AUTOMATIC SYNTHESIS OF BURST-MODE ASYNCHRONOUS CONTROLLERS

Steven Mark Nowick

Technical Report: CSL-TR-95-686

December 1995

This research was supported by the Semiconductor Research Corporation under Contracts 91-DJ-205 and 92-DJ-205; and by the Stanford Center for Integrated Systems, Research Thrust in Synthesis and Verification of Multi-Module Systems.

AUTOMATIC SYNTHESIS OF BURST-MODE ASYNCHRONOUS CONTROLLERS

Steven Mark Nowick

Technical Report: CSL-TR-95-686

December 1995
(revised version of PhD dissertation, March 1993)

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305-4055

Abstract

Asynchronous design has enjoyed a revival of interest recently, as designers seek to eliminate penalties of traditional synchronous design. In principle, asynchronous methods promise to avoid overhead due to clock skew, worst-case design assumptions and resynchronization of asynchronous external inputs. In practice, however, many asynchronous design methods suffer from a number of problems: unsound algorithms (implementations may have hazards), harsh restrictions on the range of designs that can be handled (single-input changes only), incompatibility with existing design styles and inefficiency in the resulting circuits.

This thesis presents a new locally-clocked design method for the synthesis of asynchronous controllers. The method has been automated, is proven correct and produces high-performance implementations which are hazard-free at the gate-level. Implementations allow multiple-input changes and handle a relatively unconstrained class of behaviors (called "burst-mode" specifications). The method produces state-machine implementations with a minimal or near-minimal number of states. Implementations can be easily built in such common VLSI design styles as gate-array, standard cell and full-custom. Realizations typically have the latency of their combinational logic.

A complete set of state and logic minimization algorithms has been developed and automated for the synthesis method. The logic minimization algorithm differs from existing algorithms since it generates two-level minimized logic which is also hazard-free.

The synthesis program is used to produce competitive implementations for several published designs. In addition, a large real-world controller is designed as a case study: an asynchronous second-level cache controller for a new RISC processor.

Key Words and Phrases: asynchronous, sequential synthesis, hazards, hazard-free logic, state machines, cache controller, logic minimization.

Acknowledgments

Many people have contributed to my enjoyment and education at Stanford.

My adviser, David Dill, has given me superb guidance throughout my graduate studies. I began working with Dave on the problem of verification of asynchronous circuits. My work eventually shifted to the problem of designing correct circuits. In both areas, Dave has been a source of quality insights, patience and encouragement. He has taught me much of what I know about research, writing and problem-solving. Working with Dave has made my Stanford experience immensely rewarding.

The other members of my committee — Teresa Meng, Giovanni De Micheli and Al Davis — have been unusually helpful. Teresa Meng first introduced me to many of the intricacies of asynchronous design. Our discussions prompted me to consider more deeply the problem of gate-level hazards. Her enthusiasm and insights have benefited me immensely.

Giovanni De Micheli has helped me with several technical issues in hazard-free logic minimization. He has also given me insight into the relationship between my work and synchronous synthesis methods. He has been a real source of encouragement, and I have enjoyed our discussions on future extensions to this work.

Al Davis was originally my project leader at HP Labs for two summers, where I worked on asynchronous verification for a chip design in his Mayfly Parallel Processing System. Much of my understanding of asynchronous state machine design comes from our conversations. The work that he and his group have done on asynchronous design has had a direct impact on this thesis. I have greatly enjoyed working with Al.

This thesis work has benefited from the support provided by the Semiconductor Research Corporation, under Contracts 91-DJ-205 and 92-DJ-205; and by the Stanford Center for Integrated Systems, Research Thrust in Synthesis and Verification of Multi-Module Systems.

I would also like to thank Bill Coates and Ken Stevens, of the HP Labs Mayfly Project, for two great summers at HP and many insightful discussions since then.

At Stanford, the past and present members of the asynchronous group have made life enjoyable. They have also contributed to my understanding of asynchronous design: Peter Beerel, Mark Dean, Chris Myers, Polly Siegel, Ted Williams and Ken Yun. I have especially benefited from collaborations with Mark and Ken.

Polly Siegel and Bill Coates have given me useful comments on a draft of this thesis.

I have also enjoyed many discussions on asynchronous design with Jeremy Gunawardena and Alan Marshall, who visited Stanford from HP Bristol.

Kurt Keutzer and Rick Rudell of Synopsys organized an informal summer seminar on asynchronous design in 1991 which raised many important issues in synthesis and testing. The seminar also helped to develop a sense of community among asynchronous researchers.

Many others have increased my understanding of asynchronous design: Mark Horowitz and Ed McCluskey at Stanford; Luciano Lavagno and Cho Moon at Berkeley; Ganesh Gopalakrishnan, Erik Brunvand, Nick Michell, Luli Josephson and John Hurdle at the University of Utah; Tam-Anh Chu at Cirrus Logic; Venkatesh Akella at UC Davis; Jo Ebergen at the University of Waterloo; Alain Martin at Caltech; Steve Burns at the University of Washington; Mark Yu and P.A. Subrahmanyam at Bell Labs; David Kung at IBM; Farhad Aghdasi at the University of Bristol; and Stephen Unger at Columbia.

I have also enjoyed knowing several of the non-asynchronous students and researchers working with Dave Dill at Stanford: Jerry Burch, Andreas Drexler, Alan Hu, Shoshi Wolf, Howard Wong-Toi and Han Yang.

A number of other friends made life pleasant at Stanford: Marcia Derr, Tom Henzinger, Richard Ho, Andrew Kosoresow, Janet Murdock, Geoff Phipps, Tom Rokicki, Sam Srinivas and Shoshi Wolf.

I would also like to thank my parents and grandparents.

Finally, I especially want to thank Martha Helfer, for her continued support, love and friendship. This thesis is dedicated to her.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Why Asynchronous Design?

Most modern digital systems are *synchronous*. They are organized around a global clock, and system events are synchronized to the clock. The clock produces ticks at regular intervals. On each clock tick, data is latched into storage elements and a new computation begins. Computation must be completed before the next clock tick.

A synchronous paradigm is one of centralized control. Because of the simplicity of this approach, synchronous designs have dominated the industry for years. However, as devices become smaller and faster, and hardware systems become more complex and concurrent, a synchronous approach has become increasingly unwieldy.

Problems with Synchronous Design

There are a number of difficulties with synchronous design.

Clock Skew. In a synchronous system, if the clock is not distributed evenly, *clock skew* results and the system may malfunction. Clock skew is an inherent problem in most synchronous systems. However, in practice, the effects of clock skew can be eliminated in two ways. First, the clock can be slowed down to insure correct operation. That is, a safety margin is added to each clock cycle to insure that the clock has been broadcast throughout the system and all components are stable before a new cycle begins. However, the cost of this approach is a performance loss. Alternatively, clock skew can be minimized by use of carefully balanced clock trees. However, the cost of this approach is an increase in system area.

Asynchronous External Inputs. In a synchronous system, there is a reliability problem when attempting to synchronize inputs which can arrive at arbitrary times. Such inputs may cause synchronous storage elements to enter into undefined states. This problem is called *metastability* [17]. No known method exists to eliminate metastability. However, the probability of entering a metastable state is significantly reduced by using a pair of storage elements to "resynchronize" an asynchronous input to the clock [60]. However, such resynchronization results in a performance loss.

Worst-Case Design. Synchronous designs have difficulty taking advantage of data-dependent processing delays. If a component can process particular inputs or data quickly, its performance is still bound by the global clock speed. In fact, the speed of the clock is usually set assuming worst-case conditions for: *process*, *temperature*, *voltage* and *data*. As a result, even when the system operates under nominal conditions, performance is limited by worst-case design assumptions. In practice, the cumulative "derating" of system performance based on these factors can be significant [30, 100]. Dean [30] indicates that, if such design-for-worst-case could be avoided, many systems would actually run almost twice as fast on average.

Power Consumption. At a time when designers are increasingly interested in low-power applications, the distribution of the clock throughout the system is a large source of power consumption. For example, power consumption of the clock in the recent DEC *Alpha* chip is approximately 17 watts. The problem of power consumption will only grow worse as clock frequency increases and feature size decreases.

Modularity. In a synchronous system, a component cannot be replaced *without* global implications. If the new component is slow, the system may malfunction unless the global clock speed is reduced. If the new component is fast, system performance will not change unless the clock speed can safely be increased. The contrast to modern object-oriented software systems is illuminating. In an object-oriented system, a software module can be replaced without global implications. Such modularity increases the lifetime of a system, allows rapid development, and simplifies system organization. Modularity is an important feature in system design; however it does not fit well with a synchronous paradigm.

Composability. Finally, at a time when designers are interested in constructing large multi-chip systems, synchronous designs have limited composability: it is difficult to combine synchronous subsystems operating at different clock speeds.

An alternative approach is to build *asynchronous* systems. These are systems which contain no global clock; instead, they operate under distributed control. Asynchronous systems promise to avoid many of these problems by eliminating the global clock. However, in spite of much research over the last 40 years [91], asynchronous designs are notoriously difficult to build.

Problems with Asynchronous Design

Most of the problems with asynchronous design center around the phenomenon of *hazards*, or potential glitches on wires [91]. In a synchronous discipline, glitches are permitted at all times except near the clock edge. Glitches are avoided near the clock edge by correct tuning of the clock speed. In other words, glitches are allowed throughout the computation which occurs between clock ticks; a stable result is then latched into storage elements on the next clock tick.

In an asynchronous discipline, however, there is no global clock. Instead, the system may respond to input transitions at any time. As a result, any undesired glitch may cause the system to malfunction. Because of this sensitivity to glitches, asynchronous designs often suffer from a number of problems.

Correctness. Many existing asynchronous design methods do *not* guarantee hazard-free implementations.

Flexibility. Many design methods impose harsh restrictions on the range of behaviors that can be handled, to insure correct operation. Typically, designs are limited to *single-input change* only: once an input changes, no new input change can occur until the system is stable. This restriction aids in the design of correct circuits, since techniques to eliminate hazards for single-input changes are better-known and simpler than those used for more general multiple-input changes [91]. However, the resulting circuits are of limited use.

Compatibility. Many asynchronous methods are incompatible with existing interfaces, such as synchronous interfaces. Instead, they may require the use of particular protocols, such as *four-phase handshaking* only (discussed below). This constraint limits the practicality of asynchronous designs for existing interfaces.

Performance. Finally, in practice, many asynchronous designs have poor performance. Hazards are often eliminated by slowing down circuits by adding delays. This strategy guarantees correct operation, but abandons the potential performance benefits of asynchronous design.

In the past, such difficulties have made asynchronous circuits largely unusable in practical system design. However, there has been substantial progress in overcoming these obstacles in recent work.

The contribution of this dissertation is a new automated method for designing correct asynchronous controllers. In particular, this design method addresses each of these problems: it is correct by construction, allows flexible multiple-input changes, allows compatibility with existing interfaces, and produces high-performance implementations.

Before considering the contribution of this work, it is important to understand the overall context of asynchronous design and the various design styles which have been developed.

1.2 Background and Related Work

1.2.1 Delays, Circuits and Environments

Fundamental to a circuit implementation is the notion of delay. Delays are inherent in any physical circuit; they may also be added explicitly to a circuit using *delay elements*. There are two common models of delay: a *pure delay* model and an *inertial delay* model [91, 92]. A pure delay does not alter the waveform of a signal, but delays it in time. An inertial delay may alter the waveform of a signal. In particular, an "ideal" inertial delay has a threshold period D; pulses of duration less than D are filtered out by the delay.

Delays are also characterized by their timing models. In an *unbounded delay* model, a delay can assume an arbitrary finite value. In a *bounded delay* model, a delay can assume any value within a given time interval. In a *fixed delay* model, a delay is assigned a fixed value.

Delays can be used to model the behavior of wires and gates in a circuit. Typically, a delay is associated with every wire in a circuit. In a *simple-gate*, or *gate-level*, model, a delay is associated with each gate in the circuit. In a *complex-gate* model, a single delay is associated with a network of gates. In this model, the network is assumed to behave like an individual gate.

A *circuit model* is defined using delay models for the gates and wires. The functionality of a gate or component is usually modelled by an instantaneous operator [62].

Equally important is the *environmental model*, describing how a circuit interacts with its surroundings. The circuit and its environment form a closed system called a *complete circuit* [66]. Using the terminology of Brzozowski and Ebergen, if an environment responds to circuit outputs without timing constraints, the circuit operates in *input/output mode* [13]. Otherwise, environmental timing constraints are assumed; for example: the environment cannot respond until the circuit is stable; it must supply input transitions within a fixed time interval; and so on.

1.2.2 The Hierarchy of Asynchronous Circuits

There is a wide spectrum of asynchronous circuits. The simple hierarchy of Figure 1.1 can be used to distinguish designs based on different models of a circuit and its environment.

A *delay-insensitive (DI)* circuit is one which functions correctly regardless of gate and wire delays. That is, an unbounded gate and wire model is assumed. The concept of a delay-insensitive circuit is based on work of Clark and Molnar on Macromodules [22]. Formalizations have been described by Udding [90] and Dill [32]. Few practical DI circuits can be built from simple gates [56]; however, useful circuits can be built from more complex components [33, 44].

A *speed-independent (SI)* circuit is one which functions correctly regardless of gate delays; wires are assumed to have negligible or zero-delay. The original formulation of SI circuits is due to David Muller (see [66]).

A *self-timed* circuit, as described by Seitz [63], consists of a collection of self-timed *elements*. An element is contained in an *equipotential region*, where wires have negligible or known delays. Typically, elements are designed as SI circuits or using timing information. However, no timing assumptions are made on communication between elements.

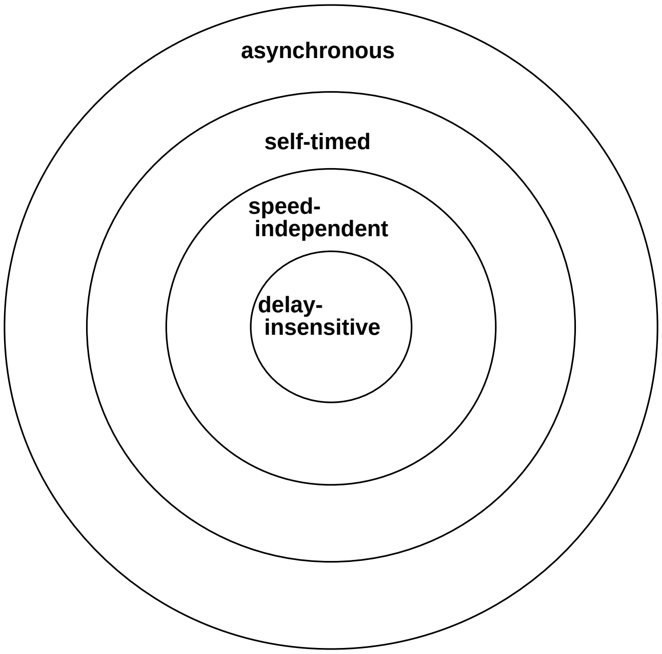


Figure 1.1: A hierarchy of asynchronous circuits.

The above circuits all operate in input/output mode: no timing assumptions are made on communication with the environment. The most general category is *asynchronous* circuits [91]. These circuits have no global clock, but can make use of timing assumptions both within a circuit and in its interaction with the environment. Latches and flipflops [60], with set-up and hold times, fall into this category.

1.2.3 Asynchronous Datapaths and Processors

This thesis is concerned with the design of digital controllers. However, since controllers interact with the datapath, a brief summary of work on asynchronous datapaths and processors will be useful.

Asynchronous datapaths often use a structure which Sutherland calls a *micropipeline* [87]. A micropipeline consists of alternating computation stages separated by storage elements and control circuitry. Asynchronous datapaths have been designed for multiplication [70], division [101], DSP [65] and other applications [50] (see references in [100]). Micropipelines have been generalized to *ring* [100] and *multi-ring* structures [85]. Techniques to eliminate overhead between stages have been developed by Dean [31] and

Williams [100]. Several of the controller design methods described below are also used for datapath synthesis.

Asynchronous microprocessors have been designed by Martin *et al.* [58] and Brunvand [11]. Other approaches include Dean's *STRiP* processor [30], and work by David, Ginosar and Yoeli [24], Ginosar and Michell [37] and Unger [94].

1.2.4 Asynchronous Controllers

There have been many design styles for asynchronous controllers in the last 30 years or so. These fall roughly into three categories: (i) *translation methods*; (ii) *Petri-net*, or *graph-based*, *methods*; and (iii) *asynchronous state machines*.

Translation Methods

Translation methods begin with a specification as a program in a high-level language of concurrency [12, 55, 14, 33, 95, 3]. The program is compiled into an asynchronous circuit by a series of transformations.

In Martin's method [55, 57], specifications are based on Hoare's *CSP* [42]. A specification describes a set of concurrent processes which communicate on channels. The description is transformed, through a series of steps, into a collection of gates and components which communicate on wires. The synthesis method has been automated by Burns [16], who has also developed techniques to analyze and improve circuit performance [15]. The method has been applied to a number of designs: a distributed mutual exclusion element [54], a multiply-accumulate unit [70], an asynchronous microprocessor [58], and several other controllers [14]. Designs typically assume communication using a *four-phase handshaking* protocol. In four-phase handshaking, a request is asserted on a request wire; an acknowledgment is asserted in response on an acknowledge wire. The request and acknowledgment are then deasserted in turn.

The resulting circuits are delay-insensitive, except for one assumption: Martin assumes that, when a wire branches, delays through the different branches are comparable. That is, *isochronic forks* are assumed. The circuits therefore fall between speed-independent and delay-insensitive designs in the circuit hierarchy, and are called *quasi-delay-insensitive* [15].

A similar approach, developed by Brunvand and Sproull [12, 10], uses a variant of *CSP*, called *occam*, to describe a concurrent system. In this method, an *occam* description is first compiled into an unoptimized circuit using syntax-directed translation. The circuit is then improved using automatic local refinements similar to peephole optimizations used in software compilers. Unlike Martin, Brunvand assumes all communication is through a *two-phase handshaking* protocol, also called *transition-signalling* [63, 87]. In this protocol, a request is made by a transition on a request wire; in response, a transition is made on an acknowledge wire.

Brunvand's control logic is delay-insensitive, but his datapaths are self-timed. In particular, two wires — a request and acknowledge — are associated with each collection of data wires. These two wires use transition-signalling to indicate when the data is valid. Sufficient delays are added to insure that these are slower than the datapath. This approach is called a *bundled data* protocol [63, 87].

A different approach has been developed by Ebergen [33] for the design of pure delay-insensitive circuits. Specifications are notated using programs called *commands*, which describe sequences of possible events. These commands are written in a language based on *trace theory* [78], developed by Rem, Snepscheut and Udding as an outgrowth of Hoare's *CSP*. A command is refined through decomposition into a delay-insensitive network of components. As in Brunvand's method, all communication is through a transition-signalling protocol.

Other methods have been developed by van Berkel [95], who has produced a complete automatic synthesis system at Philips Laboratories, and Akella and Gopalakrishnan [3, 38], who generalize the communication style of Martin to include shared variables and broadcast communication.

Translation methods are attractive in several ways: they allow elegant high-level descriptions of concurrent systems, are amenable to formal verification, and can synthesize circuits which have non-deterministic output behavior using arbiters and synchronizers [16, 10]. However, because these approaches rely on local transformations, they often lack flexibility in global optimization, such as state minimization, state assignment and logic minimization. They also rely on slow, specialized sequential components, such as *C-elements* and *toggles*. Two-phase methods in particular rely heavily on the use of *exclusive-or* gates and other slow components such as *decision-wait* elements. The methods also require interface behavior to fit into fixed protocols (e.g. *2-* or *4-phase handshaking*).

Petri-net and Graph-Based Methods

An alternative approach in asynchronous synthesis is to specify behavior using a *Petri net* [77]. A Petri-net is a compact graphical notation which can describe both concurrency and choice between events. A Petri-net has two types of vertices: *places* and *transitions*. Initially, *tokens* are assigned to particular places in the net. These tokens can pass through the net, *firing* transitions which they encounter according to certain *firing rules*. The firing of a transition corresponds to the occurrence of an associated signal or event. Therefore this simulation, or *token game*, can describe many different interleaved sequences of events.

Patil [76] proposed the synthesis of a Petri net as an *asynchronous logic array*, which is a direct mapping of the structure of a Petri net into hardware. Since each place and transition of the original net is mapped to hardware components, the result is quite inefficient.

Recent methods have focused on a behavioral implementation of a Petri-net. These methods typically begin with a constrained class of nets, because of the difficulty of implementing arbitrary concurrent behavior in hardware. Behaviors described by a net are determined using a *reachability analysis*. These behaviors are transformed into a more explicit representation called a *state graph* [19]. The state graph indicates the desired functionality of the implementation, and can be mapped into hardware.

Several simple notations are based on a constrained class of Petri-nets called *marked graphs*: Seitz's *M-Nets* [83], Molnar's *I-Nets* [67], and Rosenblum and Yakovlev's *Signal Graphs* [80]. These nets model concurrency between events, but cannot describe conditional behavior, such as a choice between inputs.

Chu proposed a more general notation called *Signal Transition Graphs* [19], or *STGs*. These are interpreted *free-choice* Petri nets allowing both concurrency and a limited form of choice between inputs. These nets have simple structural properties which can easily be checked. Chu has developed a synthesis method for speed-independent circuits which has been applied to a number of designs: an A-to-D controller, a Resource Locking Module, a distributed mutual exclusion element and several controllers for a routing chip [19, 20]. In general, Chu's STGs cannot be synthesized directly into a circuit without further modifications. Arcs may need to be added to the STG to satisfy a *persistency* requirement. These arcs constrain the concurrency of the net. Also, state variables may need to be added explicitly in the net to avoid *state assignment* problems. This requirement is called the *complete state coding (CSC)* property by Moon *et al.* [68]. A more restrictive requirement is called the *unique state coding (USC)* property by Vanbekbergen [96].

Meng [64, 65] extended this work, developing a complete automated synthesis system. As in Chu's method, arcs may need to be added to an STG before a circuit can be synthesized. However, Meng's algorithms add these arcs in such a way as to allow optimal concurrency in the final circuit. The algorithms take advantage of known delays in the circuit's environment.

An alternative method for adding constraining arcs was developed by Vanbekbergen [96]. The method is based on the concept of a *lock class*, which describes ordering constraints on signals in a marked graph. Vanbekbergen uses timing information when adding arcs, to improve area and speed of the resulting circuits.

Other recent methods include the work of: Lin and Lin [51], who avoid the use of an intermediate state graph during synthesis; and Myers and Meng [69], who use precise timing constraints to optimize the synthesized circuits. Both methods are based on marked graphs; in the latter case, they are notated as *Event-Rule Systems* [15].

These methods have several limitations. First, the syntactic requirements of STGs are restrictive. Though an STG can describe choice between inputs, this ability is severely limited by structural requirements on the underlying net. An additional restriction is that no signal may have more than one up- and down-transition in a net. These constraints make STGs impractical in describing most large systems.

Progress has been made in overcoming these STG restrictions through several generalizations: Chu's *dummy* or *epsilon* transitions [19]; Moon's *don't-care* and *toggle* transitions [68]; and Yakovlev's use of *OR-causality* [102]. However, the modelling of input choice is still often awkward or difficult using extended STG models.

Second, earlier STG-based methods do not include systematic techniques to add state variables. This step can be thought of as *state minimization* and *state assignment* [60].

Progress in this area has been made in recent work by Lavagno *et al.* [48] and Vanbekbergen *et al.* [97]. In general, though, because of the need for *critical race-free codes* (discussed further below), state assignments for STGs may require the use of more state variables than for synchronous implementations.

Finally, and most importantly, the methods of Chu, Meng and Vanbekbergen ignore the issue of gate-level hazards. Instead, they use a complex-gate model, which assumes that combinational logic is built out of large monolithic blocks with no internal delay. However, in fact, the resulting gate-level network may have hazards. To insure correct implementations, Chu and Meng both rely on an automatic verifier by Dill [32].

Some progress has been made in synthesizing hazard-free designs from STGs. Moon *et al.* [68] propose heuristics for hazard-elimination for two-level combinational logic. This method has been successfully applied to a number of small circuits, but in general is not guaranteed to succeed. Lavagno describes a promising approach [47, 49] to synthesizing gate-level hazard-free circuits from STGs. The method requires the use of added delay elements to eliminate hazards. However, the method does not demonstrate freedom from *dynamic hazards* (glitches which can occur when an output is changing value).

An alternative approach by Beerel and Meng [5] is to synthesize speed-independent circuits directly from state graphs. This method avoids many of the syntactic requirements of STGs. Though the method produces hazard-free gate-level implementations, the gates may be quite large. Current techniques to decompose these gates into realistic gate networks may introduce hazards.

Petri-net and graph-based methods have a number of strengths. Specifications can often model fine-grained concurrency and partially-ordered events in a simple and concise way. Several of the synthesis methods have been effective in the design of small, highly concurrent controllers. Furthermore, progress has been made on problems of state assignment and in describing input choice. However, the problem of producing hazard-free gate-level circuits from STGs or state graphs is a major obstacle towards the practical use of these methods.

Asynchronous State Machines

Huffman Machines.

An alternative approach, adopted in this thesis, is the more traditional *asynchronous state machine*. In its simplest form, an asynchronous state machine is a *Huffman machine* (see Figure 1.2), with primary inputs, primary outputs, and fed-back state variables [91]. State is stored on the feedback loops, which may have attached delay elements. Other approaches use unclocked latches to store state. These structures are similar to synchronous state machines [91, 60], but there are no clocked storage elements.

