges have appeared. A signal ending with an asterisk is a *directed don't care.* If a is a directed don't care, there must be a sequence of state transitions in the machine labeled with a^{*}. If a state transition is labeled with

a^*, the following state transitions in the machine must be labeled with a^* or with a^+ or a^- (the terminating edge for the directed don't care). Figure 2.1 describes a state machine having a conditional input (cntgt1), 3 edge inputs (ok, frin, dackn), and 2 outputs (dreq, faout). Consider the state transitions out of state 4. The behavior of the machine at this point is: "if cntgt1 is low when dackn falls, change the current state from 4 to 5 and lower the output dreq; if cntgt1 is high when dackn falls, change the current state from 4 to 2 and lower output dreq."

A directed don't care may change at most once during a sequence of state transitions it labels, i.e., directed don't cares are *monotonic* signals, and, if doesn't change during this sequence, it must change during the state transition its terminating edge labels. A terminating edge which is not immediately preceded by a directed don't care is called *compulsory*, since it *must* appear during the state transition it labels. In figure 2.1, frin is low when the specification is in state 4. It can rise at any point as the machine moves through states 5 and 6 or through state 2, depending on the level of cntgt1, and it must have risen by the time the machine moves to states 6 or 2, because the terminating edge frin^+ appears between states 5 and 6 and between 4 and 2.

The input signals are globally partitioned into level signals (conditionals), which can never be used as edge signals, and edge signals (terminating or directed don't care), which can never be used as level signals. If a level signal is not mentioned on a particular state transition, it may change freely. If an edge signal is not mentioned, it is not allowed to change.

More generally, an extended-burst-mode asynchronous finite state machine [75] is specified by a state diagram which consists of a finite number of states, a set of labeled state transitions connecting pairs of states, and a start state. Each state transition is labeled with a set of conditional signal levels and two sets of signal edges: an input burst and an output burst. An *output burst* is a set of output edges, and an *input burst* is a non-empty set of input edges (terminating or directed don't care), at least one of which must be *compulsory*.

In a given state, when all the specified conditional signals have correct values and when all the specified terminating edges in the input burst have appeared, the machine generates the corresponding output burst and moves to a new state. Specified edges in the input burst may appear in arbitrary temporal order. However, the conditional signals must stabilize to correct levels before any compulsory edge in the input burst appears and must hold their values until after all of the terminating edges appear. The minimum delay from the conditional stabilizing to the first compulsory edge is called the *setup time*. Similarly, the minimum delay from the last terminating edge to the conditional change is called the *hold time*. Actual values of setup and hold times of conditional signals with respect to the first compulsory edge and the last terminating edge depend on the implementation. The period starting at the specified setup time before the first compulsory edge and ending at the specified hold time after the last terminating edge is called the *sampling period*. Conditional signal levels need not be stable outside of the specified sampling periods. Each signal specified as a directed don't care may change its value monotonically at any time including during output bursts, unless it is already at the level specified by the next terminating edge. Outputs may be generated in any order, but the next set of compulsory edges from the next input burst may not appear until the machine has stabilized. This requirement — the environment must wait until the circuit stabilizes before generating the next set of compulsory edges — is a variation of the *multiple-input change fundamental-mode environmental constraint.*

The following are some examples of labels on state transitions:

- $\langle c_1{}^+\rangle\langle c_2{}^-\rangle\, x^+/z_1{}^+z_2{}^-$ means "if $c_1 = 1$ and $c_2 = 0$ when x rises, then the machine raises z_1 and lowers z_2."

- $x_1{}^+x_2{}^-/z^+$ means "the machine raises z when x_1 rises and x_2 falls."

There is an additional restriction to extended-burst-mode specifications, called *distinguishability constraint*, which prevents ambiguity among multiple input bursts emanating from a single state: For every pair of input bursts i and j from the same state, either the conditions are mutually exclusive, or the set of compulsory edges in i is not a subset of the set of all possible edges in j.

For instance, the input bursts from state 0 in figure 2.2a are legal because $\langle c^+\rangle$ and $\langle c^-\rangle$ are mutually exclusive. However, the input bursts from state 0 in figure 2.2c are illegal because the conditions are *not mutually exclusive* and $\{b^+\} \subset \{a^+, b^+\}$. Moreover, the input bursts from state 0 in figure 2.2b violate the distinguishability constraint because the set of all possible edges for the input burst a^+b^* is $\{a^+, b^+\}$ and $\{b^+\} \subset \{a^+, b^+\}$.

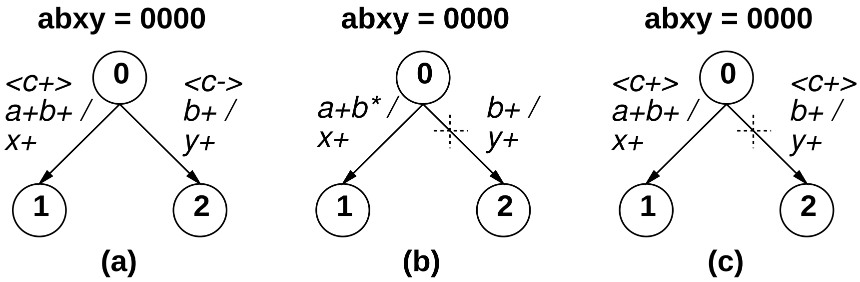


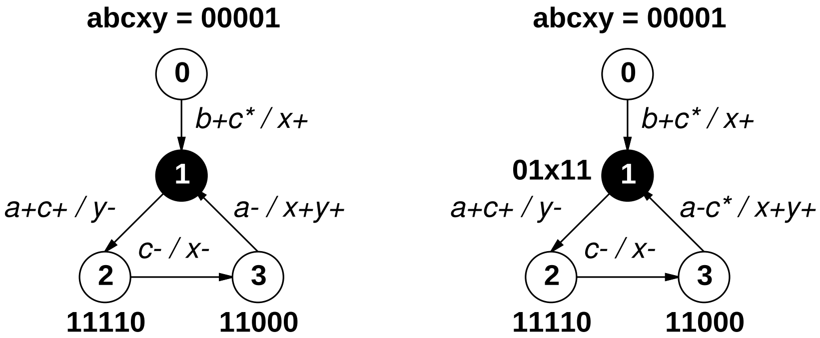
Figure 2.2: Distinguishability constraints.

To simplify exposition, we assume that the *unique entry condition* is satisfied. The set of possible *entry points* into a state (input and output values entering a state) from every predecessor state must be identical. This is a simplifying assumption that does not constrain the range of permissible behaviors since an extended-burst-mode specification can always be transformed into an equivalent specification satisfying the unique entry condition by duplicating some states.

For example, the set of valid entry points to state 1 from state 0 in figure 2.3a is $\{01011, 01111\}$, but from state 3 to state 1 it is $\{01011\}$. Thus the unique entry condition is not met in this specification. A specification satisfying the unique entry condition is shown in figure 2.3b.

2.2.1 Formal Definition of Extended-Burst-Mode

The following formal definition of the extended-burst-mode specification is adapted from the definition of the burst-mode specification in [49]. An extended-burst-mode specification is a directed graph, $G = (V, E, C, I, O, v_0, \mathsf{cond}, \mathsf{in}, \mathsf{out})$, where V is a finite set of states; $E \subseteq V \times V$ is the set of state transitions; $C = \{c_1, \ldots, c_l\}$ is the set of conditional inputs; $I = \{x_1, \ldots, x_m\}$ is the set of edge inputs; $O = \{z_1, \ldots, z_n\}$ is the set of outputs; $v_0 \in V$ is the unique start state; cond labels each state transition



(a) no unique entry (b) unique entry

Figure 2.3: Example (unique entry condition).

with a set of conditional inputs; in and out are labeling functions used to define the unique entry cube of each state. The function $\mathsf{cond} : E \to \{0, 1, *\}^l$ defines the values of the conditional inputs. The function $\mathsf{in} : V \to \{0, 1, *\}^m$ defines the values of the edge inputs and the function $\mathsf{out} : V \to \{0, 1\}^n$ defines the values of the outputs upon entry to each state.

Labeling functions trans_{IN} and trans_{OUT} are derived from graph G. $\mathsf{trans}_{IN} : E \to \mathcal{P}(I)$ defines the set of edge input changes and $\mathsf{trans}_{OUT} : E \to \mathcal{P}(O)$ defines the set of output changes. ($\mathcal{P}(I)$ and $\mathcal{P}(O)$ denote the power set of inputs and the power set of outputs respectively.) Given a state transition, $(u, v) \in E$, $x_i \in \mathsf{trans}_{IN}(u, v)$ iff $\mathsf{in}_i(u) \neq \mathsf{in}_i(v) \vee \mathsf{in}_i(v) = *$. That is, ${x_i}^+$ is in the input burst iff $\mathsf{in}_i(v) = 1 \wedge \mathsf{in}_i(u) \neq 1$, ${x_i}^-$ is in the input burst iff $\mathsf{in}_i(v) = 0 \ \wedge \ \mathsf{in}_i(u) \neq 0$, and ${x_i}^*$ is in the input burst iff $\mathsf{in}_i(v) = *$. Similarly, $z_j \in \mathsf{trans}_{IN}(u, v)$ iff $\mathsf{out}_j(u) \neq \mathsf{out}_j(v)$. That is, ${z_j}^+$ is in the output burst iff $\mathsf{out}_j(v) = 1 \ \wedge \ \mathsf{out}_j(u) = 0$, ${z_j}^-$ is in the output burst iff $\mathsf{out}_j(v) = 0 \wedge \mathsf{out}_j(u) = 1$. Finally, ctrans_{IN} defines the set of compulsory edge input changes: $\mathsf{ctrans}_{IN}(u, v) = \{x_i \in \mathsf{trans}_{IN}(u, v) \mid \mathsf{in}_i(u) \neq * \ \wedge \ \mathsf{in}_i(v) \neq *\}$.

The unique entry condition is satisfied by the above definition. The remaining requirements to ensure well-formed specifications are:

- Every input burst must contain a compulsory edge. That is, for every state transition (u, v), there exists $x_i \in \mathsf{trans}_{IN}(u, v)$ such that $\mathsf{in}_i(u) \neq * \wedge \mathsf{in}_i(v) \neq *$.

- Every pair of state transitions emanating from the same state must satisfy the distinguishability constraint. That is, for every pair, $(u,v),(u,w) \in E$, $\mathsf{ctrans}_{IN}(u,v) \subseteq \mathsf{trans}_{IN}(u,w)$ implies that either $v = w$ or $\mathsf{cond}(u,v)$ and $\mathsf{cond}(u,w)$ are mutually exclusive, that is, there exists k such that $\mathsf{cond}_k(u,v) \neq \mathsf{cond}_k(u,w) \wedge \mathsf{cond}_k(u,v) \neq * \wedge \mathsf{cond}_k(u,w) \neq *$.

- For every sequence of state transitions, $u \to v_1 \to \cdots \to v_n \to w$, with $n \geq 1$ and $\mathsf{in}_i(u) = \mathsf{in}_i(w) \neq *$, there exists $k \in 1,\ldots,n$ such that $\mathsf{in}_i(v_k) \neq *$.

2.3 Implementation Overview

In all sequential machines, the machine output depends not only on the inputs but also on the state of the machine, which keeps track of the history of input changes. All sequential machines use feedback to store the state of the machine. In Huffman-mode state machines [30, 69], the state of the machine is stored only in internal state variables — primary outputs do not store any state information. In our 3D machines, however, primary outputs are used to store the state of the machine whenever possible in order to minimize the number of internal state variables.

A 3D asynchronous finite state machine is formally defined as a 4-tuple (X, Y, Z, δ) where

- X is a non-empty set of primary input symbols;
- Y is a non-empty set of primary output symbols;
- Z is a (possibly empty) set of internal state variable symbols;
- $\delta : X \times Y \times Z \to Y \times Z$ is a *next-state function.*

The hardware implementation of a 3D state machine (see figure 2.4) is a combinational network, which implements the next-state function, with the outputs of the network fed back as inputs to the network. There are no explicit storage elements such as latches, flip-flops or C-elements in a 3D machine.

A 3D implementation of an extended-burst-mode specification is obtained from the *next-state table*, a 3-dimensional tabular representation of δ. The next state of

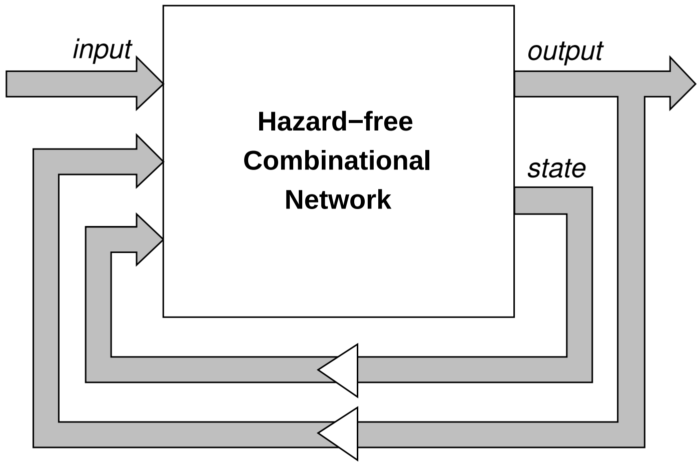


Figure 2.4: 3D asynchronous state machine.

every *reachable* state must be specified in the next-state table; the remaining entries are don't cares.

2.3.1 A Simple Example

A simple example is used to illustrate the synthesis and operation of a 3D machine (see figure 2.5). We describe the desired machine behavior according to an extended-burst-mode specification and the next-state table entries needed to make the machine behavior conform to the specification. From a completed next-state table, we can extract the logic equations directly, because next-state tables describe the next values of outputs and state variables for every combination of inputs, outputs and state variables.

In S_0 (the initial state), the machine awaits the input burst a^+b^*. During the input burst a^+b^*, abc changes from 000 to 100, if a^+ arrives first, or from 000 to 110 via 010, if b^+ arrives first. The outputs cannot change until a rises, regardless of b, because the extended-burst-mode specification mandates that outputs remain unchanged until the final *terminating* input edge (a^+ in this case) of the input burst arrives. Thus the next outputs, XY, for $abcxy$ = 00000 and 01000 are specified to be 00. Once a has risen, regardless of the value of b, the outputs xy must change monotonically to 11; thus the next output entries for $abcxy$ = 1x0xx are specified to

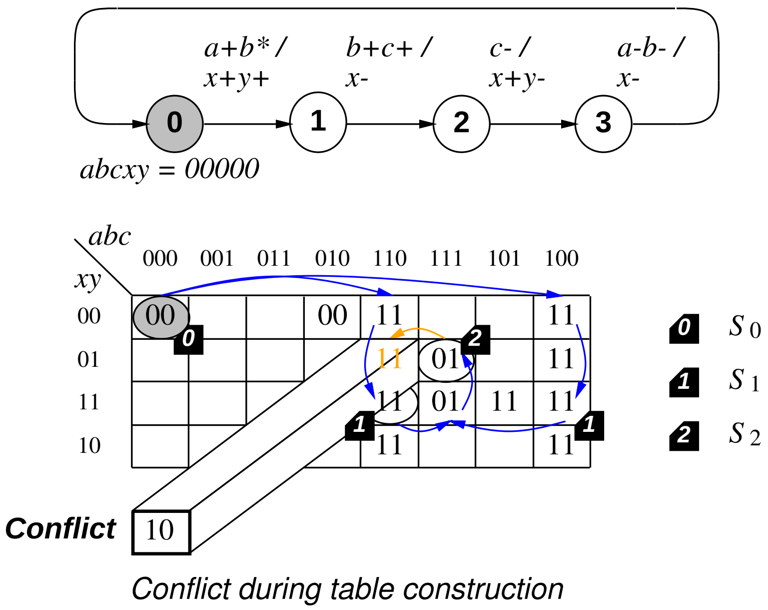


Figure 2.5: Simple example.

be 11. When the outputs xy stabilize to 11, the machine is in S_1. Note that b may be 0, 1, or rising in S_1.

In S_1, the machine waits for the input burst b^+c^+ — during this input burst, abc changes from 100 to 111 via 101 or 110, if b is 0 initially, and from 110 to 111, if b is already 1. The next outputs, XY, for $abcxy$ = 10011, 10111, and 11011 are specified to be 11, so that the outputs xy remain unchanged until both b and c have risen. The next outputs, XY, for $abcxy$ = 111x1 are specified to be 01, so that x then falls monotonically. When x stabilizes to 0, the machine is in S_2, where it awaits the input burst c^-. When the input c falls, the outputs xy must change to 10 monotonically. A problem (see figure 2.5) occurs when we attempt to specify the next outputs for $abcxy$ = 110xx to be 10. The next outputs, XY, for $abcxy$ = 110xx have already been specified to be 11 (during $S_0 \rightarrow S_1$).

Such conflicts can be avoided by adding state variables, which can be viewed as transitioning between *layers* of the next-state table (see the bottom table in figure 2.5). Conflicting entries can be placed in different layers. Our strategy, in this case, is to back up to the state following the last input burst before the conflict

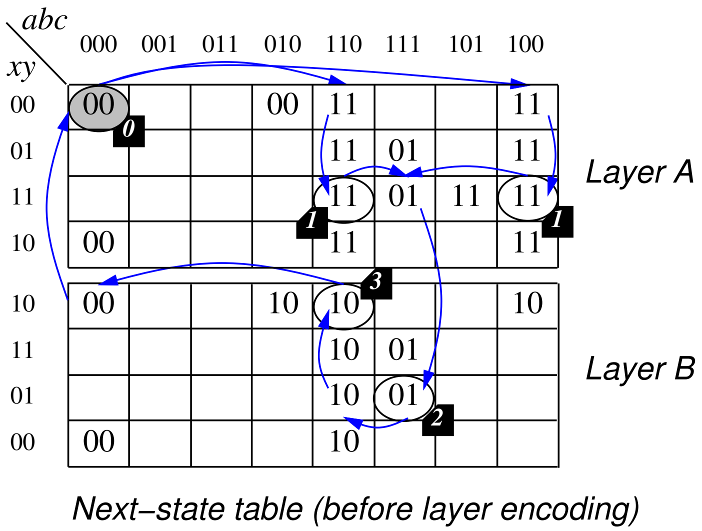


Figure 2.6: Simple example (next-state table before layer encoding).

($abcxy$ = 11111) and change the internal state and the outputs concurrently. When the output/state burst is complete, the machine is in S_2. We continue the construction of the next-state table (for outputs xy) by applying the same procedure for the state transitions from S_2 to S_3 and from S_3 to S_0.

The resulting table (in figure 2.6) has two layers, so just one state bit is needed to encode them. The code value of 0 is assigned to layer A, and 1 to layer B. We can complete the construction of the next-state table by encoding the layers in binary and adding the resulting state bits to the next-state entries as shown in figure 2.7. At this point, all reachable entries of the next-state table are specified; next states of the remaining entries are *don't cares*. We can then synthesize the logic directly from the next-state table. A Karnaugh map for the next output function Y is shown in figure 2.8, and the circuit implementation is shown in figure 2.9. Of course, care must be taken to avoid hazards in the logic, when translating a Karnaugh map to logic.

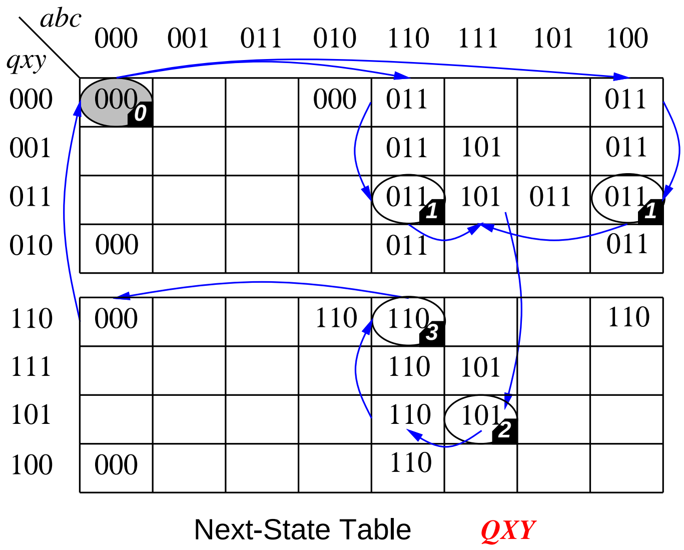


Figure 2.7: Simple example (next-state table after layer encoding).

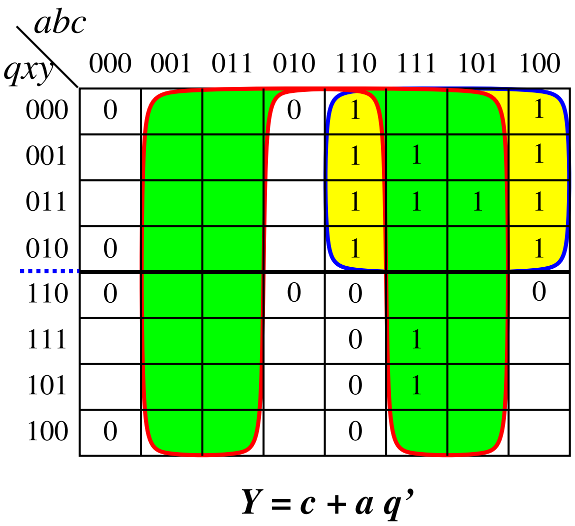


Figure 2.8: Simple example (Karnaugh map for Y).

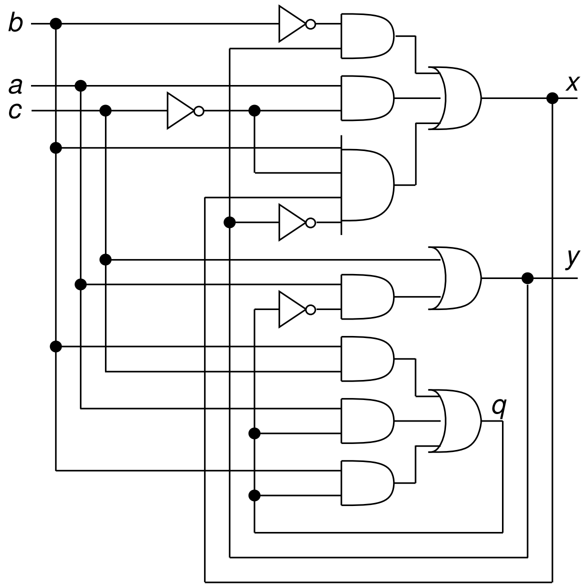


Figure 2.9: Simple example (3D implementation).

2.4 3D Machine Operation

There are three types of machine cycles in a 3D state machine:

Type I. an input burst followed by a concurrent output and state burst;

Type II. an input burst followed by an output burst followed by a state burst;

Type III. an input burst followed by a state burst followed by an output burst.

The selection of a machine cycle depends on the required level of concurrency and the combinational synthesis method used. Normally, the user of the 3D synthesis tool selects Type I or II. Type III is used only to avoid a dynamic hazard that arises in two-level AND-OR due to undirected don't cares, which will be discussed in detail in chapter 3.

At power-up or after completion of the previous machine cycle, the machine waits for an input burst to arrive. In a Type I machine cycle when the machine detects that all of the terminating edges of the input burst have appeared, it generates a concurrent output/state burst (which may be empty), completing a 2-phase machine

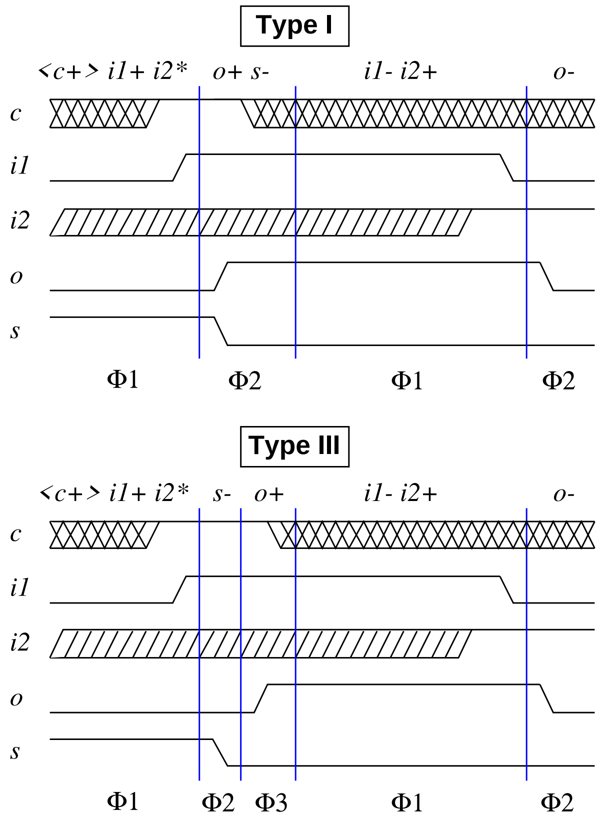


Figure 2.10: 3D machine cycles (Types I and III).

cycle. In a Type II machine cycle, when the machine detects that all of the terminating edges of the input burst have occurred, it generates an output burst (which may be empty). A state burst (which may also be empty) immediately follows the output burst, completing the 3-phase cycle. Note that an output burst enables a state burst in the "burst-mode fashion" — the state variable changes are enabled only after all the changes of the output burst have fed back. In a Type III machine cycle, a state burst is enabled by the input burst and an output burst is enabled by the state burst. Note that the state assignment used in the simple example in the last section forced the machine cycle in S_1 to be of Type I; however, a state assignment scheme that generates a different type of machine cycle can be used just as well.

Figure 2.10 illustrates examples of two machine cycles (Type I and Type III). The first machine cycle begins with input burst (phase 1) $\langle c^+ \rangle\, {i_1}^+ {i_2}^*$. The conditional signal c stabilizes to 1 before ${i_1}^+$ fires. The directed don't care signal i_2 may remain at 0 or change to 1. In the Type I machine cycle, this input burst enables a concurrent output/state burst (phase 2), $o^+ s^-$. In the Type III machine cycle, this input burst enables the state burst (phase 2), s^-, which, in turn, enables the output burst (phase 3), o^+. In the second machine cycle, an input burst, ${i_1}^- {i_2}^+$, enables an output burst, o^-, and no state burst is required. Thus both the Type I and III machine cycles are identical.

Chapter 3

Hazard Considerations

3.1 Introduction

In this chapter, we pay close attention to the correctness of the implementation and the requirements for correctness. An implementation is correct if and only if the range of possible behavior in the environment of the implementation is a subset of the range of behavior allowed by the specification. One way to guarantee that an implementation is correct is to transform the specification using a procedure each step of which preserves correctness.

The main problem in ensuring the correctness of asynchronous circuits is avoiding the possibility of hazards. A hazard is broadly construed as a potential for malfunction of the implementation. We review precise characterization of various kinds of hazards and describe how each is avoided. We show that the 3D machine synthesis problem reduces to one of synthesizing hazard-free combinational logic and then show how the various sources of hazards are systematically eliminated.

Figure 3.1 illustrates how the 3D machine can be viewed as a combinational logic function during each burst (Type II machine cycle is used in this example). Assume that no fed-back output change arrives at the network input until all of the specified changes of the output burst have appeared at the network output. The same assumption applies to the fed-back state variable changes and the state burst. These conditions will be met by inserting delays in the feedback paths as necessary. The machine then can be viewed as a combinational logic function

1. excited by the input changes during the input bursts (phase 1);

2. excited by the fed-back output changes during the output bursts (phase 2);

3. excited by the fed-back state variable changes during the state burst (phase 3).

Note that the machine is stable at the beginning of each phase.