The view of computation in this approach is state-based: in each state, a machine can receive inputs, generate outputs and move to a next state. Specifications can therefore be described by a *flow table* or *state table* [91, 60]. These are symbolic tables describing the output and next-state behavior of the machine as a function of its inputs and current state (see Figure 1.3).

Asynchronous synthesis methods typically follow the same general approach used for synchronous designs [91, 60]. First, *state minimization* is used to compact a flow table by merging states. Symbolic states are then assigned binary codes, so that they can be implemented in digital logic; this step is called *state assignment*. Finally, Boolean output and next-state functions are mapped to combinational logic using *logic minimization*. A major attraction of state machines is that this synthesis method performs global optimization. That is, the synthesis method allows freedom to choose among many possible state reductions, state assignments and logic implementations, which would be difficult or impossible to do with local transformations.

There is a bewildering variety of asynchronous state machine design methods and assumptions. A simple classification scheme is described by Unger [91, 92]. This scheme describes a hierarchy of designs based on input constraints.

1. *single-input change (SIC)*: Only one input change at a time is permitted. Consecutive input changes are separated by a minimal time interval δ.

2. *multiple-input change (MIC)*: Several input signals may change within an interval of length δ_1 and are treated as "simultaneous signals". No further input changes may occur until an interval of length δ_2 has elapsed.

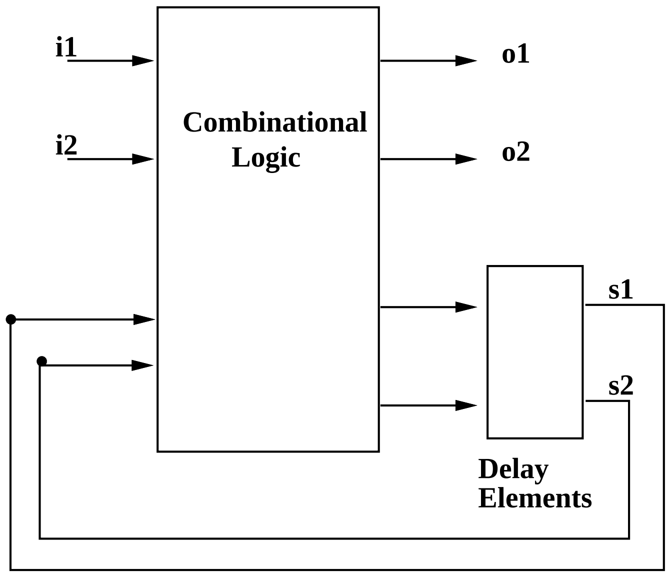


Figure 1.2: Block diagram of Huffman machine.

Next State, Outputs X, Y

inputs a b c

State	000	001	011	010	110	111	101	100
A	A,00	--	--	A,00	B,11	--	--	A,00
B	--	--	--	--	B,11	C,01	--	--
C	--	--	--	--	D,10	C,01	--	--
D	--	--	--	--	D,10	--	--	E,01
E	A,00	--	--	--	--	--	--	E,01

Figure 1.3: An asynchronous flow table.

3. *unrestricted input change (UIC)*: Any input may change at any time, provided that no given input signal changes twice within any period of length δ.

A common more restrictive assumption for SIC circuits is that no input change can occur until a circuit has stabilized. This mode of operation is known as *fundamental mode* [91]. This term is sometimes applied to MIC circuits as well.

In designing SIC circuits, a number of problems must be solved. First, when an input changes, the combinational logic may glitch. In this case, a *logic hazard* occurs. SIC logic hazards can always be avoided using special constraints on logic minimization [91].

Second, an input change may cause a state change, from the current state to the next state. If several state bits change, however, the machine may stabilize in an incorrect transient state instead of the next state. In this case, a *critical race* occurs. This problem can be avoided by using one-hot [91], one-shot [91], Liu [52] or Tracey [89] *critical-race free* state codes. In general these codes require extra state bits.

In addition, logic hazards may occur during a state change. One common approach is to tolerate output hazards. Much of Unger's book [91] is concerned with *S-proper* or *properly-realizable* circuits, for which "output transients" are permitted. Alternatively, hazards can be avoided by constrained logic minimization [34, 4] or by adding inertial delays [91].

Finally, an *essential hazard* can occur if a state change occurs before the machine has "absorbed" the effect of the input change [91]. Essential hazards can always be avoided by adding delays to the feedback path. In certain cases, they can be removed using special logic factoring [4].

In summary, SIC asynchronous machine design is well-understood but complex. Methods usually require added delays, extra state bits and specialized state codes. Some approaches allow output hazards; others rely on inertial delays.

The generalization to MIC design introduces further complications. Friedman and Menon [36] describe methods which require delayed inputs or special "delay boxes" for outputs. Mago [53] presents methods which require delayed inputs. The above methods use augmented flow tables, careful timing requirements and specialized state codes. The delaying of inputs, of course, slows down a circuit. Furthermore, as noted above, MIC designs require that multiple inputs change almost simultaneously. This restriction severely limits the usefulness of such designs.

Unger proposed a further generalization to UIC designs [92]. The method uses inertial delays whose magnitude depends on the number of machine inputs. In fact, the size of delay required by this method may be quite large. In addition, it should be noted that the reliance on inertial delays in this method and others is problematic. Inertial delays are difficult to design, slow down a circuit, are sensitive to process variation, produce slowly rising and falling transitions which are susceptible to noise, and require "recovery time" [92]. Approaches to construct inertial delays from non-inertial components [91] include timing constraints between successive glitches. A general use of inertial delays may in fact be unsound.

An important alternative operating mode, introduced by Davis, fits between MIC and UIC mode in Unger's classification. Unlike MIC mode, Davis makes *no* timing requirements on inputs during a multiple-input change. Instead, inputs in a multiple-input change may arrive at arbitrary times. Such operation allows machines to handle uncorrelated, or concurrent, inputs more easily than in MIC mode. However, unlike the more general UIC mode, inputs are still grouped together into an input change, and fundamental mode is observed. This *data-driven* mode was first used by Davis in the design of the *DDM Machine* [27]. More recently, Davis, Coates and Stevens have developed an automated synthesis system at HP Laboratories [26]. The system has been used to design controllers for an asynchronous Post Office chip [29, 86]. The method produces high-performance implementations; however, it relies on a verifier [32] to guarantee hazard-free designs.

Self-Synchronized Machines.

The difficulty and overhead of hazard-elimination in Huffman machines has led to a design style called *self-synchronized machines*. These are Huffman machines with an attached self-synchronization unit, generating an *aperiodic strobe* which acts like a clock on internal latches (see Figure 1.4). These designs attempt to combine advantages of synchronous and asynchronous machines. Self-synchronized designs exist for SIC [41, 88] and MIC [2, 21, 79] operation.[1] The approach of Ladd and Birmingham [46] requires a single external completion signal, similar to a local clock.

[1] Rey and Vaucher [79] claim that their designs operate in UIC mode, but Unger disproves this claim [93].

Typically a machine is idle until an input change occurs. Clock circuitry then generates a pulse which is applied to the flipflops. In the SIC designs of Hayes [41], the clock is generated by an exclusive-or gate network. In the MIC designs of Chuang and Das [21], it is generated by a combinational logic circuit. An inertial delay is attached to the clock to eliminate hazards. These designs have advantages similar to synchronous designs: they are simple, do not require critical-race free state codes and do not require hazard-free logic for outputs and next-state. In other words, these designs solve many of the problems of Huffman machines. The disadvantage of these designs is poor performance due to worst-case design, added delays and edge-triggered flipflops.

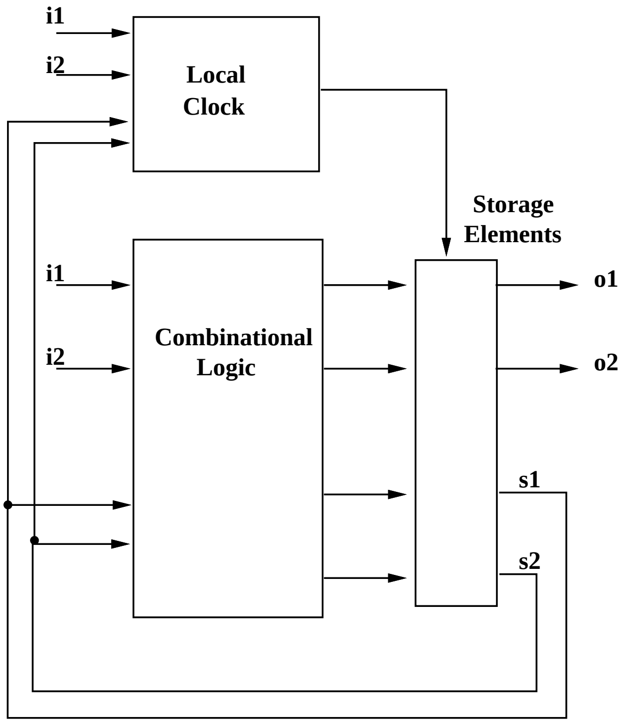


Figure 1.4: Block diagram of self-synchronized state machine.

Mixed-Operation Mode Machines.

The performance problems of self-synchronized machines has led to a hybrid approach, called *mixed-operation mode* [103]. This approach attempts to combine the advantages of Huffman machines and self-synchronized designs. The machines normally operate as Huffman machines, but are self-synchronized as needed to avoid particular problems. The MIC designs of Yenersoy [103] use self-synchronization to avoid critical races. The SIC designs of Chiang and Radhakrishnan [18] use self-synchronization to avoid essential hazards. Yenersoy's approach has the disadvantage of large, complex flipflops; it also does not demonstrate freedom from hazards. Chiang and Radhakrishnan suggest extensions to MIC operation, but do not address the issue of output hazards.

Summary.

There are two important observations based on the above survey of asynchronous state machines. First, both SIC and MIC modes are too constrained to be of general practical use. SIC mode does not allow multiple-input changes; MIC mode is limited to multiple-input changes which are near-simultaneous. Neither mode can handle concurrent, uncorrelated multiple-input changes. On the other hand, UIC mode seems too general to implement in a satisfactory way. Of the modes discussed above, the only manageable but expressive compromise is Davis' data-driven mode. The design style described in this thesis therefore uses a modified form of data-driven mode.

Second, there are three fundamental parameters in asynchronous state machine design: *correctness*, *flexibility* and *performance*. Attempts to increase flexibility of operation, from SIC towards UIC mode, have resulted in designs which are either hazardous or have poor performance. It is desirable to optimize all three parameters at once.

1.3 Scope of the Thesis

This thesis describes a new approach to designing asynchronous controllers. To produce useful results, this work is constrained and focused in a number of ways.

First, it is limited to the design of asynchronous state machines. State machine designs are of interest for several reasons: (a) State machines support global optimization. (b) A state-based approach is often a natural model of computation. This model complements the alternative models used by translation and Petri-net methods. (c) Some existing state machine methods – particularly SIC Huffman methods – demonstrate that high-performance design is possible. And, finally, (d) in many ways, existing approaches have been unsatisfactory for the design of asynchronous state machines.

Second, the thesis is concerned with the synthesis of designs which are hazard-free by construction. This thesis does not consider the use of automatic verification to determine whether designs are correct.

Third, when considering the correctness of designs, it is crucial for circuits to be modelled at a realistic level. It is unacceptable, for example, to regard a large combinational circuit as a monolithic component with no internal structure. Such a model ignores the real possibility of hazards resulting from internal delays in combinational logic networks.

Fourth, the thesis does not consider the use of inertial delays to eliminate hazards. These delays are difficult to build, slow and are potentially unrealizable for many applications.

Finally, the algorithms presented in this thesis are simple. In particular, state and logic minimization algorithms can easily be improved by incorporating more sophisticated techniques. The philosophy of this thesis has been to solve a fundamental synthesis problem, present basic practical algorithms, and leave for future work the refinement and improvement of these algorithms.

1.4 Contributions of the Thesis

The contribution of this dissertation is a new method for designing asynchronous state machines. To insure simplicity, designs are self-synchronized. However, the method avoids the performance problems of earlier self-synchronized designs.

In Section 1.2.4, three parameters of asynchronous design were discussed: *correctness*, *flexibility* and *performance*. The major advance of this thesis is a design style which optimizes all three parameters: (1) Designs are hazard-free at the gate-level. (2) Designs allow multiple-input changes where inputs have no timing constraints. The specifications are based on Davis' data-driven mode. Finally, (3) designs have high performance. Typically, the latency of a machine is comparable to its combinational logic delay. No asynchronous state machine method in the earlier survey has comparable advantages.

In summary, the main contributions of the thesis are the following:

Burst-mode specifications. A new specification style for asynchronous controllers is introduced. This specification style is a constrained and formalized version of Davis' data-driven mode, currently in use at HP Labs [25]. The specifications are in the form of Mealy machines. Unlike many traditional asynchronous state machine specifications, these impose no timing constraints on inputs within a multiple-input change.

Locally-clocked implementations. A new self-synchronized implementation style is presented to implement burst-mode specifications. There are two guiding goals in this design style: (1) designs are guaranteed correct by construction; and (2) designs have high performance (low latency). In practice, there is often a tradeoff in asynchronous design styles. Those with highest performance suffer from hazards; hazards are eliminated by slowing down the circuit. This dissertation presents a solution to this problem: a design style which is hazard-free at the gate-level and yet has low latency.

Automated synthesis algorithms. The dissertation presents a complete set of locally-clocked synthesis algorithms which have been automated. Automation is critical to the practicality of any modern digital design method.

Hazard-free logic minimization. The logic minimization algorithm that is used is actually independent of the particular asynchronous sequential design style. In fact, it is a solution to a general, previously unsolved problem: Given an incompletely-specified Boolean function and a set of multiple-input changes, find a minimal sum-of-products realization that will have no hazards for the specified input changes, if such a solution exists.

Large example. Most asynchronous controller synthesis methods focus on the design of fairly small circuits. For a design method for asynchronous controllers to become acceptable, it is critical to demonstrate that the synthesis method can scale to realistic design problems. The locally-clocked method is applied to a large, realistic example: the design of a high-performance cache controller for a new self-timed RISC architecture. The controller is significantly more complex than existing examples in the literature. In addition, the resulting asynchronous cache subsystem is approximately twice as fast as a comparable synchronous subsystem.

1.5 Overview of the Thesis

The thesis is organized as follows:

After giving basic definitions, Chapter 2 discusses the problem of hazards in combinational logic. Necessary and sufficient conditions are described to insure that combinational logic is hazard-free during a multiple-input change. The conditions only consider the case of an AND-OR implementation. The underlying combinational circuit model assumes arbitrary gate and wire delays. The conditions hold regardless of when the individual inputs change value. This work appeared in the literature 20 years ago, but some of it is not widely known. The chapter concludes by indicating that these conditions cannot always be satisfied for a given collection of multiple-input changes. That is, a hazard-free two-level logic implementation is not always guaranteed to exist.

Chapter 3 presents a new automated method for the synthesis of *locally-clocked state machines*. The machines are self-synchronized, but have some novel features. First, the clock is generated *selectively*: some transitions do not require a clock transition. Second, unlike many self-synchronized methods, the clock unit does not use inertial delays to eliminate hazards. Finally, unlike most self-synchronized design styles, these machines require combinational logic which is hazard-free for given input transitions. From Chapter 2 it is known that such logic may not exist. A major result of this chapter is that, by using small constraints during early steps of synthesis, hazard-free logic can always be produced. In particular, these constraints are added during state minimization.

Once it is shown that hazard-free logic can be produced, the correct sequential behavior of the machine must be guaranteed. One-sided timing requirements are presented, which can always be satisfied by adding delays to the circuit. (Delays are not on the critical path of operation.) Simple algorithms for state minimization and state assignment are then discussed. Logic minimization is deferred until the next chapter. An unoptimized hazard-free logic implementation can be always be constructed using a *primitive cover*.

The machines typically assume a fundamental-mode of operation. However, variants are proposed which function correctly if the environment responds more quickly. The chapter includes a design example: a controller from the HP Post Office chip [29]. The method is also applied successfully to several examples from the literature.

Chapter 4 considers the problem of hazard-free logic minimization. The general problem of hazard-free minimization of two-level logic is considered, where multiple-input changes occur. This problem has not been previously solved. First, the requirements for a *hazard-free cover* are formalized. These requirements describe a constrained covering problem on Karnaugh maps. This problem is solved using a restricted Quine-McCluskey method [60]. The method has been automated, and is applied to a number of designs. The overhead of added logic to insure no hazards is shown to be negligible.

In Chapter 5, the synthesis method is applied to a large example: a second-level cache controller for a new self-timed RISC architecture, called *STRiP* [30]. This chapter represents joint work with Mark E. Dean. The design is significantly more complex than existing state machines in the literature. This chapter demonstrates the ability of the synthesis method to handle large designs. It also shows that a locally-clocked design can fully support the memory interface of STRiP. The resulting cache subsystem has a cache read access which is twice as fast as in a comparable synchronous system.

Chapter 6 presents conclusions, describes recent developments and considers remaining open problems.

Chapter 2

Combinational Circuits and Hazards

2.1 Introduction

This chapter presents a basic model for combinational circuits and discusses the problem of hazards in combinational logic.

The elimination of all hazards from asynchronous designs is an important and difficult problem. Many existing design methods do not guarantee freedom from all hazards; other methods are limited by harsh restrictions on input behavior (single-input changes only) or implementation style (the use of large, slow inertial delays) to insure correct operation. The design of hazard-free combinational logic, in particular, is critical to the correctness of most asynchronous designs.

The focus of this chapter is on combinational circuits which function correctly assuming arbitrary gate and wire delays. Circuits are not considered which depend on bounded delay assumptions for correct operation, or which use added delay elements to fix or filter out glitches.

2.2 Definitions

The following definitions are taken from Rudell [81, 82] with minor modifications (see also [7, 60]). Only single-output functions having binary input and output variables are considered.

Define sets $P = \{0,1\}$ and $B = \{0,1,*\}$. A *Boolean function*, *f*, of n variables, x_1, x_2, $\ldots$, x_n, is defined as a mapping: *f*: $P^n \rightarrow B$. The value "*" in B represents a *don't-care* value of the function. Each element in the domain P^n of function *f* is called a *minterm* of the function. A minterm is also called an *input state* of the function. The value of an input variable x_i in minterm m is denoted by m_i.

The *ON-set* of a function is the set of minterms for which the function has value 1. The *OFF-set* is the set of minterms for which the function has value 0. The *DC-set* (don't-care set) is the set of minterms for which the function has value "*".

Each variable, x_i, has two corresponding *literals*: an *uncomplemented* literal, x_i, and a *complemented* literal, $\overline{x_i}$. The uncomplemented literal, x_i, *evaluates to* 1 for a given minterm m if $m_i = 1$, otherwise it evaluates to 0. The complemented literal $\overline{x_i}$ evaluates to 0 for m if $m_i = 1$, otherwise it evaluates to 1.

A *product* term is a Boolean product, or *AND*, of literals. A product term *evaluates to* 1 for a minterm, m, if each literal in the product evaluates to 1. In this case, the product term is said to *contain* the minterm.

A *cube* is a set of minterms which can be described by a product term.

A *sum-of-products* represents a set of products; it is denoted by Boolean sum of product terms. A sum-of-products is said to contain a minterm if some product in the set contains the minterm.

A product Y *contains* a product X ($X \subseteq Y$) if the cube for X is a subset of the cube for Y. The *intersection* of products X and Y is the set of minterms contained in the intersection of the corresponding cubes.

An *implicant* of a function is a product term which contains no minterm in the function's OFF-set. A *prime implicant* of a function is an implicant contained in no other implicant of the function. An *essential prime implicant* is a prime implicant containing an ON-set minterm contained in no other prime implicant.

A *cover* of a Boolean function is a sum-of-products which contains all of the minterms of the ON-set of the function and none of the minterms of the OFF-set. A cover may also include minterms from the DC-set.

The *two-level logic minimization problem* is to find a minimum-cost cover of a function.

2.3 Background

2.3.1 Circuit and Delay Model

This chapter considers combinational circuits having arbitrary finite gate and wire delay [62, 59]. Each wire is modelled as a connection with an attached or "lumped" delay element, describing the total wire delay. Each gate is modelled as an *instantaneous* Boolean operator with a delay element attached to its output wire, describing the total gate delay. Delays are assumed to have arbitrary finite values. Since delay elements are attached only to wires, this model has been called the *unbounded wire delay* model.

In addition, a *pure delay* model is assumed (see [6]). That is, unlike the *inertial delay* model, it is conservatively assumed that glitches are not filtered out by delays on gates and wires.

The above formalism models a combinational circuit where there is no knowledge about the delays in the individual gates and wires. Since the goal of this chapter is the design of combinational circuits which are hazard-free regardless of actual delays, this is an appropriate model. An interesting feature of this model is that input wires have arbitrary delays. So, even if a "simultaneous" multiple-input change is assumed, the model allows inputs to be skewed by these delays. That is, the model can be used to describe a multiple-input change where the inputs change at arbitrary times.

A *delay assignment* is an assignment of fixed, finite delay values to every gate and wire in a circuit.

2.3.2 Multiple-Input Changes

A *transition cube* (*cf.* [34, 6, 9]) is a cube with a *start point* and an *end point.* Given input states A and B, the transition cube [A,B] from A to B has start point A and end point B and contains all minterms that can be reached during a transition from A to B. More formally, if A and B are described by products, with i-th literals A_i and B_i, respectively, then the product for [A,B] contains precisely those literals A_i such that $A_i = B_i$.

The *open transition cube* [A,B) from A to B is defined as: [A,B] - {B}. In general, an open transition cube is covered by a set of cubes, which contain all minterms in [A,B]

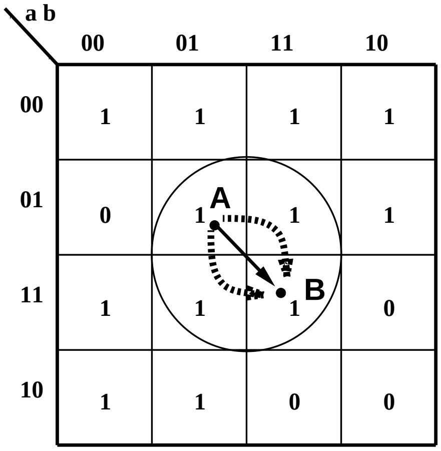


Figure 2.1: Transition cube from A to B.

other than B.

A transition cube captures the intuitive notion of a multiple-input change. A transition cube is characterized by its start point A and end point B, as shown in Figure 2.1. There are a number of possible "trajectories" from A to B. Each trajectory describes a different ordering of input changes from A to B. The set of all trajectories describes all possible orderings of the individual input changes. This set is modelled by the single transition cube [A,B]. Since the goal is to allow inputs in a multiple-input change to arrive at arbitrary times, the transition cube describes the set of all paths from A to B for which the logic must be hazard-free.

A *multiple-input change* or *input transition* from input state A to B is therefore formally described by transition cube [A,B]. There are three properties which characterize a multiple-input change. First, inputs change *concurrently*, in any order and at any time. Equivalently, a *simultaneous* input change can be assumed, since inputs may be skewed arbitrarily by wire delays (see Section 2.3.1). Second, inputs change *monotonically*: each input changes value at most once. And, finally, the input change occurs in *fundamental mode*: once a multiple-input change occurs, no further input changes may occur until the circuit has stabilized.

An input transition occurs during a *transition interval*, $t_I \leq t \leq t_F$, where the input change begins at time t_I and the circuit returns to a steady state at time t_F [6].

An input transition from input state A to B for a Boolean function f is a *static* *transition* if $f(\mathrm{A}) = f(\mathrm{B})$; it is a *dynamic transition* if $f(\mathrm{A}) \neq f(\mathrm{B})$. Function f is said to have a 0→0 transition in transition cube [A,B] if $f(\mathrm{A}) = f(\mathrm{B}) = 0$. Similar definitions apply for 0→1, 1→1 and 1→0 transitions. This chapter considers only static and dynamic transitions where f is fully defined over the transition cube; that is, for every minterm $X \in [A,B]$, $f(X) \in \{0,1\}$. In other words, the function is fully defined within specified input transitions, and is otherwise undefined.

2.3.3 Function Hazards

A function f which does not change monotonically during an input transition is said to have a *function hazard* in the transition. The following definitions are from Bredeson and Hulina [9] (see also [34, 8, 6, 61, 104]).

Definition. A Boolean function f contains a *static function hazard* for the input transition from A to C if and only if:

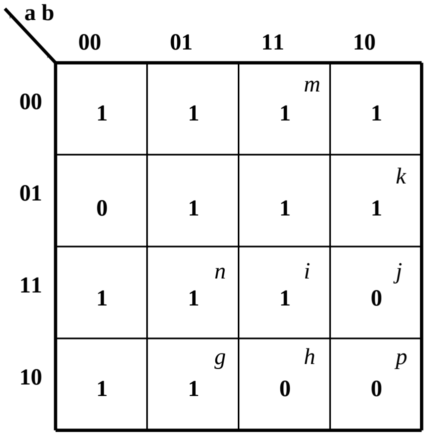
1. $f(A) = f(C)$, and
2. there exists some input state $B \in [A,C]$ such that $f(A) \neq f(B)$.

Definition. A Boolean function f contains a *dynamic function hazard* for the input transition from A to D if and only if:

1. $f(A) \neq f(D)$.
2. There exist a pair of input states B and C ($A \neq B$, $C \neq D$) such that
   (a) $B \in [A,D]$ and $C \in [B,D]$ and
   (b) $f(B) = f(D)$ and $f(A) = f(C)$.

Example. The function f of Figure 2.2 has a static function hazard for the multiple-input change from i to k, since $f(i)=f(k)=1$, $f(j)=0$, and $j \in [i,k]$. The function has a dynamic function hazard for the transition from g to j, since $f(g)=1$, $f(j)=0$, $h \in [g,j]$, $i \in [h,j]$, $f(g)=f(i)=1$ and $f(h)=f(j)=0$. The input transition from k to m is free of static function hazards, and the input transition from n to p is free of dynamic function hazards. □

It is well known that, if a transition has a function hazard, *no* implementation of the function is guaranteed to avoid glitches during the transition, assuming arbitrary



a b	00	01	11	10
00	1	1	*m* 1	1
01	0	1	1	*k* 1
11	1	*n* 1	*i* 1	*j* 0
10	1	*g* 1	*h* 0	*p* 0

Figure 2.2: Boolean function with function hazards.

gate and wire delays [34, 9]. Therefore, in the remainder of this thesis, transitions are assumed to be free of function hazards except where otherwise indicated (*cf.* [34, 8, 6]).

2.3.4 Logic Hazards

If *f* is free of function hazards for a transition from input state A to B, it may still have hazards due to delays in the actual logic realization [91, 9, 6].

Definition. A combinational circuit for a function *f* contains a *static logic hazard* for the input transition from minterm A to minterm B if and only if:

1. *f(A) = f(B).*
2. No static function hazard exists in the transition from A to B.
3. For some delay assignment, the circuit's output is not monotonic during the transition interval.

Definition. A combinational circuit for a function *f* contains a *dynamic logic hazard* for the input transition from minterm A to minterm B if and only if:

1. *f(A)* $\neq$ *f(B).*
2. No dynamic function hazard exists in the transition from A to B.
3. For some delay assignment, the circuit's output is not monotonic during the transition interval.

These definitions formalize the notion that a logic hazard occurs if, for *some* particular gate and wire delays, the combinational circuit output *will* glitch during the transition (a pure delay model is of course assumed).

2.3.5 Hazard-Free Covers

A *hazard-free cover* of a Boolean function *f* is a cover of *f* whose AND-OR implementation is free of logic hazards for a *given set* of specified input transitions.

2.4 Previous Work

Much of the original work on combinational hazards was limited to the case of single-input changes. Methods for detecting and eliminating combinational hazards for single-input changes were developed by Huffman, McCluskey and Unger and are described in [91].

Eichelberger [34] considered the case of static function and logic hazards for multiple-input changes. He indicated that static function hazards cannot be removed; however, static logic hazards can always be eliminated using a sum-of-products implementation containing every prime implicant. Others have developed improved algorithms for selective static hazard elimination.

Dynamic combinational function and logic hazards for multiple-input changes were identified in [91, 9, 6]. Unger [91], Bredeson and Hulina [9], Bredeson [8], Beister [6] and Frackowiak [35] presented conditions to avoid dynamic logic hazards in two-level and multi-level circuits during a multiple-input change. They also indicate that these conditions cannot always be satisfied.

2.5 Conditions for a Hazard-Free Transition

Necessary and sufficient conditions can now be described to insure that a sum-of-products implementation is hazard-free for a given input transition. Assume that [A,B] is the transition cube corresponding to a *function-hazard-free* transition from input state A to B for a combinational function *f*. In the following discussion, assume that C is any cover of *f* implemented in AND-OR logic. It is assumed throughout that no product contains a pair of complementary literals, otherwise additional hazards are possible; see [91].

The following lemmas present necessary and sufficient conditions to insure that the AND-OR implementation of f has *no logic hazards* for the given transition:

Lemma 2.1. If f has a $0 \rightarrow 0$ transition in [A,B], then the implementation is free of logic hazards for the input change from A to B.

Lemma 2.2. If f has a $1 \rightarrow 1$ transition in [A,B], then the implementation is free of logic hazards for the input change from A to B *if and only if* [A,B] is contained in some cube of cover C.

The conditions for the $0 \rightarrow 1$ and $1 \rightarrow 0$ cases are symmetric. Without loss of generality, we consider only a dynamic $1 \rightarrow 0$ transition, where f(A)=1 and f(B)=0. (A $0 \rightarrow 1$ transition from A to B has the same hazards as a $1 \rightarrow 0$ transition from B to A.)

Lemma 2.3. If f has a $1 \rightarrow 0$ transition in [A,B], then the implementation is free of logic hazards for the input change from A to B *if and only if* every cube $c \in C$ intersecting [A,B] also contains A.

Proof. These results follow immediately from pp. 128-9 in [91] and Theorem 3.4 in [35]. See also, Theorem 4 in [9], Lemmas 2 and 3 in [8], Theorem 4.5 in [91], and [6]. □

Lemma 2.2 requires that in a $1 \rightarrow 1$ transition, *some* product holds its value at 1 throughout the transition. Lemma 2.3 insures that no product will glitch *in the middle* of a $1 \rightarrow 0$ transition: all products change value monotonically during the transition. In each case, the implementation will be free of hazards for the given transition.

An immediate consequence of Lemma 2.3 is that if a dynamic transition is free of logic hazards, then every static sub-transition will be free of logic hazards as well.

Corollary 2.1. If f has a $1 \rightarrow 0$ transition from input state A to B which is hazard-free in the implementation, then, for every input state X $\in$[A,B) where f(X)=1, the transition subcube [A,X] is contained in some cube of cover C.

Proof. Since C is a cover of function f, there exists some cube $c \in C$ which contains X. Since f is hazard-free in the transition from A to B, then, by Lemma 2.3, cube c contains A as well; therefore c contains [A,X]. □

Corollary 2.2. If f has a $1 \rightarrow 0$ transition from input state A to B which is hazard-free in the implementation, then for every input state X $\in$ [A,B) where f(X)=1, the static $1 \rightarrow 1$ transition from input state A to X is free of logic hazards.

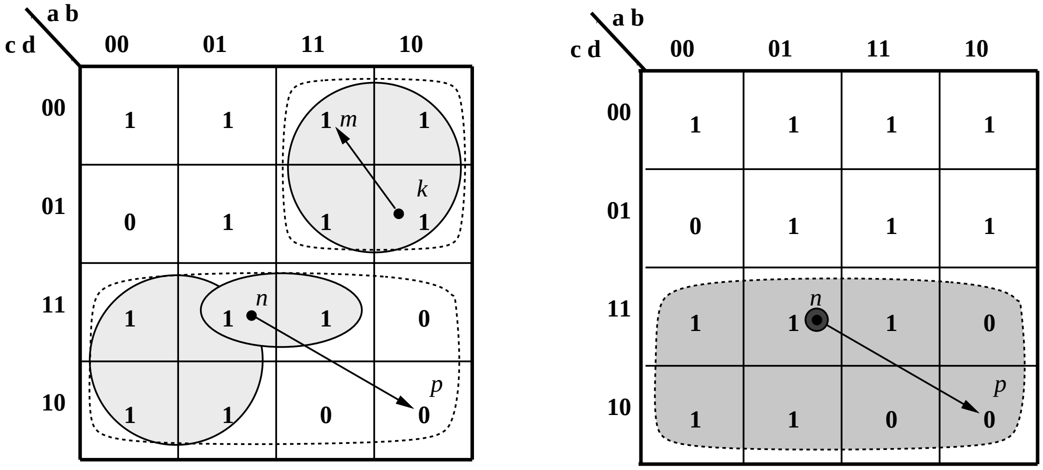
Proof. Immediate from Lemma 2.2 and Corollary 2.1. □

Corollary 2.1 considers the set of transition subcubes of the form [A,X], where input state X $\in$ [A,B) and f(X)=1. If a cube in such a set is contained in no other cube in the set, it is called *maximal* with respect to the set.

Lemma 2.2 and Corollary 2.1 define the covering requirement for a hazard-free transition. The cube [A,B] in Lemma 2.2 and the maximal subcubes [A,X] in Corollary 2.1 are called *required cubes.* These cubes define the ON-set of the function in a transition. Each required cube *must* be contained in some cube of cover C to insure a hazard-free implementation.

Lemma 2.3 constrains the cubes which may be included in a cover C. Each $1 \rightarrow 0$ transition cube is called a *privileged cube*, since no cube c in the cover may intersect a privileged cube unless c contains its *start point.* If a cube intersects a privileged cube but does not contain its start point, it *illegally intersects* the privileged cube and may not be included in the cover.

Figure 2.3 show the required and privileged cubes for two input transitions. The input transition from k to m in Figure 2.2 is free of static function hazards. The corresponding transition cube is a required cube, and is shown in Figure 2.3(a). The input transition from n to p in Figure 2.2 is free of dynamic function hazards. Each maximal ON-set subcube of the transition cube is a required cube, as shown in Figure 2.3(a). Each required cube must be contained in a cube of the cover to avoid hazards, by Lemma 2.2 and Corollary 2.1. In addition, since the transition is dynamic, the entire transition cube, along with its start point, is a privileged cube, as shown in Figure 2.3(b). By Lemma 2.3, to avoid a dynamic hazard, no cube in the cover may intersect this privileged cube if it does not contain its start point, n.



(a) required cubes

(b) privileged cube

Figure 2.3: *Required* and *privileged cubes* for two input transitions.

Hazard Example

Figures 2.4 and 2.5 illustrate the conditions of Lemmas 2.2 and 2.3 and the two Corollaries. Each figure shows a multiple-input change where inputs a and b both change from 0 to 1. The transition is described by a state graph, which represents a portion of a Karnaugh map for the given transition.

Figure 2.4 shows covers for a 1 → 1 transition. The cover in Figure 2.4(a) is hazardous. The dotted ovals represent products in the cover. These products, M and N, correspond to AND-gates in the final AND-OR implementation. Initially, the M AND-gate is high and the N AND-gate is low. To see this graphically, note that cube M contains the start point of the transition, while N does not. To continue the graphical view, a multiple-input change corresponds to a *walk* through the graph. In this example, when input a goes high, the walk never leaves the oval M. That is, AND-gate M remains high. Similarly, N remains low, since the walk never enters oval M.

When input b goes high, the walk leaves oval M and enters oval N. That is, during the transition, the M AND-gate goes low and the N AND-gate goes high. For certain delays, however, the M AND-gate goes low before the N AND-gate goes high, and the circuit output glitches (see timing diagram). In this case, the cover violates the condition of Lemma 2.2.

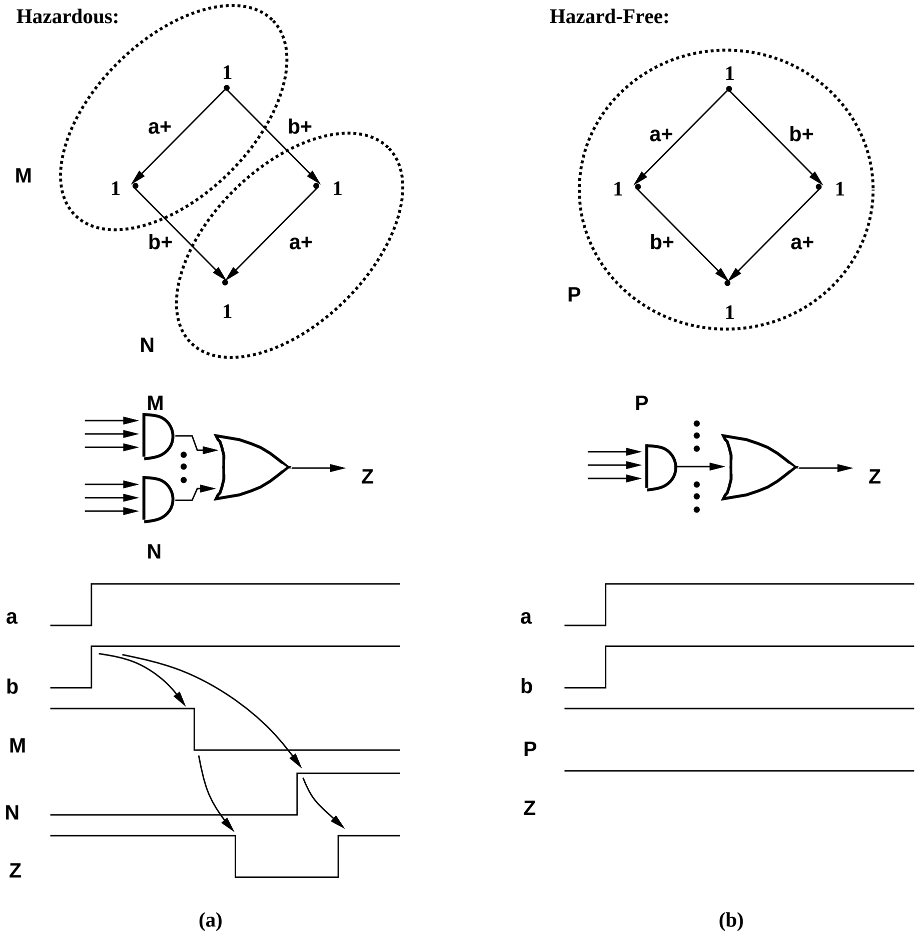


Figure 2.4: Hazardous and hazard-free covers for a 1 → 1 input transition.

The cover in Figure 2.4(b) is hazard-free. As required by Lemma 2.2, the cover contains a product, *P,* which *completely contains* the transition cube. This product corresponds to an AND-gate in the implementation which *holds its value at 1* throughout the transition. Therefore, the circuit output will not glitch (see timing diagram).

Figure 2.5 shows covers for a 1 → 0 transition. The cover in Figure 2.5(a) is hazardous: cubes R and S both illegally intersect the transition.

First, consider the sub-transition where only input a changes; the output must remain at 1. Therefore, this sub-transition is a 1 → 1 transition. However, no single product in the cover contains this sub-transition cube, so the *sub-transition* has a static hazard. In this case, the cover violates the condition of Corollary 2.1. (This case is not described

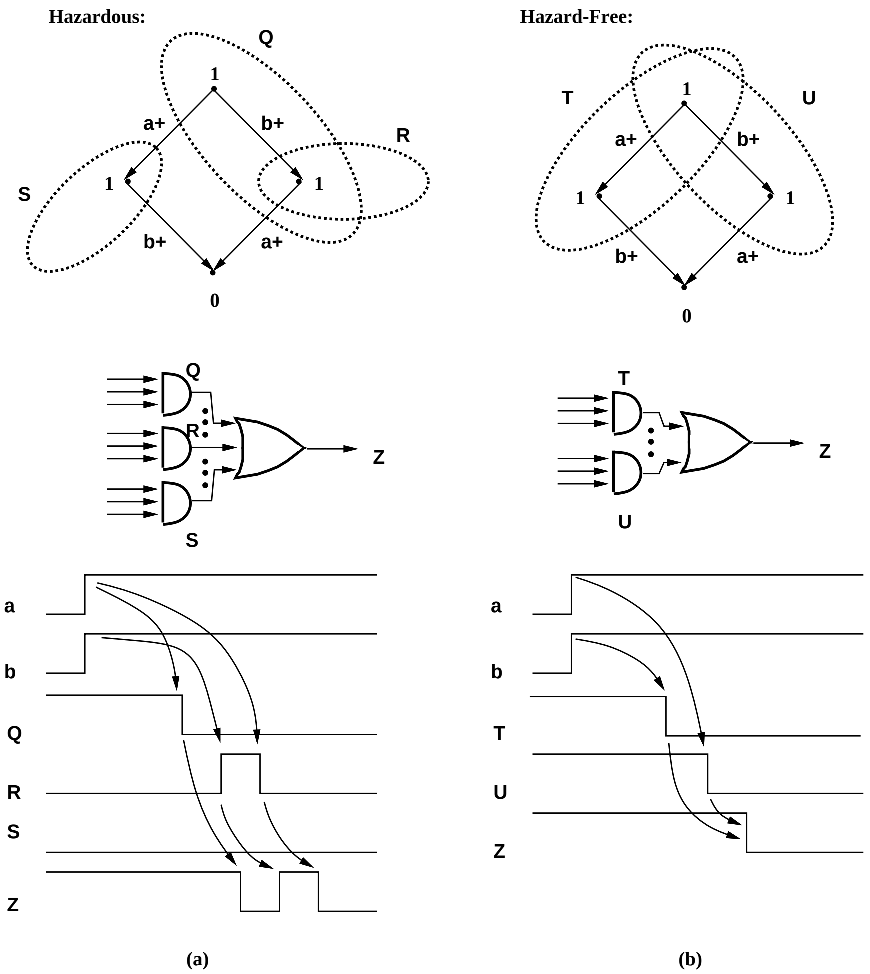


Figure 2.5: Hazardous and hazard-free covers for a 1 → 0 input transition.

by a timing diagram.)

Alternatively, consider the case where input b changes first. This sub-transition is free of static hazards, since product Q covers the static sub-transition. However, a problem remains for the dynamic transition: product R intersects the transition cube in the middle. This stray product corresponds to an AND-gate in the implementation. Initially, this AND-gate is low; it may then go high and then eventually it will go low. During a 1 → 0 transition, such a glitch on an AND-gate can propagate as a glitch to the AND-OR circuit output (see timing diagram). In this case, the cover violates the condition of Lemma 2.3.

The cover in Figure 2.5(b) is hazard-free. Each 1 → 1 sub-transition is completely contained in a product of the cover and there are no "stray" cubes which intersect the transition in the middle (see timing diagram).

2.6 Burst-Mode Transitions

Section 2.5 presented conditions to insure a that a sum-of-products implementation of a function is hazard-free for an arbitrary function-hazard-free transition. In this section, a useful special case is considered: burst-mode transitions. The locally-clocked synthesis method described in Chapter 3 produces combinational functions having burst-mode transitions only. In a burst-mode transition, a function may change only after *every* input in the burst has changed:

Definition. A *burst-mode input transition* from input state A to B, for a combinational function *f*, is an input transition where for every input state C ∈ [A,B), f(A) = f(C).

The following corollary is immediate from the definitions of burst-mode transitions and function hazards:

Corollary 2.3. If a function *f* has a burst-mode transition from input state A to B, then *f* is free of function hazards for the transition.

Lemma 2.3 described the symmetric cases of $1 \rightarrow 0$ and $0 \rightarrow 1$ transitions which were free of function hazards. The symmetry still holds, but the conditions for a $0 \rightarrow 1$ transition are now degenerate and can be simplified:

Lemma 2.4. If *f* has a $0 \rightarrow 1$ *burst-mode* transition in [A,B], then a sum-of-products implementation is free of logic hazards for the input change from A to B.

Proof. This result follows immediately from Lemma 2.3 and the definition of burst-mode transition. (As with the earlier lemmas, we assume that no product has a pair of complementary literals, otherwise additional hazards may be possible.) □

In a burst-mode $0 \rightarrow 1$ transition from input state A to B, the function *f* changes to 1 *only after every input has changed*, that is, in input state B. In this case, no cube c in cover C can intersect [A,B] illegally. That is, under a burst-mode restriction, a $0 \rightarrow 1$ transition can never be illegally intersected: Lemma 2.3 is trivially satisfied by any cover.

To conclude, a burst-mode transition is always free of function hazards. Also, conditions to eliminate logic hazards are simpler than the more general conditions described earlier.

2.7 Existence of a Hazard-Free Cover

Section 2.5 presented conditions to eliminate logic hazards in a given function-hazard-free multiple-input transition. For each such transition, a sum-of-products implementation exists which satisfies these conditions [35]. In general, though, it is desirable is to eliminate logic hazards in a given *set* of multiple-input transitions.

An important result is that, for certain Boolean functions and sets of input transitions, *no hazard-free cover exists* [91, 6]. This fact has important implications for asynchronous sequential circuit synthesis. Any synthesis method which requires hazard-free two-level combinational logic must insure that a hazard-free cover exists for the synthesized Boolean functions. In the synthesis method described in the next chapter, careful constraints are imposed to insure that hazard-free two-level logic can always be synthesized.

Chapter 3

Locally-Clocked Asynchronous State Machines

3.1 Introduction

This chapter describes an automated, correct design methodology for asynchronous state-machine controllers. The goal of this work is a design style which has much of the simplicity of a synchronous design, but with the advantages of an asynchronous method. Our implementations realize asynchronous state-machine specifications using standard combinational logic, level-sensitive latches as storage elements, and a locally-generated clocking signal that pulses whenever there is a change in state.

This design style allows multiple-input changes, where inputs in an input change may arrive at arbitrary times. The implementations use a minimal or near-minimal number of states. The design style also allows arbitrary state encoding and flexibility in logic minimization and gate-level realization, so it can take advantage of systematic CAD optimization techniques.

A novelty of our approach is that for many transitions, the clock is not used – our implementations can function as combinational logic with essentially *zero-overhead penalty.* The machines usually operate in a generalized fundamental mode. With modification, the machines can operate under a variety of environmental assumptions. The designs can be implemented easily in such common VLSI styles as gate-array, standard cell and full-custom. Designs are guaranteed to be free from logic hazards.

The chapter is organized as follows. Burst-mode specifications are described in Section 3.2. The locally-clocked implementation style is described in Section 3.3. An overview of the synthesis method appears in Section 3.4. This section is followed by a detailed formal analysis of the synthesis method in Section 3.5, including a demonstration of the correctness of our method. Different environmental assumptions are considered in Section 3.6, and optimizations are discussed briefly in Section 3.7. State minimization and state assignment algorithms are presented Section 3.8. A real design example is presented in detail in Section 3.9: an asynchronous finite-state controller used in the Post Office communications processor chip [86] developed at HP Laboratories for the Mayfly Parallel Processing System [28, 29]. State-splitting is discussed in Section 3.10, and related work is considered in Section 3.11. The chapter concludes with Section 3.12, where results of our synthesis method are compared with published asynchronous designs.

3.2 Burst-Mode Specifications

An asynchronous state machine allowing multiple-input changes is specified by a form of state diagram, called a *burst-mode specification*, which is derived and formalized from the asynchronous specification style currently in use at HP Laboratories [25]. A state diagram contains a finite number of states, a number of labelled arcs connecting pairs of states, and a distinguished start state (initial wire values are either specified or assumed 0).

Arcs are labelled with possible transitions, taking the system from one state to another. Each transition consists of a non-empty set of input changes (an *input burst*) and a set of output changes (an *output burst*). Note that every input burst must be non-empty; if no inputs change, the system is stable (we do not allow counters).

In a given state, when all the inputs in some input burst have changed value, the system generates the corresponding output burst and moves to a new state. Only specified input changes may occur, and inputs may arrive in arbitrary order and at arbitrary times.

There are two further restrictions to specifications. First, no input burst in a given state can be a subset of another, since otherwise the behavior may be ambiguous (*cf.*

the more general requirements for *delay-insensitive codes* in [98]). This restriction is called the *maximal set property.* Second, a given state is always entered with the same set of input values; that is, each state has a *unique entry point* (this restriction simplifies minimization and helps to guarantee a hazard-free implementation). The maximal set property is a fundamental property of burst-mode behavior. However, the unique-entry point property is simply a syntactic constraint on burst-mode state diagrams: if a state diagram does not satisfy this property, it can be transformed into an equivalent satisfactory diagram by splitting states.

The maximal set property, the unique entry point property, the requirement that inputs can always change in arbitrary order, and the requirement of non-empty input bursts, all distinguish our burst-mode specifications from the data-driven specifications developed by Davis *et al.* [26, 29, 25].

Examples of burst-mode specifications are shown in Figures 3.1 and 3.2. Each transition is labelled with an input burst followed by an output burst. Input and output bursts are separated by a slash, /. A rising transition is indicated by a "+" and a falling transition is indicated by a "–". The burst-mode specification in Figure 3.1 describes a simple controller having 3 inputs (*a, b, c*) and 2 outputs (*x, y*). The specification of Figure 3.2 describes a more complex controller which has been implemented (using a different design style) for the Post Office communication chip developed for the Mayfly Parallel Processing System at HP Laboratories [28, 86]. The machine has 5 inputs (*req-send, treq, rd-iq, adbld-out, ack-pkt*) and 3 outputs (*tack, peack, adbld*).

A more precise definition of a burst-mode specification will be useful for later analysis. Formally, a burst-mode specification is a rooted, labelled, directed graph, $G = (V, E, I, O, v_0, in, out)$, where: V is a finite set of *vertices* (or *states*); $E \subseteq V \times V$ is the set of *edges* (or *transitions*); $I = \{x_1, \ldots, x_m\}$ is the set of *inputs*; $O = \{z_1, \ldots, z_n\}$ is the set of *outputs*; $v_0 \in V$ is the unique *root* (or *start state*); and in and out are labelling functions used to define the unique entry point of each state. In particular, function $in : V \rightarrow \{0,1\}^m$ defines the values of the m inputs and function $out : V \rightarrow \{0,1\}^n$ defines the values of the n outputs at the entry point of each state. The value of input x_i on entering state v is denoted by $in_i(v)$, and the value of output z_j on entering v is denoted by $out_j(v)$.