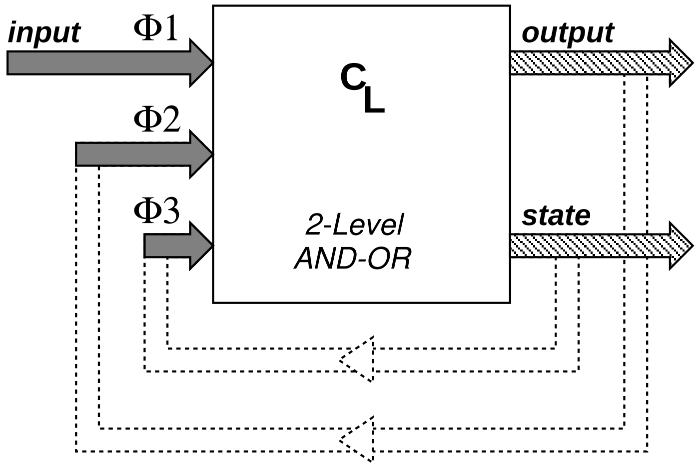


Figure 3.1: Combinational view of the 3D state machine.

Therefore, the 3D machine synthesis procedure follows the steps outlined below:

1. specifying a hazard-free combinational logic function that can be transformed into a hazard-free logic circuit;

2. implementing a hazard-free combinational circuit from the specified combinational function;

3. ensuring that the sequential circuit created by connecting feedback paths are free of hazards.

The first step of the synthesis procedure is to correctly specify a combinational logic function that conforms to the specification. This step must ensure that the specified function is free of *function hazards*, that is, for every set of input changes and feedback signal changes with all the signals not specified to change set to correct values, both the static and dynamic behavior of every output is exactly as specified. In addition, this functional synthesis step must take measures to ensure that a hazard-free circuit exists for the specified function.

The second step of the synthesis procedure is to correctly implement a combinational logic circuit from the combinational function specified in the last step. That is, this step must implement a circuit free of *logic hazards*.

The last step of the synthesis procedure is to turn this combinational circuit into a sequential circuit by connecting outputs of the network to the inputs, that is, creating feedback paths. This step must ensure that the sequential circuit created by connecting feedback paths is free of *sequential hazards*, that is, the circuit behaves as specified as a sequential machine.

In the remainder of this chapter, we examine the sources of hazards (sequential hazards, function hazards, and combinational logic hazards) in detail and provide remedies for each. The synthesis procedure itself and the algorithms are presented in chapter 4.

3.2 Sequential Hazard

The correct operation of the 3D machine relies on the assumption that all of the specified changes of the outputs of the combinational network excited by a set of changes at the network inputs are completed before the next set of changes arrives at the network inputs. A violation of this assumption may result in a *sequential hazard*, the hazard that exists regardless of the correctness of the underlying combinational circuit. Both the timing characteristics of the circuit itself and the environment of the circuit can cause sequential hazards.

3.2.1 Essential Hazard

We examine how the internal timing of the circuits can introduce sequential hazards. It has been assumed up to now that no change at the network output is fed back to the input of the combinational network until all the changes at the network outputs that are concurrently enabled have taken place. However, this assumption may be violated if feedback delays are short compared to the difference between the maximum and minimum feedforward delays. The hazard that arises due to the race between the arrivals of input edges and one or more fed-back output edges, enabled by the same input changes, at the network input is called *essential hazard*.

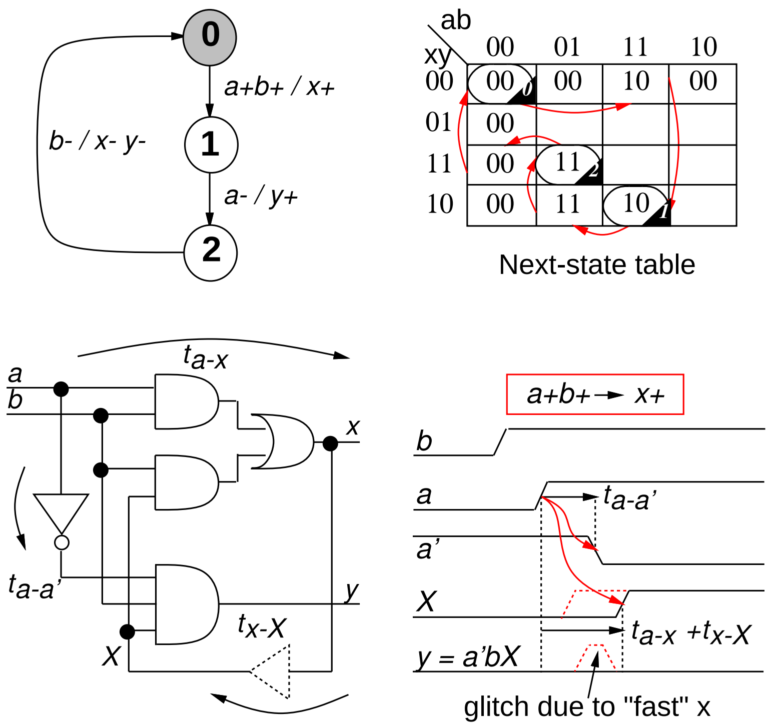


Figure 3.2: Essential hazard.

The possibility of an essential hazard during a $0 \rightarrow 0$ transition of an output is illustrated in figure 3.2. During the input/output burst $(a^+b^+ \rightarrow x^+)$, y is specified to remain 0. However, if x^+ is fed back to the network input before a' goes low, then a $0-1-0$ glitch may propagate to output y. Thus, we need to make sure that the feedback delay (t_{x-X}) is sufficiently large to avoid essential hazards.

Essential hazards, in general, can be avoided simply by *inserting sufficient delays* in the feedback paths. However, the delays in the feedback paths increase the delay constraint between last output change and next compulsory input change that must be obeyed by the environment of the circuit. Hence, it is desirable to minimize feedback delays to improve system performance. Sometimes, it is possible to find tighter constraints, i.e., reduce feedback delays, if the details of the implementation technology are known.

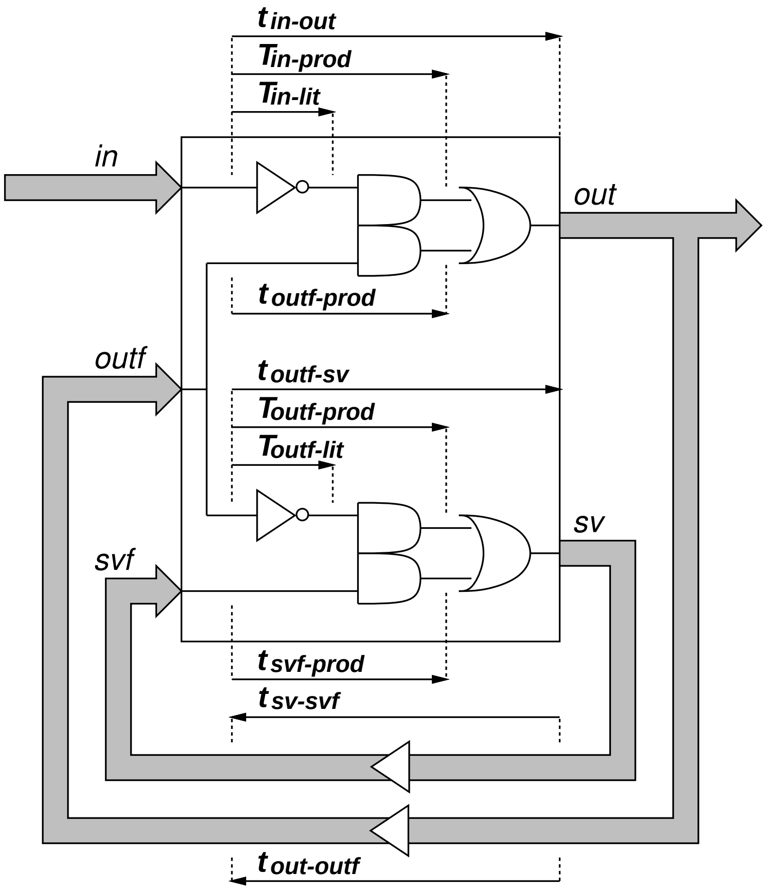


Figure 3.3: Timing requirements for minimum feedback delay.

Sufficient Conditions for Freedom from Essential Hazards in 2-Level AND-OR

If 3D machines are implemented in two-level AND-OR, a set of simple one-sided timing constraints can be used to characterize the minimum required feedback delay. We show below a set of timing constraints required for Type II machine cycles. t_{x-y}

denotes the minimum delay from a transition of type x to a transition of type y, while T_{x-y} denotes the maximum delay (see figure 3.3).

1. $t_{in-out} + t_{out-outf} > T_{in-lit}$.

2. $t_{in-out} + t_{out-outf} + t_{outf-prod} > T_{in-prod}$.

3. $t_{outf-sv} + t_{sv-svf} > T_{outf-lit}$.

4. $t_{outf-sv} + t_{sv-svf} + t_{svf-prod} > T_{outf-prod}$.

Usually, these inequalities are satisfied without adding delays, as should be clear by comparing the lengths of the paths followed on each side of the inequalities. Note that the requirements for a Type I machine cycle are simpler, because state variable changes are concurrent with output changes: only the first two inequalities are needed.

3.2.2 Environmental Constraints

An inherent feature of the 3D implementation is that parts of the circuit may still be unstable after a change at the network output has taken place. In some sense, this feature can help improve the performance of the system by effectively making the stabilization of the circuit and the reaction of the environment concurrent, provided that the environment is slow to react to the changes in the circuit outputs. However, if the environment reacts so fast that the circuit detects the new input arrivals before the arrival of feedback variable changes, then the circuit may malfunction. Therefore, we must have the environment delay generating certain changes. This is called the *fundamental-mode* environmental constraint. In practice, this is usually not a problem, because of the delays in wires between the circuit and the environment and the time for the environment to react are generally longer than it takes for the circuits to stabilize. In addition, not all the input signals have to meet this constraint, because some signals are specified as don't cares in the extended-burst-mode.

Another form of the environmental constraint required by the extended-burst-mode 3D machine are the *setup time* and *hold time* requirements: all conditional signals specified to stabilize must stabilize for some interval before any compulsory (sampling) edge appears and must remain stable until the output/state burst is completed. This requirement is similar to the setup and hold requirements on data signals with respect to clock of synchronous flip-flops.

3.2.3 Summary

The following are the timing requirements imposed by the synthesis method to guarantee correctness of the implementation.

1. *feedback delay requirement* : feedback variable changes are not fed-back until all enabled feedback variable changes are completed;

2. *fundamental-mode environmental constraint* : no compulsory edges of the next input burst may arrive until the machine is stabilized;

3. *setup and hold time requirements* : all conditional signals specified to be stable must be stabilize before any compulsory (sampling) edge appears and must remain stable until the output/state burst is completed.

Assuming these timing constraints are met, we need only analyze the hazards in the combinational circuit that results from cutting feedback paths.

3.3 Function Hazard

A *function hazard* is a non-monotonic change, i.e., more than one change, of a combinational function during a multiple-input change [23, 69]. Function hazards are problematic because they are present in *every* gate-level implementation of the function, if inputs to functions have arbitrary delay. Consequently, function hazards must be prevented before combinational synthesis. We consider function hazards during multiple-input changes in which some inputs are non-monotonic, i.e., change more than once. We examine the implications of allowing certain input changes to be non-monotonic, define what a function hazard is in this setting, and explain how function hazards are avoided in the 3D implementations.

3.3.1 Definitions

We summarize some definitions and concepts from [5, 4, 53, 60] that are used in the following subsections.

A *logic function* f is a mapping from $\{0,1\}^n$ to $\{0,1,*\}$. A *minterm* of f is an n-tuple $[x_1, x_2, \ldots, x_n]$ where x_i, the value of the i-th input of f, is 0 or 1.

The *on-set* of f is the set of minterms for which f is 1; the *off-set* of f is the set of minterms for which f is 0; the *dc-set* of f is the set of minterms for which f is $*$.

A *cube* c, written as $[c_1, c_2, \ldots, c_n]$, is a vector in $\{0,1,*\}^n$. A minterm $[x_1, x_2, \ldots, x_n]$ is a cube such that for every $i \in 1, \ldots, n$, $x_i \neq *$.

A cube $[a_1, a_2, \ldots, a_n]$ is said to *contain* another cube $[b_1, b_2, \ldots, b_n]$ iff, for all i in $1, \ldots, n$, $a_i = b_i$ or $a_i = *$.

A cube $[a_1, a_2, \ldots, a_n]$ is said to *intersect* another cube $[b_1, b_2, \ldots, b_n]$ iff, for all i in $1, \ldots, n$, $a_i = b_i$ or $a_i = *$ or $b_i = *$.

A *literal* is a variable or its complement. A *product term* is a boolean product of literals, and a *sum of products* is a boolean sum of product terms. We consider only product terms satisfying the restriction that *no product term can have both a variable and its complement as inputs.* With this restriction, there is a one-to-one correspondence between product terms and cubes, so we use the terms *cube* and *product term* interchangeably. Thus a product term $x_1\overline{x_3}x_4$ is equivalent to a cube $[1, *, 0, 1, *, \ldots, *]$.

An *implicant* of f is a product term which contains no off-set minterms of f.

A *cover* C of a logic function f is a set of implicants of f such that every on-set minterm of f is contained in some cube of C but no off-set minterm. A cover is isomorphic to a sum-of-products implementation of f.

If $A = [a_1, a_2, \ldots, a_n]$ and $B = [b_1, b_2, \ldots, b_n]$, the *transition cube* $C = [c_1, c_2, \ldots, c_n]$ is determined so that, for $i = 1, \ldots, n$, $c_i = a_i = b_i$, if $a_i = b_i$, and $c_i = *$, if $a_i \neq b_i$. The transition cube C, denoted as $[A, B]$, is the smallest cube that contains both A and B.

A *trajectory* in $[A, B]$ is a vector of minterms contained $[A, B]$, denoted as $[m_1, m_2, \ldots, m_p]$, such that, for every j in $1, \ldots, p-1$, the minterms m_j and m_{j+1} differ in just one bit position.

A combinational function has a function hazard if it changes more than once during a specified multiple-input change. Assume, for now, that all input changes are monotonic (we will generalize it so that some input changes can be non-monotonic in the following subsection). There is a corresponding transition cube for every multiple-input change. The transition cube contains all of the minterms in every possible trajectory of the specified input changes. If the function changes its value more than once along a certain trajectory, then there is a function hazard. The following "classical" definition of function hazard adapted from [4] captures this notion precisely.

Definition 3.1 *A combinational function f contains a* **function hazard** *during a multiple-input change from A to B iff there exists a pair of minterms X and Y in $[A, B]$ ($A \neq X$ and $Y \neq B$) such that*

1. $X \in [A, B]$ *and* $Y \in [X, B]$ and
2. $f(A) \neq f(X)$ *and* $f(Y) \neq f(B)$.

If $f(A) = f(B)$, it is a *static function hazard*, that is, a $1-0-1$ or $0-1-0$ function hazard. Otherwise, it is a *dynamic function hazard*, that is, a $1-0-1-0$ or $0-1-0-1$ function hazard.

3.3.2 Generalized Transition

If some inputs are allowed to change non-monotonically during multiple-input changes, the classical definition of function hazard is inadequate. We develop a notion of generalized transition to remedy this deficiency and to provide a vehicle to discuss functional synthesis in analytical terms in chapter 4.

A *generalized transition* (T, A, B) defines a set of all *legal* trajectories in $[A, B]$, where A is a *start cube*, B is an *end cube*, and T is a mapping from a set of inputs to a set of *input types*. There are three types of inputs: *rising edge*, *falling edge*, and *level* signals. Edge inputs can only change monotonically; therefore, edge inputs change at most once in a legal trajectory. Level inputs must remain constant or be undefined (don't care), which implies that each level input must hold the same value in both

A and B or be undefined in both A and B. Level inputs, if they are undefined, may change non-monotonically,

A generalized transition cube $[A, B]$ is the smallest cube that contains the start and end cubes A and B, as defined before. *Open generalized transition cubes*, $[A, B) = [A, B] - B$, $(A, B] = [A, B] - A$, and $(A, B) = [A, B) - A$ respectively. Note that $[A, B) = \emptyset$, if $A = B$. The *start subcube* A' is a maximal subcube of A such that:

1. the value of every rising edge input i in A' is 0, if it is $*$ in A;
2. the value of every falling edge input j in A' is 1, if it is $*$ in A.

The *end subcube* B' is a maximal subcube of B such that:

1. the value of every rising edge input i in B' is 1, if it is $*$ in B;
2. the value of every falling edge input j in B' is 0, if it is $*$ in B.

Intuitively, if edge signals have weight 1 and level signals have weight 0, the trajectories from A' to B' are the maximum-weight trajectories. If every don't care input is an edge signal in (T, A, B), $[A', B'] = [A, B]$ and A' and B' reduce to minterms.

Lemma 3.1 *For every minterm X in $[A, B]$, all of the minterms in every legal trajectory from X to B is contained in $[X, B']$.*

Proof: We prove by contradiction. Assume that there exists a trajectory such that one of the minterms in the trajectory is outside of $[X, B']$.

1. $[X, B'] = [X, B]$
   $[X, B]$ contains all of the minterms in every trajectory from X to B, which contradicts the assumption.
2. $[X, B'] \subset [X, B]$
   Then there exists Y contained in $[X, B] - [X, B']$ such that Y can be reached from X legally. Let $X = [x_1, \ldots, x_n]$, $Y = [y_1, \ldots, y_n]$, $B' = [b'_1, \ldots, b'_n]$, and $[X, B'] = [c_1, \ldots, c_n]$. For every i in $1, \ldots, n$, $c_i = *$ or $c_i = b'_i$. Since Y is not contained in $[X, B']$, there exists an edge signal j such that $y_j \neq * \wedge y_j \neq b'_j \wedge c_j \neq *$. Because $c_j \neq *$, $x_j = b'_j$, which means x_j changed once. Thus there is no legal trajectory from X to Y, which contradicts the assumption.

□

During a generalized transition (T, A, B), each output signal is assumed to change its value at most once. Furthermore, no output change is allowed in A and B. If not, a function hazard is said to be present. Below is the new definition of function hazard adapted for generalized transitions:

Definition 3.2 *A combinational function f contains a* **function hazard** *in (T, A, B) iff*

1. *there exists a pair of minterms X, Y in A such that $f(X) \neq f(Y)$,* or
2. *there exists a pair of minterms X, Y in B such that $f(X) \neq f(Y)$,* or
3. *there exists a pair X, Y in (A, B) such that $Y \in [X, B')$* (or, equivalently, $X \in (A', Y]$) *and $f(A) \neq f(X)$ and $f(Y) \neq f(B)$.*

The last criterion states that there is a function hazard if there exist two minterms X and Y in a legal trajectory from A to B such that $f(A) \neq f(X)$ and $f(Y) \neq f(B)$.

A generalized transition (T, A, B) is a static transition for f iff $f(A) = f(B)$; it is a dynamic transition for f iff $f(A) \neq f(B)$. No change in level inputs can enable output changes directly, that is, at least one edge input must change from 0 to 1 or from 1 to 0 in a generalized dynamic transition.

Examples of generalized transitions are shown in figure 3.4. a, b, and c are rising edge signals, and s is a level signal. Figures 3.4ac show function-hazard-free static and dynamic transitions respectively. Figure 3.4b illustrates a 1-0-1 static function hazard, and figure 3.4d does a 0-1-0-1 dynamic function hazard on the trajectory, $abc : 000 \to 100 \to 101 \to 111$.

3.3.3 Extended-Burst-Mode Transition

An extended-burst-mode transition is a generalized transition with the following requirements:

1. For every pair of minterms X and Y in $A \cup [A, B)$, $f(X) = f(Y)$.

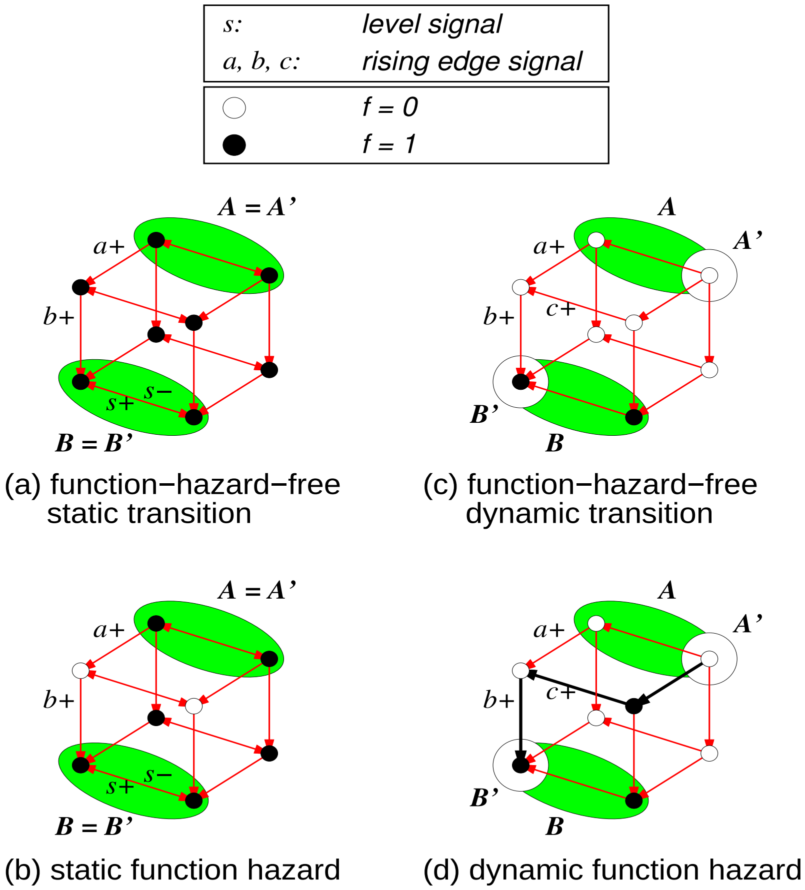


Figure 3.4: Generalized transitions.

2. For every pair of minterms X and Y in B, $f(X) = f(Y)$.

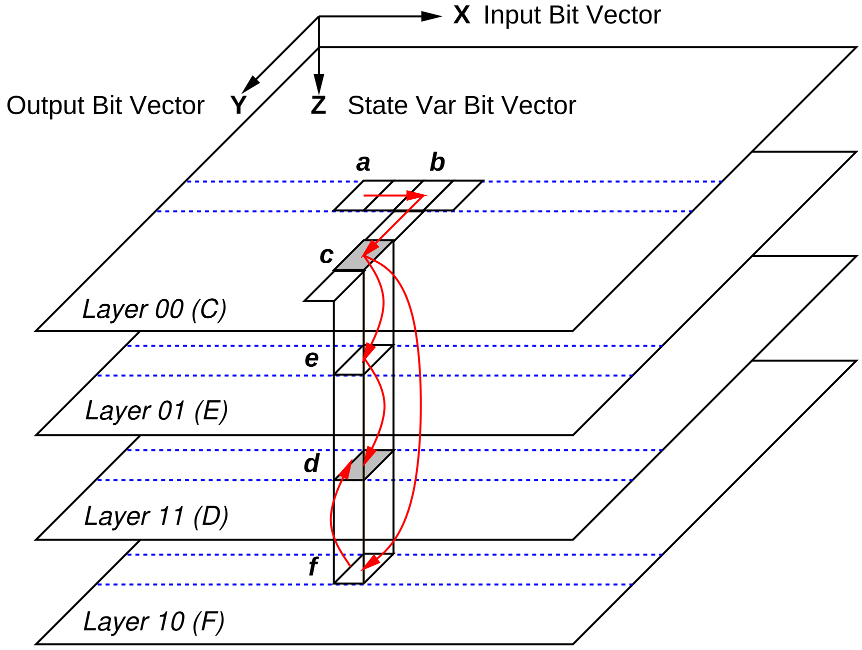
Theorem 3.1 *Every extended-burst-mode transition is function-hazard-free.*

Proof: Consider f during an extended-burst-mode transition from A to B. Since $A \subseteq A \cup [A, B)$, for every pair of minterms X and Y in A, $f(X) = f(Y)$ by requirement 1 of the definition of extended-burst-mode transition. This contradicts criterion 1 of Definition 3.2. For every pair of minterms X and Y in B, $f(X) = f(Y)$ by requirement 2, which contradicts criterion 2 of Definition 3.2. Finally, for all X in (A, B), $f(X) = f(A)$ by requirement 1, which contradicts criterion 3 of Definition 3.2. Therefore, f is free of function hazards. $\square$

An edge signal that changes from 0 or $*$ to 1 or from 1 or $*$ to 0 during an extended-burst-mode transition from A to B is a *terminating* signal in (T, A, B). An edge signal whose value is $*$ in B is a *directed don't care* in (T, A, B). A level signal whose value is $*$ in (T, A, B) is an *undirected don't care*. In a dynamic extended-burst-mode transition, the output is enabled to change only after all of the terminating edges appear.

Another way of describing terminating signals and don't cares is as follows: Let minterms $X = [\ldots, x_s, \ldots]$ and $X' = [\ldots, \overline{x_s}, \ldots]$, where x_s and $\overline{x_s}$ are the values of s in X and X'. s is a terminating signal iff $X \in B$ implies $X' \notin B$. s is a don't care (directed or undirected) iff $X \in B \Leftrightarrow X' \in B$ or, equivalently, $X \in [A, B) \Leftrightarrow X' \in [A, B)$.

A 3D machine cycle that requires no conditional signals to stabilize has transitions corresponding to an input burst and a concurrent output/state burst, if it is of Type I, or an input burst, an output burst and a state burst, if it is of Type II or III. A 3D machine cycle that requires conditional signals to stabilize has an additional transition for setting up conditional signals. Each of these transitions by itself is free of function hazards, since these are all extended-burst-mode transitions. However, as we have seen in the simple example in section 2.3.1, a function-hazard-free next-state assignment requirement for one transition may conflict with another transition. The 3D state assignment algorithm avoids this type of conflict by adding state variables when necessary, as described in the next chapter.



State		Next-state
a	□ beginning of input burst	***a***
b	□ end of input burst / beginning of output burst	***c***
c	■ end of output burst / beginning of state burst	***d***
e, f	□ transient states ***(critical race if d is not next-state)***	
d	■ end of state burst	***d***

Figure 3.5: Critical race.

3.3.4 Critical Race

If a transition between layers requires multiple state bit changes (see figure 3.5), the machine traverses intermediate layers (E or F) before it settles down to the final stable state (d). In traditional asynchronous state machines [30, 69], a *critical race* is said to be present if the final stable state depends on the order in which the state bits change. In 3D machines, *a* **critical race** *is said to be present if the transient states during a layer transition have different next values of outputs and state variables from those of the start-state of the transition.* Hence, in 3D machines, a critical race is simply a manifestation of a function hazard during a state burst. We insure that the machine is free of critical races by encoding layers so that the next states of the transient states during layer transitions are the same as those of the start-state of the transition.

3.4 Combinational Logic Hazards

Hazards in combinational circuits can also be introduced by the delay variations of physical gates and wires, even if the logic functions are completely and correctly specified, i.e., function-hazard-free. In this section, we present two different methods to implement hazard-free combinational logic: the two-level AND-OR implementation [53] and the multiplexor tree implementation derived from a Binary Decision Diagram (BDD) representation of the next-state function [35].

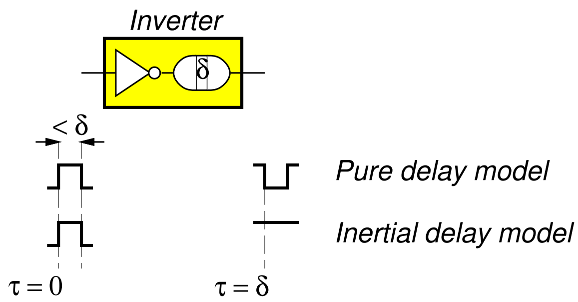


Figure 3.6: Delay model.