Given graph G, two edge-labelling functions, $trans_i$ and $trans_o$, can be derived which

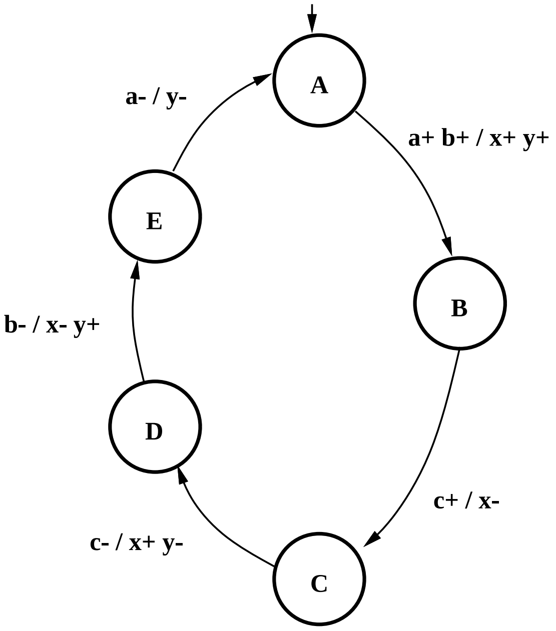


Figure 3.1: Simple example specification.

are useful in specifying the graph. First, for any set S, define the *power set* $\mathrm{P}(S)$ as the set of all subsets of S [43]. Using this notation, $trans_i : E \rightarrow \mathrm{P}(I)$ defines the set of input changes (or *input burst*) and $trans_o : E \rightarrow \mathrm{P}(O)$ defines the set of output changes (or *output burst*) for each edge in the graph. Intuitively, $trans_i$ and $trans_o$ are labelling functions, which associate a set of input changes and output changes, respectively, with each edge in G. For convenience, a "+" or "$-$" can be added to each input and output in a burst to indicate a rising or falling transition. Given any edge, $e = (u, v) \in E$, an input x_i is in the input burst of e precisely if it changes value between u and v. More formally , $x_i \in trans_i(e)$ if and only if $in_i(u) \neq in_i(v)$ ($trans_o$ is similarly defined).

The graph, as defined, guarantees the "unique entry point" property for each state in G. In particular, each state, v, is always entered with input value $in(v)$ and output value $out(v)$. The remaining burst-mode requirements are imposed by adding "well-formedness" constraints on the labelling functions, $trans_i$ and $trans_o$. First, to insure that every input burst in a graph G is non-empty, it is required that, for every edge $e \in E$, $trans_i(e) \neq \phi$. Second, to insure the "maximal set property", it is required that, for each pair of edges, $(u, v), (u, w) \in E$, $trans_i(u, v) \subseteq trans_i(u, w) \rightarrow v = w$. Intuitively, this latter requirement insures that the input changes on edge (u, v) are not

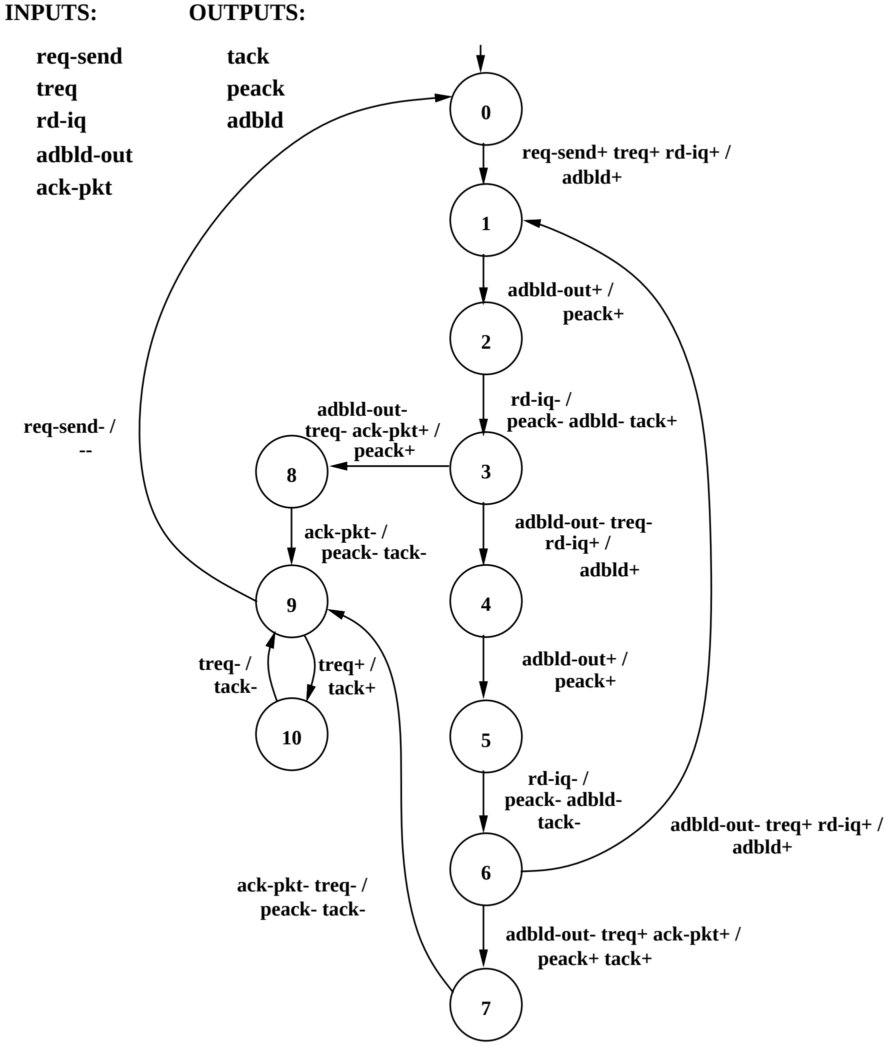

INPUTS:
req-send
treq
rd-iq
adbld-out
ack-pkt
OUTPUTS:
tack
peack
adbld
0
req-send+ treq+ rd-iq+ / adbld+
1
adbld-out+ / peack+
2
rd-iq- / peack- adbld- tack+
3
adbld-out- treq- ack-pkt+ / peack+
8
ack-pkt- / peack- tack-
9
req-send- / --
adbld-out- treq- rd-iq+ / adbld+
4
adbld-out+ / peack+
5
rd-iq- / peack- adbld- tack-
6
adbld-out- treq+ rd-iq+ / adbld+
adbld-out- treq+ ack-pkt+ / peack+ tack+
7
ack-pkt- treq- / peack- tack-
treq- / tack-
treq+ / tack+
10

Figure 3.2: Specification for HP controller (*pe-send-ifc*).

a subset of the input changes on edge (u, w), and vice-versa.

The following proposition elaborates the notion of the maximal set property; it indicates that the input changes on (u, w) never pass through the input state, $in(v)$, which terminates the input changes on (u, v). This proposition follows immediately from the above definitions; it will be useful in later proofs.

Proposition 3.1. Let $G = (V, E, I, O, v_0, in, out)$ be any burst-mode specification, and $(u, v), (u, w) \in E$ be any pair of edges emanating from any state $u \in V$, where v and w are distinct vertices. Then $in(v) \notin [in(u), in(w)]$.

Proof. Recall that a transition cube $[in(u), in(w)]$ has start point $in(u)$, end point $in(w)$, and contains all minterms that can be "reached" in a multiple-input change from $in(u)$ to $in(w)$. Since v and w are distinct vertices, then by the maximal set property, the input changes on (u, v) are not a subset of the input changes on (u, w); that is, $trans_i(u, v) \not\subseteq trans_i(u, w)$. Let $i \in I$ be any input in $trans_i(u, v)$ which is not in $trans_i(u, w)$. By definition of $trans_i$, the value of i in $in(u)$ is the same as the value in $in(w)$ but it is different from the value in $in(v)$; that is, $in(u)_i = in(w)_i$ and $in(u)_i \neq in(v)_i$. Therefore, for each minterm x in the transition cube $[in(u), in(w)]$, $x_i = in(u)_i$ (by definition of transition cube). Since $in(u)_i \neq in(v)_i$, it follows immediately that $in(v) \notin [in(u), in(w)]$. □

Before proceeding to the synthesis of burst-mode specifications, it is necessary to understand the target machine implementation and its operation.

3.3 Locally-Clocked State Machine Implementation

A block diagram of a locally-clocked asynchronous state machine is shown in figure 3.3. The machine consists of combinational logic, storage elements with clock control, primary inputs (*i1, i2*) and outputs (*o1, o2*), and state variables (*s1, s2*) which are fed back as machine inputs.

The local clock, or self-synchronization unit, is used to eliminate a number of possible hazards; it also controls state changes of the machine. However, unlike synchronous design, state changes can occur only when a new input burst arrives; *there is no fixed cycle-time.* Therefore the clock depends only on the current inputs and state. It is

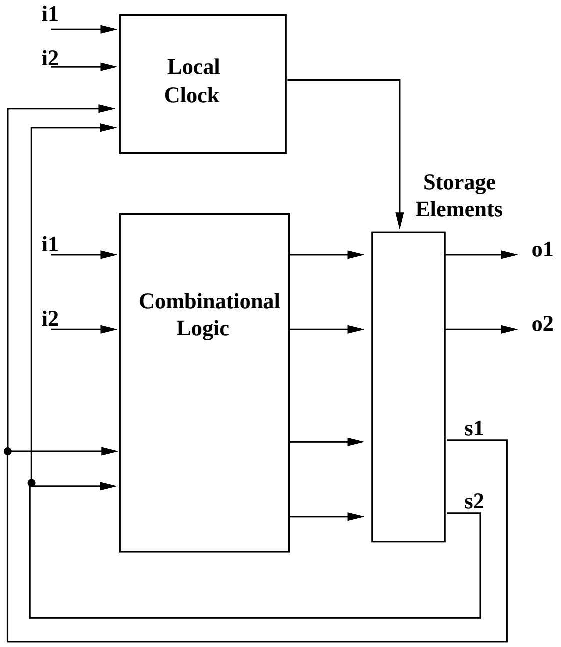


Figure 3.3: Block diagram of locally-clocked asynchronous state machine.

generated *locally* for each module.

Figure 3.4 shows the implementation style in more detail. The *clock* logic itself is unlatched; it is inverted to generate a *two-phase clock* which controls the *phase-1* and *phase-2* D-latches. Each *output variable* has a single phase-1 latch, which passes data when the clock is low. Dynamic latches can be used for phase-1 (or static latches). Each *state variable* has a pair of latches, phase-1 and phase-2, forming an edge-triggered flipflop. Phase-2 latches are static. State variables are latched when the clock goes high.

The operation of the machine is illustrated by an example. Figure 3.5 shows a locally-clocked implementation for the simple burst-mode specification shown earlier in Figure 3.1. States A and B in the specification have been merged into a single machine state (q=0); similarly states C, D and E have been merged (q=1). Consider transitions $A \rightarrow B$ and $B \rightarrow C$.

Initially the circuit is quiescent, and the clock is low; the inputs, outputs and state variable are low as well, as shown in Figure 3.5. The machine is in *phase-1*. The phase-2 latch in Figure 3.5 is shaded, indicating that it is disabled. The phase-1 latches are transparent: data passes directly through them. Therefore no hazards are permitted on

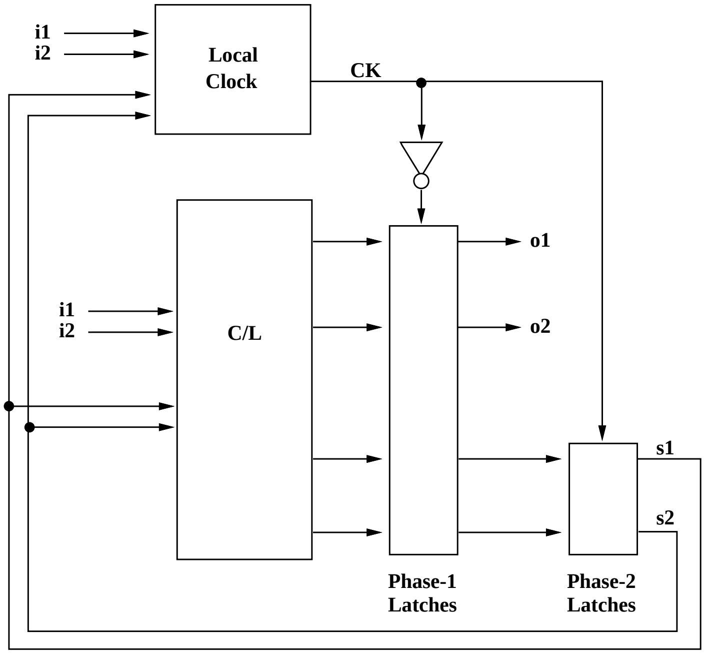


Figure 3.4: Detailed block diagram of locally-clocked asynchronous state machine.

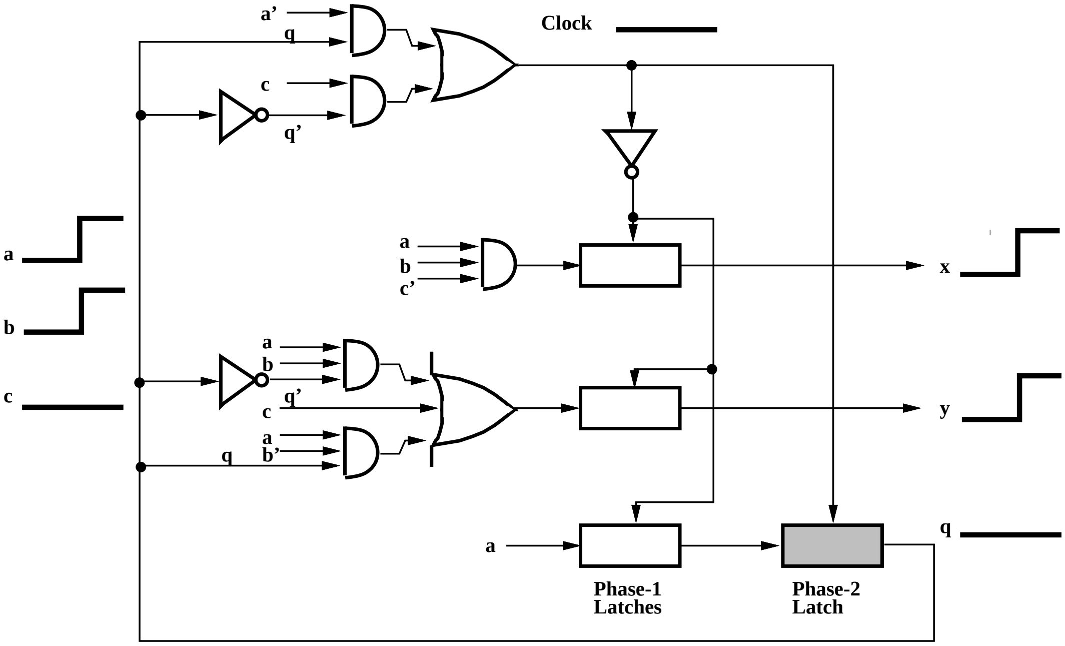


Figure 3.5: Simple example implementation: $A \rightarrow B$ transition (*Phase-1*)

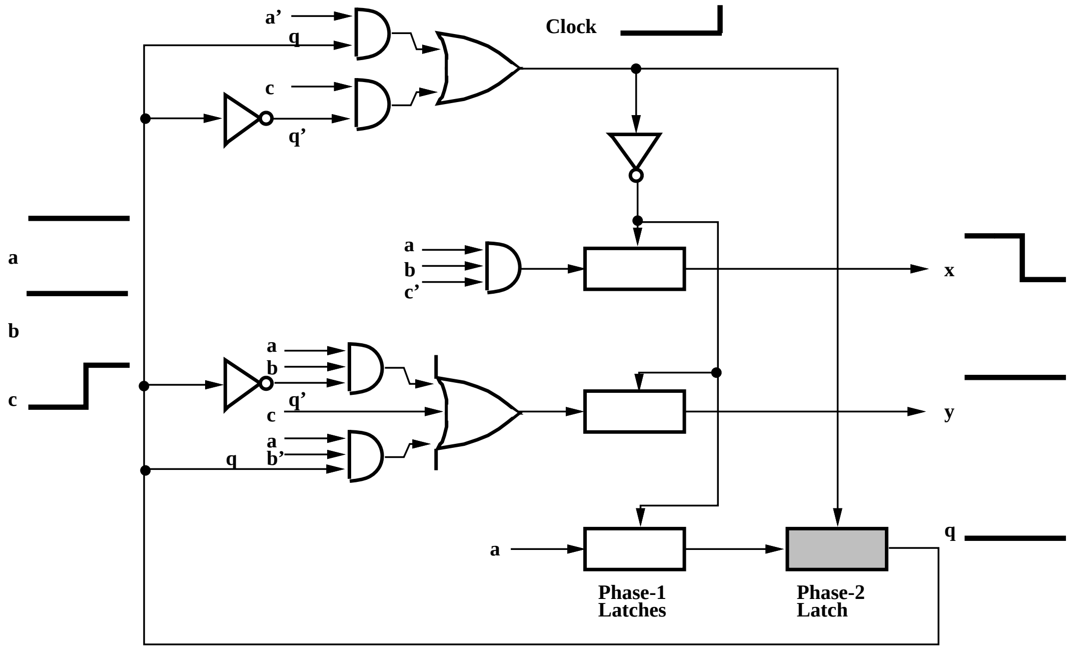


Figure 3.6: Simple example implementation: $B \rightarrow C$ transition (*Phase-1, initial*)

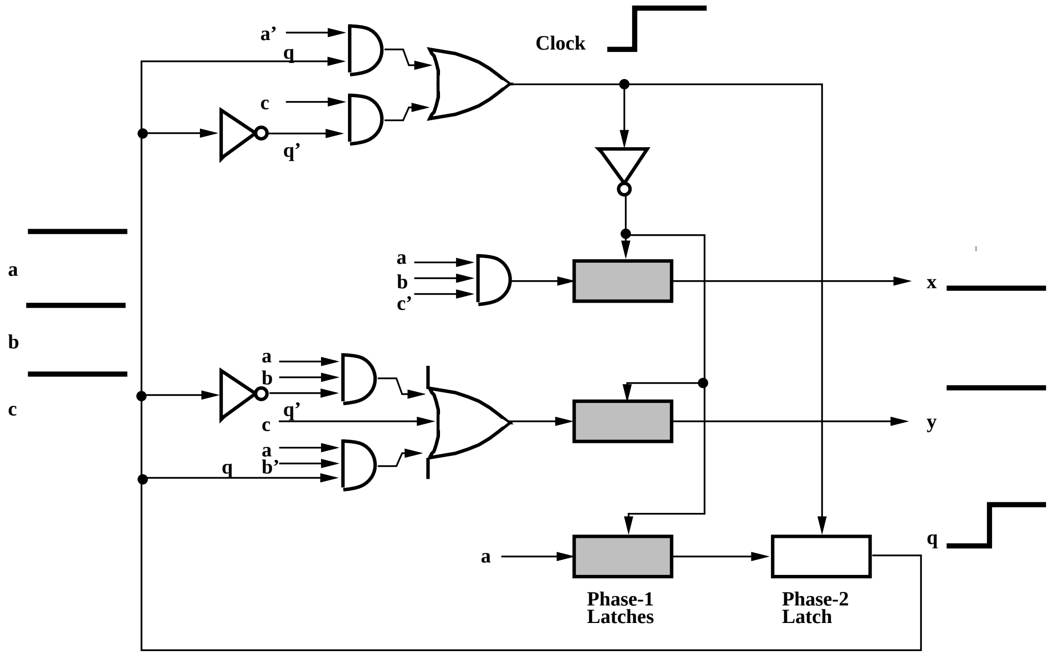


Figure 3.7: Simple example implementation: $B \rightarrow C$ transition (*Phase-2*)

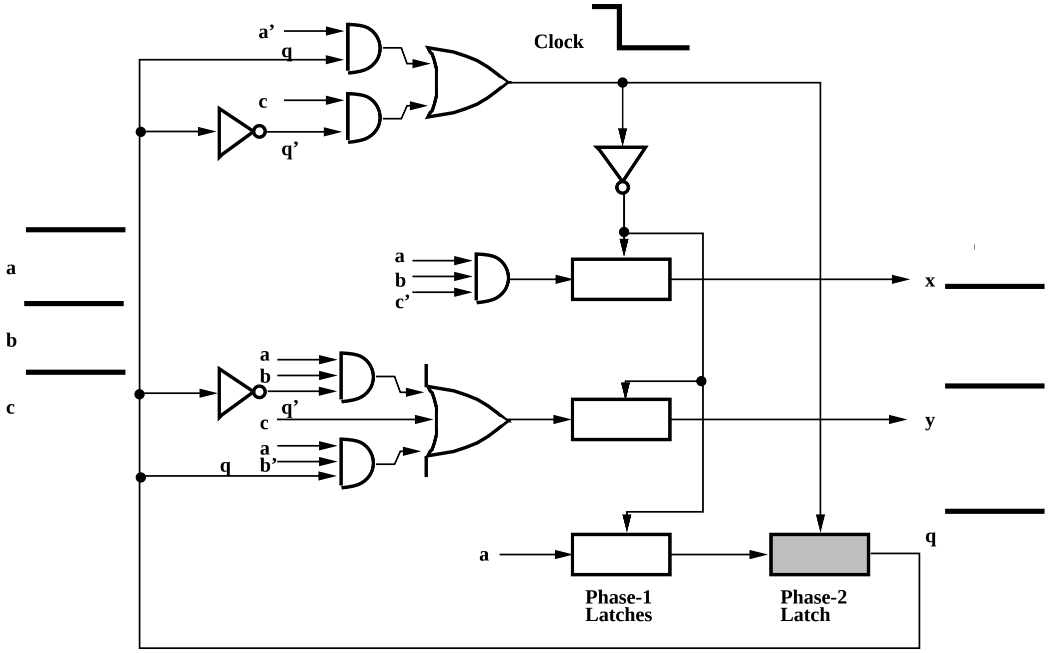


Figure 3.8: Simple example implementation: $B \rightarrow C$ transition (*Phase-1, final*)

the outputs during phase-1: as inputs a and b go high for the $A{\rightarrow}B$ transition, outputs x and y *must remain low* until the input burst is complete. At this point, both outputs go high monotonically. Since there is no state change for this transition, the clock remains low and the machine is ready to receive new inputs.

In the next transition, $B{\rightarrow}C$, input c goes high, as shown in Figure 3.6. The output latches are still transparent, so outputs cannot change until the input burst is complete. Once input c goes high, output x goes low monotonically. At this point, *a state change occurs* (see Figure 3.7): the state logic generates the correct next state (q=1), the clock goes high, the next state is latched on the rising edge of the clock and the phase-1 latches are disabled. (Note that outputs have already passed through the latches before the clock goes high.) The machine is in *phase-2*. No further changes pass through the phase-1 latches. Therefore, the output and state logic are permitted to glitch during phase-2. Once the output and state logic stabilizes in the new state, the clock logic is reset (see Figure 3.8), enabling the phase-1 latches and completing the cycle; the machine returns to phase-1.

Note that a clock pulse is generated only for transitions where the state changes, such as $B \rightarrow C$. Remaining transitions are unclocked even if output changes occur, such as $A \rightarrow B$ (unlike [21, 79]).[1] There are two benefits to our *selective clocking* approach: the clock implementation tends to be simpler, and some of the transitions will have no clock-cycle overhead.

Throughout most of this chapter we assume that outputs can change at the same time that the state changes, and no new inputs arrive until a clock cycle is complete and all logic is stable. That is, we assume a generalized *fundamental mode* of machine operation, where multiple-input changes are permitted [91]. In this case, the local clock never delays an output change, and there is no clock-cycle overhead (though it is still desirable to minimize clock area). However, we later consider cases where new inputs may arrive quickly, and it is necessary to delay either input or output changes to insure correct machine operation (see Section 3.6). In such cases, the *cycle time* of the local clock has direct effect on overall performance, and the benefits of selective clocking become more important.

3.4 Synthesis Method: Overview

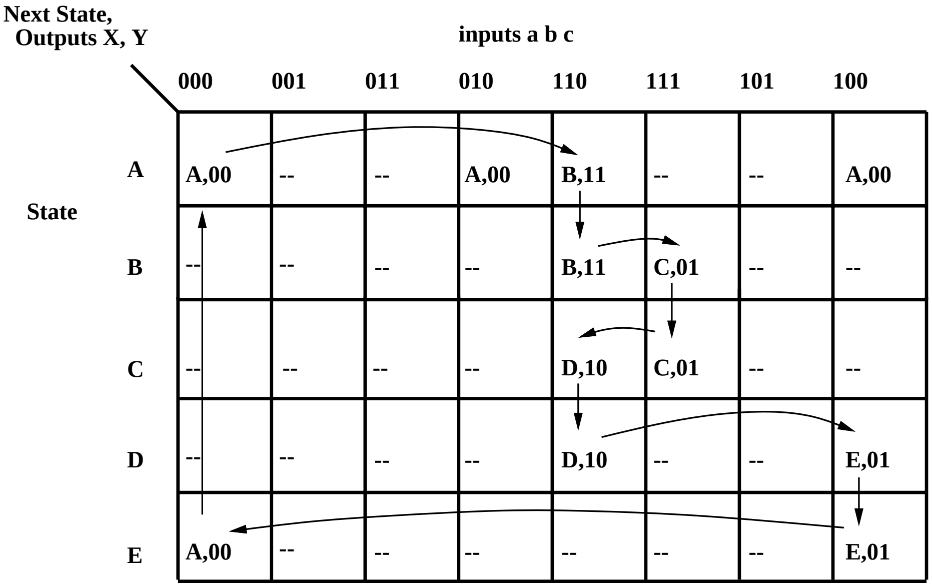
Now that the implementation style has been explained, the synthesis method can be presented in a top-down fashion, using the simple specification of Figure 3.1 as an illustration. This section is an overview of the synthesis of locally-clocked state machines. Later sections formalize the method described below.

3.4.1 Functional Synthesis

The first step in state-machine synthesis is to generate and reduce a *flow table* for the specification, assign state codes, and generate Boolean functions for the clock and each output and state variable. Questions of logic and timing implementation are deferred to later sections. The synthesis method at this stage is based on standard sequential synthesis techniques [60].

[1] The implementation essentially has a 2-phase clock which remains in phase-1 except when a state change is required.

The burst-mode specification is first transformed into an *unminimized* flow table, indicating output and next-state values. Each state in the original specification is mapped to a *unique row* in the table, having a unique, stable *entry point* (or input vector where the state is entered). An input burst begins at the entry point of some state; it terminates in an unstable entry which leads directly to a new stable state. If only single-input changes occur, the flow table is called a *primitive flow table* [91]. The term "primitive" indicates that the table is unminimized. In burst-mode specifications, however, multiple-input changes can occur, and so the table is called a *generalized primitive flow table.* [2] The generalized primitive flow table for the specification of Figure 3.1 is shown in Figure 3.9. Input bursts are indicated by horizontal arrows and state changes by vertical arrows. All reachable entries in the primitive flow table must have specified output and next-state values; remaining entries are don't-cares.

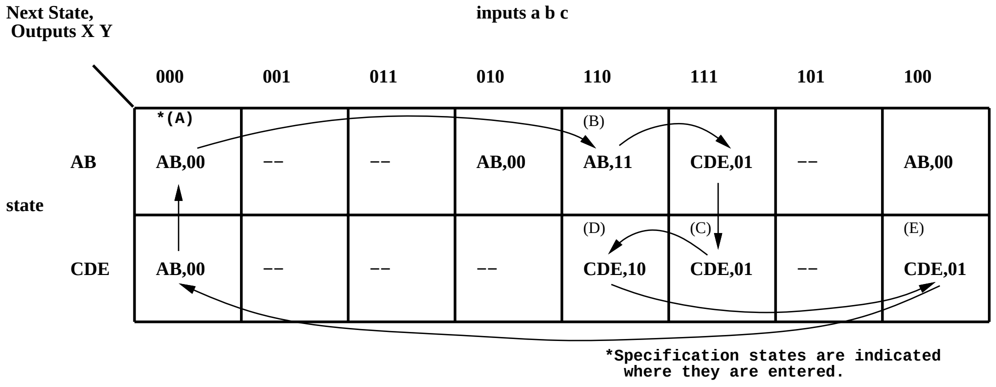


	000	001	011	010	110	111	101	100
A	A,00	--	--	A,00	B,11	--	--	A,00
B	--	--	--	--	B,11	C,01	--	--
C	--	--	--	--	D,10	C,01	--	--
D	--	--	--	--	D,10	--	--	E,01
E	A,00	--	--	--	--	--	--	E,01

Figure 3.9: Primitive flow table for simple example.

The flow table is reduced by merging states. The resulting minimized flow table is shown in Figure 3.10. States A and B of the primitive table are merged into state AB, and states C, D and E are merged into state CDE.

[2] Using the terminology decribed in Unger [91], the flow tables considered in this section are *single-output change* (*SOC*) tables in *standard form.*



Next State, Outputs X Y — inputs a b c

state	000	001	011	010	110	111	101	100
AB	*(A) AB,00	--	--	AB,00	(B) AB,11	CDE,01	--	AB,00
CDE	AB,00	--	--	--	(D) CDE,10	(C) CDE,01	--	(E) CDE,01

*Specification states are indicated where they are entered.

Figure 3.10: Reduced flow table for simple example.

Next, merged states are assigned arbitrary unique state codes. Since the reduced table has two states, only one Boolean state variable, q, is required. In the encoded flow table of Figure 3.11, state AB is assigned the Boolean code $q = 0$, and state CDE is assigned the code $q = 1$.[3]

Next State Q, Outputs X Y — inputs a b c

state q	000	001	011	010	110	111	101	100
0	0,00	--	--	0,00	0,11	1,01	--	0,00
1	0,00	--	--	--	1,10	1,01	--	1,01

Figure 3.11: Encoded flow table for simple example.

The final flow table is generated by augmenting the table with unassigned state codes, and filling in these added entries with don't-care values. In the given example, this step is not necessary since every possible state code for state variable q has already been assigned. Finally, next-state and and output functions are generated for each state variable and output, respectively. The clock function is generated as well. Since the implementation uses only D-latches, these functions are also "excitation functions" [60] for the D-latches. The Boolean functions for the given example are shown in the Karnaugh

[3] The state assignments used in this chapter are called *single-transition time* (STT) assignments [91], since an unstable state leads directly to a stable state without "multi-stepping" .

maps of Figure 3.12.

inputs a b c

Outputs X Y state q	000	001	011	010	110	111	101	100
0	(A)* 00	--	--	00	(B) 11	01	--	00
1	00	--	--	--	(D) 10	(C) 01	--	(E) 01

(a) Output Function

*Specification states are indicated when they are entered.

inputs a b c

Next-State Q state q	000	001	011	010	110	111	101	100
0	(A) --	--	--	--	(B) --	1	--	--
1	0	--	--	--	(D) --	(C) --	--	(E) --

(b) Next-state Function

inputs a b c

Clock state q	000	001	011	010	110	111	101	100
0	(A) 0	--	--	0	(B) 0	1	--	0
1	1	--	--	--	(D) 0	(C) 0	--	(E) 0

(c) Clock Function

Figure 3.12: Final Karnaugh maps for simple example.

In generating Karnaugh maps from the final flow table, don't-cares are added in different ways for the outputs, next-state and clock. These don't-cares reflect the actual structure and operation of the locally-clocked machine (see Figure 3.4). For the following discussion, recall that in phase-1, the machine receives an input burst, which corresponds to a horizontal (or row) transition in the flow table. In phase-2, the machine changes state, which corresponds to a vertical (or column) transition in the flow table.

- **Outputs.** Each output variable has a single attached latch. In phase-1, the output latches are transparent, so every Karnaugh map entry that can be reached during an input burst must be specified. In phase-2, the output latches are disabled, so every Karnaugh map entry that can be reached in the middle of a state change (*i.e.* transient state) is unspecified (*i.e.* is a don't-care). That is, outputs are only specified during phase-1.

- **Next-State.** Each next-state variable has a pair of latches forming an edge-triggered flipflop. These flipflops are clocked only when a state change occurs. Therefore, the next-state function is specified only at the *end point* of an input burst, and *only* if the input burst results in a state change. All remaining Karnaugh map entries in phase-1 (where the next-state is stable) and all transient entries in phase-2 (after the clock edge) are unspecified (*i.e.*, don't-cares). That is, only *unstable* flow table entries are specified for the next-state function.

- **Local Clock.** The local clock is unlatched. The clock is low during an input burst in phase-1. If the input burst results in a state change, the clock is driven high at the end of the input burst, and then is driven low by the state change. If there is no state change, the clock remains low. Since the clock is never disabled, every Karnaugh map entry that can be reached during a phase-1 input burst must be specified. However, every Karnaugh map entry that can be reached during a state change (*i.e.* transient state) will be *unspecified* (*i.e.* is a don't-care). *That is, we shall ignore the correct phase-2 specification of the clock: the clock is only specified during phase-1!* Since the clock is never disabled, these transient states may produce incorrect clock values if function hazards exist. This problem is addressed in the next subsection.

The transformation of a flow table into Karnaugh maps is illustrated in Figures 3.11 and 3.12. Every output function entry that is reachable during the input bursts of transitions A→B and B→C is specified in the Karnaugh map of Figure 3.12(a). In contrast, the next-state function has only a single specified entry for these transitions (Figure 3.12(b)). This entry ($abcq$=1110) is the point where the next state is latched by the local clock for transition B→C. This entry corresponds to an unstable state in the flow table. Note that entry $abcq$=1100, representing the completion of transition A→B, is unspecified, since the machine does not change state during this transition.

Each clock function entry which can be reached during the A→B transition is 0, since this transition does not require a state change (see Figure 3.12(c)). In contrast, the B→C transition *does* require a state change. For this transition, every stable (*i.e.*, non-final) entry is 0 (in this case, only *abcq*=1100), and the transition terminates in a single 1 entry (*abcq*=1110) where the state change begins. Once the state change is completed, the machine returns to a stable state (*abcq*=1111), and the clock logic is reset to 0.

3.4.2 Logic and Timing Requirements

Given the above functions for the clock, outputs and state variables, correct logic and timing requirements can now be added for their implementation. It is assumed that the machine is in a stable state, no new input burst has begun and all internal logic is stable. Any specified input burst may occur, with inputs changing at arbitrary times.

> **Requirement 1.** (*Phase-1*) The clock and output logic must be free of hazards for every permitted input burst in every state.

During phase-1, the phase-1 latches are transparent; therefore, every output must be free of glitches. Since the local clock is unlatched, it must be glitch-free as well.

> **Requirement 2.** (*Phase* 1 → 2*, Phase* 2 → 1) The minimum propagation delay through clock logic is greater than the maximum propagation delay through the logic for every output and state variable.

This requirement is a simple one-sided timing constraint to insure correct operation of the local clock.

> **Requirement 3.** (*Phase-2*) Once the clock is set, it must be reset without hazards.

During phase-2, output logic is permitted to glitch since the phase-1 latches are disabled. However, the local clock is unlatched, so it must be glitch-free.

> **Requirement 4.** (*Phase-2*) The delay between the enabling of the phase-2 latches and the disabling of the phase-1 latches must be less than the minimum delay in the feedback path.

This final requirement prevents race conditions through the phase-1 latches during phase-2. This requirement is needed because the machine generates a two-phase clock from a single clock source. As a result, the clock may have *overlapping* phases. In particular, in the transition from phase-1 to phase-2, the phase-2 latches may be enabled before the phase-1 latches have been disabled. To guarantee correct operation in this case, Requirement 4 insures sufficient delay on the feedback path to avoid races through the phase-1 latches.[4]

These requirements divide the implementation problem into two parts. Requirement 1 is concerned with the implementation of hazard-free combinational logic for the outputs and clock. It insures correct operation of the state machine in phase-1. Requirements 2 through 4, and other minor requirements discussed later, are concerned with the resetting of the clock and with timing constraints. These requirements guarantee the correct sequential operation of the machine in phase-2 and in the transitions between phase-1 and phase-2.

3.4.3 Hazard-Free Logic Implementation

A simple implementation style is presented below to satisfy Requirement 1. In this solution, the local clock and each output are implemented using AND-OR logic. Multi-level logic realizations and realizations which use other logic families (NAND, NOR, etc.) can be derived from these implementations using techniques discussed in chapter 4.

To satisfy Requirement 1, the outputs and clock must be free of hazards during each specified input burst. In an input burst, there are four possible transitions for any output — $0 \rightarrow 0, 0 \rightarrow 1, 1 \rightarrow 1, 1 \rightarrow 0$ — and two possible transitions for the local clock — $0 \rightarrow 0, 0 \rightarrow 1$. These transitions are guaranteed free of function hazards by the synthesis method (see Section 3.5.1). In fact, each transition is a "burst-mode transition": the function can change value only at the end point of the transition. Conditions to insure hazard-free logic for burst-mode transitions were described in Section 2.6. We review these conditions below.

[4] Alternatively, a two-phase *non-overlapping* clock could be used. This approach would simplify the timing requirements of the machine, but could result in a longer cycle time.

An AND-OR realization is hazard-free for all burst-mode $0 \to 0$ and $0 \to 1$ transitions. For each $1 \to 1$ transition, it is sufficient to have *at least one* product term which *covers* (remains 1 during) the transition cube ("required cube"). For the remaining transition type, $1 \to 0$, there must be *some* product covering each required cube (*i.e.*, maximal ON-set subcube) of the input transition. In addition, no product may intersect the transition cube unless it also intersects its start point.

The clock trivially satisfies these conditions, since it has only $0 \to 0$ and $0 \to 1$ transitions during input bursts. That is, *any sum-of-products realization of the local clock is permitted.* Note that we are *only* concerned here with the operation of the clock during phase-1. We address the phase-2 operation of the clock in Section 3.4.4, where we discuss Requirement 3.

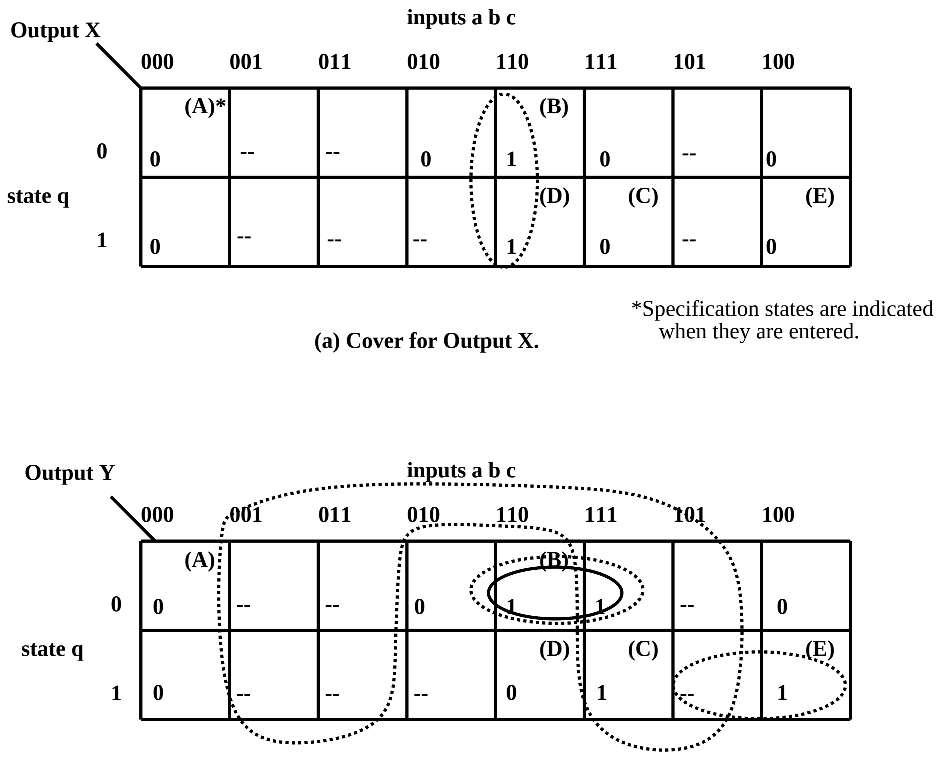
For the outputs to satisfy these conditions, we require products to cover certain cubes in the output functions and forbid certain other products. Covers which satisfy these conditions for the given example are shown in Figure 3.13. (Required cubes which contain more than one minterm are shown as well.)

As indicated earlier, there are cases where these conditions are unsatisfiable: given a Boolean function, it is not always possible to synthesize a hazard-free cover for a particular set of input transitions. In particular, products covering required cubes in $1 \to 1$ or $1 \to 0$ transitions may illegally intersect other $1 \to 0$ transitions. However, it can be shown that these conditions are *always* satisfiable for the output functions if there is no state minimization. That is, *it is always possible to synthesize a burst-mode specification by restricting state minimization.* We therefore add safe constraints on state merging to insure that the conditions can always be satisfied. (Cases where minimized tables cannot be covered correctly seem rare for small- and medium-sized specifications.)

3.4.4 Clock Reset and Timing Implementation

Requirements 2, 3 and 4 and other minor requirements are considered in Sections 3.5.2 and 3.5.3. Requirements 2 and 4 are simple one-sided timing constraints. They can always be satisfied by adding appropriate delay to the clock output and feedback path, if necessary.

Finally, to insure that the clock is hazard-free when it resets (Requirement 3) we



Output X — inputs a b c

state q	000	001	011	010	110	111	101	100
0	(A)* 0	--	--	0	(B) 1	0	--	0
1	0	--	--	--	(D) 1	(C) 0	--	(E) 0

*Specification states are indicated when they are entered.

(a) Cover for Output X.

Output Y — inputs a b c

state q	000	001	011	010	110	111	101	100
0	(A) 0	--	--	0	(B) 1	1	--	0
1	0	--	--	--	(D) 0	(C) 1	--	(E) 1

(b) Cover for Output Y.

Note: required cubes are indicated by solid lines, and cover is indicated by broken lines.

Figure 3.13: Hazard-free output covers for simple example.

forcibly reset the clock. To do this, circuitry is added: each feedback line is ANDed with the inverted clock to generate a new, *resettable feedback* input to the clock. With appropriate timing, this circuitry guarantees that the clock always resets without hazards.

3.4.5 Summary

To summarize the synthesis method, Karnaugh maps are constructed for the clock, state variables and outputs for a given burst-mode specification. There are no additional restrictions on state variables, so any implementation of the state variables is permitted. Any sum-of-products implementation of the clock is permitted, provided reset circuitry and correct delays are added. Any sum-of-products implementation of the outputs is correct, if all of the hazard-free covering conditions are met. (Additional minor timing requirements must be met as well.) These requirements can always be satisfied and are sufficient to insure a correct locally-clocked implementation.

3.5 Synthesis Method: Details and Formal Analysis

The synthesis method described in the previous section can now be formalized. Sufficient conditions are presented to insure a correct implementation, and it is shown that these conditions can always be satisfied. This section also includes details on the elimination of logic hazards, design of the clock reset logic, and sequential timing requirements.

The first subsection 3.5.1 formalizes the logic synthesis method described in Section 3.4. The goal of logic synthesis is the implementation of *combinational logic* for outputs, state variables and local clock which is correct for every transition in phase-1. The transformation of combinational behavior into correct *sequential* (phase-2) *behavior* is determined by the clock reset, the use of storage elements and the implementation of timing requirements, and will be discussed in Sections 3.5.2 and 3.5.3.

3.5.1 Logic Synthesis

There are five basic steps to state machine synthesis:

1. Generate a primitive flow table
2. State minimization
3. State assignment
4. Generate Boolean functions
5. Logic minimization

Steps 1–4 involve the functional synthesis of the outputs, state variables and local clock. Step 5 concerns the hazard-free logic implementation of these functions.

Functional Synthesis

The main result of this section is that, given any specified transition in a burst-mode specification, the final synthesized state, output and clock functions implement the desired behavior, and, moreover, the outputs and clock are free of function hazards.

For the following, recall that set $P = \{0, 1\}$ and $B = \{0, 1, *\}$, as defined in Chapter 2, where "*" represents a *don't-care*, or undefined, value.

A *flow table*, F, is a sextuple, $F = (S, I, O, s_0, f, n)$, where: S is a finite set of *states*; $I = \{x_1, \ldots, x_m\}$ is the set of *inputs* (*i.e.,* binary input variables); $O = \{z_1, \ldots, z_n\}$ is the set of *outputs* (*i.e.,* binary output variables); $s_0 \in S$ is the unique *start state*; $f : S \times P^m \rightarrow B^n$ is the *output function*; and $n : S \times P^m \rightarrow S \cup \{*\}$ is the *next-state function*. The value of an output z_j in state s and input value $x \in P^m$ is denoted by $f_j(s, x)$. Given a state $s \in S$ and input value $x \in P^m$, functions f and n are said to be *undefined* at (s, x) if $f_j(s, x) = *$ for every $j \in [1..n]$ and if $n(s, x) = *$.

Each pair $(s, x) \in S \times P^m$ is called a *total state*. A total state describes the input and current state of a state machine. Given flow table F and total state (s, x) such that $n(s, x)$ is defined, (s, x) is a *stable state* if $n(s, x) = s$ and it is an *unstable state* if $n(s, x) \neq s$.

A flow table $F' = (S', I, O, s'_0, f', n')$ is a *reduced flow table* of F if there exists a mapping, $h : S \rightarrow S'$, such that $s'_0 = h(s_0)$, and for every state $s \in S$ and input value $x \in P^m$:

1. $f_j(s, x) = f'_j(h(s), x)$, whenever $f_j(s, x)$ is defined; and

2. $h(n(s,x)) = n'(h(s),x)$, whenever $n(s,x)$ is defined.

Two distinct states, $s, t \in S$, are said to be *merged* by mapping h if and only if h maps both to the same state of F', that is, $h(s) = h(t)$. Condition 1 insures that the output behavior of every state $s \in S$ is preserved after state merger. Condition 2 insures that the next-state behavior of every state $s \in S$ after merger is "consistent" with the behavior before merging. (The above definition considers only mappings which produce *partitions* of the original states. *State splitting* is considered in a later section.)

A *k-variable state assignment* for a flow table $F = (S, I, O, s_0, f, n)$ is an isomorphism $g : S \leftrightarrow H$, where $H \subseteq P^k$ is the set of *assigned (Boolean) state codes*, for some positive integer k. Flow table $F' = (H, I, O, s_0', f', n')$ is an *encoded flow table* for flow table F under state assignment g if F' is a reduced flow table for F under g. Functions f' and n' are (partial) combinational functions of k state variables and m inputs, and are represented by the *output table* and *transition table*, respectively, of F' under state assignment g.

The set of states of an encoded flow table can be augmented to include unassigned state codes. Flow table $F' = (A, I, O, s_0, f', n')$ is called the *augmented (encoded) flow table* of encoded flow table $F = (H, I, O, s_0, f, n)$, if $A \equiv P^k$, and for every state $s \in A$ and input value $x \in P^m$:

$$f_j'(s,x) = \begin{cases} f_j(s,x) & \text{if } s \in H, \\ *, & \text{otherwise;} \end{cases}$$

and

$$n'(s,x) = \begin{cases} n(s,x) & \text{if } s \in H, \\ *, & \text{otherwise.} \end{cases}$$

f' is called the *output function* and n' is called the (Boolean) *next-state* or *transition function*. (Functions f' and n' are represented as *Karnaugh maps*.) The injective identity function $Z : H \to A$ defines a mapping from states of F to equivalent states of F'.

For the next definition, recall that the transition cube $[x, y]$ from input state x to input state y contains all input states that can be reached in a transition from x to y. Also, recall that the open transition cube $[x, y)$ is equivalent to $[x, y] - \{y\}$.

Given a burst-mode specification $G = (V, E, I, O, v_0, in, out)$, a flow table $F = (V, I, O, v_0, f, n)$ is called a *primitive flow table for G* if the following holds for every edge $(u, v) \in E$:

1. $f(u, x) = out(u)$, for every $x \in [in(u), in(v))$;
2. $f(u, v) = out(v)$;
3. $n(u, x)) = u$, for every $x \in [in(u), in(v))$;
4. $n(u, in(v)) = v$.

(Functions f and n are otherwise undefined.) These conditions simply insure that the primitive flow table is "filled in" as described in Section 3.4.1.

Lemma 3.1. There exists a primitive flow table for every burst-mode specification $G = (V, E, I, O, v_0, in, out)$.

Proof. The only reason that G might not have a primitive flow table is if f and n, as defined in Conditions 1–4, were not functions. This could occur only if f or n were assigned two different values for the same input and state. In such a case, G must have two distinct edges, (u, v) and (u, w), where $in(w) \in [in(u), in(v)]$. However, as shown in Proposition 3.1 in Section 3.2, the "maximal set property" of burst-mode specifications insures that this will never occur. □

The functional synthesis method of Section 3.4.1 can now be formalized. Let $G = (V, E, I, O, v_0, in, out)$ be any burst-mode specification, $F = (V, I, O, v_0, f, n)$ be the primitive flow table for G, $F' = (S', I, O, s'_0, f', n')$ be any reduced flow table for F under mapping $h : V \rightarrow S'$, $F'' = (H, I, O, s''_0, f'', n'')$ be any encoded flow table for F' under k-variable state assignment $g : S' \leftrightarrow H$, where $H \subseteq P^k$, and let $F''' = (A, I, O, s''_0, f''', n''')$ be the augmented flow table for F'', where $A \equiv P^k$. F''' is called a *synthesized flow table* for G. Flow tables F, F', F'' and F''' represent the results of Steps 1, 2, 3 and 4, respectively, of the synthesis method.

The synthesis method defines a mapping $k \equiv Z \circ g \circ h : V \rightarrow A$ from states of the burst-mode specification G to encoded states of flow table F''', where Z is the injective mapping from the encoded states of H to equivalent states in A. For each *state* $u \in V$

of the specification G, there *corresponds* a state $s = k(u) \in A$ of synthesized flow table F'''. Similarly, for each specified transition $(u, v) \in E$ of G, there *corresponds* an input transition from $in(u)$ to $in(v)$ in state s of F'''.

For the following lemmas, it is assumed that $G = (V, E, I, O, v_0, in, out)$ is any burst-mode specification, $F''' = (A, I, O, s_0'', f''', n''')$ is any synthesized flow table for G, and $k : V \to A$ is the resulting mapping from states of G to states of F'''.

Lemma 3.2 indicates that the output behavior described by a burst-mode specification for a transition (u, v) is preserved under functional synthesis. Lemma 3.3 indicates that the resulting next-state function is stable in a state corresponding to u until the transition is complete, and then changes to a state corresponding to v.