The existence of hazards depends on the delay assumptions in the circuit model used and on the models of the delay itself. Lots of delay models have been proposed [69, 63, 9], here are two commonly used examples: the *inertial* delay model which assumes that no input pulse of duration shorter than the gate delay is transported to the output of the gate, and the *pure* delay model which assumes that a pulse of any duration computed by the logic function of the gate is asserted on the gate output.

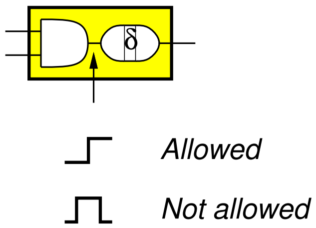


Figure 3.7: Delay model used in 3D synthesis.

Our synthesis method works for *all* delay models, because we use a strategy to avoid multiple input changes to a delay before output (see figure 3.7). In addition, we assume that both the gates and the wires connecting gates in the combinational network have finite but arbitrary delays.

The following definitions of logic hazards are from [4, 53].

Definition 3.3 *A combinational network contains a* **static logic hazard** *during a function-hazard-free input change from A to B iff*

1. $f(A) = f(B)$;

2. *A momentary pulse may be present during the input change from A to B.*

Definition 3.4 *A combinational network contains a* **dynamic logic hazard** *during a function-hazard-free input change from A to B iff*

1. $f(A) \neq f(B)$;

2. *A momentary 0 and a momentary 1 output may appear during the input change from A to B.*

3.4.1 Two-Level AND-OR Implementation

First, we consider the implementation of the next-state functions in two-level AND-OR logic. We develop a set of hazard-free covering requirements for the 2-level AND-OR implementation of a logic function during an extended-burst-mode transition. The hazard-free combinational logic synthesis for multiple *monotonic* input changes is described in [23, 5, 3, 4, 53, 77]. The *new* results presented here are simple extensions of the theory in [53] to account for non-monotonic input changes. We apply these results to the 3D machine combinational logic synthesis.

Below we state and prove necessary and sufficient conditions for hazard freedom for a two-level AND-OR circuit during an extended-burst-mode transition. Note that the *product term* refers to an *on-set* cube for the remainder of this section.

Lemma 3.2 *A product term that does not intersect the generalized transition cube* $[A, B]$ *remains 0 during a function-hazard-free transition* (T, A, B).

Proof: Every product term that does not intersect $[A, B]$ has a literal whose value remains 0 during the input change. Thus a product term not intersecting $[A, B]$ remains 0. □

Lemma 3.3 *A product term that contains* A' *(or* B'*) changes monotonically during an extended-burst-mode transition* (T, A, B).

Proof: First, consider the case in which a product term p contains both A' and the start-point of a trajectory in the transition (T, A, B). The initial values of all the literals of p are 1. Level signals are either constants or don't cares in A'. If a level signal is a don't care in A', then it is a don't care in the cube that contains A'; therefore, it does not appear as a literal in the corresponding product term. Since all other input changes are monotonic, values of the literals change monotonically from 1 to 1 or from 1 to 0. Thus the output of p changes monotonically.

Now consider the case in which the product term, p, contains A' but not the start-point. By the definition of A', at least one monotonic change of an edge signal is needed to traverse from A' to a start-point in $A - A'$; no additional change of the same signal occurs in $[A, B]$. The value of the literal in p which corresponds to this input signal falls during a transition from A' to the start-point and remains 0. Thus the output of p remains 0 in $[A, B]$, if the start-point of the trajectory is not contained in p.

Thus the output of a product term that contains A' changes monotonically ($1 \rightarrow 1$, $1 \rightarrow 0$, or $0 \rightarrow 0$).

Using the same argument, the output of a product term that contains B' also changes monotonically ($0 \rightarrow 0$, $0 \rightarrow 1$, or $1 \rightarrow 1$). □

Theorem 3.2 *The output of a two-level AND-OR circuit is hazard-free during a $0 \rightarrow 0$ extended-burst-mode transition.*

Proof: No product term intersects the transition cube since the transition is function-hazard-free. Thus all the product terms in the network remain 0 during the transition by lemma 3.2. □

Theorem 3.3 *The output of a two-level AND-OR circuit is hazard-free during a $1 \rightarrow 1$ extended-burst-mode transition iff the circuit contains a product term which contains the transition cube $[A, B]$.*

Proof: ($\Rightarrow$) Assume that the circuit does not contain a product term that contains $[A, B]$. In order for the transition to be function-hazard-free, $[A, B]$ is covered by more than one product term. During a transition from A to B, one or more product terms rise, one or more product terms fall, and the rest remain 0. If a falling edge of a product term precedes all rising edges, the output of the circuit may change from 1 to 0 to 1, which is a hazard, contradicting the hypothesis.

($\Leftarrow$) The output of a product term that contains $[A, B]$ remains 1 during a transition from any point in A to any point in B, because there is no literal in this product term that can change in $[A, B]$. Hence, the sum of products remains 1 throughout the trajectory. □

Theorem 3.4 *The output of a two-level AND-OR circuit is hazard-free during a $1 \rightarrow 0$ extended-burst-mode transition iff every product term intersecting the transition cube $[A, B]$ also contains the start subcube A'.*

Proof: ($\Rightarrow$) Assume that there exists a product term p that intersects $[A, B]$ but does not contain A'. Consider a trajectory from a point in A' not contained in p to any point in B. The initial value of p is 0 since p does not contain the start-point. The final value of p is 0 because the final value of the output of the network must be 0. Because p intersects $[A, B]$, p changes from 0 to 1 to 0 on some trajectories from A' to B. All other product terms that contain A' fall during a transition from A' to B. Since the wire delay on p can be arbitrary, the output of the network may undergo a $1 \to 0 \to 1 \to 0$ transition. Thus the circuit is not hazard-free, which contradicts the hypothesis.

($\Leftarrow$) The final values of all the product terms are 0, because the final value of the output of the network must be 0. By lemma 3.3, the product terms that contain A' change monotonically during a transition from A to B. Thus the product terms that intersect $[A, B]$ fall monotonically. The product terms that do not intersect $[A, B]$ remain 0, by lemma 3.2. Thus the output of the network changes monotonically, i.e., hazard-free. □

Theorem 3.5 *The output of a two-level AND-OR circuit is hazard-free during a* $0 \to 1$ *extended-burst-mode transition iff no product term intersects the transition cube* $[A, B]$ *unless it also contains the end subcube* B'*.*

Proof: Exchange 0 and 1 and reverse trajectories in proof of Theorem 3.4. □

The hazard-free covering requirements for two-level AND-OR logic for extended-burst-mode transitions can be summarized as below:

1. For every $1 \to 1$ transition:

   There exists a product term that contains $[A, B]$.

2. For every $1 \to 0$ ($0 \to 1$) transition:

   Every product term that intersects $[A, B]$ must also contain A' (B').

Each maximal subcube of $[A, B]$ needed to satisfy the covering requirements above is called a *required cube* of $[A, B]$ [53, 49]. Just one cube is required for a $1 \to 1$ transition whereas n cubes are required for a $1 \to 0$ transition enabled by n terminating input edges. Figure 3.8 illustrates the hazard-free covering requirements for the simple example from section 2.3.1. Each unshaded circle represents a required cube for a $1 \rightarrow 1$ transition; shaded circles constitute a set of required cubes for a $1 \rightarrow 0$ transition.

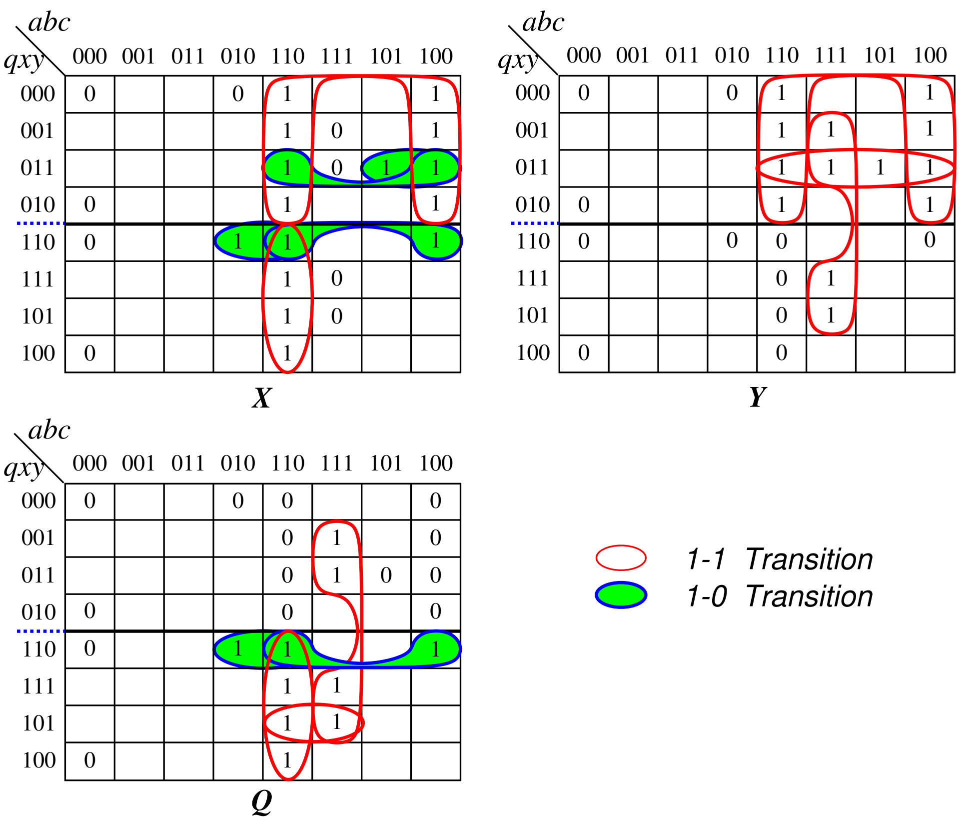


Figure 3.8: Simple example (required cubes).

Suppose a generalized transition cube $[A, B]$ for a $1 \rightarrow 0$ extended-burst-mode transition is intersected by a required cube c_r (required for another transition). If c_r does not contain A' and cannot be expanded (by assigning 1 to don't care entries) to contain A', then the implementation has a dynamic logic hazard. Figure 3.9 illustrates four examples of illegal intersections of transition cubes. In each of these cases, a dynamic logic hazard is present in the implementation of x — for instance, in figure 3.9b, the output of c_3 may glitch $(0 - 1 - 0)$, if s rises momentarily before a rises but the output of c_3 is slow to change, and this glitch may propagate to the output. This observation leads to the notion of privileged cube.

A generalized transition cube $[A, B]$ for a $1 \to 0$ extended-burst-mode transition is said to be a *privileged cube* [53, 49] iff $[A, B)$ contains more than one minterm. Likewise, a generalized transition cube $[A, B]$ for a $0 \to 1$ extended-burst-mode transition is said to be a *privileged cube* iff B contains more than one minterm. A cube that intersects a privileged $1 \to 0$ transition cube must also contain the start subcube, and a cube that intersects a privileged $0 \to 1$ transition cube must also contain the end subcube. Otherwise, the cube is said to intersect the privileged cube *illegally*.

Let C be a cover of a logic function f that implements an output or a state variable of the 3D machine. C is free of logic hazards iff it includes all of the required cubes and no cube in C intersects a privileged cube illegally.

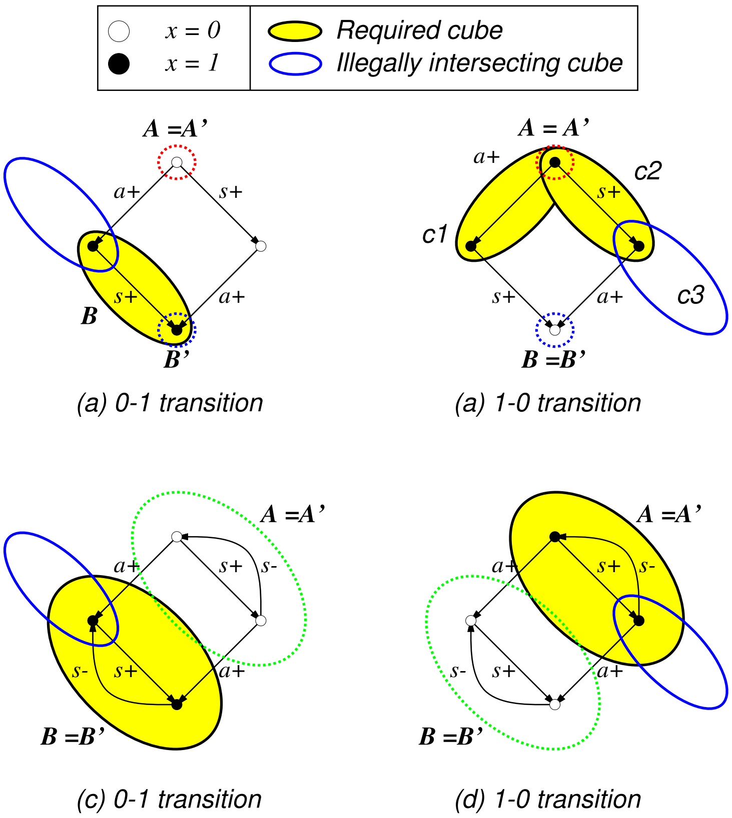


Figure 3.9: Illegal intersection of privileged cube.

3.4.2 BDD Implementation

Here we present a radically different alternative method for implementing hazard-free combinational circuits. In this method, combinational networks that describe next-state functions are multiplexor trees constructed from BDD (binary decision diagram) descriptions. Lin and Devadas presented a method to synthesize a combinational logic, which is hazard-free for a set of multiple-input changes, for a specified function by building a BDD for the function and deriving a multi-level circuit from it in [35]. We observed that it is possible to apply this technique for synthesizing a combinational logic which is hazard-free for a set of *extended-burst-mode transitions*. This requires an extension of their theory to handle non-monotonically changing signals.

The following definition of a Binary Decision Diagram is from [8].

Definition 3.5 *A* **Binary Decision Diagram** *is a rooted, directed graph with vertex set V containing two types of vertices. A* **non-terminal** *vertex v has as attributes an argument index* $\mathsf{index}(v) \in \{1, \ldots, n\}$ *and two children* $\mathsf{low}(v), \mathsf{high}(v) \in V$*. A* **terminal** *vertex v has as attributes a value* $\mathsf{value}(v) \in \{0, 1\}$.

The correspondence between a BDD and a Boolean function is defined as below:

Definition 3.6 *A Binary Decision Diagram G having root vertex v denotes a function f_v defined recursively as:*

1. *If v is a terminal vertex:*
   (a) If $\mathsf{value}(v) = 1$*, then* $f_v = 1$.
   (b) If $\mathsf{value}(v) = 0$*, then* $f_v = 0$.

2. *If v is a non-terminal vertex with* $\mathsf{index}(v) = i$*, then*
   $f_v(x_1, \ldots, x_n) = \overline{x_i} \cdot f_{\mathsf{low}(v)}(x_1, \ldots, x_n) + x_i \cdot f_{\mathsf{high}(v)}(x_1, \ldots, x_n)$.
   x_i is called the **decision variable** *for vertex v.*

In addition,

1. *Each decision variable occurs at most once on every path from a terminal vertex to the root vertex,*

2. *A* **reduced** *BDD is a BDD in which* $\mathsf{low}(v) \neq \mathsf{high}(v)$ *for every vertex* v *and no two subgraphs are identical.*

In a reduced BDD, each path from a terminal vertex to the root vertex corresponds to a cube of the logic function the BDD represents.

A *reduced ordered BDD* (ROBDD) is a canonical form with the following restriction: for any non-terminal vertex v, if $\mathsf{low}(v)$ is a non-terminal, then $\mathsf{index}(v) < \mathsf{index}(\mathsf{low}(v))$, and if $\mathsf{high}(v)$ is a non-terminal, then $\mathsf{index}(v) < \mathsf{index}(\mathsf{high}(v))$.

A *reduced free BDD* (free BDD) is a BDD which does not require a strict variable ordering (unlike in an OBDD) but still requires that each decision variable is encountered at most once when traversing a path from a terminal vertex to the root vertex.

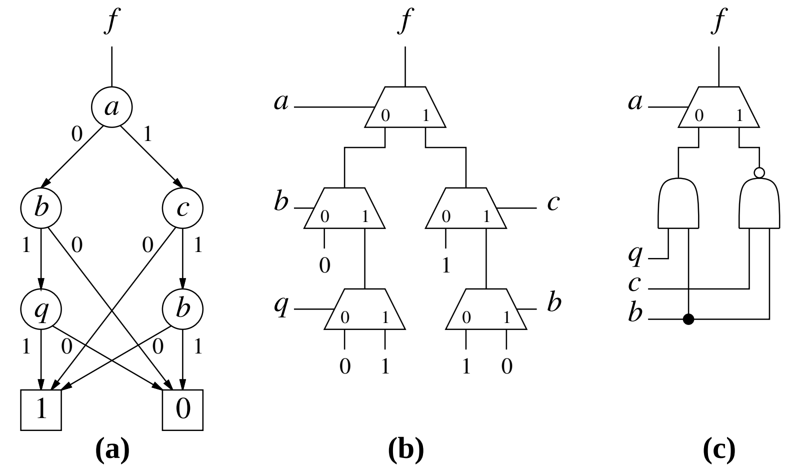


Figure 3.10: (a) BDD (b) MUX network derived from BDD (c) Simplified network (by constant propagation).

A multi-level network can be derived directly from a BDD by replacing each vertex with a two-input MUX with the decision variables as the select inputs of the MUXes. Figure 3.10b shows a MUX network derived from the BDD in figure 3.10a. If one or more input of a MUX is constant, the MUX can be replaced with a simpler gate, such as a NAND or a NOR. This *constant propagation* is carried out topologically from inputs to outputs. Constant propagation cannot introduce hazards. Figure 3.10c shows an equivalent network after constant propagation. The basic gates that comprise the resulting combinational network are ANDs, ORs, NANDs, NORs, inverters, and MUXes. We assume that every basic gate is *atomic*, i.e., a single change of a gate input cannot cause a multiple change at the output. Figure 3.11 shows a CMOS pass transistor implementation of a multiplexor. This implementation is atomic, if the delays t_1 and t_2 are closely matched.

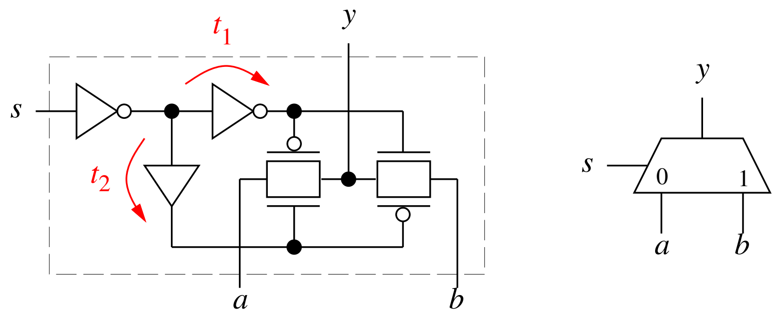


Figure 3.11: A CMOS multiplexor.

To ensure that the resulting multi-level circuit is hazard-free, a requirement called *trigger signal ordering* (TSO) must be satisfied. This requirement imposes constraints on the variable ordering of the BDD. It was shown in [35] that if this variable ordering is satisfied, then the resulting multi-level circuit is free of logic hazards for a set of specified transitions. Note that every input change in [35] was assumed to be monotonic during each transition. We will prove that the resulting circuit is free of logic hazards for a set of specified extended-burst-mode transitions, in which some inputs may change non-monotonically, as long as the TSO requirement is satisfied.

A *trigger state* is a state in which a legal input change enables the output to change. A *trigger signal* is an input signal whose transition enables the output to change in a trigger state; a *non-trigger signal* is an input signal which is enabled to change but cannot by itself enable the output to change. The TSO requirement states that *trigger signals in a trigger state must appear before the non-trigger signals of the same trigger state in the variable ordering.*

In the generalized transition cube that corresponds to an extended-burst-mode dynamic transition, all *terminating* signals are trigger signals in one or more minterms, because terminating edges can appear in any temporal order and the last one that appears is a trigger signal. Note that no terminating signal can be a non-trigger signal, because no output change can be enabled until *all* terminating edges appear. Furthermore, all *don't care* signals (directed or undirected) are non-trigger signals in one or more minterms, because their values may change anywhere, including in the trigger states, in the generalized transition cube. Clearly, no don't care signal can be a trigger signal in any minterm in the generalized transition cube, because don't care signals can never enable outputs to change. Therefore, we can impose a set of ordering requirements, which do not conflict, as a sufficient condition for hazard freedom *per generalized transition cube*, although the TSO requirement in [35] is an imposition on each trigger state in the transition cube.

Now we can state the variable ordering requirements for the extended-burst-mode transitions as follows:

> *Along every path from root to terminal of the BDD whose corresponding cube intersects the generalized transition cube, no don't care signal of a dynamic transition appears before a terminating signal of the same generalized transition cube.*

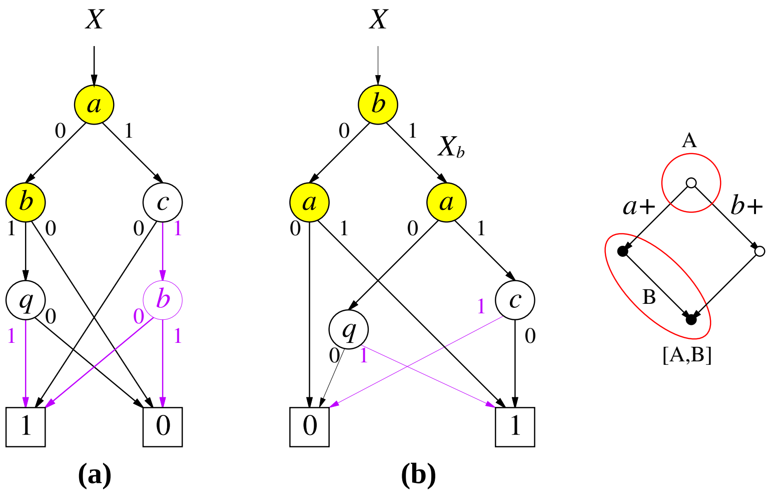


Figure 3.12: Simple example (BDD representations of X).

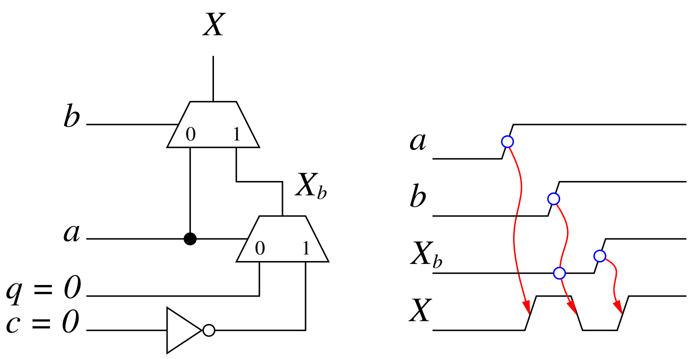


Figure 3.13: Dynamic hazard in BDD-based implementation.

Figure 3.12 shows two different implementations of the next-state function X from the simple example in section 2.3.1. Every path from the root to a leaf corresponds to a cube. Solid arrows represent the cubes that intersect the generalized transition cube corresponding to the extended-burst-mode transition shown on the right. In figure 3.12a, the terminating signal a appears before the don't care b in every path, but the order is reversed in figure 3.12b. Hence, figure 3.12a is hazard-free and figure 3.12b is hazardous. Figure 3.13 illustrates a possibility of a dynamic hazard in figure 3.12b, a direct consequence of the violation of the variable ordering requirement. Initially, $a = b = c = q = X_b = X = 0$. A rising edge of a enables X_b to rise. Since b is a don't care, it may rise anytime. If b rises after a but before X_b, a $0-1-0-1$ glitch may propagate to X. Note that, in order to prevent this hazard, we must ensure that one terminal of the multiplexor remains constant. It is always possible to achieve this for the extended-burst-mode transitions as long as the variable ordering requirement stated above is satisfied, as we will prove below.

Here, we prove that the combinational network derived from a reduced BDD (ordered or free) description is hazard-free during an extended-burst-mode transition as long as the BDD satisfies the variable ordering requirement for the transition: No don't care signal appears before a terminating signal in a dynamic transition.

Lemma 3.4 *If (T, A, B) is an extended-burst-mode transition for f, then $f_{\overline{s}}(X) = f_s(X)$ for every don't care signal s in (T, A, B) and for every minterm X in $[A, B]$.*

Proof: Suppose that s is a don't care in (T, A, B) and a minterm X is in $[A, B]$.

$X = [\ldots, x_s, \ldots]$ and $X' = [\ldots, \overline{x_s}, \ldots]$, where x_s and $\overline{x_s}$ are the values of s in X and X' and all other components are the same. Because s is a don't care in (T, A, B), $X \in B$ implies $X' \in B$ and $X \in [A, B)$ implies $X' \in [A, B)$. Thus $f(X) = f(X')$. If $x_s = 1$, $f(X) = f_s(X)$ and $f(X') = f_{\overline{s}}(X')$. $f_{\overline{s}}(X') = f_{\overline{s}}(X)$ because $f_{\overline{s}}$ is independent of s. Thus $f_s(X) = f_{\overline{s}}(X)$. On the other hand, if $x_s = 0$, $f(X) = f_{\overline{s}}(X)$ and $f(X') = f_s(X') = f_s(X)$. Therefore, $f_s(X) = f_{\overline{s}}(X)$. $\square$

Definition 3.7 *Subtransitions:*

1. *If s is not a constant 0 in (T, A, B), (T, A_s, B_s) is a subtransition of (T, A, B) with the value of s fixed to 1.*

2. *If s is not a constant 1 in (T, A, B), $(T, A_{\overline{s}}, B_{\overline{s}})$ is a subtransition of (T, A, B) with the value of s fixed to 0.*

Note that A_s is not necessarily a subset of A with $s = 1$. For example, if s is a rising edge signal and s is 0 in A, then A_s is not a subset of A with $s = 1$, because $A_s \subset [A_s, B_s]$ but A with $s = 1$ is $\emptyset$. Also note that $(T, A_s, B_s) = (T, A, B)$, if s is a constant 1 in (T, A, B), and $(T, A_{\overline{s}}, B_{\overline{s}}) = (T, A, B)$, if s is a constant 0 in (T, A, B). If $B = [\ldots, b_s, \ldots]$, then $B_s = [\ldots, 1, \ldots]$ and $B_{\overline{s}} = [\ldots, 0, \ldots]$.

Lemma 3.5 *If (T, A, B) is an extended-burst-mode transition for f, then*

1. *(T, A_s, B_s) is an extended-burst-mode transition for f_s, if s is not a constant 0;*

2. *$(T, A_{\overline{s}}, B_{\overline{s}})$ is an extended-burst-mode transition for $f_{\overline{s}}$, if s is not a constant 1.*

Proof: First, we will prove that (T, A_s, B_s) is an extended-burst-mode transition for f_s, if s is not a constant 0.

1. s is a constant 1 in (T, A, B).
   Then $f_s = f$ in $[A, B]$ and $(T, A_s, B_s) = (T, A, B)$. Thus (T, A_s, B_s) is an extended-burst-mode transition for f_s.

2. s is a don't care in (T, A, B).
   Then s is a don't care in B. Thus $B_s \subset B$, so $f(B_s) = f(B)$. $[A_s, B_s) = [A_s, B_s] - B_s = [A_s, B_s] - B \subset [A, B)$, so $f([A_s, B_s)) = f([A, B))$. Thus (T, A_s, B_s) is an extended-burst-mode transition for f_s.