Lemma 3.2. For each transition (u, v) in G, the corresponding input transition for f''' is free of function hazards. In particular, $[(s, x), (s, y)]$ is a burst-mode transition for function f''', where $s = k(u), x = in(u), y = in(v)$, and:

- $f'''(s, w) = out(u)$, for every $w \in [x, y)$; and
- $f'''(s, y) = out(v)$.

Lemma 3.3. For each transition (u, v) in G, the corresponding input transition for n''' is free of function hazards. In particular, $[(s, x), (s, y)]$ is a burst-mode transition for function n''', where $s = k(u), t = k(v), x = in(u), y = in(v)$, and:

- $n'''(s, w) = s$, for every $w \in [x, y)$; and
- $n'''(s, y) = t$.

Proof. These results follow immediately from the definitions of burst-mode specification, primitive flow table, reduced flow table, encoded flow table, synthesized flow table and corresponding input transition. □

A useful corollary of Lemma 3.3 is that every input transition begins in a stable state. This is an immediate consequence of Lemma 3.3 and the requirement on burst-mode specifications, in Section 3.2, that every input burst must be non-empty:

Corollary 3.1. For each transition (u, v) in G, the corresponding input transition for n''' begins in a stable state. In particular, $n'''(s, x) = s$, where $s = k(u)$ and $x = in(u)$.

The next-state function n''', characterized in Lemma 3.3 describes the desired next-state behavior of the locally-clocked machine. However, this behavior will be implemented by the combined use of a local clock, next-state logic and latches. The final part of Step 4 is therefore to transform the next-state function, n''', into the final clock function, c_{lc}, and next-state function, n_{lc}, used in locally-clocked synthesis. For every state $s \in A$ and input value $x \in P^m$:

$$c_{lc}(s,x) = \begin{cases} * & \text{if } n'''(s,x) = *, \\ 0 & \text{if } n'''(s,x) = s, \\ 1, & \text{otherwise;} \end{cases}$$

and

$$n_{lc}(s,x) = \begin{cases} * & \text{if } n'''(s,x) = *, \\ * & \text{if } n'''(s,x) = s, \\ n'''(s,x), & \text{otherwise.} \end{cases}$$

Functions c_{lc} and n_{lc} formalize the desired behavior of the clock and next-state logic described in Section 3.4.1. The clock remains at 0 unless a state change is required. When a state change is required, the clock goes to 1. Because the rising edge of the local clock controls the next-state latches, the next-state function is only specified when the clock goes to 1. Corollaries 3.2 and 3.3 are immediate from Lemmas 3.2 and 3.3 and the above definitions.

Corollary 3.2. For each transition (u,v) in G, the corresponding input transition for the final clock function c_{lc} is free of function hazards. In particular, $[(s,x),(s,y)]$ is a burst-mode transition for c_{lc}, where $s = k(u), t = k(v), x = in(u), y = in(v)$, and:

- $c_{lc}(s,w) = 0$, for every $w \in [x,y)$;

- $c_{lc}(s,y) = 0$, if $s = t$; and

- $c_{lc}(s,y) = 1$, if $s \neq t$.

Corollary 3.3. For each transition (u,v) in G, the final next-state function $n_{lc}(s,y) = t$ if $s \neq t$, where $s = k(u), t = k(v)$, and $y = in(v)$.

Corollary 3.4 is immediate from Corollaries 3.1 and 3.2, and indicates that the clock is always 0 at the start of an input transition:

Corollary 3.4. For each transition (u, v) in G, the clock function $c_{lc}(s, x) = 0$, where $s = k(u)$ and $x = in(u)$.

Hazard-Free Logic Implementation

The synthesis method described above is adequate to insure the desired output, clock and state functions for every input burst that can occur in phase-1, but does not guarantee that the output logic can be implemented without logic hazards.

This section has three main results. First, using the synthesis method defined above, it is shown that it is *not always possible* to avoid logic hazards on outputs in a sum-of-products implementation. Second, using the same synthesis method but with *no* state reduction, it is *always* possible to avoid output logic hazards. And, finally, sufficient conditions on state merging are given to insure that output logic hazards can always be avoided.

The section concludes by discussing the clock implementation. No constraints on state merging are required to avoid logic hazards in the clock. In fact, it will be shown that any sum-of-products implementation of the clock function is free of logic hazards for each specified input burst.

Example. Figure 3.14 is a burst-mode specification having 6 states, 3 inputs (a, b, c) and 2 outputs (y, z). Initially, $abc = 000$ and $yz = 01$. The unminimized, or primitive, flow table is shown in Figure 3.15. Using the previous definitions, a minimum reduced flow table can be derived having a partition of states: $\{(AD), (C), (BEF)\}$.

Figures 3.16 and 3.17 illustrate the problem. Figure 3.16 shows two states, A and D, of the primitive flow table of Figure 3.15. The input transitions corresponding to specified transitions $A \rightarrow B$, $A \rightarrow C$ and $D \rightarrow E$ are also shown. For simplicity, only output z is shown.

Output z has a $1{\rightarrow}0$ transition in state A during input burst $a{+}b{+}$, and it has a $1{\rightarrow}1$ transition in state D during input burst $c+$. The start point of each input transition is indicated with an asterisk in Figure 3.16. Each start point is the entry point of a state in the primitive flow table. The $1{\rightarrow}0$ transition in state A has two required cubes: one covers the $a+$ input change and the other covers the $b+$ change. The $1 \rightarrow 1$ transition

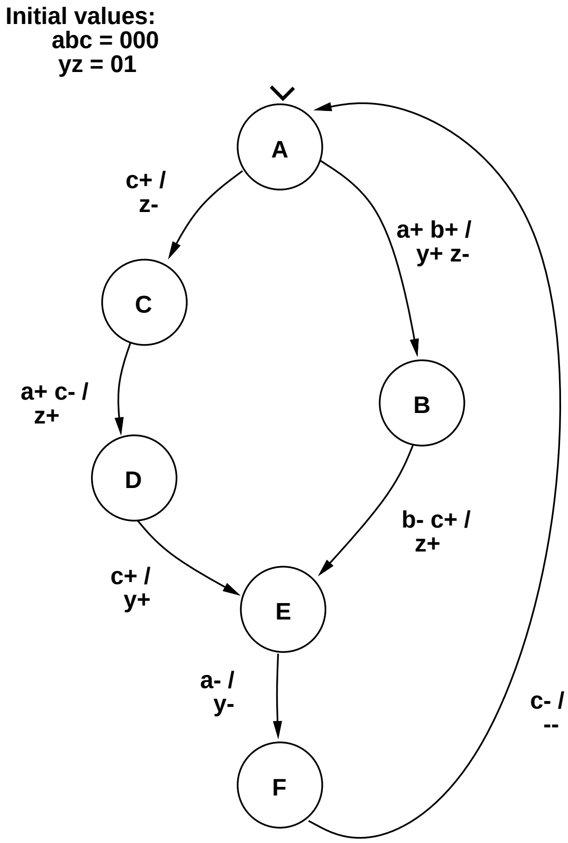
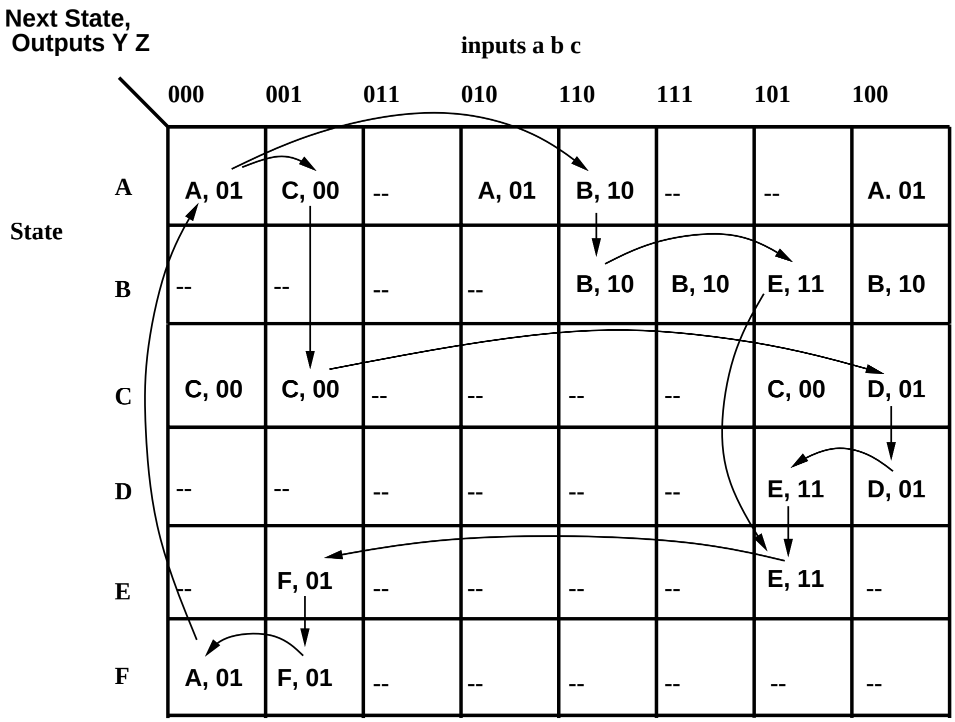


Figure 3.14: Example burst-mode specification.

in state D has one required cube, covering the entire transition. To avoid logic hazards, each required cube must be contained in a product of the cover. In addition, no product may intersect the 1→0 transition cube unless it also contains the cube's start point.

Before state minimization, the two transitions can be covered with hazard-free logic. In fact, from Figure 3.16, *the required cubes themselves* form a hazard-free cover for states A and D and can be used in the final cover, *regardless* of the final state assignment.

However, suppose states A and D are merged, as shown in Figure 3.17. In this case, *no cover exists* which satisfies the hazard-free covering conditions, *regardless* of the final state assignment. In particular, any product which contains the required cube for the transition of state D will illegally intersect the 1→0 transition of state A. As a result, every sum-of-products implementation produced from the given state partition will have a logic hazard for output z for either the 1→0 transition of state A or the 1→1 transition



Next State, Outputs Y Z

State \ inputs a b c	000	001	011	010	110	111	101	100
A	A, 01	C, 00	--	A, 01	B, 10	--	--	A. 01
B	--	--	--	--	B, 10	B, 10	E, 11	B, 10
C	C, 00	C, 00	--	--	--	--	C, 00	D, 01
D	--	--	--	--	--	--	E, 11	D, 01
E	--	F, 01	--	--	--	--	E, 11	--
F	A, 01	F, 01	--	--	--	--	--	--

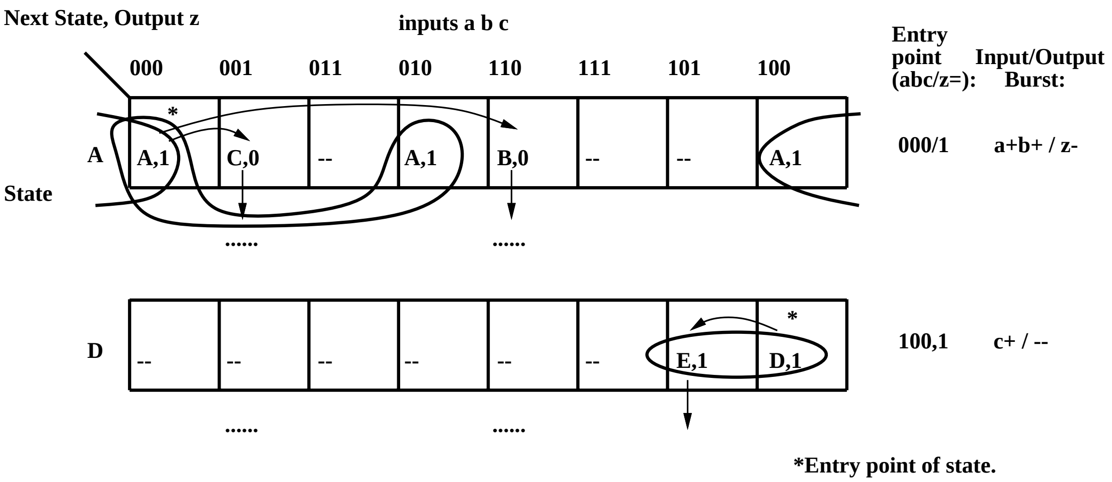
Figure 3.15: Example primitive flow table.

of state D. Therefore, *states A and D should not have been merged.* □.

In this example, output functions generated from the *unminimized* flow table had hazard-free covers. In fact, the next theorem demonstrates that this is always the case. This result holds regardless of the final state assignment. In particular, the *set of required cubes* for such an output function always forms a hazard-free cover. The theorem therefore is a constructive proof of a hazard-free implementation.

Theorem 3.1. Let G be any burst-mode specification, let z be any output variable of G, let F be an unminimized flow table synthesized from G using an arbitrary state assignment, and let f_z be the output function for z in table F. Then the set of required cubes for f_z is a hazard-free cover.

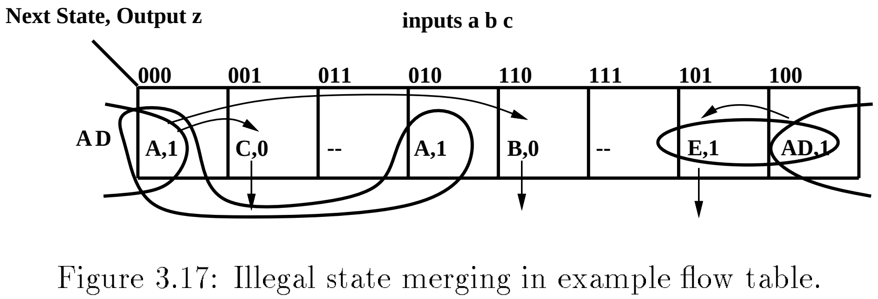
Proof. From Section 2.5, the set of required cubes forms a cover of function f_z. It must be shown that the the cover is hazard-free for every given transition. Function f_z is free of function hazards for every transition by Lemma 3.2. The cover is free of logic hazards for each $0 \to 0$ transition by Lemma 2.1. It is free of logic hazards for a $1 \to 1$ transition, by Lemma 2.2, since the required cubes themselves form the cover. It is



State	000	001	011	010	110	111	101	100	Entry point (abc/z=):	Input/Output Burst:
A	A,1	C,0	--	A,1	B,0	--	--	A,1	000/1	a+b+ / z-
D	--	--	--	--	--	--	E,1	D,1	100,1	c+ / --

*Entry point of state.

Figure 3.16: Fragment of example flow table.



	000	001	011	010	110	111	101	100
AD	A,1	C,0	--	A,1	B,0	--	E,1	AD,1

Figure 3.17: Illegal state merging in example flow table.

also free of logic hazards for a 0 → 1 transition, by Lemma 2.4, since only burst-mode transitions are used. For a 1 → 0 transition, the condition of Corollary 2.1 is satisfied: each required cube of the transition is contained in some cube of the cover, since the cover consists of the required cubes.

Finally, it must be shown that Lemma 2.3 is satisfied, so that there are no illegal intersections. Suppose some product, r, in the cover intersects a 1→0 transition cube, c, where c corresponds to some transition $s \rightarrow t$ in the specification. Since r is a required cube, it must be contained in a transition cube, c', which also corresponds to a transition in the specification. By construction, each transition cube is contained within a single state of the final table, F. In fact, since $r \subseteq c'$ intersects transition cube c, c and c' belong to the *same state* in F. Since states have not been merged, c and c' must therefore both correspond to the same specification state, s. That is, c' corresponds to some transition $s \rightarrow u$ in the specification. As a result, transition cubes c and c' have the same start point in function f_z. Since transition cube c describes a $1 \rightarrow 0$ transition, c' must describe either a $1 \rightarrow 0$ or a $1 \rightarrow 1$ transition. In either case, required cube r contains the common start point of c and c', so the intersection of r and c must be legal. That is, every transition is free of hazards in the cover. □

The goal of restrictions on state merging is to insure that the conditions for hazard-free logic continue to be met even after states are merged. It is possible to add simple, safe constraints on state merging to guarantee that some hazard-free cover exists for the final output functions after state minimization. In particular, we focus on the cover consisting only of required cubes *after* state merger. Cubes which are contained in other cubes are deleted. The resulting cover is a canonical sum-of-products implementation, which we call a **primitive cover**. Safe constraints are added to insure that the primitive cover resulting after state reduction will be hazard-free.

Given a transition τ from specification state p to s, define $start(\tau) = in(p)$ as the input value at the entry point of state p, and $cube(\tau) = [in(p), in(s)]$ as the set of all input values that can occur during τ.

Restrictions on State Merging to Satisfy Covering Conditions

Restriction 1. Let p and q be any two specified states, let z be any output, let τ_1 be any specified transition of state p and let τ_2 be any specified transition of state q. If output z has a $1 \rightarrow 0$ transition in τ_1 and a $1 \rightarrow 1$ transition in τ_2, where (a) $\mathrm{cube}(\tau_1) \cap \mathrm{cube}(\tau_2) \neq \phi$, and (b) $\mathrm{start}(\tau_1) \notin \mathrm{cube}(\tau_2)$, then p **and** q **cannot be merged**.

Restriction 2. If output z has a $1 \rightarrow 0$ transition in τ_1 and a $1 \rightarrow 0$ transition in τ_2, where (a) $\mathrm{start}(\tau_1) \neq \mathrm{start}(\tau_2)$, (b) $\mathrm{cube}(\tau_1) \not\subseteq \mathrm{cube}(\tau_2)$, and (c) $\mathrm{cube}(\tau_2) \not\subseteq \mathrm{cube}(\tau_1)$, then p **and** q **cannot be merged.** □

The first condition insures that the required cube which covers the $1 \rightarrow 1$ transition of z will not illegally intersect cubes used to cover the $1 \rightarrow 0$ transition, regardless of the final state merger. By the second condition, if two $1 \rightarrow 0$ transition subcubes have the same start point then, after state merger, their required cubes will not intersect illegally. Also, if a $1 \rightarrow 0$ transition subcube is contained in another $1 \rightarrow 0$ transition subcube, the required cubes which are used to cover the enclosing transition will also cover the enclosed transition correctly and without hazards after state minimization. In all other cases, we conservatively avoid merger of states. These conditions are incorporated into the state minimization algorithms described in Section 3.8.1. More sophisticated constraints can be developed to permit state merger for additional cases.

We conclude by discussing the hazard-free implementation of the clock logic. Unlike the outputs, there are only two possible transitions of the clock during an input burst in phase-1: $0 \rightarrow 0$ and $0 \rightarrow 1$. The following lemma indicates that *any* sum-of-products implementation of the clock is free of logic hazards.

Lemma 3.4. Let G be any burst-mode specification, let F be any synthesized flow table for G with clock function c_{lc}, and let C be any sum-of-products implementation of function c_{lc}. For each transition (u, v) in G, the corresponding input transition in F is free of logic hazards for cover C.

Proof. By Corollary 3.2, each specified transition (u, v) corresponds to a burst-mode $0 \rightarrow 0$ or $0 \rightarrow 1$ transition of function c_{lc}, which is free of function hazards. By Lemmas 2.1 and 2.4 of the previous chapter, any sum-of-products implementation is free of logic hazards for such a burst-mode transition. Therefore, C is free of logic hazards. □

3.5.2 Clock Reset Implementation

The previous section discussed the synthesis of combinational logic for the outputs, next-state and clock having the desired phase-1 behavior. This section is concerned with an important component of phase-2 behavior: the clock reset structure.

In Section 3.4.2, Requirement 3 states that, once the clock goes high, it must be reset without hazards. The synthesis method described above makes no attempt to eliminate hazards in the clock during state changes. This section proposes a simple solution to avoid such hazards: the use of clock reset circuitry with appropriate timing requirements.

To understand the problem, consider the role of the clock. Section 3.5.1 insured that outputs are hazard-free in phase-1. The clock eliminates the remaining hazards. In phase-2, outputs are permitted to glitch, since the clock disables the output latches. Since edge-triggered flipflops are used for the state, hazard-free state logic is not required. Because of the simplicity of this approach, arbitrary state assignment is used: there is no need for critical-race free codes.

Unfortunately, this approach does not eliminate the problem of hazard-free logic. Instead, the problem has now been pushed from the output and next-state logic into the clock design itself! In particular, *the clock must be hazard-free at all times*. Section 3.5.1 insured that the clock is hazard-free during phase-1. However, after state assignment, the clock may have function or logic hazards during phase-2 . That is, the clock may glitch during phase-2, allowing incorrect values to pass through the latches.

This problem has a simple solution. Instead of attempting to eliminate function and logic hazards for the clock during each possible state change, the clock can be regarded as a form of ring oscillator: once the clock goes high, its output is inverted and fed back as an input which *forcibly resets* it. To do this, additional circuitry is added: each feedback line is ANDed with the inverted clock to generate a new, *resettable feedback* input to the clock. The resulting reset structure is shown in Figure 3.18. Using this structure, it is shown below that the clock can be reset without hazards.

The timing diagram of Figure 3.19 illustrates the resulting hazard-free behavior of the clock using reset logic. Figure 3.20 illustrates the clock operation in detail. To simplify the figure, the inputs and part of the clock logic are omitted. The figure illustrates a

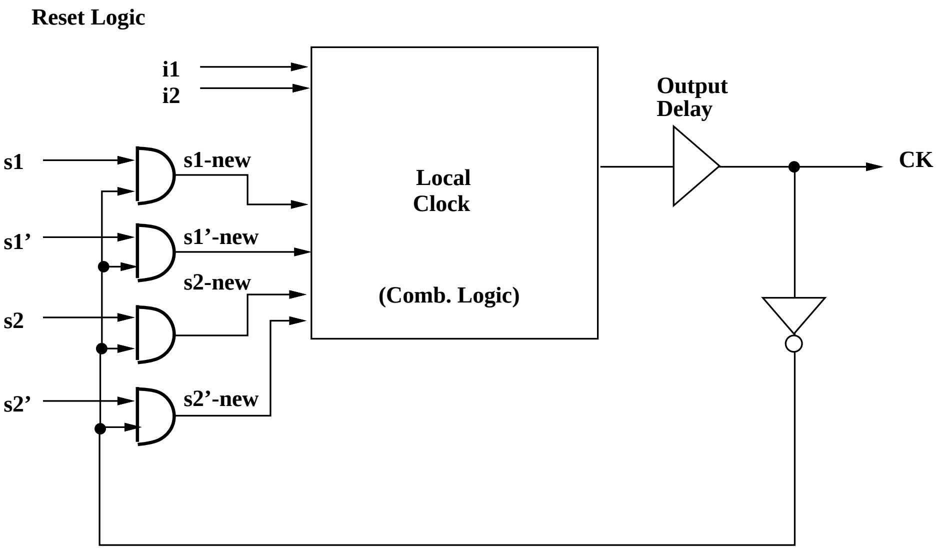


Figure 3.18: Block diagram of clock with attached reset-logic and output delay.

state change from state $s_1s_2 = 10$ to $s_1s_2 = 11$.

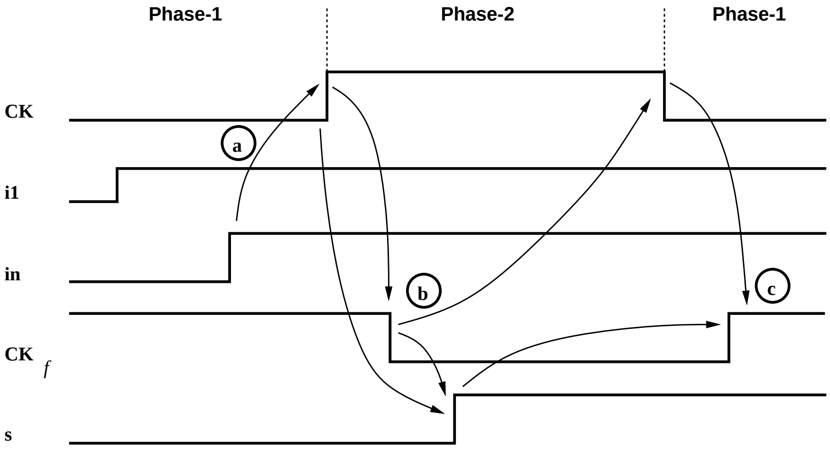
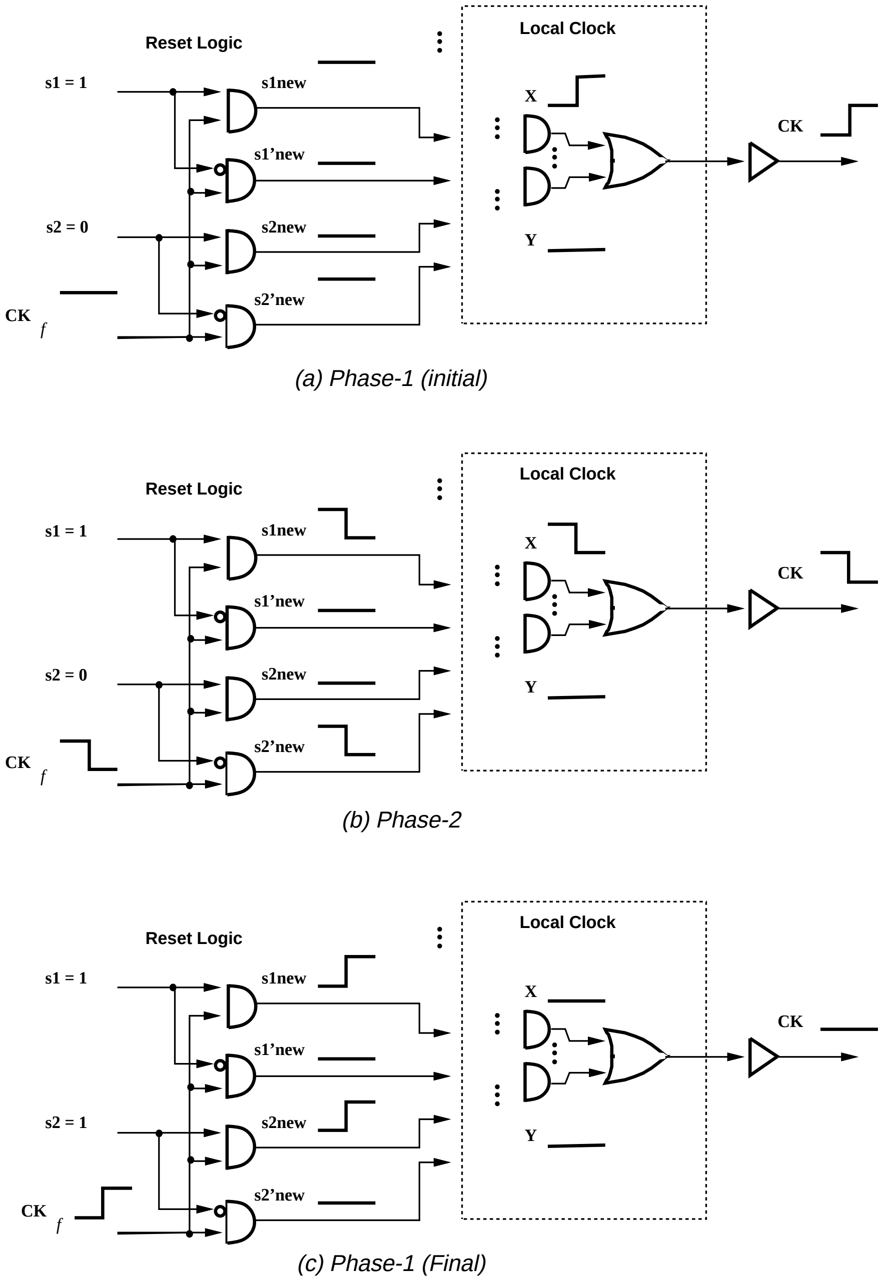


Figure 3.19: Timing diagram for clock operation using clock reset logic.