3. s is a rising terminating signal in (T, A, B).
   Then $s = 1$ in B, which implies $B_s = B$ and $f(B_s) = f(B)$. $[A_s, B_s) = [A_s, B_s] - B \subset [A, B)$, so $f([A_s, B_s)) = f([A, B))$. Thus (T, A_s, B_s) is an extended-burst-mode transition for f_s.

4. s is a falling terminating signal in (T, A, B).
   Then $s = 0$ in B. Thus $[A_s, B_s] \cap B = \emptyset$, which implies $[A_s, B_s] \subset [A, B)$. Therefore, $f([A_s, B_s]) = f([A, B))$, which means that (T, A_s, B_s) is an extended-burst-mode transition for f_s.

Similarly, $(T, A_{\overline{s}}, B_{\overline{s}})$ is an extended-burst-mode transition for $f_{\overline{s}}$, if s is not a constant 1. $\square$

Corollary 3.1 *If (T, A, B) is an extended-burst-mode dynamic transition for f and s is a don't care in (T, A, B), then (T, A_s, B_s) is an extended-burst-mode dynamic transition for f_s and $(T, A_{\overline{s}}, B_{\overline{s}})$ for $f_{\overline{s}}$.*

Corollary 3.2 *Static transitions of cofactors:*

1. *(T, A_s, B_s) is a static transition for f_s if (T, A, B) is an extended-burst-mode transition for f and s is a falling terminating signal.*

2. *$(T, A_{\overline{s}}, B_{\overline{s}})$ is a static transition for $f_{\overline{s}}$ if (T, A, B) is an extended-burst-mode transition for f and s is a rising terminating signal.*

Theorem 3.6 *The combinational network C derived from a reduced BDD (ordered or free) description of f is hazard-free during an extended-burst-mode transition if it satisfies the variable ordering requirement for the transition: no don't care signal appears before a terminating signal.*

Proof: We prove this by induction on the number of variables.

Base case: The sole input of the network is connected to the select input of the multiplexor. The other inputs terminals are connected to a constant 1 or 0. Since the multiplexor is atomic and only the select input can change, f is hazard-free.

Inductive hypothesis: Now assume that a combinational network derived from a reduced BDD representation of an n-input function ($n \geq 1$), which satisfies the variable ordering requirements for an extended-burst-mode transition, is hazard-free during the extended-burst-mode transition.

Now consider the network C derived from a reduced BDD representation of a function f with $n+1$ input variables and an extended-burst-mode transition (T, A, B) for f. Assume that the select input of the multiplexor driving the output of C is s and the data inputs are $f_{\overline{s}}$ and f_s, the Shannon cofactors of f with respect to $\overline{s}$ and s. Then (T, A_s, B_s) is an extended-burst-mode transition for f_s if s is not a constant 0, and $(T, A_{\overline{s}}, B_{\overline{s}})$ is for $f_{\overline{s}}$ if s is not a constant 1, by Lemma 3.5. Since f satisfies the variable ordering requirements, so do f_s and $f_{\overline{s}}$. Therefore, f_s is hazard-free if s is not a constant 0, and $f_{\overline{s}}$ is hazard-free if s is not a constant 1, by the inductive hypothesis. We will consider 3 cases: s is a constant, s is a don't care, and s is a terminating signal.

1. s is a constant:
   Thus, if $s = 0$, $f = f_{\overline{s}}$ is hazard-free since s is not a constant 1. Likewise, f is hazard-free, if $s = 1$.

2. s is a don't care:
   First we prove by contradiction that (T, A, B) must be a static transition for f if s is a don't care. Assume that (T, A, B) is a dynamic transition for f. By Corollary 3.1, (T, A_s, B_s) is an extended-burst-mode dynamic transition for f_s. Suppose s remains at 1 while f_s changes. Then the change in f_s propagates to f, which means that there is a terminating signal that enables f to change, regardless of s, violating the variable ordering requirement. Thus (T, A, B) cannot be a dynamic transition, if s is a don't care.

   By Lemma 3.4, $f(X) = f_s(X) = f_{\overline{s}}(X)$ for every X in $[A, B]$. Therefore,

(T, A_s, B_s) and $(T, A_{\overline{s}}, B_{\overline{s}})$ are static transitions of same type, that is, both $0 \to 0$ or both $1 \to 1$, for f_s and $f_{\overline{s}}$ respectively. By the inductive hypothesis, f_s and $f_{\overline{s}}$ are hazard-free, therefore constant. Since the multiplexor is atomic, f is hazard-free.

3. s is a terminating signal:
   Without loss of generality, consider only the case in which s rises. By Corollary 3.2, $f_{\overline{s}}$ undergoes a static transition. By the inductive hypothesis, both f_s and $f_{\overline{s}}$ are hazard-free. Consider the case in which $f_{\overline{s}} = 0$. We prove by contradiction that f_s must rise or remain a constant. Assume that f_s is initially 1 and falls to 0, but s rises first. Since $f_{\overline{s}} = 0$ and $f_s = 1$ initially, f is enabled to rise as s rises. This is a static function hazard, since f is assumed to be 0 at the end of the transition. Thus f_s must rise or remain a constant; in both cases, f is hazard-free. Similarly, we can prove that f is hazard-free when $f_{\overline{s}} = 1$, by proving that f_s must fall or remain a constant.

□

Corollary 3.3 *The combinational network C derived from a reduced BDD (ordered or free) description of f is hazard-free during an extended-burst-mode static transition.*

Table 3.1 summarizes the correspondence between the classifications of signals in the extended-burst-mode transition and in [35]. Note that a signal can be a trigger or a non-trigger signal only in the trigger states. Therefore, the classifications, "trigger" and "non-trigger", in the trigger/non-trigger column of table 3.1 represent that each signal is either trigger or non-trigger in some trigger states in the generalized transition cube.

3.4.3 Summary

We presented two methods of synthesizing hazard-free combinational logic: two-level AND-OR implementation and the BDD-based multi-level implementation. We showed that these methods require different constraints to guarantee that implementations are hazard-free. In the next chapter, we will show how these constraints are

input signal level at		trigger or	terminating
start cube	end cube	non-trigger	or don't care
0	0	–	–
1	1	–	–
0	1	trigger	terminating
1	0	trigger	terminating
$*$	1	trigger	terminating
$*$	0	trigger	terminating
0	$*$	non-trigger	don't care
1	$*$	non-trigger	don't care
$*$	$*$	non-trigger	don't care

Table 3.1: Trigger/non-trigger signal.

satisfied and how the selection of the combinational synthesis method affects the state assignment.

Chapter 4

Automatic Synthesis Procedure

One of the most significant reasons for the resurrection of asynchronous circuits in the 1980's was the advent of automatic synthesis, which meant tedious and complex tasks, such as hazard-free state assignment and logic minimization, could be carried out by computer. During the course of this research, it was observed that the exact algorithms generally tend to be intractable and do not improve the results significantly over simple heuristics that run in polynomial time. Whenever possible, this thesis presents heuristics that find near-optimal solutions in polynomial time, instead of exact algorithms that find optimal solutions in exponential time.

The synthesis procedure consists of the following steps:

1. **next state assignment**: A primitive next-state table with each specification state assigned to a unique layer is constructed from the extended-burst-mode specification.

2. **layer minimization**: Layer minimization is performed by merging compatible layers.

3. **layer encoding**: A layer diagram, which represents connectivities and encoding restrictions among the layers, is generated, and then a critical-race-free layer encoding is performed.

4. **combinational logic synthesis**: Combinational logic synthesis is carried

out.

4.1 Next State Assignment

We describe an algorithm for assigning next states in a primitive next-state table. In a primitive next-state table, a *state* is a combination of the values of primary inputs and outputs and the current specification state of a 3D machine; a *next state* is a combination of the next output values and the next specification state of a state. A *layer* of a primitive next-state table contains $(l + m) \times n$ entries, each of which represents the next state of a state, where $l + m$ and n are the total numbers of primary inputs and outputs respectively. A primitive next-state table consists of a set of layers; there is a one-to-one correspondence between the set of layers in a primitive next-state table and the set of specification states of the 3D machine the table represents. This algorithm assigns a next state for every reachable entry in the table, according to the extended-burst-mode semantics.

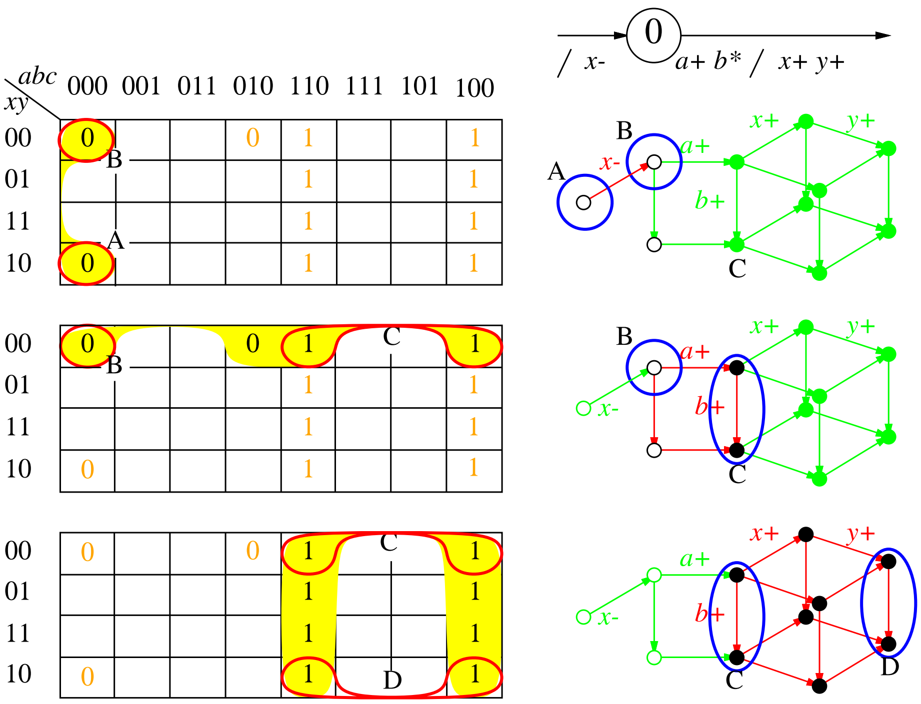


Figure 4.1: Next state assignment.

A Type I 3D machine cycle that requires no conditional signal to stabilize has transitions corresponding to an input burst and a concurrent output/state burst. A Type II or III machine cycle that requires no conditional signal to stabilize has transitions corresponding to an input burst, an output burst and a state burst. Figure 4.1 illustrates the next state assignments for the layer that corresponds to S_0 of the simple example from section 2.3.1. The Karnaugh maps on the left side represent the next output function, X, of the output x during the output/state burst from S_3 to S_0 and the input and output/state bursts from S_0 to S_1. The first transition, which is a static extended-burst-mode transition, corresponds to the output burst x^-. The start cube A of this transition is $abcxy$: 00010, and the end cube B is $abcxy$: 00000. The second transition corresponds to the input burst a^+b^*. The start cube B of this transition is $abcxy$: 00000, and the end cube C is $abcxy$: 1x000. $f([B, C)) = f(B) = 0$ and $f(C) = 1$; thus it is a dynamic extended-burst-mode transition. The third transition, which is a static extended-burst-mode transition, corresponds to the output burst, x^+y^+. The start cube C of this transition is $abcxy$: 1x000, and the end cube D is $abcxy$: 1x011.

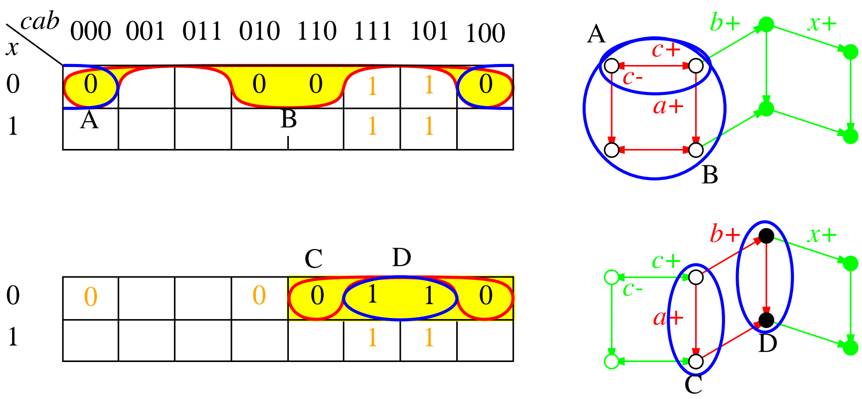


Figure 4.2: Conditional input setup transition.

A 3D machine cycle that requires conditional signals to stabilize has an additional transition for setting up conditional signals. Consider an input burst, $\langle c^+ \rangle\, a^*b^+$ (see figure 4.2). Initially, $c = *$ and $a = b = 0$. Conditional signals must stabilize before any compulsory edge appears. The conditional signal c may change non-monotonically until some setup time before b^+ appears. Since a is a directed don't care, it may rise before c stabilizes to 1. Therefore, the start cube A of this conditional input setup transition is $cabx$: x000, and the end cube B is $cabx$ = xx00. All conditional input setup transitions are static, because outputs cannot be enabled to change until the setup is complete. The next transition corresponds to the input burst, a^*b^+. The start cube C of this transition is $cabx$: 1x00, and the end cube D is $cabx$: 1x10. $f([C, D]) = f(C) = 0$ and $f(D) = 1$; thus it is a dynamic extended-burst-mode transition.

Given an extended-burst-mode specification, $G = (V, E, C, I, O, v_0, \mathsf{cond}, \mathsf{in}, \mathsf{out})$, with $C = \{c_1, \ldots, c_l\}$, $I = \{x_1, \ldots, x_m\}$, and $O = \{y_1, \ldots, y_n\}$, let W be the set of conditional input bit vectors $\{(c_1, \ldots, c_l) \mid c_i \in \{0, 1\}, i \in 1, \ldots, l\}$, let X be the set of edge-input bit vectors $\{(x_1, \ldots, x_m) \mid x_j \in \{0, 1\}, j \in 1, \ldots, m\}$, and let Y be the set of output bit vectors $\{(y_1, \ldots, y_n) \mid y_k \in \{0, 1\}, k \in 1, \ldots, n\}$. A *primitive next-state table* is defined as $T = (V, W, X, Y, \delta, \lambda)$, where V is the set of specification states, and $\delta : V \times W \times X \times Y \rightarrow V \cup \{*\}$ and $\lambda : V \times W \times X \times Y \rightarrow \{0, 1, *\}^n$ define the *next specification state function* and the *next output function* respectively.

Below, we describe the next state assignment for Type I (an input burst followed by a concurrent output/state burst) and Type II (an input burst followed by an output burst followed by a state burst) machine cycles. $\mathsf{cond}(u, v)$, $\mathsf{in}(u)$, and $\mathsf{out}(u)$ denote the values of conditional inputs from u to v, edge-inputs in u, and outputs in u respectively. $L(u)$ is a symbolic code assigned to specification state u. $[\mathsf{cond}(u, v), \mathsf{in}(v), \mathsf{out}(u), L(u)]$ denotes a cube in $\{0, 1, *\}^{l+m+n+r}$, where r is the length of state variable bit vectors. Of course, the actual value of r will not be determined until the layer encoding is done. $*$ in the place of conditional inputs is a shorthand notation, meaning that all conditional inputs are $*$. in' is defined as: for all $j \in 1, \ldots, m$,

$$\mathsf{in}'_j(u) = \begin{cases} \mathsf{in}_j(u) & \text{if there exists } (u, v) \text{ such that } \mathsf{in}_j(v) \neq * \\ * & \text{otherwise} \end{cases} .$$

$\mathsf{in}'_j(u) \neq *$ iff there exists a state transition (u, v) in which j is compulsory or a constant. Thus, $\mathsf{in}'(u)$ represents the allowed values of edge-inputs immediately prior to the first compulsory input edge during a state transition from u. $*$ means both 0 and 1 are possible.

The next state assignment for an input burst is as follows: For all $k \in 1, \ldots, n$ and for every state transition, (u, v),

- Conditional input burst:
  - Conditional input setup:
    for every minterm M in $[A, B]$, where $A = [*, \mathsf{in}(u), \mathsf{out}(u), L(u)]$ and $B = [*, \mathsf{in}'(u), \mathsf{out}(u), L(u)]$,

    $\lambda_k(M) = \mathsf{out}_k(u)$;

    $\delta(M) = u$,
  - Input burst:
    for every minterm M in $[A, B)$, where $A = [\mathsf{cond}(u, v), \mathsf{in}'(u), \mathsf{out}(u), L(u)]$ and $B = [\mathsf{cond}(u, v), \mathsf{in}(v), \mathsf{out}(u), L(u)]$,

    $\lambda_k(M) = \mathsf{out}_k(u)$;

    $\delta(M) = u$,
- Unconditional input burst:
  for every minterm M in $[A, B)$, where $A = [*, \mathsf{in}(u), \mathsf{out}(u), L(u)]$ and $B = [*, \mathsf{in}(v), \mathsf{out}(u), L(u)]$,

  $\lambda_k(M) = \mathsf{out}_k(u)$;

  $\delta(M) = u$,

Note that $[A, B] = B$ during conditional input setup transitions because $A \subseteq B$. The only edge-inputs that may change during conditional input setup transitions are non-compulsory signals.

The next state assignment for output and state bursts is as follows: For all $k \in 1, \ldots, n$ and for every state transition, (u, v),

- Type I machine cycle (Output/state burst):
  for every minterm M in $[A, B]$, where $A = [\mathsf{cond}(u,v), \mathsf{in}(v), \mathsf{out}(u), L(u)]$ and $B = [\mathsf{cond}(u,v), \mathsf{in}(v), \mathsf{out}(v), L(v)]$,

  $\lambda_k(M) = \mathsf{out}_k(v);$

  $\delta(M) = v,$

- Type II machine cycle:

  - Output burst:
    for every M in $[A, B)$, where $A = [\mathsf{cond}(u,v), \mathsf{in}(v), \mathsf{out}(u), L(u)]$ and $B = [\mathsf{cond}(u,v), \mathsf{in}(v), \mathsf{out}(v), L(u)]$,

    $\lambda_k(M) = \mathsf{out}_k(v);$

    $\delta(M) = u,$

  - State burst:
    for every M in $[A, B]$, where $A = [\mathsf{cond}(u,v), \mathsf{in}(v), \mathsf{out}(v), L(u)]$ and $B = [\mathsf{cond}(u,v), \mathsf{in}(v), \mathsf{out}(v), L(v)]$,

    $\lambda_k(M) = \mathsf{out}_k(v);$

    $\delta(M) = v.$

Finally, for all the remaining entries, $\lambda_k(M) = *$ and $\delta(M) = *$.

We prove that a function-hazard-free next state assignment exists for every output and state variable if each specification state is assigned to a unique layer of the next-state table and if the layers can be encoded so that every state burst is function-hazard-free. In section 4.3, we will show that such layer encoding is always possible as well.

Theorem 4.1 *The above next state assignments are free of function hazards, if each specification state is assigned to a unique layer and if the layers can be encoded so that every transition crossing the layer boundary is function-hazard-free.*

Proof: We prove this for Type I machine cycles:

Consider a layer which corresponds to specification state u. According to the next state assignment algorithm for Type I machine cycles, next states must be assigned for the following transitions: output/state burst transitions into u, conditional input setup transitions from u, and input burst and output/state burst transitions from u. Since all these transitions are extended-burst-mode transitions and only the input bursts can be dynamic transitions, each output must have the same next output throughout all the output/state burst transitions into u, the conditional input setup transition, and all the input burst transitions excluding the end cubes of the transitions. The next output values may change in the end cubes of the input burst transitions but remain at those values during the corresponding output/state transitions from u. Therefore, to show that the next state assignment for layer u is free of function hazards, it suffices to show that no output/state burst transition from u intersects (1) an output/state burst transition into u, (2) the conditional input setup transitions from u, (3) the other output/state burst transitions from u or the input burst transitions which enable those output/state bursts.

Without loss of generality, consider an output/state burst transition from u to v.

1. Since every input burst must contain a compulsory edge, there exists $j \in 1, \ldots, m$ such that $\mathsf{in}_j(u) \neq \mathsf{in}_j(v)$ and $\mathsf{in}_j(u) \neq *$ and $\mathsf{in}_j(v) \neq *$. Therefore, the generalized transition cube for the output/state burst from u to v, $[\ldots, \mathsf{in}(v), \ldots]$, does not intersect any generalized transition cube for the output/state bursts into u, $[\ldots, \mathsf{in}(u), \ldots]$.

2. Since every input burst must contain a compulsory edge, there exists $j \in 1, \ldots, m$ such that $\mathsf{in}_j(u) \neq \mathsf{in}_j(v)$ and $\mathsf{in}_j(u) \neq *$ and $\mathsf{in}_j(v) \neq *$. Moreover, $\mathsf{in}_j(v) \neq *$ implies $\mathsf{in}'_j(u) = \mathsf{in}_j(u)$, which means that $\mathsf{in}'_j(u) \neq \mathsf{in}_j(v)$ and $\mathsf{in}'_j(u) \neq *$ and $\mathsf{in}_j(v) \neq *$. Therefore, the generalized transition cube for the output/state burst of (u, v), $[\ldots, \mathsf{in}(v), \ldots]$, does not intersect the conditional input setup transition from u, $[\ldots, \mathsf{in}'(u), \ldots]$.

3. The distinguishability constraint requires that, for every pair of state transitions (u, v) and (u, w), either the conditions are mutually exclusive, or the set of compulsory edges in the input burst of (u, v) is not a subset of the set of all possible

edges in the input burst of (u, w). Therefore, either there exists $i \in 1, \ldots, l$ such that $\mathsf{cond}_i(u, v) \neq \mathsf{cond}_i(u, w)$ and $\mathsf{cond}_i(u, v) \neq *$ and $\mathsf{cond}_i(u, w) \neq *$, or there exists $j \in 1, \ldots, m$ such that j is compulsory in (u, v) but a constant in (u, w), that is, $\mathsf{in}_j(v) \neq \mathsf{in}_j(u)$, $\mathsf{in}_j(v) \neq *$, $\mathsf{in}_j(u) \neq *$, and $\mathsf{in}_j(w) = \mathsf{in}_j(u)$.

In the first case, neither the input burst transition nor the output/state burst transition of (u, w) can intersect the output/state burst transition of (u, v), because the generalized transition cubes of both the input and output/state bursts of (u, w) are of the form $[\mathsf{cond}(u, w), \ldots]$ but the generalized transition cube of the output/state burst of (u, v) is of the form $[\mathsf{cond}(u, v), \ldots]$. The same is true of the second case, because the generalized transition cube of the output/state burst of (u, v) is of the form $[\ldots, \mathsf{in}_j(v), \ldots]$ but the generalized transition cube of the input and output/state bursts of (u, w) are of the form $[\ldots, \overline{\mathsf{in}_j(v)}, \ldots]$.

Therefore, the only possible intersections are between the input burst transitions and the corresponding output/state burst transitions. Since the end cube of an input burst transition is the same as the start cube of the corresponding output burst transition and the next states are specified the same, there is no function hazard. □

4.1.1 Next State Assignment for Two-Level AND-OR

Now we must examine whether it is possible to find a hazard-free logic implementation from the next-state table constructed using the above next state assignment algorithms. In the last chapter, we derived hazard-free covering requirements for two-level AND-OR logic: for a $1 \to 1$ transition of output, a product term must contain $[A, B]$, and for $1 \to 0$ and $0 \to 1$ transitions every product term that intersects $[A, B]$ must also contain A' and B' respectively.

These covering requirements can be satisfied trivially for each transition individually, as shown in figure 3.8. It is not clear, however, whether it is possible to satisfy the requirements for every transition *simultaneously.* First, we will show that it is always possible to satisfy the covering requirements for every transition without violating a requirement for another transition, provided every input burst is unconditional, that is, no level input exists in the specification. Second, we will show how level signals, in certain situations, cause dynamic hazards and then show how to handle level signals by further constraining the next state assignment.

Assume that no level input exists. Consider a state u. In a Type I machine cycle, only the input bursts can be dynamic transitions; thus only the input burst transition cubes can be privileged cubes. To show that there are no hazards, we use Theorems 3.4 and 3.5. By the *unique entry condition* imposed on the extended-burst-mode specification, the end cube of every output/state burst transition into u is the same as the start cubes of the input bursts. Thus all the required cubes that intersect input burst transitions and enable outputs or state variables to fall, also contain the start cubes of the corresponding input burst transitions. As shown in Theorem 4.1, output/state burst transitions intersect only the corresponding (enabling) input burst transitions, and the start cube of every output/state burst transition is the same as the end cube of the corresponding input burst transition. Therefore, all the required cubes that intersect input burst transitions and enable outputs or state variables to rise, also contain the end cubes of the corresponding input burst transitions.

In a Type II machine cycle, both the input and output bursts can be dynamic transitions, because outputs can enable state variables to change. As before, all the required cubes that intersect input burst transitions and enable outputs to fall, also contain the start cubes of the input burst transitions, because of the unique entry condition. The start cube of every output burst transition is the same as the end cube of the corresponding input burst transition; furthermore, the output burst transition intersects no other input burst transitions, by Theorem 4.1. Therefore, all the required cubes that intersect input burst transitions and enable outputs to rise, also contain the end cubes of the corresponding input burst transitions, and all the required cubes that intersect output burst transitions and enable state variables to fall, also contain the start cubes of the corresponding output burst transitions. The start cube of every state burst transition is the same as the end cube of the corresponding output burst transition; moreover, the state burst transition intersects no other output burst transitions. Thus, all the required cubes that intersect output burst transitions and enable state variables to rise, also contain the end cubes of the corresponding output burst transitions.

In both of these cases, we showed that no required cubes illegally intersect privileged cubes. Thus, by Theorems 3.4 and 3.5, there exist hazard-free two-level AND-OR implementations.

Effects of Undirected Don't Cares on Next State Assignment

We examine the effects of allowing undirected don't cares, that is, how conditional signals cause dynamic hazards. Then we show how making state variable changes before output changes can avoid these hazards. We begin by analyzing an example.

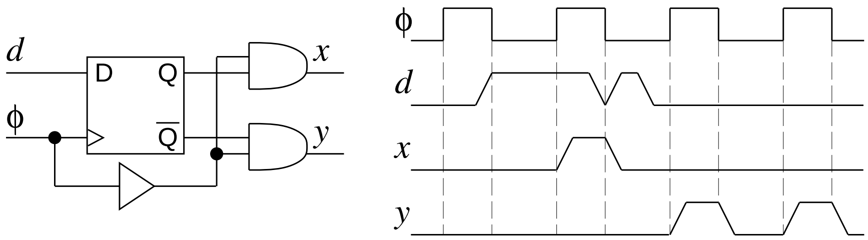


Figure 4.3: Example 2 (synchronous implementation).

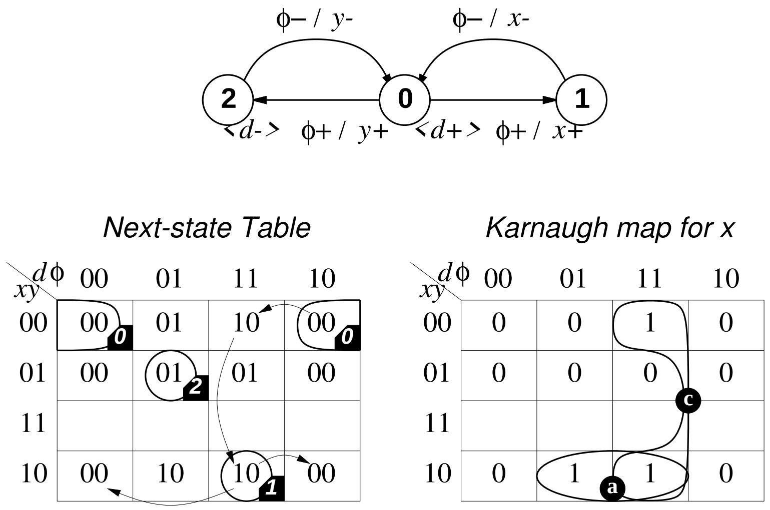


Figure 4.4: Example 2 (specification and next-state table).