Initially, in phase-1, the clock output CK is low and the inverted clock CK_f is high, as shown in Figure 3.20(a). For each state variable, s_i, the reset logic generates two values: $s_{i_{new}}$ and $s_{i'_{new}}$. These dual-rail *state literals* are the uncomplemented and complemented signals, respectively, for state variable s_i. These signals are used as inputs to the clock logic, replacing the original state variable.

In phase-1, at some point, an input burst arrives and the clock is driven high (if the



(a) Phase-1 (initial)

(b) Phase-2

(c) Phase-1 (Final)

Figure 3.20: Hazard-free clock operation using clock reset logic.

input burst results in a state change). The synthesis method of the previous section insures that the clock is free of function and logic hazards during such a phase-1 transition. Since the clock goes high, at least one product, X, will go high. Other products, such as Y, may remain low. This transition corresponds to transition (a) in Figure 3.19.

Figure 3.20(b) shows the clock in phase-2. This transition corresponds to transition (b) in the timing diagram of Figure 3.19. After the clock, CK, is driven high, two events occur in parallel. First, the inverted clock, CK_f, is driven low. At the same time, the clock enables the phase-2 latches, allowing an internal state change from $s_1s_2 = 10$ to $s_1s_2 = 11$. To insure correct resetting of the clock, there is a timing constraint: CK_f must go low *before* the fed-back state variables change value. Under this constraint, since each AND-gate of the reset logic receives a 0 input (CK_f) before the fed-back state variables change, each state literal is driven low without hazards. Changes in the feedback variables may arrive later; they will be masked out by the disabled reset logic, so the state literals will not glitch.

It can easily be seen that, by resetting the state literals, each clock product is driven or held low without hazards. Consider any product of the clock, such as X or Y. Product X is high in the current state $s_1s_2 = 10$. At the completion of the state change, the synthesis method insures that the clock function is 0. Therefore, product X must be low in the destination state $s_1s_2 = 11$. Therefore, at least one input to product X must be a state literal (*e.g.*, $s_{2'_{new}}$), otherwise X could not have been reset by the state change. Alternatively, consider product Y, which is low in the current state. For some other transition, Y must be driven high.[5] Therefore, by similar reasoning, Y must contain some state literal as well. In summary, *every clock product contains at least one state literal.* Since each state literal is driven low, every product is driven or held low in phase-2. Since no other inputs go high during phase-2, the clock resets without hazards.

Figure 3.20(c) shows the clock after it returns to phase-1. This transition corresponds to transition (c) in Figure 3.19. Again, two events occur in parallel. First, after the clock is driven low, the inverted clock, CK_f, is driven high. At the same time, the internal state

[5] If Y is never driven high during an input burst, Y covers only don't-cares of the clock function, and can be removed from the cover. However, even if Y is included, the clock will still reset correctly. In this case, Y can only be driven high during a state change. At the start and end points of the state change, it will be low; it will go high at a transient point. Therefore, Y is set and reset by a state change, so it must contain a state literal.

may still be changing. To insure correct operation, there is a second timing constraint: the state change must be completed *before* CK_f goes high. Each state literal is then driven to its new value before the reset is enabled again.

It can easily be seen that the clock remains low without hazards. The synthesis method insures that the clock function is low at the completion of the state change. Therefore, each clock product must have some literal which is low in this state. If this literal is a primary input, the product is held low regardless of changes to the state literals, since a primary input may not change during phase-2; it must therefore have been low at the end of phase-1. If the low input is a state literal, it will remain low even after CK_f goes high, since the reset logic AND-gate will not go high. Therefore, every clock product remains low, and so the clock remains low without hazards. In conclusion, the clock is hazard-free throughout the entire machine cycle.

To insure that the clock reset operates correctly, three timing requirements are necessary. Additional timing requirements to insure correct machine operation are discussed in the next section. For the following definitions, refer to Figure 3.21. Let

- d_{ckcl} be the delay of the combinational logic for the clock;
- d_{ckout} be any additional delay at the clock output;
- d_{ck2} be the delay of the feedback path of the clock, including reset logic;
- d_{en2} be the delay from the clock output to the phase-2 latches;
- d_{pcq2} be the propagation delay from clock edge to latch output of the phase-2 latches; and
- d_f be the delay along the feedback path.

Also, let $d_{ck1} \equiv d_{ckcl} + d_{ckout}$ be the total input-to-output delay of the clock, including any added delay on the clock output; and let $d_F \equiv d_{en2} + d_{pcq2} + d_f$ be the total delay from a transition of the clock to a change in a fed-back state variable.

First, the clock logic must be stable before it is reset by the inverted clock in phase-2. This requirement is necessary to avoid glitches when the clock is reset:

CK1. $d_{ck1} + d_{ck2} \geq d_{ckcl}$.

The above notation means that, for all possible values of d_{ck1}, d_{ck2} and d_{ckcl} in a given implementation, the inequality must always hold.

Next, the inverted clock, CK_f, must be driven low in phase-2 before any state changes reach the clock reset:

CK2. $d_F \geq d_{ck2}$.

Finally, the state change must be completed before CK_f is driven high and the machine returns to phase-1:

CK3. $d_{ck1} + d_{ck2} \geq d_F$.

To conclude, the above clock reset structure, along with these three timing requirements, is sufficient to guarantee the hazard-free operation of the clock.

3.5.3 Timing Analysis and Correctness

Sections 3.5.1 and 3.5.2 were concerned with the synthesis of two critical components of a locally-clocked machine: (i) combinational logic for the outputs, clock and next-state, and (ii) the clock reset structure. In this section, we combine these components, analyze the machine's operation and derive sufficient timing requirements to insure correct sequential behavior. These timing requirements are in the form of linear inequalities. The set of inequalities describes one-sided timing constraints which can always be satisfied by adding appropriate delays to the machine. Under a typical fundamental-mode operating assumption, added delays are not on the critical path and therefore do not affect the latency of the machine.

For the following analysis, refer to Figure 3.21, which illustrates the principal signals and paths considered in the analysis of a locally-clocked machine. Let

- d_{oscl} be the delay of the combinational output and state logic;
- d_{scl} be the delay of the combinational state logic;
- d_{ckcl} be the delay of the combinational logic for the clock;
- d_{ckout} be any additional delay at the clock output;

- d_{ck2} be the delay of the feedback path of the clock, including reset logic;
- d_{en1} be the delay from the clock output to the phase-1 latches;
- d_{en2} be the delay from the clock output to the phase-2 latches;
- d_f be the delay along the feedback path;
- d_{h1}, d_{s1}, and d_{w1} be the hold time, set-up time and clock width, respectively, for the phase-1 latches;
- d_{h2}, d_{s2}, and d_{w2} be the hold time, set-up time and clock width, respectively, for the phase-2 latches;
- d_{pcq1} and d_{pcq2} be the propagation delay from clock edge to latch output of the phase-1 and phase-2 latches, respectively; and
- d_{piq1} and d_{piq2} be the propagation delay from input to output of enabled phase-1 and phase-2 latches, respectively.

In addition, two composite paths are defined, to simplify the final delay requirements:

- $d_{ck1} \equiv d_{ckcl} + d_{ckout}$; and
- $d_F \equiv d_{en2} + d_{pcq2} + d_f$.

d_{ck1} is the total input-to-output delay of the clock, including any added delay on the clock output. d_F is the total delay from a clock transition to a change in fed-back state variables.

We now analyze the machine's operation during a single machine cycle. In the remainder of this section, it is assumed that we are given a burst-mode specification, G. The output, clock and next-state logic are synthesized as described in Section 3.5.1. The clock reset logic is designed as described in Section 3.5.2. We consider a single specified transition (u, v) of G. It is assumed that the machine is initially stable, the input state is $in(u)$, and the machine is in the state corresponding to specification state u. By Corollary 3.4 of Section 3.5.1, the clock is initially at 0.

By Lemma 3.3, there are two cases to consider: the specified input burst (i) does not cause a state change ("stable transition"), or (ii) causes a state change ("unstable transition").

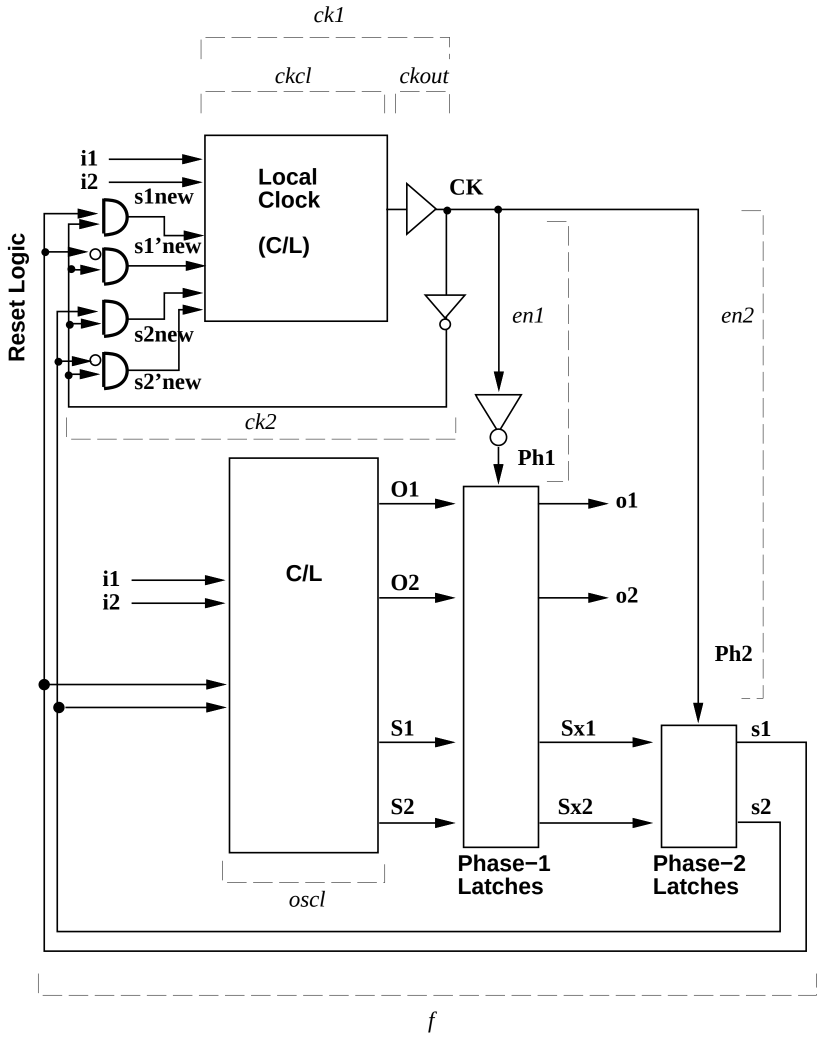


Figure 3.21: Timing parameters for locally-clocked machine.

Transition without a State Change.

Figure 3.22 gives a timing diagram for the phase-1 behavior of the machine where no state change occurs. This scenario occurs when states u and v have been merged into the same final machine state. Inputs i_1 and i_n are the first and last inputs, respectively, to change value in the input burst. Output O is any output signal before it enters its phase-1 latch, and o is the corresponding phase-1 latch output. State variable S is any next-state signal before it enters its phase-1 latch, Sx is the corresponding phase-1 latch output and s is the corresponding phase-2 latch output.

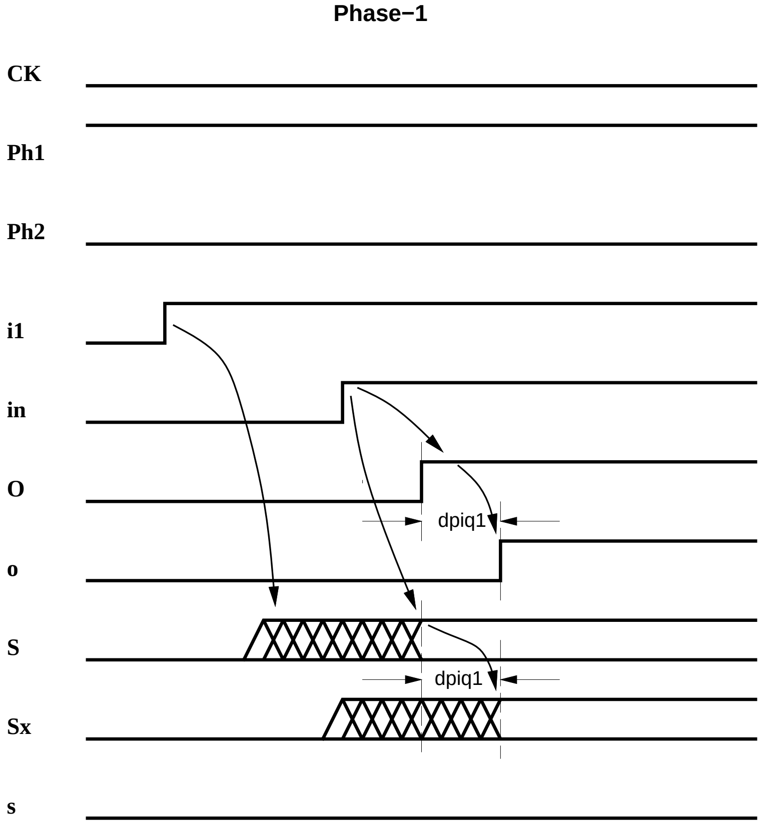


Figure 3.22: Timing diagram for locally-clocked machine: no state change (*phase-1*).

The functional and logical correctness of the outputs, state variables and clock *during phase-1* are guaranteed in Section 3.5.1. In particular, the output and clock are hazard-free, and the clock remains at 0. The final next-state function is unspecified, so the next-state logic may change value arbitrarily without affecting the internal state. The machine remains in phase-1, and no timing requirements are necessary. The machine state corresponds to specification state v; no state change is required. (It is assumed throughout that hazard-free phase-1 D-latches are used: when they are transparent, glitch-free inputs are passed through as glitch-free outputs.)

Transition with a State Change.

Figure 3.23 gives the timing diagram for a machine cycle where the machine changes state. In this case, a clock transition occurs.

Phase-1 behavior is similar to the previous case. However, in this case, the machine goes to a new state corresponding to specification state v. Timing requirements are needed to insure correct sequential behavior. In particular, output and next-state changes must pass through the phase-1 latches before the latches are disabled. Similarly, the next-state must be set-up before the phase-2 latches are enabled. These requirements were stated informally as Requirement 1. The formal requirements are described by Equations 3.1 and 3.2. Figure 3.24 illustrates the requirements, and they are also indicated by numbers 1 and 2 in Figure 3.23.

$$d_{ck1} + d_{en1} \geq d_{oscl} + d_{s1}; \tag{3.1}$$

$$d_{ck1} + d_{en2} \geq d_{scl} + d_{piq1} + d_{s2}. \tag{3.2}$$

Once the clock goes high, the machine enters phase-2. The next-state is fed back as input to the machine. Since overlapping clock phases are allowed, the phase-2 latches may be enabled before the phase-1 latches are disabled. Equation 3.3 insures that the phase-1 latches are disabled before the fed-back state variables can pass through them. This requirement was stated informally as Requirement 4. Figure 3.25 illustrates the requirement, and it is also indicated by numbers 3 in Figure 3.23.

$$d_F + d_{oscl} \geq d_{en1} + d_{h1}. \tag{3.3}$$

In phase-2, the clock will eventually be reset. This must not occur until the output

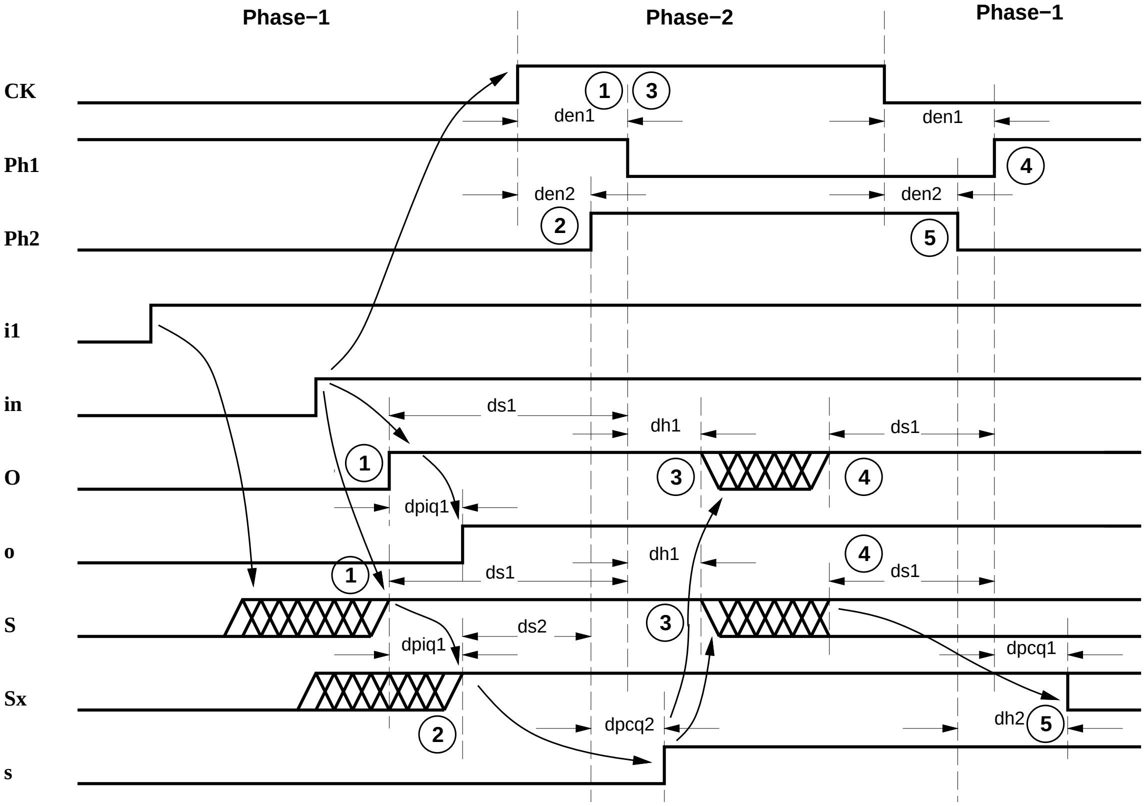


Figure 3.23: Timing diagram for locally-clocked machine: state change.

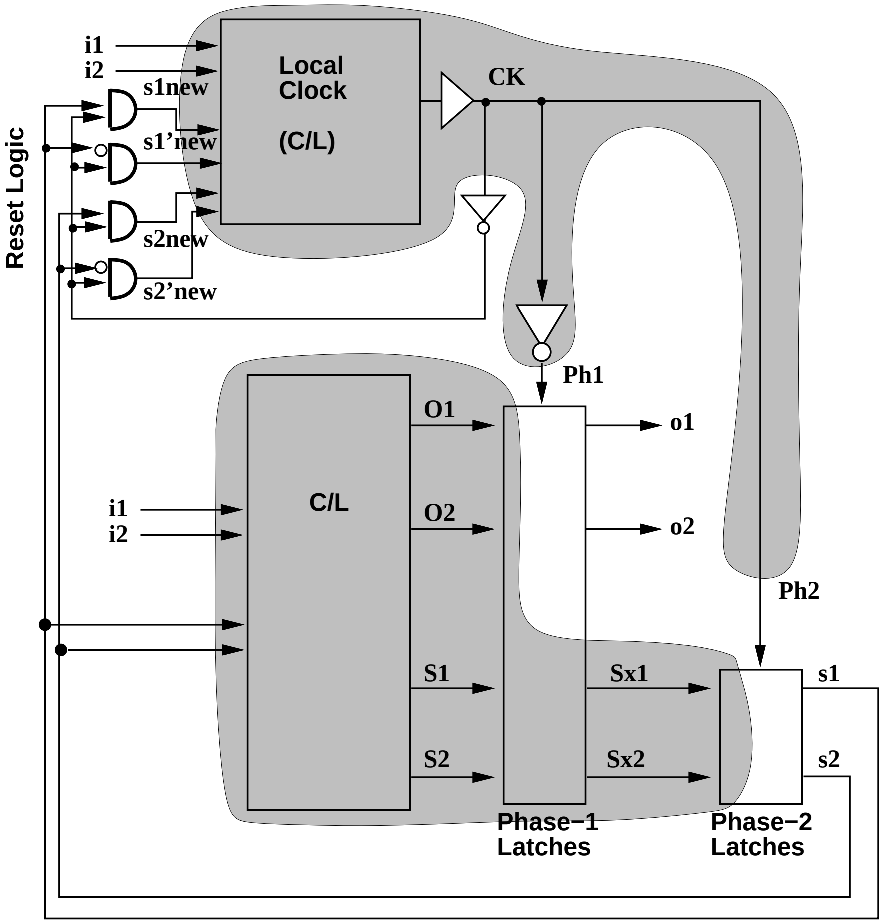


Figure 3.24: Sequential timing requirements (A).