Figure 4.4 shows a specification of a circuit described below and figure 4.3 shows one possible synchronous implementation and the timing diagram.

> If the mode bit d sampled at the rising edge of the clock ϕ is 1, the output x follows the clock for that cycle and the output y remains 0. Otherwise, y follows the clock and x remains 0.

Consider the input burst ϕ^- in S_1, which causes the output x to fall. The covering requirement states that no cube may intersect the transition cube $[A, B]$ ($A = d\phi xy$: x110 and $B = d\phi xy$: x010) unless it also contains A', which is the same as A here. However, cube c, required to cover the output burst x^+ in S_0, shown in figure 4.4 intersects the cube a (part of the transition cube $[A, B]$), but it cannot be expanded to cover A' — there is a dynamic hazard.

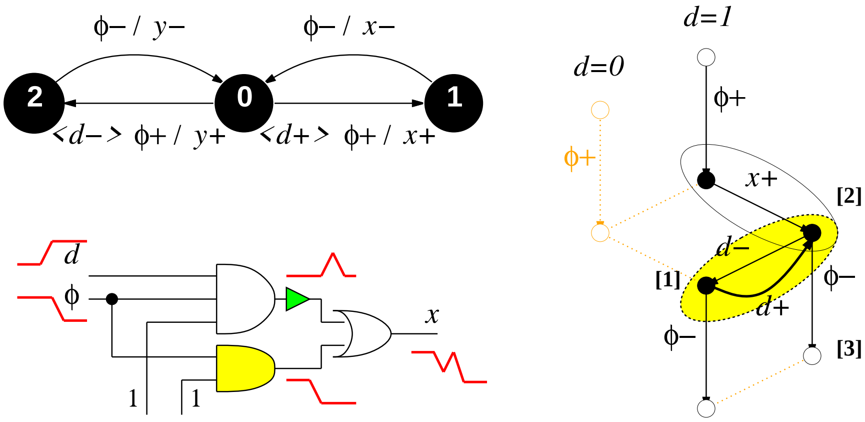


Figure 4.5: Example 2 (problem).

To see how the hazard can actually occur, let's examine a fragment of the state graph in figure 4.5. After the output burst x^+, the machine is at [2] waiting for the next input change ϕ^-. Since we allow conditional signals to vary freely, d may fall, taking the machine to point [1] (product terms corresponding to cube c and cube a are 0 and 1 respectively). If the next set of events is a concurrent input change $d^+\phi^-$, the machine may change from point [1] to [2] to [3]. During this change, product term a falls, and product term c glitches $0-1-0$ which may propagate to the output causing a $1-0-1-0$ glitch at the output (dynamic hazard). Note that this phenomenon can occur long after the *hold time* constraint is satisfied, because, according to the hold time requirement, d only needs to be stable until x becomes 1, and the cause of this glitch, a concurrent input change $d^+\phi^-$, can occur long after x^+. This phenomenon would not occur if the conditional signal change is monotonic.

This is a general problem that occurs when an output burst transition enabled by a conditional input burst intersects an unconditional input burst transition that enables an output to fall. Consider an output/state burst transition (T_o, E, F), from z to u, enabled by a conditional input burst, where $E = [\mathsf{cond}(z,u), \mathsf{in}(u), \mathsf{out}(z), L(z)]$ and $F = [\mathsf{cond}(z,u), \mathsf{in}(u), \mathsf{out}(u), L(u)]$, and an unconditional input burst transition (T_i, A, B) from u, where $A = [*, \mathsf{in}(u), \mathsf{out}(u), L(u)]$ and $B = [*, \mathsf{in}(v), \mathsf{out}(u), L(u)]$. Note that there are no setup transitions preceding unconditional input bursts. Assume that an output x undergoes $1 \to 1$ and $1 \to 0$ transitions in $[E,F]$ and $[A,B]$ respectively. If there exists i such that $\mathsf{cond}_i(z,u) \neq *$, then $A' \not\subseteq F$ because the level signal i is a don't care in A'. If F cannot be expanded to contain A', then the covering requirement for the $1 \to 0$ transition is violated, which induces a dynamic hazard.

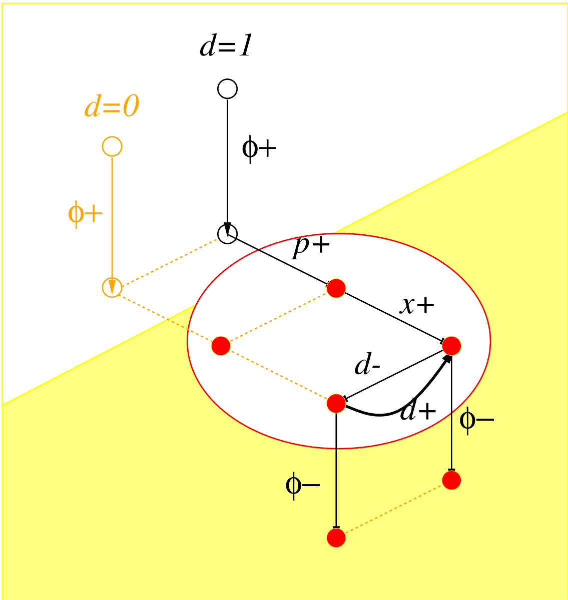


Figure 4.6: Example 2 (solution – state graph).

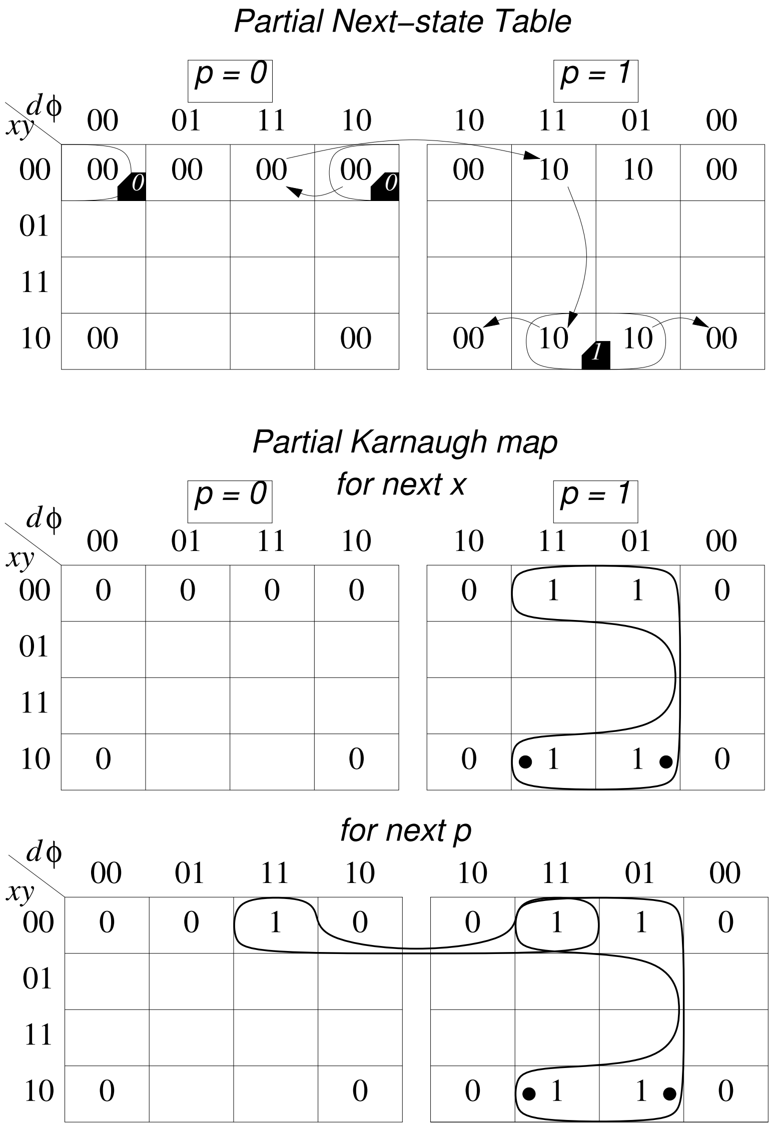


Figure 4.7: Example 2 (solution – next-state table).

Solution

Our solution to avoid this dynamic hazard is to add a new layer and to move to it via a state burst before enabling outputs to change *if the next input burst is unconditional and enables an output to fall.* Intuitively, the trick is to "store" the conditions in the state variables. Once the conditions are stored in the state variables, the conditional signals can change freely. To eliminate hazards on the state variables, the state variables are "latched" by a "strobe" generated by the last sampling edge followed by output changes. The conditional signals must remain stable only until the strobe is turned off by the output changes.

Figures 4.6 and 4.7 illustrate our solution. In figure 4.6, if $d = 1$ when ϕ rises, we move to a new layer via the state burst p^+ before raising x. Thus the next x entry for $d\phi xy = 1100$ in the $p = 0$ part of the table in figure 4.7 is specified to be 0. Now we need to specify the next x for the output burst transition x^+. The trick here is that we expand the generalized transition cube *as if* d were allowed to change, although the environment is not allowed to change d until x is stable because of the hold time constraint. This is possible because the generalized transition cube for this output burst is on a *new layer*, the $p = 1$ part of the table. When output x stabilizes to 1, the machine is in S_1. The next x for $d\phi xy = \mathsf{x}110$, in the $p = 1$ part of the table, is specified to be 1 so that output x remains unchanged until the compulsory edge ϕ^- appears.

The start cube of the output burst x^+ is $pd\phi xy = 1\mathsf{x}100$, and the end cube $pd\phi xy = 1\mathsf{x}110$. The required cube ($pd\phi xy = 1\mathsf{x}1\mathsf{x}0$) for this output burst now contains the *start cube* of the next input burst ϕ^- — there is no violation of covering requirements.

Now examine the required cubes for state variable p we added (see the bottom table in figure 4.7). We require one cube ($pd\phi xy = \mathsf{x}1100$) to cover the state burst p^+ enabled by the conditional input burst $\langle d^+ \rangle \phi^+$ and another cube ($pd\phi xy = 1\mathsf{x}1\mathsf{x}0$) to cover the output burst x^+. Since the required cube ($pd\phi xy = \mathsf{x}1100$) does not intersect the start cube of the next input burst ϕ^-, the covering requirements are not violated.

We can also understand this in the physical circuit (see figure 4.8). The sampling

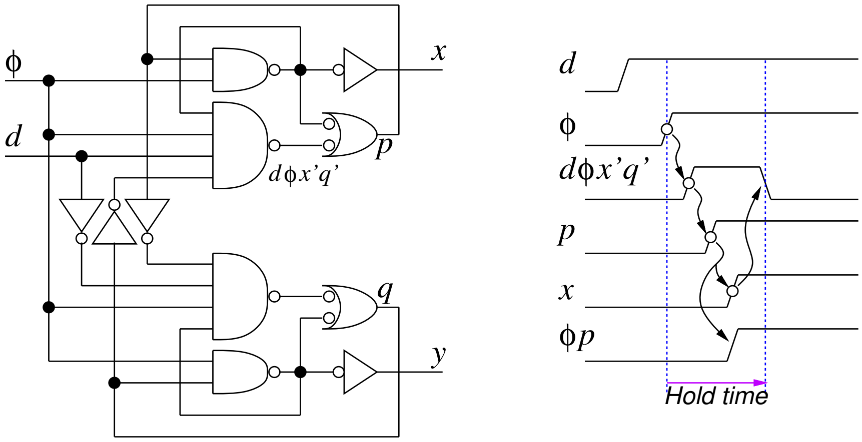


Figure 4.8: Example 2 (circuit and timing).

signal ϕ generates a "strobe" ($d\phi x'y' = 1$) to latch p; once the output x rises, the strobe signal is turned off blocking out the effects of changing d. No glitch occurs if d remains stable until x^+ turns off the strobe signal.

Formally, the next state assignments for the state and output burst portions of a Type III machine cycle (an input burst followed by a state burst followed by an output burst) are as follows: For all $k \in 1, \ldots, n$ and for every edge, (u, v),

- State burst:
  for every M in $[A, B)$, where $A = [\mathsf{cond}(u,v), \mathsf{in}(v), \mathsf{out}(u), L(u)]$ and $B = [\mathsf{cond}(u,v), \mathsf{in}(v), \mathsf{out}(u), L(v)]$,

  $$\lambda_k(M) = \mathsf{out}_k(u);$$

  $$\delta(M) = v,$$

- Output burst:
  for every M in $[A, B]$, where $A = [*, \mathsf{in}(v), \mathsf{out}(u), L(v)]$ and $B = [*, \mathsf{in}(v), \mathsf{out}(v), L(v)]$,

  $$\lambda_k(M) = \mathsf{out}_k(v);$$

  $$\delta(M) = v.$$

Note that the output burst in Type III machine cycle must not be empty. If the output burst is empty in the specification, a *dummy* output edge is added.

4.1.2 Next State Assignment for BDD-Based Multi-Level Circuit

We show that both Type I and II next state assignments are free of logic hazards for a BDD-based implementation, if each specification state is assigned to a unique layer and if the layers can be encoded so that every transition crossing the layer boundary is critical-race-free. The BDD-based implementation is hazard-free during an extended-burst-mode static transition and, if the variable ordering requirement is satisfied, hazard-free during an extended-burst-mode dynamic transition as well. Again, this ordering requirement can be satisfied trivially for each transition individually, as shown in chapter 3. however, we need to check whether it is possible to satisfy the requirements for every transition simultaneously.

Assume that each specification state is assigned to a unique layer. In a Type I machine cycle, only the input bursts can be dynamic transitions. Therefore, it suffices to check whether there are *conflicting* ordering requirements among the input bursts from the same specification state. In a Type II machine cycle, both the input and output bursts can be dynamic transitions. Since the terminating edges of output bursts are output edges and the terminating edges of input bursts are input edges, there are no conflicting ordering requirements between input bursts and output bursts. Since outputs cannot be don't cares, there are no conflicting ordering requirements among output bursts. Again, it suffices to check whether there are conflicting ordering requirements among the input bursts from the same specification state.

Lemma 4.1 *There always exists a free BDD that satisfies the variable ordering requirements for two dynamic input burst transitions from a specification state.*

Proof: If there are no conflicting ordering requirements among the input bursts, then the variable ordering requirements are trivially satisfied in an ordered BDD.

Assume that the input bursts from state transitions (u, v) and (u, w) have conflicting ordering requirements. By the distinguishability constraint, either the conditions are mutually exclusive, or the set of compulsory edges in the input burst of (u, v) is not a subset of the set of all possible edges in the input burst of (u, w).

If the conditions of two input bursts are mutually exclusive, then there exists a conditional signal such that it is a constant in both input bursts but its value in one input burst is different from that in the other input burst. If this conditional variable appears before any variable involved in the ordering, the variable ordering for each input burst is satisfied in left or right partition created by this conditional variable.

If the conditions of two input bursts are not mutually exclusive, then there exist compulsory signals i and j in the input bursts of (u, v) and (u, w) respectively such that i is a constant in the input burst of (u, w) and j is a constant in the input burst of (u, v). Suppose that we select a variable ordering such that i appears before j and before any variable involved in the orderings. Let the output of the multiplexor with i as its select signal be g. Without loss of generality, assume that i rises. For the input burst of (u, w), g_i is irrelevant, because i is a constant 0 in the input burst of (u, w). Therefore, the variable ordering requirement for the input burst of (u, w) is satisfied by selecting an appropriate variable ordering in the sub-BDD $g_{\bar{i}}$.

Let (T, A, B) be the input burst transition of (u, v). Since (T, A, B) is an extended-burst-mode transition for g, $(T, A_{\bar{i}}, B_{\bar{i}})$ is an extended-burst-mode transition for $g_{\bar{i}}$, by Lemma 3.5. By Corollary 3.2, $(T, A_{\bar{i}}, B_{\bar{i}})$ is a static transition for $g_{\bar{i}}$. Thus, there is no ordering requirement in the sub-BDD $g_{\bar{i}}$ for $(T, A_{\bar{i}}, B_{\bar{i}})$. We can select an appropriate variable ordering in the sub-BDD g_i to satisfy the requirement for the input burst of (u, v).

By symmetry, a free BDD with j appearing before i can also satisfy the variable ordering requirement. □

Example

Consider two input bursts, $a^+b^*c^+$ and $a^*b^+d^+$, which correspond to (T_b, A, B) and (T_c, A, C) respectively with $A = 0000$, $B = 1\text{x}10$, and $C = \text{x}101$, as shown in figure 4.9a. (T_b, A, B) requires that $a < b$ and $c < b$, because b is a directed don't care and a and c are terminating signals in (T_b, A, B). (T_c, A, C) requires that $b < a$ and $d < a$, because a is a directed don't care and b and d are terminating signals in (T_c, A, C). Obviously, we cannot satisfy $a < b$ and $b < a$ globally. Since c and d are compulsory edges in (T_b, A, B) and (T_c, A, C) respectively and $c = 0$ in (T_c, A, C) and

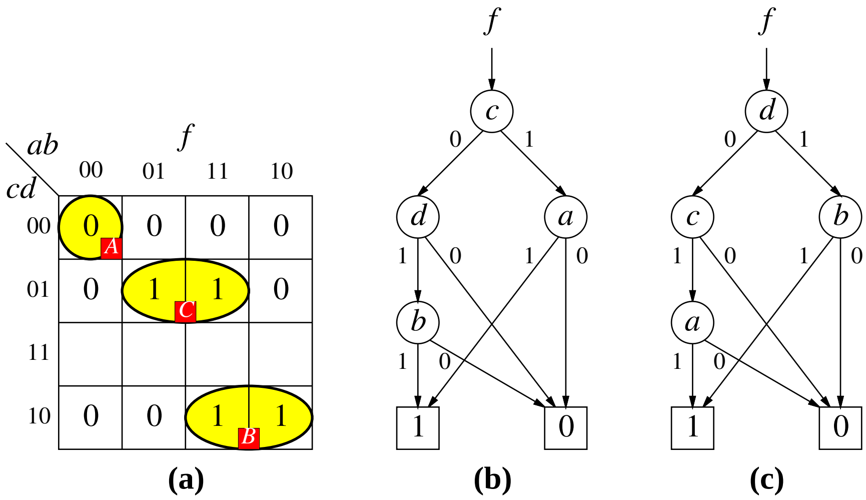


Figure 4.9: Satisfying variable ordering locally.

$d = 0$ in (T_b, A, B), free BDD implementations that satisfy $a < b$ and $b < a$ "locally", such as the ones in figure 4.9bc, exist.

4.2 Layer Minimization

In the previous section, we showed that a hazard-free implementation (two-level AND-OR or BDD-based multi-level circuit) exists if each specification state is assigned to a unique layer and if the layers can be encoded so that every transition crossing the layer boundary is function-hazard-free. In this section, we present a heuristic for hazard-free layer minimization and contrast it with a classical state minimization algorithm [69]. The goal of hazard-free layer minimization is to reduce the number of layers required for the next state table while insuring that a hazard-free implementation can be found for every output and state variable. This is done by merging compatible specification states, as defined below, into a common layer.

4.2.1 Definitions

A *partially encoded total state* (u, s), where $u \in V$, $s \in W \times X \times Y$, is a member of the set $V \times W \times X \times Y$.

(u, s) and (v, s) are *output-compatible* iff $\delta(u, s) = \delta(v, s)$ or $\delta(u, s) = *$ or $\delta(v, s) = *$.

u and v are *dhf-compatible* [49] if no dynamic hazard results from specifying the next states of u and v on a single layer of the next-state table.

u and v are *SOP-dhf-compatible* (sum-of-products dynamic-hazard-free compatible) if no required cube in u illegally intersects a privileged cube in v and vice versa, when the next states of u and v are specified on a single layer of the next-state table. In figure 4.10, for example, if specification states i and j are merged, the layers i and j, represented as Karnaugh maps, would be superimposed, resulting in a violation of a hazard-free covering requirement: a required cube $abxy$: 11x1 illegally intersects a privileged cube $abxy$: x111. The generalized transition cube of every input burst for Type I machine cycles is treated as a privileged cube; since we do not yet have encoding of layers, we must assume that any input burst may enable a state variable to fall. Likewise, the generalized transition cubes of output bursts for Type II and input bursts for Type III are treated as privileged cubes.

Note that two specification states bridged by a Type III machine cycle are always incompatible, because the next outputs of the minterms in the start cube of the state burst are different from those in the end cube. That is, outputs are enabled to change in the end cube of the state burst in Type III machine cycles.

u and v in V are *SOP-compatible* ($u \sim_s v$) iff u and v are SOP-dhf-compatible and, for every s in $W \times X \times Y$,

1. (u, s) and (v, s) are output-compatible and
2. $\delta(u, s) = *$ or $\delta(v, s) = *$ or $\delta(u, s) \sim_s \delta(u, s)$.

u and v are *BDD-dhf-compatible* (BDD dynamic-hazard-free compatible) iff for every pair of state transitions (u, w_u) and (v, w_v),

1. there exists $k \in 1, \ldots, n$ such that $\mathsf{out}_k(u) \neq \mathsf{out}_k(v)$ or
2. i is a terminating signal in (u, w_u) and j is a don't care in (u, w_u) imply that i is a terminating signal in (v, w_v) or j is a don't care in (v, w_v), that is, $\mathsf{in}_i(w_u) \neq$

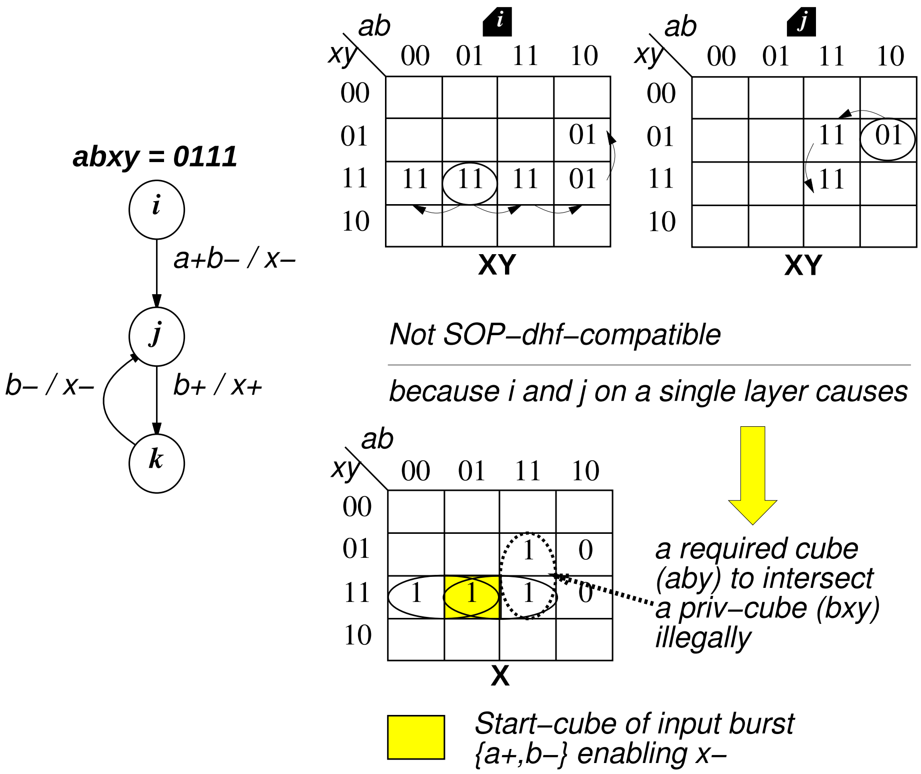


Figure 4.10: Output-compatible but not SOP-dhf-compatible.

$\mathsf{in}_i(u) \wedge \mathsf{in}_i(w_u) \neq * \wedge \mathsf{in}_j(w_u) = *$ implies $\mathsf{in}_i(w_v) \neq \mathsf{in}_i(v) \wedge \mathsf{in}_i(w_v) \neq *$ or $\mathsf{in}_j(w_v) = *$.

This criterion states that no input burst from u has conflicting ordering requirements with an input burst in v that has identical values of fed-back outputs.[1] In figure 4.11, merging specification states i and j would result in conflicting variable ordering requirements: $a < b$, $c < d$, $b < a$, and $d < c$. Therefore, i and j are not BDD-dhf-compatible.

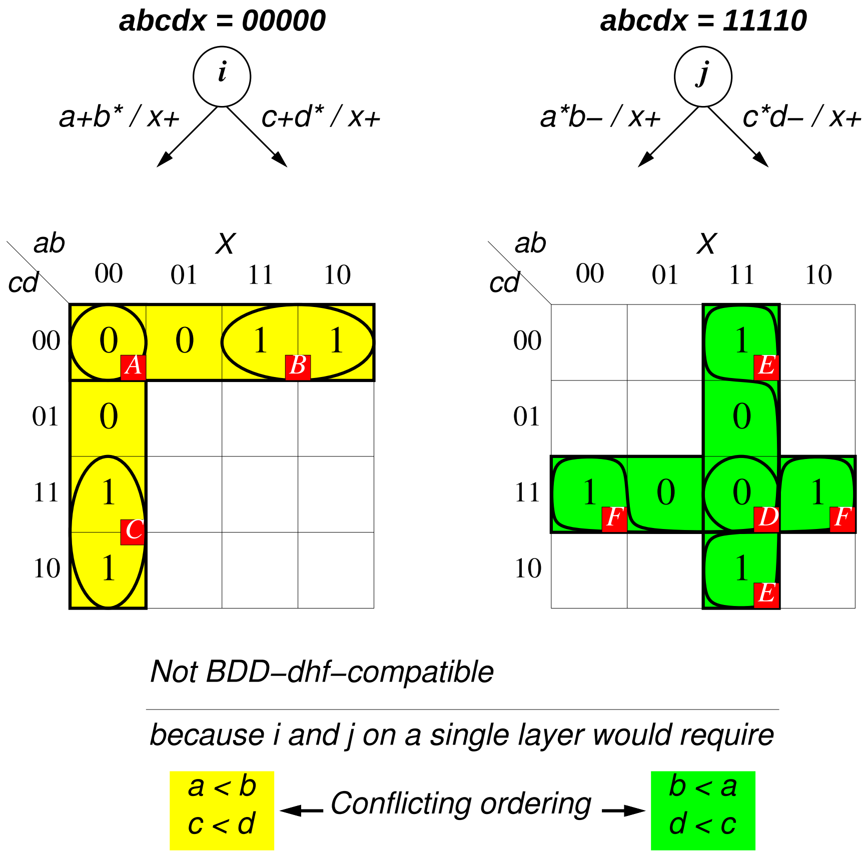


Figure 4.11: Output-compatible but not BDD-dhf-compatible.

u and v in V are *BDD-compatible* ($u \sim_b v$) iff u and v are BDD-dhf-compatible and, for every s in $W \times X \times Y$,

[1]This is just a sufficient condition for dynamic hazard freedom. It is possible to state a necessary condition; however, it is much more complex and not very useful. In practice, we have never encountered an example with two specification states incompatible solely due to BDD-dhf-incompatibility.

1. (u, s) and (v, s) are output-compatible and
2. $\delta(u, s) = *$ or $\delta(v, s) = *$ or $\delta(u, s) \sim_b \delta(v, s)$.

$\sim_s$ and $\sim_b$ are reflexive and symmetric but not transitive. Thus $\sim_s$ and $\sim_b$ are not equivalence relations. Henceforth, we will use the terms *compatible* and *incompatible* to mean SOP-compatible and SOP-incompatible, when the two-level synthesis is used, and to mean BDD-compatible and BDD-incompatible, when the BDD synthesis is used. Moreover, we will use the notation $u \sim v$ to mean that u and v are compatible and $u \not\sim v$ to mean that u and v are incompatible.

A *compatible*, C, is a set of specification states, of which every pair of specification states u and v in C is compatible. A *maximal compatible* is a compatible which is not a proper subset of any other compatible.

A set of compatibles is said to *cover* the extended-burst-mode specification if every specification state is included in at least one compatible. An *irredundant cover* is a cover in which each specification state appears in exactly one compatible.

(u', v') belongs to the *implied set* of (u, v) if $u \sim v$ implies $u' \sim v'$.

A compatible C is *closed* if, for every pair u and v in C, every member of the implied set of (u, v) is a subset of some compatible C' in the cover. A compatible C is *self-contained* if, for every pair u and v in C, every member of the implied set of (u, v) is a subset of C.

A *closed cover* is a cover of which every compatible is closed. An *irredundant closed cover* is a closed cover in which each specification state appears in exactly one compatible.

4.2.2 Layer Minimization Algorithm

The general state minimization problem can be stated as follows, *to find an irredundant closed cover of minimum cost*, e.g., the cost is the cardinality of the cover, if the objective is to minimize the number of states. It has been shown elsewhere [27] that the exact state minimization algorithm for an incompletely specified table, such as ours, is exponential in the number of original states. Furthermore, minimizing the number of layers does not necessarily result in minimizing the area of the final implementation [15]. Therefore, we use a simple algorithm, which works well for for the most extended-burst-mode specifications, as an alternative to the exact algorithm. Our algorithm guarantees to find a (not necessarily minimal) cover in polynomial time in the number of specification states. In practice, the number of layers has been optimal or close to optimal for most examples.

Our layer minimization algorithm proceeds as follows — step 1 is exact but the step 2 is a heuristic.

1. For every pair of specification states u and v, the algorithm determines the compatibility of u and v and the implied set of (u, v).

2. The heuristic traverses the extended-burst-mode specification depth-first and merges *self-contained connected* sets of *compatible specification states* into layers.

Here we describe a classical technique [69] for the state minimization problem, as applied to the extended-burst-mode state machine, for comparison.

1. Determine the compatibility relation by checking, for every pair of states u and v in the extended-burst-mode specification, whether u and v are compatible.

2. Find a set of maximal compatibles.

3. Find an irredundant minimum cover using an algorithm such as Petrick's method [57].

4. Check if each compatible is still closed. If a member of the implied set of $(u, v) \in C$, say (u', v'), is "split" (i.e., for all C' in the irredundant cover, $u \in C' \Rightarrow v \notin C'$ and $v \in C' \Rightarrow u \notin C'$), then u and v must also be "split". Note that this step needs to be done recursively.

In general, steps 2 and 3 induce a state explosion — the number of maximal compatibles is exponential in the number of specification states.

The outcome of the layer minimization is a *reduced next-state table*, $T' = (V', W, X, Y, \delta', \lambda')$, where V' is the set of layers, and $\delta' : V' \times W \times X \times Y \rightarrow$

$V' \cup \{*\}$ and $\lambda' : V' \times W \times X \times Y \to \{0, 1, *\}^n$ define the *next layer function* and the *next output function* respectively.

Example

We use an example in figure 4.12 to compare our algorithm with the classical algorithm described above.

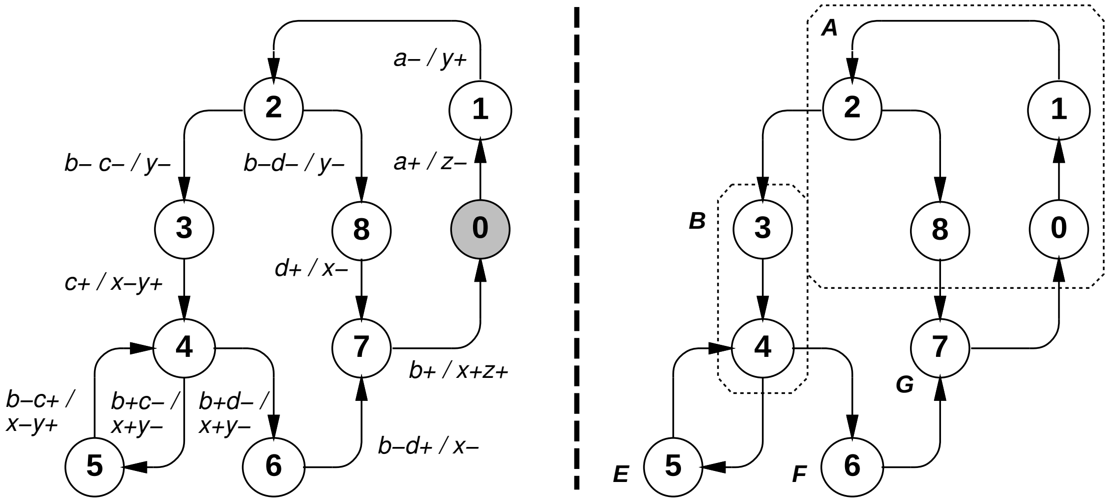


Figure 4.12: ISEND specification and layer assignment.

Both our algorithm and the classical algorithm share the same first step. For every pair of specification states, u and v, in the extended-burst-mode specification, the compatibility of u and v is decided (see figure 4.13). Note that in this example no pair of compatible specification states has a corresponding implied set.

Compatible specification states are merged into common layers as the heuristic traverses the extended-burst-mode specification depth-first. First, states 0, 1, and 2 are merged into layer A because $S_0 \sim S_1$, $S_0 \sim S_2$, and $S_1 \sim S_2$. Because $S_2 \not\sim S_3$, state 3 is assigned to a new layer B. Because $S_3 \sim S_4$, state 4 is merged to layer B. State 5 is assigned to another new layer E, because $S_4 \not\sim S_5$. Since state 5 only connects to state 4, it is the leaf node. The heuristic then backs up to the node with branches that have not been traversed. States 6 and 7 are assigned to new layers F and G, because $S_4 \not\sim S_6$ and $S_6 \not\sim S_7$. When the heuristic traverses the right branch

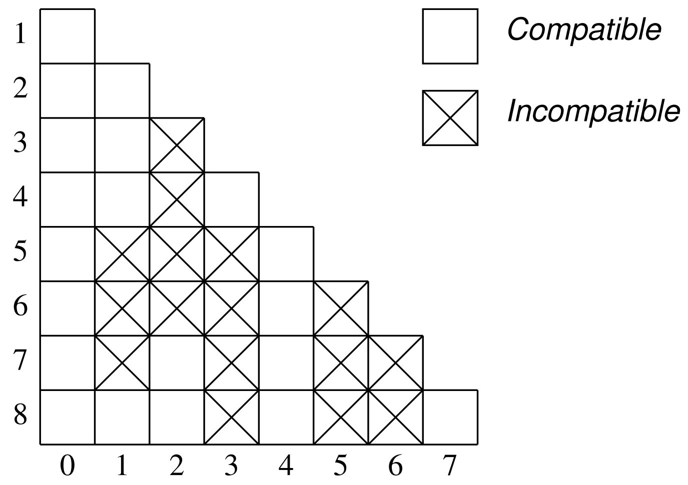


Figure 4.13: Compatibility table.

of state 2, state 8 is merged to layer A, because it is compatible with all three states 0, 1, and 2. Our heuristic layer minimization is complete once the traversal of the extended-burst-mode specification is over; the result is a set of layers each of which contains a self-contained connected compatible specification states (see right side of figure 4.12).

A	$(0,1,2,8)$	E	(5)
B	$(3,4)$	F	(6)
		G	(7)

For comparison, we also compute the results from the classical algorithm. The following is a list of maximal compatibles computed:

A	$(0,1,2,8)$	E	$(0,4,5)$
B	$(0,1,3,4)$	F	$(0,4,6)$
C	$(0,1,4,8)$	G	$(0,4,7,8)$
D	$(0,2,7,8)$		

Two sets of minimum covers are found using Petrick's method [57]:

(A, B, E, F, G) (B, D, E, F)

Possible symbolic layer assignments after removing redundancies are:

Solution I: ((A, B, E, F, G) is selected as a cover)

A	$(0, 1, 2, 8)$	E	(5)
B	$(3, 4)$	F	(6)
		G	(7)

Solution II: ((B, D, E, F) is selected as a cover)

B	$(0, 1, 3, 4)$	E	(5)
D	$(2, 7, 8)$	F	(6)

Note that the solution I is identical to the result we obtained using our algorithm.

4.3 Layer Encoding

We describe an algorithm for critical-race-free encoding of layers. It has been shown elsewhere [24, 61, 69] that there exists a universal one-shot state encoding (using *state splitting*), with a Hamming distance of 1 between any two state codes. However, universal state encoding is very costly to implement for a large number of states; furthermore, requiring a Hamming distance of 1 between any two layers is unnecessary for our 3D implementation. Instead, we use a simple algorithm derived from [67].

In the previous section, we presented an algorithm to construct the layers of the next-state table. At the end of the layer construction, we have a symbolic layer code for each state. We can then extract the constraints on the layer encoding in the form of a layer diagram (see figure 4.15). We use this diagram to do the critical-race-free layer encoding. The objective of the layer encoding is *to generate a critical-race-free layer assignment with few state bits.*

4.3.1 Layer Diagram

A *conflict* is a function hazard during a state burst. A *layer diagram* is an undirected graph. Each edge is labeled with a (possibly empty) set of pairs of vertices. The vertices represent layers of the next-state table, the edges represent transitions between the layers, and the labels on an edge correspond to potential conflicts during transitions between the layers.

If there is a transition from layer C to layer D, an undirected edge is drawn between them. The next-state table entries of all states with the same combination of primary inputs and outputs (see figure 3.5) as the initial state (c) and the final state (d) of the layer transition from C to D are checked for possible conflicts. If the next state of e, a state with the same xy-position as c and d, has already been specified, then the layer E containing e is considered a *potential conflict layer* in assigning codes to C and D, *unless the next layer of* e *is* D. The edge between C and D is then labeled with E. Furthermore, if the next state of e is in another layer, say F, then layer E, layer F, and all the transient layers between E and F become potential conflict layers. The edge between C and D is labeled with $[E, F]$ in this case.

Given a reduced next-state table, $T' = (V', W, X, Y, \delta', \lambda')$, where V' is the set of layers, and δ' and λ' define the *next layer function* and the *next output function* respectively, assume that $C, D, E, F \in V'$ are *unique* layers, s is a member of $W \times X \times Y$, and $\delta'(C, s)$ is D. Formally, a layer transition from E to F is said to be a *potential conflict transition* for the transition from (C, s) to D iff $\delta'(E, s) = F$. Furthermore, layer E is said to be a *potential conflict layer* for the transition from (C, s) to D iff $\delta'(E, s) = E$ and $\lambda'(E, s) \neq *$.

So far, we have assumed that C, D, E, and F are unique. Now consider degenerate cases: $C = E$ and $D = F$. In the first case, since $\delta'(C, s)$ cannot be both D and F, if $D \neq F$, it is impossible for $[C, F]$ to be a potential conflict transition for the transition from C to D. In the second case, the transition from (E, s) to D is not a potential conflict transition for the transition from (C, s) to D, because they merge at (D, s)

Let the codes assigned to layers C and D be c_C and c_D. A potential conflict layer

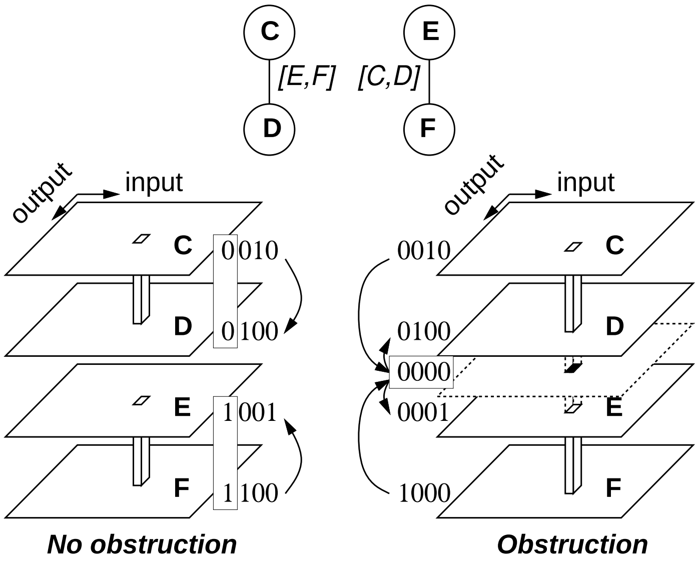


Figure 4.14: Layer encoding example.

E, if assigned the code c_E, is said to *obstruct* the transition from C to D (or from D to C) iff c_E differs from c_C and c_D only in bit positions whose values change during the transition from C to D —

$$c_C + (c_C \oplus c_D) = c_E + (c_C \oplus c_D)$$

where $+$ and $\oplus$ denote *bitwise OR* and *XOR*. For example, the potential conflict layer E obstructs the transition from C ($c_C = 001$) to D ($c_D = 010$), if c_E is 000 or 011, but does not, if c_E is 100.

We can generalize the definition of obstruction to the case in which the transition between C and D has a potential conflict with a transition $[E, F]$. The potential conflict transition $[E, F]$ is said to *obstruct* the transition from C to D (or from D to C) iff E and F are assigned the codes c_E and c_F and c_E and c_F differ from c_C and c_D only in bit positions whose values change during the transition from C to D or during the transition from E to F —

$$c_C + c_X = c_E + c_X$$

where $c_X = (c_C \oplus c_D) + (c_E \oplus c_F)$. Note that if $[E, F]$ is labeled on the edge between C and D, then $[C, D]$ is labeled on the edge between E and F ($[C, D]$ and $[E, F]$ are said to form a *dichotomy* in [67]). In figure 4.14, $[E, F]$ does not obstruct $[C, D]$ and vice versa, as long as c_C and c_D share the same value and c_E and c_F share the value opposite to c_C's at least in one bit position (the MSB in this example). The potential conflict transition $[E, F]$ obstructs the transition from C ($c_C = 0010$) to D ($c_D = 0100$), if c_E and c_F are 0001 and 1000 respectively, but does not, if c_E and c_F are 1001 and 1100.

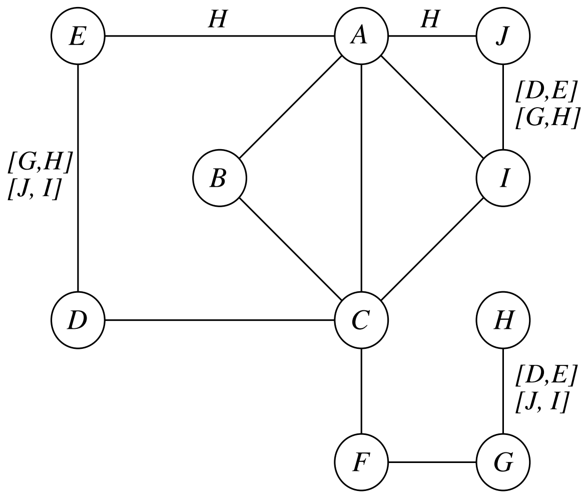


Figure 4.15: Layer diagram.

Suppose that layer transition $[C, D]$ is labeled $[E, F]$ and layer transition $[E, F]$ is labeled $[C, D]$. It is clear that we can find codes for C, D, E, and F that do not cause the obstruction. If $C \neq E$, $C \neq F$, $D \neq E$, and $D \neq F$, we can find codes with an arbitrary Hamming distance between C and D and between E and F. We simply select a bit position and assign 0 in that bit position for C and D and 1 for E and F; the remaining bit positions are used to make codes for C, D, E, and F unique. Therefore, it is always possible to encode layers so that every state burst is critical-race-free.

Now we must examine whether there is a possibility of logic hazard due to these state bursts. Recall that, in the BDD implementations, only a Type I or II machine cycle was used. In Type I and II machine cycles, every state burst is a static transition.

Thus, there can be no hazard due to state bursts in the BDD implementation. Recall that, in the two-level AND-OR implementation, dynamic hazards arise due to illegal intersections of privileged cubes. Because we took care to avoid intersecting any non-don't care entry except in the cases in which two state burst transitions merge at the destination layer, no state burst transition intersects a privileged cube and no required cube intersects a state burst transition. Thus, no hazard is introduced by state bursts in the two-level AND-OR implementation.

4.3.2 Layer Encoding Algorithm

Our goal is to encode the layers so that no layer or transition between layers obstructs the transitions (edges) on which it is labeled as a potential conflict. The layer encoding algorithm begins by resolving all potential conflicts. Initially, a code bit is reserved for each labeled layer transition so that the potential conflict layers are assigned a value different from the source and destination layers of the layer transition in that bit position. For example, 5 bits are allotted to encode the layers of the layer diagram in figure 4.15 because there are 5 labeled transitions. In the table below, each row represents a dichotomy (transition and corresponding obstructions) and each column represents a layer to be encoded.

Transition	Obstructions	A	B	C	D	E	F	G	H	I	J
[A,E]	H										
[A,I]	H										
[D,E]	$[G, H], [I, J]$										
[G,H]	$[D, E], [I, J]$										
[I,J]	$[D, E], [G, H]$										

All redundant dichotomies are then removed and symbolic code values are assigned. In the above example, $[D, E]$ from row 4 and the entire row 5 can be removed because these dichotomies are already represented in row 3. In the table below, v_i is 0 or 1, $\overline{v_i}$ is the complement of v_i, and $-$ is a don't care).

Transition	Obstructions	A	B	C	D	E	F	G	H	I	J
[A,E]	H	v_1	$-$	$-$	$-$	v_1	$-$	$-$	$\overline{v_1}$	$-$	$-$
[A,I]	H	v_2	$-$	$-$	$-$	$-$	$-$	$-$	$\overline{v_2}$	v_2	$-$
[D,E]	$[G, H], [I, J]$	$-$	$-$	$-$	v_3	v_3	$-$	$\overline{v_3}$	$\overline{v_3}$	$\overline{v_3}$	$\overline{v_3}$
[G,H]	$[I, J]$	$-$	$-$	$-$	$-$	$-$	$-$	v_4	v_4	$\overline{v_4}$	$\overline{v_4}$

Compatible rows of the above table are merged to minimize the number of state bits. Two rows i and j are compatible iff one of the following conditions is true:

- in every column, the bit positions i and j have the same value or at least one is a don't care;
- in every column, the bit positions i and j have the opposite value or at least one is a don't care.

A set of rows of the dichotomy table that can be merged is said to be a *compatible*. A *maximal compatible* is a compatible which is not a proper subset of any other compatible. Minimizing the number of rows is then posed as a covering problem (as in the state minimization case): cover all the dichotomies with as few maximal compatibles as possible. A dichotomy table with the minimum number of rows is said to be a *reduced dichotomy table*. Then the symbolic codes are replaced with binary values (0 for v_i and 1 for $\overline{v_i}$).

		A	B	C	D	E	F	G	H	I	J
[A,E],[A,I],[I,J]	$[G, H], H$	0	$-$	$-$	$-$	0	$-$	1	1	0	0
[D,E]	$[G, H], [I, J]$	$-$	$-$	$-$	0	0	$-$	1	1	1	1

Before completing the code assignment, the algorithm determines the total number of state bits required to uniquely encode each layer, which may exceed the number of rows in the table.

A *subcode* of code, c_A, is any code that has the same value as c_A in every bit position in which c_A is not a don't care. For example, c_E (00) is a subcode of c_A (0$-$)

but c_D (-0) is not. In order to differentiate c_A from all of its subcodes in the table (c_E, c_I and c_J), at least 2 bits ($\lceil \log_2(3+1) \rceil$) are needed; hence we need at least 3 bits to encode c_A (2 bits to differentiate c_A from c_E, c_I and c_J and 1 preassigned bit).

The *minimal code length* required for a partially encoded layer (a layer with don't care bit positions) A with the code assignment of c_A in the reduced dichotomy table is then:

$$\max(n_r, n_r - n_d + \lceil \log_2(n_s + 1) \rceil)$$

where n_r is the number of rows of the reduced dichotomy table, n_d is the number of don't care bit positions in c_A, and n_s is the number of subcodes of c_A in the reduced dichotomy table. $n_r - n_d$ is the number of preassigned bits, and $\lceil \log_2(n_s + 1) \rceil$ additional bits are needed to differentiate c_A from all of its subcodes.

If the longest of the minimal code lengths exceeds the number of rows of the reduced dichotomy table, the algorithm pads the table with rows of don't cares until the number of rows equals the longest of the minimal code lengths (4 in our example), which is the lower bound on the number of rows that needs to be added. Note that the upper bound is no greater than $\lceil \log_2 n_l \rceil$ where n_l is the total number of layers. In all of the examples we synthesized, the lower bound was sufficient.

The number of *available codes* for a partially encoded layer A is $2^{n_d} - n_s$ where n_s is the number of subcodes of c_A in the *padded* reduced dichotomy table and n_d is the number of don't care bit positions in c_A.

		A	B	C	D	E	F	G	H	I	J
[A,E],[A,I],[I,J]	$[G, H], H$	0	–	–	–	0	–	1	1	0	0
[D,E]	$[G, H], [I, J]$	–	–	–	0	0	–	1	1	1	1
		–	–	–	–	–	–	–	–	–	–
		–	–	–	–	–	–	–	–	–	–

The remaining task of the algorithm is to assign an available code to each partially encoded layer while sustaining the number of available codes for other partially encoded layers greater than 0. If no code can be found that keeps the number of available codes for other layers greater than 0, then the algorithm pads another row.

A final code assignment for the example is shown below.

A	B	C	D	E	F	G	H	I	J
0	1	0	0	0	0	1	1	0	0
0	0	0	0	0	1	1	1	1	1
0	0	1	1	0	1	0	0	0	0
1	0	1	0	0	0	0	1	0	1

4.4 Combinational Logic Synthesis

4.4.1 Two-Level AND-OR Implementation

The *3D synthesis tool* generates required cubes and privileged pairs (privileged cubes and associated start states) for each primary output and internal state variable function. Logic minimization is performed using exact algorithms for hazard-free logic, implemented in an automated logic minimizer [49]. This hazard-free logic minimizer, using a variation of Quine-McCluskey algorithm, finds an optimal hazard-free cover of *required cubes* using *dynamic-hazard-free implicants*, implicants that do not illegally intersect *privileged cubes*.

4.4.2 BDD-Based Multi-Level Implementation

Our strategy is to build an ROBDD using a *global* variable ordering, if such an ordering can be found, or to build a free BDD. If no global order exists, there is always a variable that can appear first. This variable partitions the function into a left and right BDD. The left and right BDDs do not have to have the same variable order, so they can be constructed recursively using the same method.

In the 3D implementation of extended-burst-mode machines using Type I or II machine cycles, no state burst (fed-back state variable changes) ever enables a state variable to change. Furthermore, all fed-back state variables remain constant in the generalized transition cubes for input and output bursts. During the state minimization step, two specification states with conflicting ordering requirements are placed in different layers, which means that two specification states with conflicting ordering requirements are placed in different partitions. We showed that conflicting ordering requirements between two input bursts from the same specification state can be resolved by selecting a variable ordering such that unique compulsory signals appear first. Therefore, it is always possible to satisfy the variable ordering requirements in each partition.

The *3D synthesis tool* generates extended-burst-mode transition cubes (start cube and end cube pairs) and the variable ordering constraints for each primary output and internal state variable function. The actual BDD construction and logic minimization is carried out using the combinational synthesis tool by Lin and Devadas [35].

4.5 Experimental Results

The synthesis procedure is completely automated. The 3D synthesis tool transforms a textual extended-burst mode specification into the next-state table with a symbolic layer assignment, derives a layer diagram, performs a critical-race-free layer encoding, generates required cubes and privileged pair sets for outputs and state variables for the two-level combinational synthesis, and generates extended-burst-mode transition cubes and variable ordering constraints for the BDD-based combinational synthesis. The logic minimization is performed by the combinational synthesis tools discussed in the last section.

Numerous experiments have shown that the 3D synthesis tool produces results that have smaller area and are much faster than competing asynchronous synthesis methods. There are two measures that characterize the performance of the 3D machines: latency and cycle time. The *latency* is the delay from the last terminating input edge of an input burst to the last edge of the resulting output burst. The *cycle time* is the delay required to avoid circuit malfunction from the last terminating input edge of an input burst to the first compulsory input edge of the next input burst.

4.5.1 Examples Using Two-Level Synthesis

Experimental results are shown in table 4.1. The latencies and the cycle times are evaluated using a 0.8μm CMOS standard cell library, developed for the Verilog simulator by the Torch group at Stanford University [36]. The library cells were characterized using the SPICE simulator under military worst-case conditions (4.5V power supply, 125°C) and derated for the nominal case (5V, 25°C). The two-level logic equations produced by the 3D tool were mapped to the standard cell library either manually or using a hazard-nonincreasing technology mapper [64].

The largest example we have evaluated is a *pipelined SCSI bus controller (asynchronous data transfer protocol).* The extended-burst-mode specification of the asynchronous data transfer protocol of the pipelined SCSI bus controller has 45 states and 62 state transitions, 10 primary inputs, and 5 primary outputs. The 3D synthesis tool added 5 internal state variables, and the implementation required 108 product terms and 378 literals. The Verilog simulation results for the output latency and the cycle time were 3.3ns and 6.1ns.

4.5.2 Comparison to Locally-Clocked Methods

We compare the 3D and the locally-clocked method [52, 49] for four controller implementations: pe-send-ifc, dramc, 5380fsm, and cache-ctrl (see table 4.2).[2] It is interesting to compare to the locally-clocked method, because (1) the locally-clocked method uses the burst-mode as the user-level specification formalism, which means that every machine synthesized using the locally-clocked method can be reimplemented in 3D, and (2) the locally-clocked method has been used for many practical large scale controller designs.

The 3D results are based on two-level AND-OR implementations. pe-send-ifc, dramc, and 5380fsm are designed for Type II machine cycles, and cache-ctrl for Type I. The delay estimates are based on comparison of output literal counts. The area

[2]No extended-burst-mode machine implementations are compared because the locally clocked synthesis only works for burst-mode machines.

	Specification				**Implementation**				
	States / Transitions		Primary In	Primary Out	State Vars	Prod Terms	Lits	Latency	Cycle Time
D-FF	4	6	2	1	1	8	21	1.8ns	2.3ns
chu-ad-opt	4	4	3	3	0	4	11	1.2ns	1.2ns
vanbek-ad-opt	3	3	3	3	0	4	9	1.3ns	1.3ns
dme	8	10	3	3	2	11	29	2.0ns	3.1ns
dme-fast	8	10	3	3	2	12	29	1.7ns	2.9ns
alloc-outbound	8	9	4	3	2	12	27	1.8ns	3.0ns
mp-forward-pkt	4	4	3	4	0	6	14	1.4ns	1.4ns
nak-pa	6	6	4	5	1	10	17	1.7ns	2.5ns
pe-send-ifc	11	14	5	3	2	21	60	2.3ns	3.7ns
pe-rcv-ifc	12	15	4	4	2	26	72	2.1ns	3.8ns
ram-read-sbuf	8	8	5	5	0	13	22	1.7ns	1.7ns
rcv-setup	6	8	3	2	0	3	8	1.4ns	1.4ns
sbuf-ram-write	6	6	5	5	1	18	41	1.9ns	3.2ns
sbuf-read-ctl	7	8	3	3	1	8	17	1.5ns	2.6ns
sbuf-send-ctl	8	9	3	3	2	14	32	2.1ns	3.3ns
sbuf-send-pkt2	7	10	4	2	2	11	30	2.2ns	3.5ns
sendr-done	3	3	2	1	1	4	8	1.0ns	2.4ns
sic-example	6	12	2	1	1	6	13	1.5ns	2.5ns
dram-controller	12	14	7	6	1	20	46	2.2ns	2.2ns
scsi-tsend-bm	11	13	5	4	2	27	58	2.3ns	3.8ns
scsi-trcv-bm	10	12	5	4	2	24	55	2.3ns	3.4ns
scsi-isend-bm	10	12	5	4	2	25	62	2.5ns	3.9ns
scsi-tsend-csm	10	11	5	4	2	24	44	2.2ns	3.3ns
scsi-trcv-csm	8	9	5	4	2	23	42	2.3ns	3.6ns
scsi-isend-csm	8	9	5	4	2	24	42	1.9ns	3.4ns
pscsi-isend	9	11	4	3	3	28	80	2.9ns	4.4ns
pscsi-ircv	6	7	4	3	2	14	31	1.7ns	3.2ns
pscsi-tsend	10	12	4	3	3	26	70	2.2ns	4.3ns
pscsi-trcv	6	7	4	3	1	14	25	2.2ns	2.6ns
pscsi-tsend-bm	10	12	4	4	3	23	60	2.0ns	3.7ns
pscsi-trcv-bm	7	9	4	4	2	21	47	2.0ns	3.8ns
pscsi	45	62	10	5	5	108	378	3.3ns	6.1ns

Table 4.1: Experimental results.

reduction is mainly due to the lack of a local clock whereas the output latency reduction is a result of greater degrees of freedom in logic minimization due to the greater number of inputs to logic functions, with the addition of primary outputs as inputs to logic functions. Literal and product counts do not include the latch overhead of the locally-clocked machine. We also ignore area overhead due to feedback delays, which we assume is negligible. In a typical 1μm gate array implementation, we estimate an additional saving of about 2ns in output latency over the locally-clocked implementation due to the lack of latches. If the locally clocked implementations use dynamic latches, then the saving would be about .7ns. The cost of latch removal in the 3D machines is an increase in state variable logic and greater constraints on state encoding, since the encodings must be critical-race-free.

	Literals				Product terms				Latency Reduction	Area Reduction
	Output		Total		Output		Total			
	LC	3D	LC	3D	LC	3D	LC	3D		
pe-send-ifc	47	45	79	60	15	15	25	21	4%	24%
dramc	51	40	64	46	20	17	23	20	22%	28%
5380fsm	217	181	459	345	59	54	114	100	17%	25%
cache-ctrl	720	494	886	532	215	161	245	172	31%	40%

Table 4.2: Comparisons to locally-clocked machine.

4.5.3 Experimental Results Using BDD Synthesis

We used the 3D synthesis tool [78] in conjunction with the combinational synthesis tool [35] to perform experiments (see table 4.3) on many examples previously synthesized by the method described in [75]. With very modest efforts to find the optimal variable order,[3] most of the examples required less area than the two-level method, primarily because of the reduction in the number of state variables due to simpler

[3]Experiments were conducted by Bill Lin at IMEC, Belgium. He tried a few random orderings and picked the best result.

state assignment. For a minority of the examples, area increased somewhat. We believe that the area results will be further improved with the development of heuristic variable ordering algorithms tuned to our application.

A more important issue is output latency. Out of 39 examples synthesized, 24 of them (the names with *) previously required state variable changes before output changes for some of the specified state transitions. In these cases, using BDD-based synthesis rather than 2-level synthesis improved the output latency by 100% or more.

	Specification				**State vars added**		**Total literals**	
	States / Transitions		Primary In	Primary Out	*2-level*	*BDD*	*2-level*	*BDD*
iccad93ex*	3	4	2	2	2	0	20	6
edac93ex*	4	5	3	2	2	1	32	20
condtest*	4	5	3	2	2	1	30	23
dff1*	4	6	2	2	2	0	28	12
dff2*	4	6	2	2	2	0	28	12
sr2*	8	12	2	3	3	2	82	55
sr2x2*	8	20	3	3	4	2	131	130
q42	4	4	2	2	1	1	27	24
select2ph*	4	8	2	2	2	0	42	56
selmerge2ph*	8	12	3	2	2	1	89	47
sin	13	17	3	4	3	4	71	108
ring-counter	8	8	1	2	1	1	45	64
binary-counter	32	32	1	4	3	3	94	80
binary-counter-co	32	32	1	5	3	3	104	88
pe-send-ifc	11	14	5	3	2	2	90	115
pe-rcv-ifc	12	15	4	4	3	2	84	85
dramc	12	14	7	6	1	0	71	76
cache-ctrl	38	49	16	19	1	1	704	1379
tsend*	22	30	7	4	5	5	328	583
isend*	24	32	7	4	5	7	490	887
trcv*	16	22	7	4	3	2	175	111
ircv*	16	22	7	4	3	2	188	124
tsend-bm*	11	14	6	4	2	2	96	88
trcv-bm*	8	10	6	4	3	1	77	104
isend-bm*	12	15	6	4	3	3	177	103
ircv-bm*	8	10	6	4	3	1	80	78
tsend-csm*	11	14	6	4	4	3	92	67
trcv-csm*	8	10	6	4	3	2	70	72
isend-csm*	12	15	6	4	3	4	142	81
ircv-csm*	8	10	6	4	3	2	80	76
abcs	23	33	9	7	3	3	199	278
stetson-p1	31	38	13	14	3	3	376	754
stetson-p2	25	27	8	12	4	4	178	319
stetson-p3	8	11	4	2	1	0	16	7
biu-fifo2dma*	11	13	5	2	5	5	125	166
fifocellctrl	3	3	2	2	1	1	16	20
scsi-targ-send*	7	8	4	2	3	3	53	50
scsi-init-send*	7	8	4	2	2	2	31	37
scsi-init-rcv-sync	4	5	3	1	1	1	20	16

Table 4.3: Comparing two-level vs BDD.

Chapter 5

Design Example: SCSI Controller

In this chapter, we describe a version of a commercial SCSI controller data path. The purpose of this exercise is to demonstrate that the extended-burst-mode specification and the 3D implementation indeed provide a powerful mechanism to design and implement controllers operating in heterogeneous environment.

All the work related to the discussions in this chapter was performed as an independent project at Advanced Micro Devices, Sunnyvale, CA., during the summer of 1993 under the supervision of Mark Knecht. All AMD proprietary information has been omitted. Although this information can be used to construct a part of a commercial SCSI controller, it does not represent any specific commercial product.

5.1 Overview

The SCSI (Small Computer Systems Interface) as defined by ANSI standard X3.131-1986 (SCSI-1) and augmented in ANSI X3T9.2/86-109 Rev. 10c (SCSI-2) is a logical and physical protocol for communication between computers and peripheral devices such as disk and tape drives (see figure 5.1). A maximum of eight SCSI devices are allowed on the SCSI bus with communication allowed between only two SCSI devices at a time — an initiator and a target. SCSI controllers support the physical layer of the SCSI bus protocol (bus arbitration and data transfer). In this chapter, we describe an implementation of the SCSI controller data path (BIU, FIFO, and SCSI

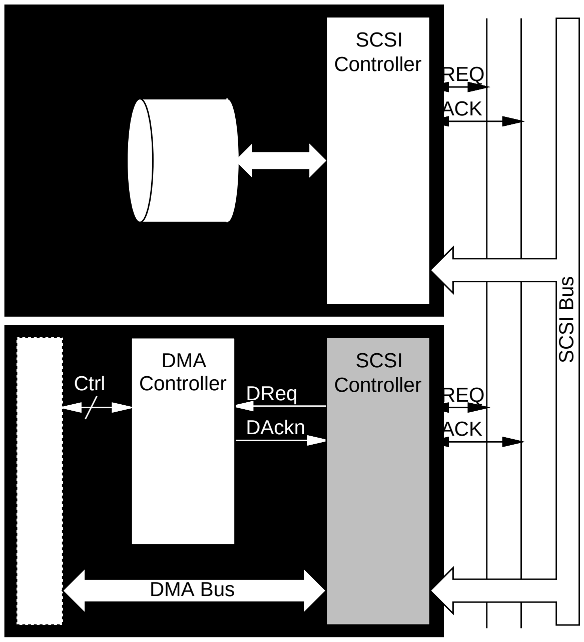


Figure 5.1: A simple configuration of SCSI bus.

Bus Interface) and associated control circuits.

The SCSI controller communicates with two interfaces during data transfer: the SCSI device's local DMA (Direct Memory Access) bus and the SCSI bus. The controller regulates the flow of data between two buses. This implementation (see figure 5.2) assumes that the DMA bus uses a common protocol obeyed by most conventional DMA controllers as described in section 5.2.1.

A SCSI device is configured in one of four operating modes before a data transfer operation begins: *Target-Send*, *Target-Receive*, *Initiator-Send*, and *Initiator-Receive*. The initiator originates the data transfer operation by requesting the target to begin a handshaking protocol. The sender moves data from the local bus to the SCSI bus; the receiver moves data from the SCSI bus to the local bus. This implementation supports all four operating modes.

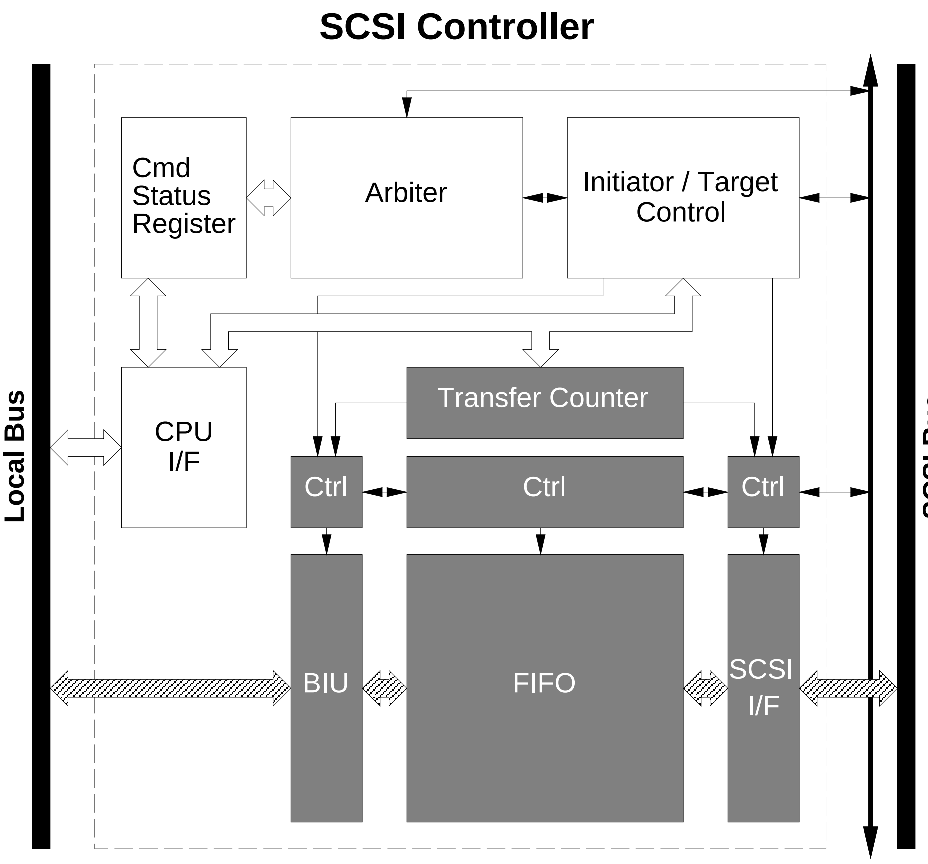


Figure 5.2: SCSI controller block diagram.

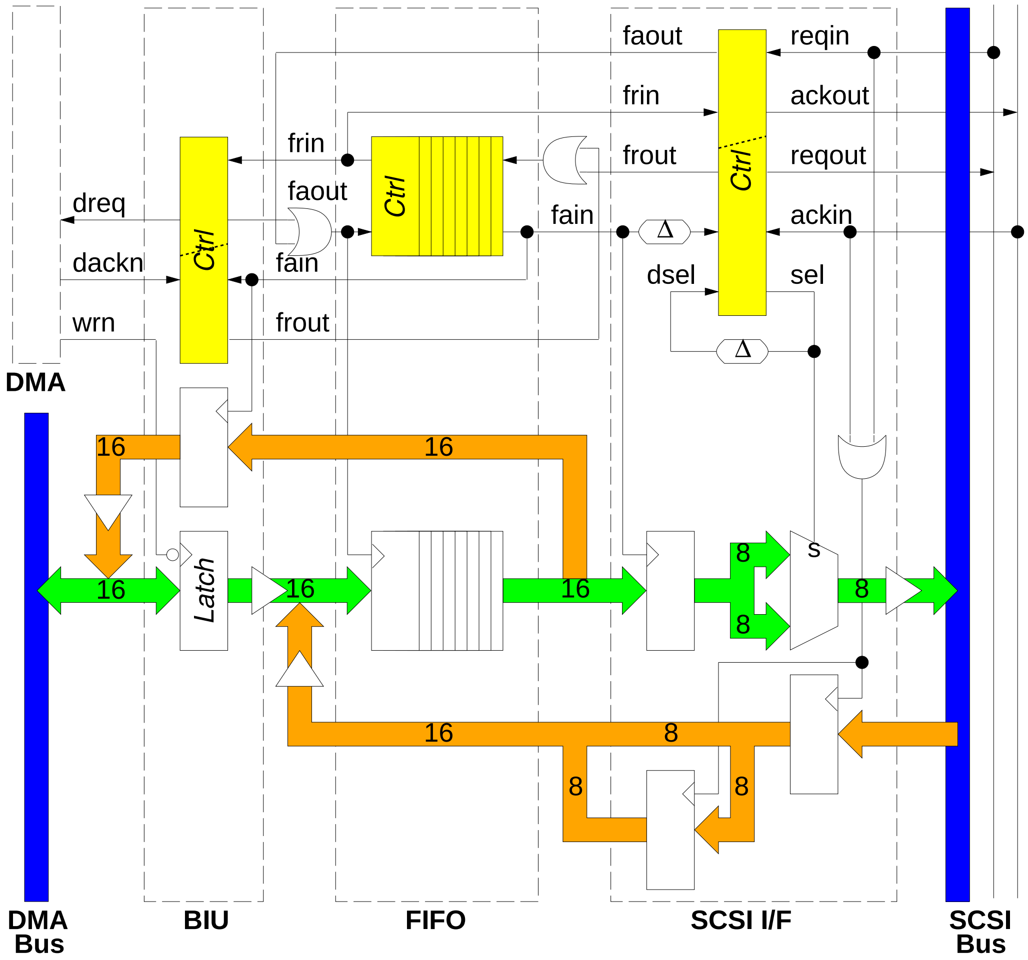


Figure 5.3: SCSI controller data path.

5.2 Implementation

The SCSI controller data path is a bidirectional asynchronous pipeline, which works as a FIFO buffer and a protocol converter. It is somewhat similar to Ivan Sutherland's micropipeline [65].

The SCSI controller data path must have a FIFO to buffer the data because of the disparity in transfer rates between two interfaces. The maximum throughput is 1-10 Mbytes/s on the SCSI side and 5-33 Mbytes/s on the DMA side. The data transfer on the DMA side tends to be "bursty" because the DMA bus is shared among many devices competing for access, but the data transfer between the SCSI devices tends to be continuous because an entire data block, which is generally quite large (sometimes up to 16Mbytes), is transferred at a time without interruption, unless the target device requests interrupts. Thus, the SCSI controller with an internal FIFO functions as a "rate smoother".

The SCSI controller must also bridge the differences in data transfer protocols of two interfaces. The data transfer protocol on the DMA side is "quasi-asynchronous" handshaking (described in detail in section 5.2.1), as defined by widely used DMA controllers [31], and the *default* data transfer protocol on the SCSI side, as defined by the SCSI specification, is asynchronous handshaking. This makes the timing of events on the DMA side and the SCSI side completely independent, and, as such, the asynchronous pipeline is a natural bridge between two interfaces. However, we cannot use a conventional delay-insensitive micropipelines for two reasons: the handshaking on the DMA side depends on timing — communications cannot be delay-insensitive — and the SCSI side must also support synchronous transfer modes. Instead, a combination of pipelined registers, as in micropipelines, and the extended-burst-mode 3D machines, as the control circuits for these registers, is used, because the extended-burst-mode machines are equally adept at handling timing-dependent communications as they are at delay-insensitive communications.

The SCSI controller data path (see figure 5.3) consists of BIU (Bus Interface Unit), FIFO, and SCSI Bus Interface, and the control circuits for these modules.

5.2.1 BIU (Bus Interface Unit)

The BIU (Bus Interface Unit) is used as the interface between the DMA bus and the FIFO. It is designed to obey a common protocol between a DMA controller and slave devices (see figure 5.4). A slave device initiates a DMA access by asserting dreq (DMA request). The DMA controller acknowledges by asserting dackn (DMA acknowledge) for a fixed assertion time (typically 35-100ns). Note that dackn is an active low signal. Valid data is driven onto the DMA bus while dackn is active, and dreq must remain asserted until dackn is asserted. After dackn is negated, dreq may remain asserted or become negated; however, if it is to be negated, it must be done before dackn is asserted again. If dreq remains asserted after dackn is negated, the DMA controller asserts dackn again after a fixed negation time (typically 35-100ns).

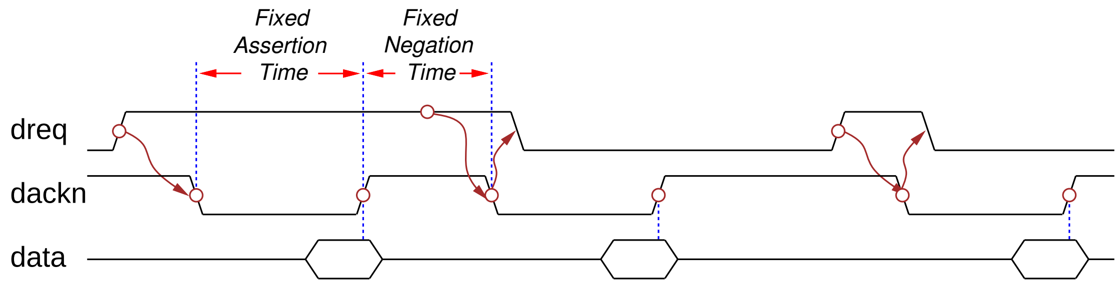


Figure 5.4: DMA protocol.

This SCSI controller has two options: (1) asserting dreq continuously until a desired amount of data is transferred or until the FIFO is full, and (2) asserting and negating dreq for each data word. In this implementation, the second option was selected.

Keeping dreq asserted continuously has a drawback because of the difficulty in synchronizing the FIFO events with the DMA access. Consider, for example, the data transfer from the DMA bus to the FIFO. The FIFO must not be overrun, that is, the SCSI controller must halt the DMA access before the FIFO becomes full. The only mechanism available to halt the DMA access is to negate dreq. Negating dreq must be done after dackn is negated but before dackn is asserted according to the protocol described above. The only way to do that is to negate dreq shortly after dackn is negated using the same reference clock as dackn, which can be viewed as "synchronizing" dreq to dackn. Since dreq is negated by a FIFO event, which is independent of DMA events, there is a finite probability of synchronization failure [12].

On the other hand, asserting and negating dreq for each data word can degrade the transfer throughput. Suppose that the SCSI controller keeps dreq deasserted for a "long" period (longer than it takes for the DMA controller to detect that dreq by the SCSI controller is negated). Then the DMA controller may opt to service another slave device actively requesting a DMA access. When the SCSI controller does request a DMA access by asserting dreq, it may need to wait a significant period before the DMA controller grants the bus access to the SCSI controller.

Note that the synchronization failure will not be eliminated from the *system* as long as this "quasi-asynchronous" handshaking, which requires *sampling* of dreq to decide the next course of action, is used. By using the second option (asserting and negating dreq for each data word), we removed the problem only from the *SCSI controller*, not from the system.

The DMA data bus is assumed to be 16 bits wide and the SCSI bus is 8 bits wide. One of the design decisions made was to use a 16-bit FIFO over an 8-bit one. A 16-bit FIFO was selected primarily to reduce the latency from the DMA bus to the FIFO.

The BIU consists of an input data latch and an output data register and two extended-burst-mode state machines (one for the data transfer direction from DMA to FIFO and the other for the direction from FIFO to DMA). We describe the data transfer direction from DMA to FIFO. The block diagram, the controller and the timing diagram of this portion of the BIU are shown in figure 5.5.

Data Transfer from DMA to FIFO

A data transfer operation begins when the BIU receives a go-ahead signal, ok^+, from the master control. The master control asserts ok^+ when the Transfer Counter is loaded with a value greater than 0. The BIU then makes a data transfer request to the DMA controller by asserting dreq. The DMA controller acknowledges the request by asserting dackn and wrn and placing a data word on the bus. The BIU then

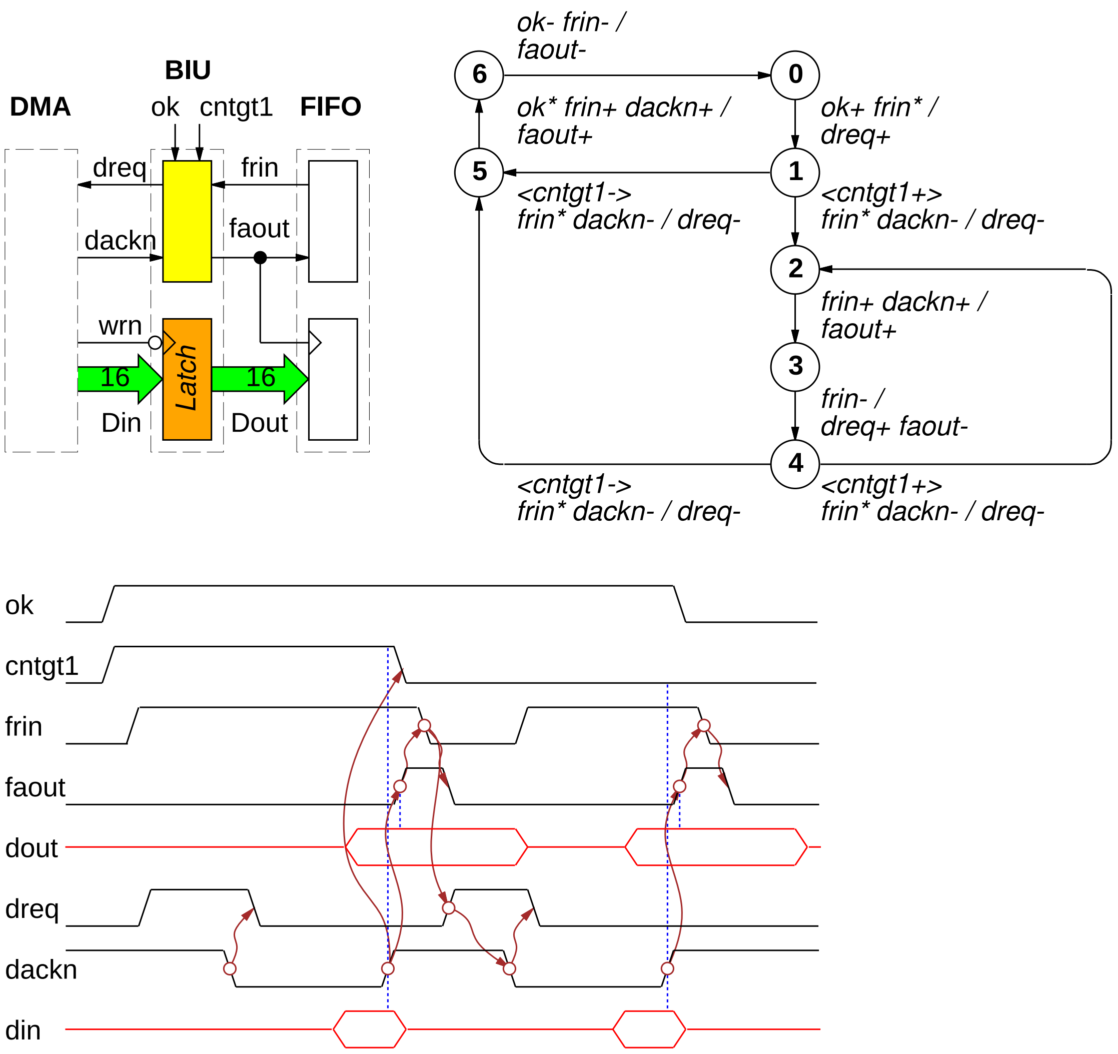


Figure 5.5: BIU (data transfer from DMA to FIFO).

negates dreq immediately. The data is latched into the input data latch, available to the FIFO, when dackn is negated. After dackn is negated and frin is asserted, signaling that the FIFO is ready to receive a data word, the BIU acknowledges the FIFO by asserting faout. The data word is latched into the FIFO cell at the rising edge of faout. When the FIFO negates frin signaling that the FIFO has received the data, the BIU negates faout and asserts dreq again, requesting a new data word. This is repeated until the transfer count is down to 1 (cntgt1 is sampled low when dackn is asserted), at which point dreq is negated for the last time. After the last data word is latched and after frin is asserted, faout is asserted. When ok and frin are negated, the BIU completes the data transfer operation by negating faout.

It is important to note the following:

1. the handshaking protocol between the BIU and the FIFO is a conventional 4-phase type ($\mathsf{frin}^+ \rightarrow \mathsf{faout}^+ \rightarrow \mathsf{frin}^- \rightarrow \mathsf{faout}^-$);

2. a transparent latch is used for storing the input data, which means that the Dout is valid even before dackn is negated. This is to ensure that data is set up in sufficient time before faout is asserted.

3. frin is specified as a directed don't care in state 4, which means it may be asserted immediately after faout is negated, although it won't be acknowledged until after dackn is negated. This is to make DMA events and FIFO events as concurrent as possible.

4. If frin is asserted before dackn is negated, that is, the FIFO can accept the data upon its arrival, then only two events (faout^+ and frin^-) are required between the negation of dackn and the new assertion of dreq, which is in the order of several gate delays. Typically, the DMA controller samples dreq one clock cycle (35-50ns) after dackn is negated, which is considerably longer than several gate delays. Therefore, the SCSI controller will not "lose" the DMA bus as long as the FIFO is not full.

5. The Transfer Counter is decremented each time dackn is negated, and the transfer count is sampled each time dackn is asserted. Because the negation time for

dackn is 35-100ns, the Transfer Counter needs to generate cntgt1 flag in about 30ns, which is not a very harsh requirement.[1]

5.2.2 FIFO

This implementation of the FIFO is a simple asynchronous pipeline, selected for the simplicity of design. The size of the FIFO is 16 bits wide and 8 cells deep. Each cell communicates with its neighbors via 4-phase handshaking protocol. One end of the FIFO interfaces to the BIU and the other end to the SCSI Bus Interface. Communications between the BIU and the FIFO and between the FIFO and the SCSI Bus Interface are done via 4-phase handshaking as well. Every cell is identical so the layout of the FIFO is very regular and the timing characteristics are predictable. Each cell has a 16-bit register and its own tiny control state machine.

Data Transfer from DMA to SCSI

Suppose the FIFO is connected to the DMA on the left and the SCSI on the right and the cells are numbered from left to right. If the data transfer direction is from the DMA to the SCSI (see figure 5.6), cell n requests a data word from its left neighbor (cell $n-1$), by asserting rout, as soon as its own cell is empty, that is, the data has been emptied to its right neighbor (cell $n+1$). When the left neighbor asserts ain signaling that the data is ready, cell n latches the data and negates rout. If the request from the right cell, rin$^+$, is pending, when the left neighbor negates ain, cell n asserts aout immediately. If not, it asserts aout when the request from the right cell arrives. When aout$^+$ is received by the right neighbor, the data is latched into cell $n+1$. When the right neighbor negates rin, signaling that it has received the data, cell n, in turn, negates aout and asserts rout again, requesting a new data word from the left neighbor.

Note that rin is specified as a directed don't care in state 0, again, to make the operation on the right side as concurrent as possible with the operation on the left

[1]This means that a slow but area and power efficient ripple counter is perfectly suitable for this 24 bit Transfer Counter (to count up to 16Mbytes).

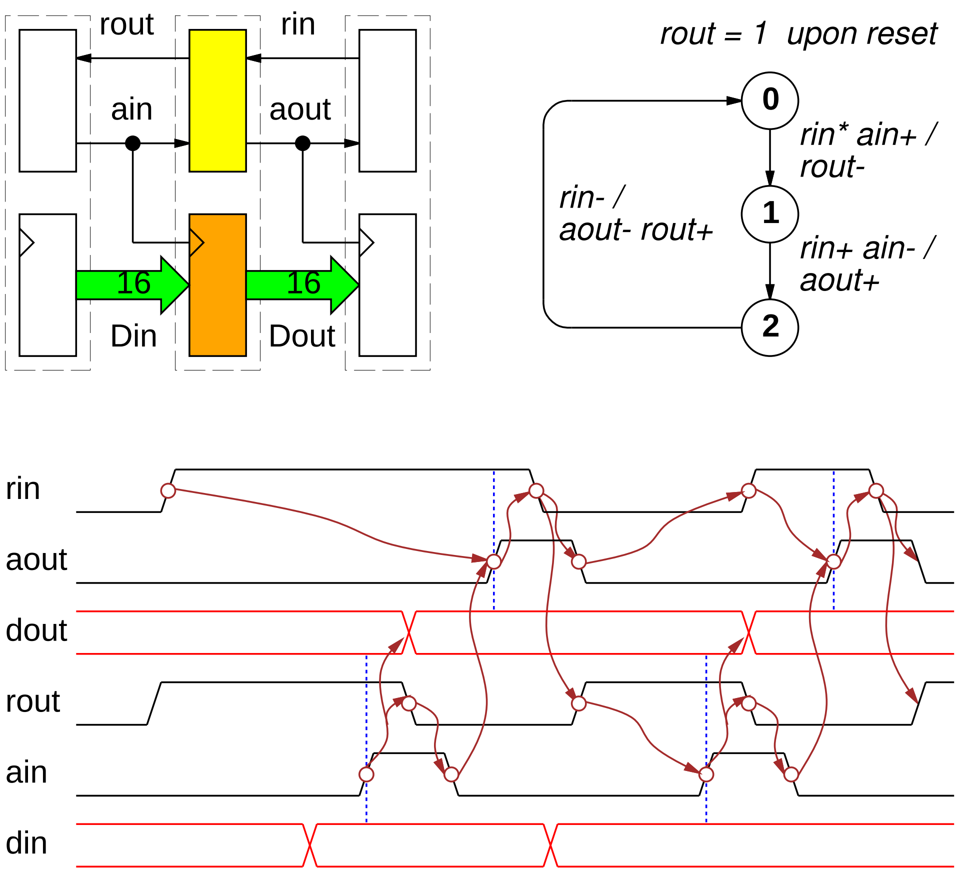


Figure 5.6: FIFO cell (data transfer from DMA to SCSI).

side.

5.2.3 SCSI Bus Interface

The SCSI Bus Interface is used as the interface between the FIFO and the SCSI bus. This implementation uses a state machine for each of four SCSI operating modes: *Initiator-Send*, *Initiator-Receive*, *Target-Send* and *Target-Receive*. There are two types of SCSI data transfer protocols used: asynchronous 4-phase handshaking and synchronous. Although this implementation supports the synchronous protocol, we will omit it here because it is not central to our discussion.[2]

Asynchronous Data Transfer from FIFO to SCSI by Initiator

In the *Initiator-Send* mode (see figure 5.7), a data transfer operation begins when the SCSI Bus Interface receives a go-ahead signal, ok^+, from the master control. The master control asserts ok when the Transfer Counter (not the same one as the one used for BIU) is loaded with a value greater than 0. The SCSI Bus Interface then requests the FIFO for a data word by asserting frout. The FIFO asserts fain, if its rightmost cell has a data word, at which time the data is latched into the SCSI Bus Interface output data register. When the SCSI Bus Interface detects fain asserted, it negates frout. After the request from the target, reqin^+, arrives and after fain is negated, the SCSI Bus Interface acknowledges the target by asserting ackout. Recall that the output data register holds a 16-bit word but the SCSI bus is only 8 bits wide. The sel signal indicates whether the upper or lower byte of the data word stored in the register should be sent out. Initially, sel is 0. When the target signals that it has received a data byte by negating reqin, the SCSI Bus Interface negates ackout and switches sel to 1, preparing for the next byte transfer.

[2]The synchronous protocol still uses Req and Ack wires. However, unlike the asynchronous protocol, the requester (target) does not have to wait for Ack for each data byte. A stream of Req can be sent out as long as the offset between the number of Req and the number of Ack is less than some prescribed amount, which is typically 4-8. Our implementation supports this protocol by adding the interface between the SCSI Bus Interface module and the SCSI bus to handle synchronization and buffering.

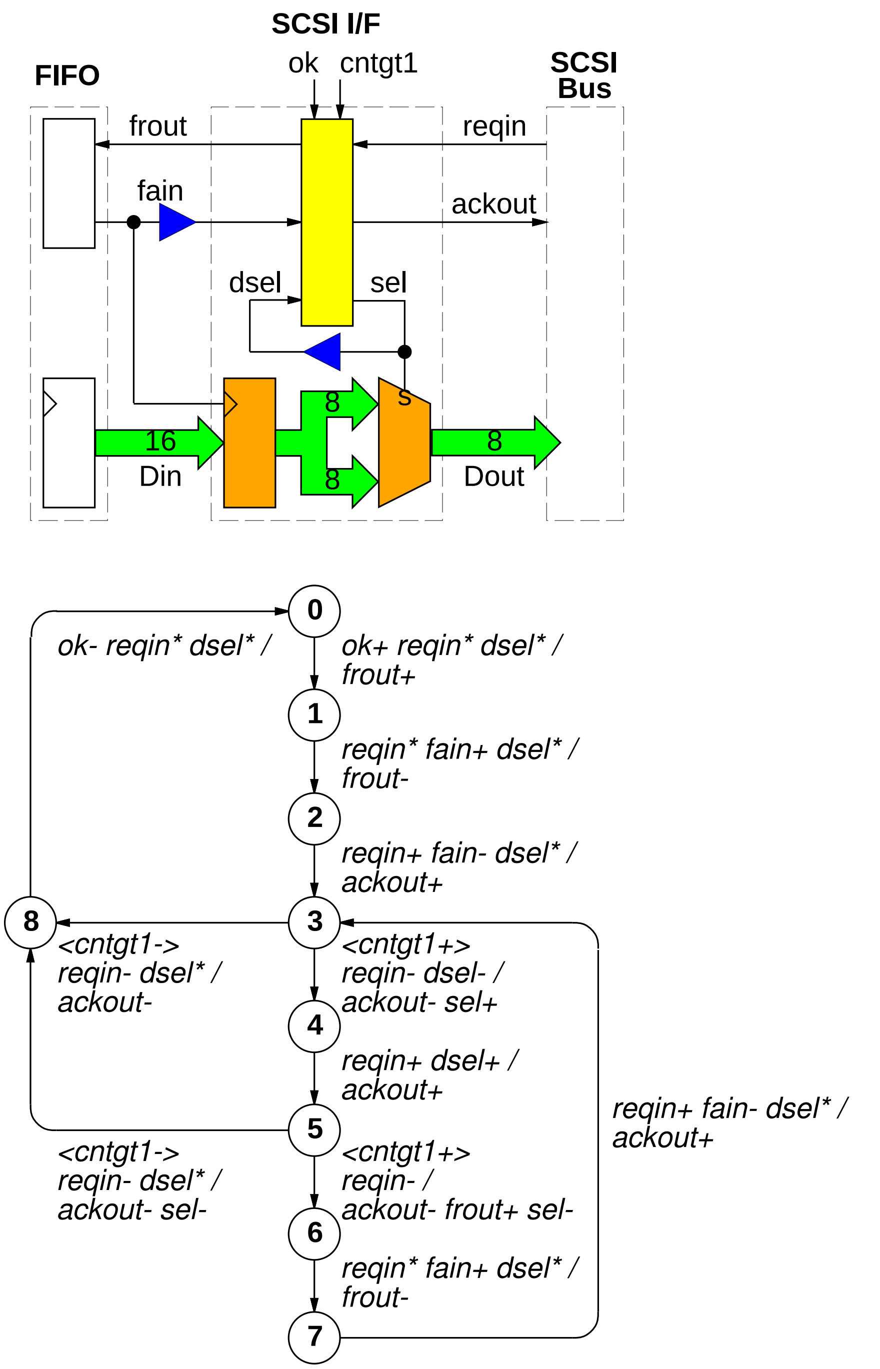


Figure 5.7: SCSI Bus Interface (initiator data transfer from FIFO to SCSI).

There is a delay from when the data is latched into the output data register to when fain assertion is detected by the state machine. This is to guarantee a specified setup time (50ns according to the SCSI bus specification) from the data validation to ackout assertion.[3] dsel, a delayed version of sel, is used for a similar purpose. As stated earlier, at the conclusion of the first byte transfer, the state machine raises sel, which enables dsel to rise after a specified delay ($\geq 50ns$). The SCSI Bus Interface does not assert ackout, even if the request from the target is pending, until dsel has gone high, guaranteeing that the data has been valid for a specified period. When the target negates reqin, signaling that the second byte has been received, the SCSI Bus Interface negates ackout, requests the FIFO to send the next word by asserting frout and lowers sel again.

The data transfer continues until cntgt1 is sampled low when reqin is negated, meaning that the transfer count is down to 1, at which point ackout is negated for the last time and sel is reset to 0, unless it is already 0. The Transfer Counter counts the number of bytes, not words, and the transfer count is updated each time ackout is negated. The last negation of ackout decrements the transfer count to 0. When the master control negates ok, the data transfer operation is complete.

reqin is specified as a directed don't care in states 8, 0, 1, and 6 to make the SCSI events and the FIFO events as concurrent as possible.

5.3 Results

We designed and simulated the SCSI controller data path and the associated control circuits, which support all four SCSI bus operating modes and both asynchronous and synchronous data transfer protocols. In addition, we designed a CPU interface and skeletal high level control circuits to handle the arbitration and various initialization tasks, in order to make all of the data transfer modes demonstrable. The completed

[3] In this implementation, this delay is generated by feeding this signal into a set of flip-flops, clocked by an external clock. The need for generating "long" delays to meet some timing requirements in asynchronous circuits is quite real and warrants further research.

design was mapped to one of AMD's 0.8μm CMOS standard cell libraries and simulated with Verilog simulator modified to handle timing, called *Verilog with Timing*. The design flow is shown in figure 5.8. The design simulated correctly with respect to all of the test vectors used. The performance is limited by the SCSI specification, not the design itself. One of the surprising results, was the competitiveness of the area. Many recent experimental asynchronous microprocessor development efforts [41, 25, 55, 6] reported that the asynchronous design had a substantially larger area compared to functionally equivalent synchronous designs. However, we do not yet know whether the area savings can be replicated in other designs.

This exercise demonstrates that an automated asynchronous design is feasible for commercial-scale circuits and that the extended-burst-mode specification and the 3D tool can be applied to designing a real, cost effective system.

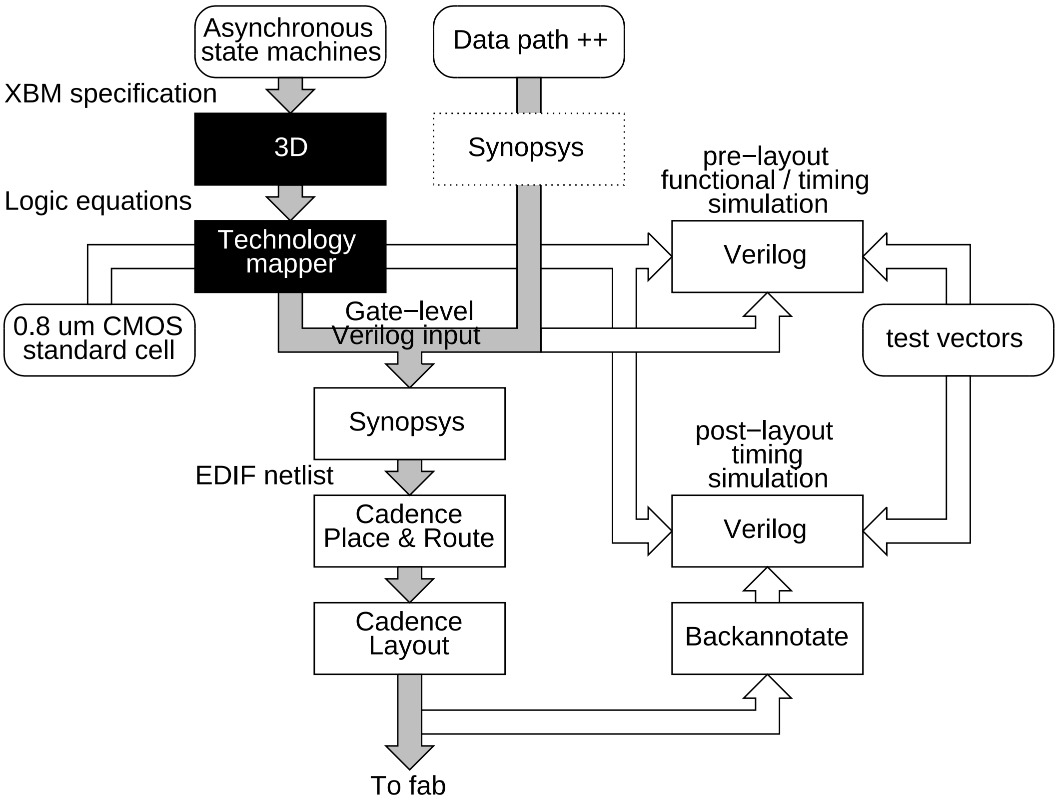


Figure 5.8: SCSI controller design flow.

Chapter 6

Conclusion

6.1 Summary

We have reviewed all aspects of an asynchronous controller design method: user-level specification formalism, hazard-free combinational synthesis theory and its application to sequential synthesis, and automated sequential synthesis algorithms. We also discussed a practical commercial-scale design example.

A summary of the results presented in this thesis is as follows:

Extended-burst-mode design style: We introduced a new specification formalism called extended-burst-mode. We showed that a wide range of practical circuits, both asynchronous and synchronous, can be specified in extended-burst-mode and synthesized using a single synthesis tool. This design style is appropriate for specifying many circuits that fall in the gray area between synchronous and asynchronous which are difficult or impossible to synthesize automatically using competing methods.

Hazard-free combinational synthesis requirements: We described two different hazard-free combinational synthesis methods: two-level sums-of-products and multiplexor trees implementation. We extended the existing theories on hazard-free combinational synthesis to handle non-monotonic input changes and developed a set of requirements for freedom from logic hazards for each combinational synthesis method.

3D automatic synthesis algorithm: We presented a complete set of automated sequential synthesis algorithms: hazard-free state assignment, hazard-free state minimization, and critical-race-free state encoding. We observed that eliminating dynamic hazard was the key factor which set the overall synthesis direction. In fact, the functional synthesis step looks ahead for any possibility of dynamic hazards and takes measures to prevent them. Because each combinational synthesis method has a different set of requirements to avoid dynamic hazards, we observed that the functional synthesis steps needed to be tuned to the selection of the combinational synthesis method. We asserted that the heuristics that find near-optimal solutions in polynomial time do not significantly degrade the quality of the final implementations. Finally, we compared the results with the locally-clocked method and determined that our 3D method produced substantially better results, both in area and in latency. We also compared the two-level AND-OR implementations with the multiplexor tree implementations.

Commercial-scale design example: We reported on the design and implementation of a commercial-scale SCSI controller data path. We showed that our automated asynchronous design completed in a very short period was simulated to be correct and compatible with an equivalent commercial design. More importantly, we demonstrated that the extended-burst-mode design style and the 3D synthesis tool are feasible for real-world designs.

6.2 Future Work

During the course of this research, it was observed that there are many important problems that need to be addressed in order to make asynchronous design truly viable.

Arbitration and synchronization: Most synthesis methods today do not tackle arbitration and synchronization problems directly. It is up to the designers to isolate these problems and manually design them. Clearly, it would be beneficial to the designers to have the synthesis tool deal with these problems automatically. One attractive solution is to extend the semantics of the specification language so that the arbitration behavior can be specified directly in the controller specification and automatically synthesized by the synthesis tool.

Timing analysis: In chapter 3, we observed that it is desirable to minimize feedback delays to improve system performance. Furthermore, in order to construct a reliable system, it is necessary to compute output latencies and cycle times precisely. These require careful timing analyses. A large body of work already exists in this area; however, a timing analysis tool that can weed out a bulk of false paths and still run in polynomial time would be very useful but does not exist today.

Electrical design issues: During the SCSI controller design, it was observed that asynchronous circuits may be more prone to errors stemming from noise than synchronous circuits, because asynchronous circuits are sensitive to every input change. Today, engineers use sound design practices and guidelines to avoid glitches due to ground bounces and reflections in the backplane buses among others. Also there are a few circuit-level remedies that remove narrow glitches caused by electrical noise. However, a systematic system level solution that can selectively ignore certain signal changes would complement sound design practices in solving electrical noise problems.

Delay generation: Asynchronous circuits operating in a heterogeneous system sometimes require lengthy signal delays. In the SCSI controller design, delays of 35-50ns were needed, which were, in fact, generated by a conventional synchronous solution using flip-flops clocked by a high speed clock. An alternative solution that does not require a high speed clock is highly desirable.

Testing: For the asynchronous circuits to be widely accepted, they must be testable. Simple strategies like adding scan latches in the feedback paths work just as well as in synchronous circuits. However, more research needs to be done in combinational logic testing and delay fault testing. In 3D machines, the multiplexor tree implementations have better testable structures than two-level implementations.

Bibliography

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

[2] P. Beerel and T. H.-Y. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proceedings of the 1992 IEEE/ACM International Conference on Computer Aided Design*, November 1992.

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

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

[5] 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, March 1972.

[6] E. Brunvand. The NSR processor. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 428–435. IEEE Computer Society Press, January 1993.

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

[8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[9] J. R. Burch. Delay models for verifying speed-dependent asynchronous circuits. In *Proceedings of the 1992 International Conference on Computer Design: VLSI in Computers and Processors*, pages 270–274. IEEE Computer Society Press, October 1992.

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

[11] S. M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits.* PhD thesis, California Institute of Technology, 1991.

[12] C. J. Chaney and C. E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers (Correspondence)*, C-22(4):421–425, April 1973.

[13] T.-A. Chu. *Synthesis of Self-timed VLSI Circuits from Graph-Theoretic Specifications.* PhD thesis, MIT, 1987. Technical Report MIT-LCS-TR-393.

[14] H. Y. H. Chuang and S. Das. Synthesis of multiple-input change asynchronous machines using controlled excitation and flip-flops. *IEEE Transactions on Computers*, C-22(12):1103–1109, December 1973.

[15] B. Coates, 1993. Private Communication.

[16] W. S. Coates, A. L. Davis, and K. S. Stevens. Automatic synthesis of fast compact self-timed control circuits. In *IFIP Working Conference on Asynchronous Design Methodologies*, Manchester, UK, 1993.

[17] 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.

[18] A. Davis. A data-driven machine architecture suitable for VLSI implementation. In *Proceedings of the Caltech Conference on Very Large Scale Integration*, pages 472–494, January 1979.

[19] Digital Equipment Corporation, Maynard, MA. *DEC Chip 21064-AA RISC Microprocessor Preliminary Data Sheet*, 1992.

[20] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* MIT Press, Cambridge, MA, 1989.

[21] D. Dobberpuhl et al. A 200MHz 64b dual-issue cmos microprocessor. *IEEE Journal of Solid-State Circuits*, 27(11):1555–1565, November 1989.

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

[23] E. B. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM Journal of Research and Development*, March 1965.

[24] A. D. Friedman, R. L. Graham, and J. D. Ullman. Universal single transition time asynchronous state assignments. *IEEE Transactions on Computers*, C-18(6):541–547, 1969.

[25] S. B. Furber. AMULET1 – an asynchronous ARM processor. In *Symposium Record of Hot Chips V*, Stanford, CA, August 1993.

[26] G. C. Gopalakrishnan, P. Kudva, and E. L. Brunvand. Peephole optimization for asynchronous macromodule networks. In *Proceedings of the 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors.* IEEE Computer Society Press, October 1994.

[27] A. Grasselli and F. Luccio. A method for minimizing the number of internal states in incompletely specified sequential networks. *IEEE Transactions on Electronic Computers*, EC-14(3):350–359, June 1965.

[28] A. B. Hayes. Stored state asynchronous sequential circuits. *IEEE Transactions on Computers*, C-30(8):596–600, August 1981.

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

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

[31] Fujitsu Microelectronics Inc. *Fast Track to SCSI: A Product Guide*. Prentice-Hall, 1991.

[32] J. Kessel et al. An error decoder for the compact disc player as an example of vlsi programming. In *Proceedings of the European Design Automation Conference*, pages 69–74, 1992.

[33] M. Ladd and W. P. Birmingham. Synthesis of multiple-input change asynchronous finite state machines. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 309–314. Association for Computing Machinery, June 1991.

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

[35] B. Lin and S. Devadas. Synthesis of hazard-free multi-level logic implementations under multiple-input changes from binary decision diagrams. In *Proceedings of the 1994 IEEE/ACM International Conference on Computer Aided Design*. IEEE Computer Society Press, November 1994.

[36] J. Maneatis and D. Ramsey, 1992. Private communication.

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

[38] A. J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 245–60. Computer Science Press, Inc., 1985.

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

[40] A. J. Martin. Programming in VLSI: From communicating processes to delay-insensitive VLSI circuits. In C. A. R. Hoare, editor, *UT Year of Programming Institute on Concurrent Programming*. Addison-Wesley, 1990.

[41] A. J. Martin, S. M. Burns, T.K. Lee, D. Borković, and P.J. Hazewindus. The design of an asynchronous microprocessor. In C. L. Seitz, editor, *Proceedings of the Decennial Caltech Conference on Very Large Scale Integration*, pages 351–373. MIT Press, 1989.

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

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

[44] C. Mead and L. Conway. *Introduction of VLSI Systems*, chapter 7. Addison-Wesley, 1980. C. L. Seitz, System Timing.

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

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

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

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

[49] S. M. Nowick. *Automatic Synthesis of Burst-mode Asynchronous Controllers*. PhD thesis, Stanford University, 1993.

[50] S. M. Nowick and B. Coates. Automated design of high-performance unclocked state machines. In *Proceedings of the 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors*. IEEE Computer Society Press, October 1994.

[51] 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, October 1993.

[52] S. M. Nowick and D. L. Dill. Synthesis of asynchronous state machines using a local clock. In *Proceedings of the 1991 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 192–197. IEEE Computer Society Press, October 1991.

[53] S. M. Nowick and D. L. Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. In *Proceedings of the 1992 IEEE/ACM International Conference on Computer Aided Design*, pages 626–630, November 1992.

[54] S. M. Nowick, K. Y. Yun, and D. L. Dill. Practical asynchronous controller design. In *Proceedings of the 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 341–345. IEEE Computer Society Press, October 1992.

[55] N. C. Paver. *The design and implementation of an asynchronous microprocessor*. PhD thesis, University of Manchester, 1994.

[56] J. L. Peterson. *Petri net theory and the modeling of systems*. Prentice-Hall, 1981.

[57] S. R. Petrick. A direct determination of the irredundant forms of a boolean function from the set of prime implicants. *AFCRC-TR-56-110 Air Force Cambridge Research Center*, April 1956.

[58] M. Rem, J. van de Snepscheut, and J. Udding. Trace theory and the definition of hierarchical components. In R. Bryant, editor, *Proceedings of the Third Caltech*

Conference on Very Large Scale Integration, pages 225–239. Computer Science Press, 1983.

[59] C. A. Rey and J. Vaucher. Self-synchronized asynchronous sequential machines. *IEEE Transactions on Computers*, C-23(12):1306–1311, December 1974.

[60] R. Rudell. Logic synthesis for VLSI design. Technical Report Memorandum UCB/ERL M89/49, University of California, Berkeley, 1989.

[61] G. Saucier. Encoding of asynchronous sequential networks. *IEEE Transactions on Electronic Computers*, EC-16(6):365–369, 1967.

[62] H. Schols. *Delay-insensitive Communication.* PhD thesis, Technische Universiteit Eindhoven, 1992.

[63] C. J. Seger. *Models and Algorithms for Race Analysis in Asynchronous Circuits.* PhD thesis, University of Waterloo, 1988.

[64] P. Siegel, G. De Micheli, and D. Dill. Automatic technology mapping for generalized fundamental-mode asynchronous designs. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 61–67, June 1993.

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

[66] J. A. Tierno, A. J. Martin, D. Borkovic, and T. K. Lee. A 100-MIPS GaAs asynchronous microprocessor. *IEEE Design & Test of Computers*, 11(2):43–49, Summer 1994.

[67] J. H. Tracey. Internal state assignments for asynchronous sequential machines. *IEEE Transactions on Electronic Computers*, EC-15(8):551–560, August 1966.

[68] J. Udding. *Classification and Composition of Delay-Insensitive Circuits.* PhD thesis, Technische Universiteit Eindhoven, 1984.

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

[70] K. van Berkel et al. A fully asynchronous low-power error corrector for the dcc player. In *Proceedings of the 1994 IEEE International Conference on Solid-state Circuits*, page TA5.4, 1994.

[71] P. Vanbekbergen. *Synthesis of asynchronous controllers from graph-theoretic specifications.* PhD thesis, Interuniversitair Micro-Elektronica Centrum, 1993.

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

[73] C. Ykman-Couvreur, B. Lin, G. Goossens, and H. De Man. Synthesis and optimization of asynchronous controllers based on extended lock graph theory. In *Proceedings of The European Conference on Design Automation with The European Event in ASIC Design*, pages 512–517. IEEE Computer Society Press, February 1993.

[74] K. Y. Yun and D. L. Dill. Automatic synthesis of 3D asynchronous finite-state machines. In *Proceedings of the 1992 IEEE/ACM International Conference on Computer Aided Design*, pages 576–580. IEEE Computer Society Press, November 1992.

[75] K. Y. Yun and D. L. Dill. Unifying synchronous/asynchronous state machine synthesis. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer Aided Design*, pages 255–260. IEEE Computer Society Press, November 1993.

[76] K. Y. Yun, D. L. Dill, and S. M. Nowick. Synthesis of 3D asynchronous state machines. In *Proceedings of the 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 346–350. IEEE Computer Society Press, October 1992.

[77] K. Y. Yun, D. L. Dill, and S. M. Nowick. Practical generalizations of asynchronous state machines. In *Proceedings of The European Conference on Design Automation with The European Event in ASIC Design*, pages 525–530. IEEE Computer Society Press, February 1993.

[78] K. Y. Yun, B. Lin, D. L. Dill, and S. Devadas. Performance-driven synthesis of asynchronous controllers. In *Proceedings of the 1994 IEEE/ACM International Conference on Computer Aided Design*, pages 550–557. IEEE Computer Society Press, November 1994.