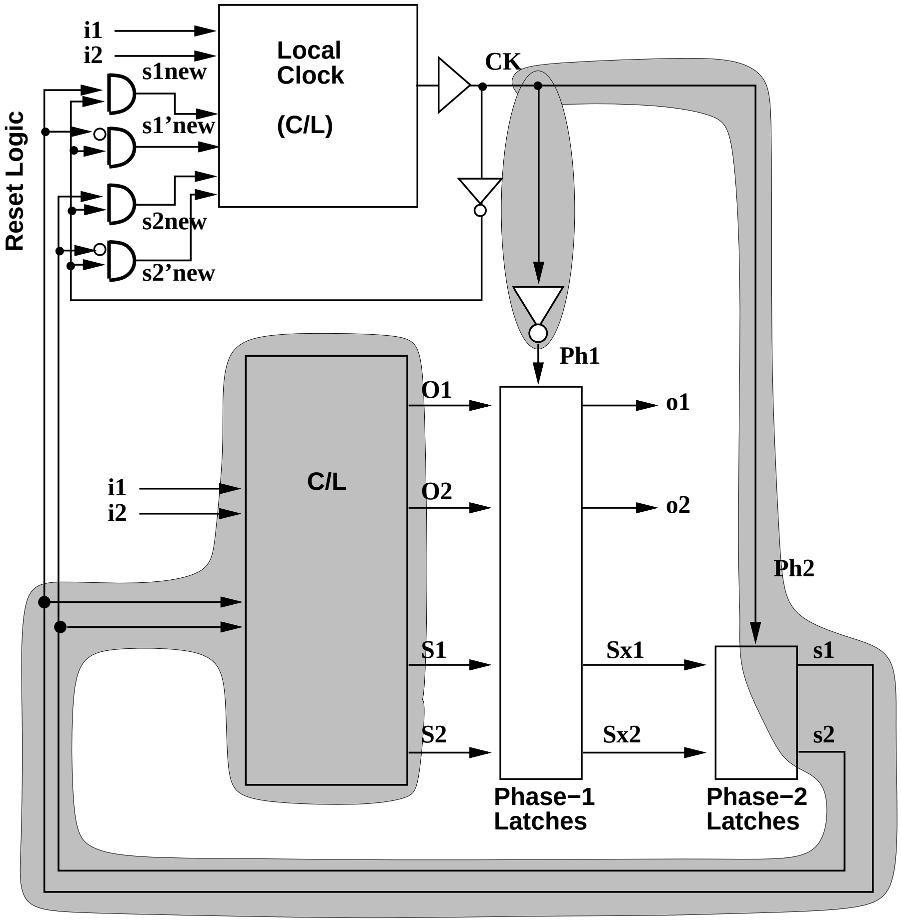


Figure 3.25: Sequential timing requirements (B).

and next-state logic have stabilized and phase-1 set-up times are satisfied. This requirement is formalized by Equation 3.4. This was stated informally as Requirement 2. Figure 3.26 illustrates the requirement, and it is also indicated by numbers 4 in Figure 3.23.

$$d_{ck1} + d_{ck2} \geq d_F + d_{oscl} + d_{s1}. \tag{3.4}$$

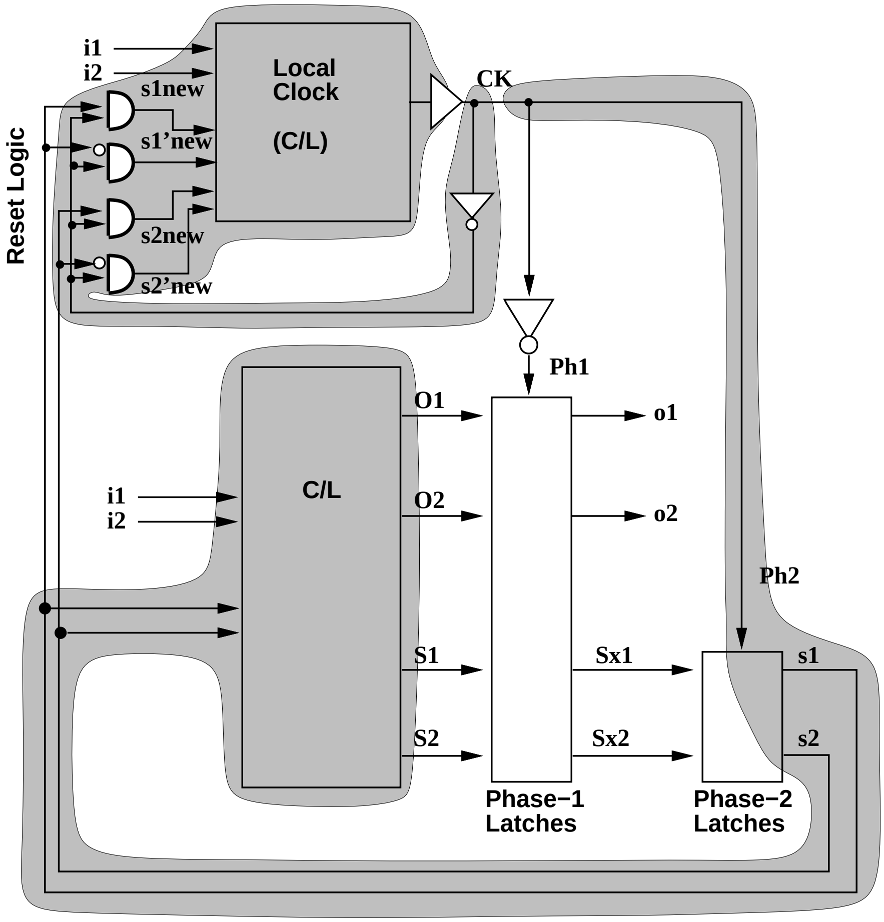


Figure 3.26: Sequential timing requirements (C).

Figure 3.27 illustrates an additional timing requirement: in the return to phase-1, there must be no race through the flipflops, even if path d_{en2} is very fast. This requirement is described by Equation 3.5. It is also indicated by number 5 in Figure 3.23. This requirement is only necessary because the next-state, S and Sx, is a don't care after the feedback cycle, and so we must prevent an incorrect next-state from passing through the latches. If we modified the next-state function to have a specified correct value at this point, the requirement would be unnecessary. However, the next-state logic might be more complicated.

$$d_{en1} + d_{pcq1} \geq d_{en2} + d_{h2}. \tag{3.5}$$

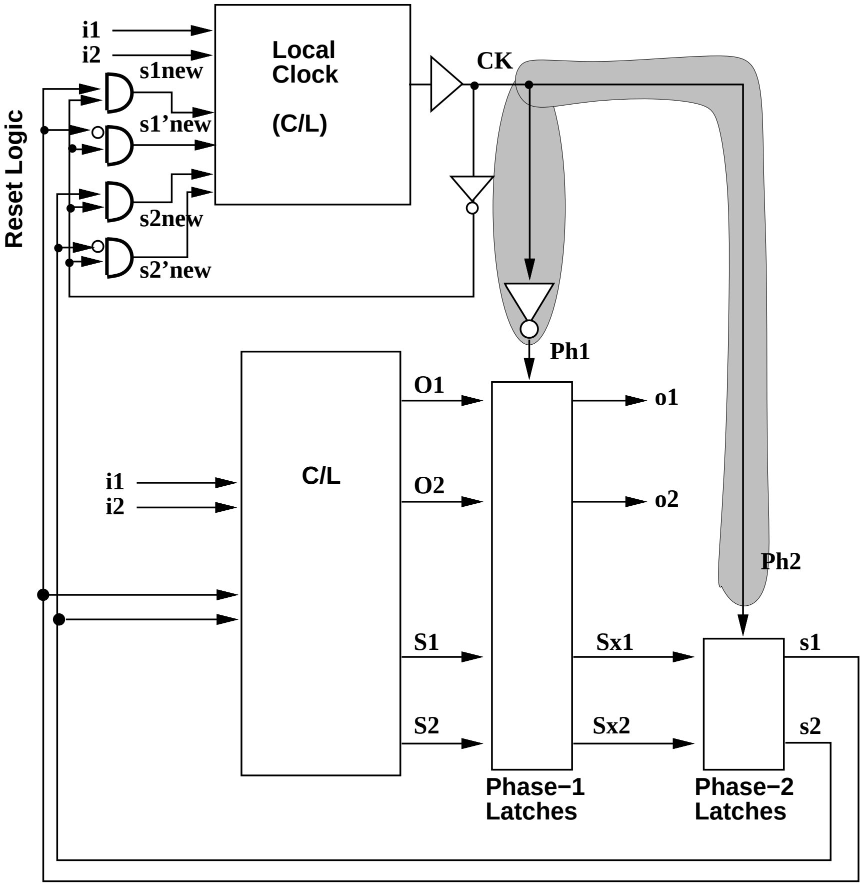


Figure 3.27: Sequential timing requirements (D).

In addition, the clock pulse width of the clock pulse must be sufficiently long for the given latches; that is:

$$d_{ck1} + d_{ck2} \geq d_{w1}, d_{ck1} + d_{ck2} \geq d_{w2}. \tag{3.6}$$

Finally, we include the timing requirements on the clock described in Section 3.5.2.

The first requirement, **CK1.**, insured that the clock was stable before it was reset:

$$d_{ck1} + d_{ck2} \geq d_{ckcl}. \tag{3.7}$$

The next requirement, **CK2.**, insured a correct resetting of the clock: $d_F \geq d_{ck2}$. The parameter d_F has the same meaning as defined in this section: $d_F \equiv d_{en2} + d_{pcq2} + d_f$:

$$d_F \geq d_{ck2}. \tag{3.8}$$

The final requirement, **CK3.**, insures that the clock returns to phase-1 correctly: $d_{ck1} + d_{ck2} \geq d_F$. However, this requirement does not need to be added, since it is subsumed by Equation 3.4.

These requirements are one-sided timing constraints on d_{ckout}, d_f, and d_{en1}. They can always be satisfied by adding appropriate delays to the clock output, feedback path and phase-1 latch enable lines, if necessary.

In particular, d_{en1} can be increased first to satisfy Equation 3.5. Next, d_f can be increased to satisfy Equations 3.3 and 3.8. Finally, d_{ck1} can be increased to satisfy the remaining requirements. There are no circular dependencies, and the result is a solution to a set of one-sided timing constraints.

Many of these requirements will be satisfied in practice without added delays. However, in general, the need for delays depends on the complexity of the implementation, as well as details of layout and technology mapping. As a rule, though, we anticipate that Equations 3.3, 3.6, 3.7 and 3.8 will easily be satisfied.

3.6 Generalizations

We have assumed that no new inputs can arrive until all output changes are generated and all logic has stabilized after the state change. This is a "generalized fundamental mode" assumption for a burst-mode implementation.

This assumption can be relaxed to allow new inputs to arrive earlier. In general, *if there is no state change*, new input changes may not occur until the combinational logic is stable. We now consider restrictions on when new inputs may arrive when there is a state change.

1. *New inputs can arrive as soon as all output changes are generated and the clock has gone high.* This assumption allows inputs to arrive during phase-2. To insure correct operation during a state change, we add a phase-1 latch to each primary input (see Figure 3.28).[6] When the clock goes high, these latches are disabled and the feedback cycle begins. At this point, new inputs may safely arrive. When the clock is reset, the input latches are re-enabled, and the machine can process the new inputs.

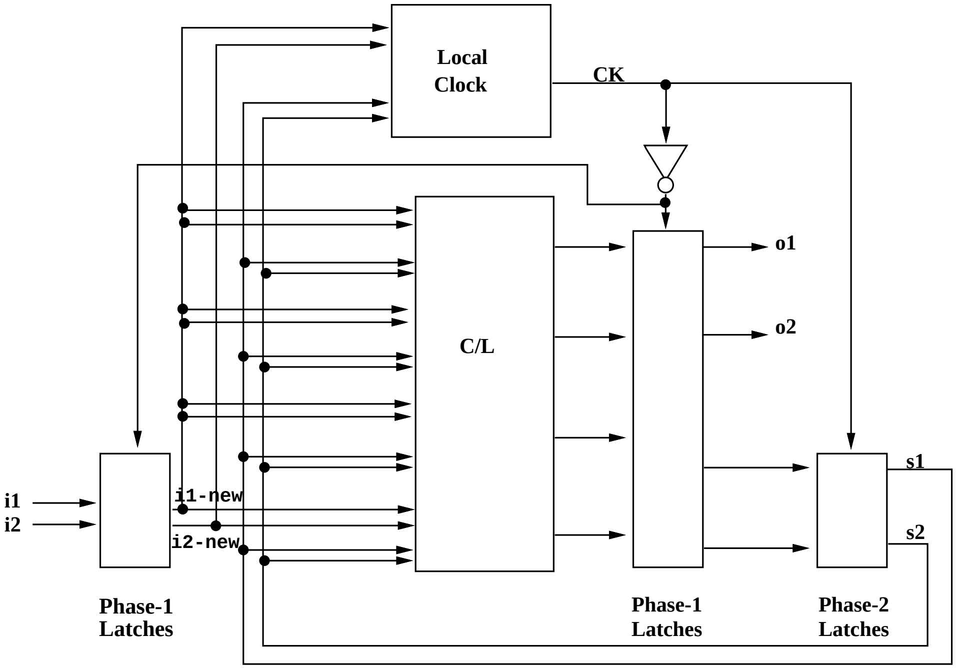


Figure 3.28: Phase-1 input latches in an asynchronous state machine.

2. *New inputs can arrive as soon as all output changes are generated.* This assumption allows even faster response by the environment. To insure safe operation, outputs themselves must be changed late: *after* a state change is completed. For example, if an input burst from $abc = 001 \rightarrow 010$ results in output burst $yz = 01 \rightarrow 11$ and a state change from P to Q, the output change is enabled only in input state $abc = 010$ in destination state Q.[7] Such changes are called *late output changes*, or *late outputs.* Output changes which are not late are called *early output changes*, or *early outputs.*

[6] This scheme is feasible only if the latches will not glitch if the inputs are changing just before the latches are enabled.

[7] This approach can be used to simplify output logic as well [25].

3.7 Optimizations

Three optimization techniques are discussed briefly.

1. *Selective clock reset.* The clock can be reset correctly using an alternative "selective" approach, which does not require reset logic. First, clock products are analyzed and partitioned into two classes: those which are hazardous during some state transition, and those which are always hazard-free. Second, to eliminate hazards during clock reset, each hazardous product is augmented to receive a reset input: the inverted fedback clock (in a different context, cf. [103]). This input is similar to the state literals produced by reset logic: it is used to mask potential glitching during the state change. The remaining clock products are unmodified; they are hazard-free during every state transition (appropriate timing assumptions are required). For the synthesis of the specifications in Figures 3.1 and 3.2, after analysis, it is determined that every clock product is hazard-free. Therefore, no fedback clock lines are required.

2. *State variable removal.* In certain cases, output variables can be substituted for state variables. An output can replace a state variable if its output-table (or its complement) "covers" the state's transition-table. That is, the output (or its complement) has the same value as the state variable wherever the state variable is specified. In this case, the state variable's logic is removed; a phase-2 latch is added to the output, which now serves as both an output and a state variable.

3. *Output latch removal.* An output latch is used to prevent hazards during state changes; if there are no such hazards the latch can be removed. In particular, each transition "cube" in the output table which corresponds to a state change must satisfy the same conditions as were satisfied by all input bursts: (a) the transition must be free of function hazards; (b) hazard-free logic must be used, that is, the cover for the given transition must not violate any covering requirement described in chapter 2; and (c) the output change must be an early change (if it is late, latch removal may cause the output change to occur earlier than is safe).

3.8 Synthesis Algorithms

Section 3.5.1 formalized the steps of logic synthesis, but did not describe synthesis algorithms. This section describes algorithms for two steps of synthesis: state minimization and state assignment. The final step — logic minimization — is discussed in the next chapter. Section 3.5.1 indicated that a *primitive cover* can always be used as an unoptimized hazard-free logic implementation. The algorithm in the next chapter is used to produce a minimized hazard-free logic implementation.

3.8.1 State Minimization

In this section, we present a simple state minimization algorithm which partitions the states of an initial flow table. Section 3.10 considers the more general case where state splitting is allowed. Before discussing the algorithm, it will be useful to review some background (see Unger [91] for further details).

State minimization algorithms usually begin with the notion of a *compatibility relation*. Informally, a pair of states is *compatible* if the two states can be merged in a final flow table. Each compatible pair has an associated set of *implied*, or *dependent*, *pairs*: if the compatible pair is merged, other state pairs must be merged as well.

Compatible pairs are grouped into larger sets called *compatibility classes*, or *compatibles*. A compatible describes a set of states which can be merged. If no state can be added to a compatible without introducing an incompatible pair, the set is called a *maximal compatible*. Each compatible has an associated set of *implied compatibles*, or *dependencies*: if the states of the compatible are merged, then the states of these other compatibles must be merged as well.

A *cover* is a set of compatibles where every state is contained in some compatible of the set. A cover is a *partition* if each state appears in only one of its compatibles. A cover is consistent, or *closed*, if each dependency of each of its compatibles is contained in a compatible of the cover. A *minimal* cover is a cover containing a minimum number of compatibles. The goal of state minimization is to find a minimal closed cover for a given flow table.

Grasselli and Luccio developed a general algorithm to produce a minimal closed cover, which makes use of *prime compatibles* [40]. Grasselli also developed an algorithm to produce minimal closed partitions, using *admissible compatibles* [39]. Our algorithm follows a simpler procedure described by Unger [91]; it makes use of maximal compatibles. The algorithm can easily be improved by incorporating techniques from the above algorithms.

The goal of state minimization is usually to minimize the number of states in the reduced flow table.[8] Our algorithm has two additional goals. First, we must insure that a hazard-free cover exists for each output. Section 3.5.1 demonstrated that, after arbitrary state reduction, there may exist no hazard-free cover for the outputs. However, a hazard-free cover is guaranteed to exist if constraints are placed on state merging. We include such constraints in our algorithm by adding restrictions to the definition of compatibility.

Our second goal is to minimize the final number of *state changes* that can occur. This goal is an important optimization because an implementation with fewer state changes has fewer clock pulses; so the local clock logic is often smaller and faster.

As an example, consider a burst-mode specification having a transition from a state s to t. Recall that, in a locally-clocked implementation, the clock remains low except when a state change is required. If states s and t are not merged in the final implementation, a state change occurs and the local clock goes high. However, if s and t are merged, the state is unchanged and the clock remains low. That is, each merger of a pair of *adjacent* states of the specification eliminates a state change — and a clock transition — from the final implementation. In practice, the clock logic is often simpler when there are fewer clock transitions. Our second goal is therefore to merge adjacent specification states whenever possible.

In the algorithm below, Steps 1 through 3 are designed to minimize the number of states. Step 1 also includes constraints on state merging to guarantee a hazard-free logic implementation. Step 4 finds a partition of the states which has a minimal number of state changes. Step 5 checks that the final partition is closed. Of these, Step 1 uses standard techniques but has a new definition of compatibility; Steps 2, 3 and 5 are standard; and Step 4 uses a new approach to generating a partition of states. Standard techniques are described in depth in Unger [91].

[8] As a rule, a smaller flow table tends to map to a simpler implementation. For a discussion of exceptions to this rule, see Unger [91] and McCluskey [60].

Step 1. Determine compatible states. A compatibility relation is defined in two steps. In the first step, an initial compatibility relation is defined, called *output compatibility.* Two states are *output incompatible* if they have different output values for some input value; otherwise they are *output compatible.* More formally, using the notation of Section 3.5.1, states s and t are output incompatible if $f_j(s,x) \neq f_j(t,x)$ for some output function f_j and input value x, where $f_j(s,x)$ and $f_j(t,x)$ are both defined.

Output compatibility determines when the output functions of two states are consistent. A final relation, called *compatibility*, determines when the next-state functions are consistent as well. A recursive definition is used: two states are *incompatible* if they are output incompatible or if their corresponding next-states are incompatible; otherwise they are *compatible.* A compatibility relation is derived from an initial output compatibility relation using a simple fixpoint computation [91].

In our synthesis method, however, the initial compatibility relation must be modified: states must be checked both for *dynamic-hazard-free (dhf-) compatibility* and *output compatibility.* Two states are called *dhf-incompatible* if their merger might result in an unavoidable hazard in a sum-of-products logic implementation; otherwise they are *dhf-compatible.* The restrictions on state merging presented in Section 3.5.1 define when two states are dhf-incompatible. Because incompatible states are never merged, this relation insures that the constraints on state merger in Section 3.5.1 will always be satisfied.

Step 2. Generate all maximal compatibility classes. Next, maximal compatibility classes are built up from compatible pairs, expanding them by adding new states whenever possible. A simple iterative algorithm to generate all maximal compatibles is described by Unger [91].

Step 3. Generate minimum (irredundant) covers of maximal compatibles. To find a minimum cover of maximal compatibles, a symbolic algorithm is used, called *Petrick's method* [60]. The compatibles which contain a state are represented as a sum; the symbolic product of all such sums describes all possible covers. A minimum-size solution can then be selected.

Example. Suppose F is a flow table having three states, (q, r, s), and four maximal compatibles, (W, X, Y, Z). Also suppose that state q is contained in W and X; state r is contained in W and Y; and state s is contained Z. The covering requirement is that each state must be covered by some compatible. This requirement can be represented symbolically using a Boolean sum. $(W+X)$ describes compatibles which cover q; $(W+Y)$ describes compatibles which cover r; and (Z) describes the compatible which covers s. The combined covering requirement for all three states is described by a Boolean product of these sums: $(W+X)\,(W+Y)\,(Z)$. This product can be multiplied out and simplified, to yield an equivalent Boolean sum: $WZ + XYZ$. This sum of products describes all minimal covers satisfying the given requirements. Cover WZ contains two maximal compatibles, while XYZ contains three. Therefore, WZ is a minimum cover of the original states. □

Step 4. Generate a final partition of the original states. Given a minimum cover of compatibles, we generate a state partition by removing states from compatibility classes until each state belongs to a unique class. A state is *assigned* if it belongs to precisely one class, otherwise it is *unassigned.*

The simple heuristic in Table 3.1 tries to leave adjacent states together in the same compatible, as states are removed. The algorithm first removes states which are not adjacent to other states in their class. When necessary, it breaks up adjacent pairs by arbitrarily assigning a state, and reiterates, until a final state partition is achieved.

Step 5. Check dependency constraints. In general, a pair of merged states may require that other pairs of states be merged. Our approach is to record dependency information, generate a minimal solution, and test that the solution is consistent with the dependency constraints. This approach suffices to find correct state minimizations for all the examples of this thesis. (Cases where solutions violate dependency constraints are rare.)

Example

As discussed in Section 3.5.1, Figure 3.14 is a burst-mode specification having 6 states, 3 inputs (a, b, c) and 2 outputs (y, z). Initially, $abc = 000$ and $yz = 01$. The unminimized, or primitive, flow table was shown in Figure 3.15. We use this example to illustrate the state minimization algorithms.

In Step 1, first *ignoring* the constraints of "dhf-compatibility" and using a standard compatibility relation instead, the resulting compatible pairs of states are: { *(A D), (B E), (B F), (D E), (D F), (E F)* }. In Step 2, four maximal compatibles are generated:

```
*Algorithm* **Remove-Isolated-States** (S,C):
    /* S = set of states; */
    /* C = $\{X_1, \ldots, X_n\}$ is any closed cover of maximal compatibles; */
    UNASSIGNED = S;
    ASSIGNED = {};

    *while* UNASSIGNED $\neq \{\}$
        *repeat*
            *for* each state $s \in$ UNASSIGNED
                *if* $s$ is in exactly one compatible $X_i \in C$
                    ASSIGNED = ASSIGNED $\cup \{s\}$;
                    UNASSIGNED = UNASSIGNED $- \{s\}$;
                *else if* $s$ belongs to some compatible $X_i$
                  where $s$ is not adjacent to any other state in $X_i$
                    $X_i = X_i - \{s\}$        /* remove s from $X_i$ */
        *until* no change;
        *if* UNASSIGNED $\neq \{\}$
            let $t \in$ UNASSIGNED be any unassigned state,
              and let $X_i$ be any compatible containing $t$;
            ASSIGNED = ASSIGNED $\cup \{t\}$;
            UNASSIGNED = UNASSIGNED $- \{t\}$;
            *for* each compatible $X_j \in C$, where $i \neq j$
                $X_j = X_j - \{t\}$        /* remove t from all other compatibles */

    *return*($C$)        /* return partition $C$ */
```

Table 3.1: Algorithm *Remove-Isolated-States*.

{ *(A D), (C), (B E F), (D E F)* }. In Step 3, a minimal irredundant cover is generated using Petrick's method: { *(A D), (C), (B E F)* }. No other cover has cardinality 3. In Step 4, no further modifications are necessary to this cover, since it is already a partition. In Step 5, all dependencies are satisfied; therefore the result is a minimum cover.

However, as indicated earlier, states A and D violate the conditions for dhf-compatibility; they should not have been merged. After incorporating dhf-compatibility into Step 1 above, states A and D are *no longer compatible*. The final list of compatible pairs of states is: { *(B E), (B F), (D E), (D F), (E F)* }. The maximal compatibles of Step 2 are then: { *(A), (C), (B E F), (D E F)* }. By Step 3, a minimal cover now has *four* compatibles: { *(A), (C), (B E F), (D E F)* }. This cover is not a partition: states E and F each appear in two compatibles. Step 4 attempts to find a state which is not adjacent in the burst-mode specification (Figure 3.14) to other states in that compatible. However, E and F are adjacent to each other, as well as to B and D. Therefore, the algorithm arbitrarily assigns one of the states to one of the classes. If E is assigned to *(B E F)*, the compatibles are then: { *(A), (C), (B E F), (D F)* }. At this point, only F appears in two compatibles. In compatible *(B E F)*, F is adjacent to E; in compatible *(D F)*, F is not adjacent to D. Therefore, F is assigned to compatible *(B E F)*, and the resulting partition is: { *(A), (C), (B E F), (D)* }. In Step 5, all dependencies are satisfied. After state assignment and generation of Karnaugh maps, each output and clock function is guaranteed to have a hazard-free sum-of-products realization. □

3.8.2 State Assignment

Our design style allows the use of unconstrained (i.e. arbitrary) state assignment, as in a synchronous design. There is no need to use a critical race-free state assignment. Therefore, state encoding can be performed using standard synchronous tools, such as *nova* [99].

However, there is a slight mismatch between *nova* and our design style. *nova* attempts to find an *optimal state encoding*, that is, one which leads to a minimal logic implementation in sum-of-products form. The algorithm uses a *symbolic cover* to describe a logic implementation in terms of symbolic states. *Our* goal, though, is to find a state encoding which leads to a minimal *hazard-free* logic implementation. As a result, the state codes produced by *nova* may not optimal for our purposes. In the future, it would be useful to modify *nova* to consider only *hazard-free* symbolic covers when it searches for an optimal state encoding.

3.9 Detailed Example

The synthesis method is now demonstrated on a larger example: a controller that has been implemented using a different design style for the Post Office communication chip developed at HP Laboratories [86, 28]. The machine has 5 inputs (*req-send, treq, rd-iq, adbld-out, ack-pkt*) and 3 outputs (*tack, peack, adbld*) (see Figure 3.2).

The state diagram is translated into a primitive flow table having 11 states and 14 state transitions. After minimization the final flow table has 4 states and 9 state transitions; remaining transitions occur without a state change. Therefore, 5/14 or 35.7% of the original transitions do not require a clock pulse in the final implementation. The final state partition is shown in Figure 3.29(a).

State codes are assigned to the states, as shown in Figure 3.29(a)). Karnaugh maps are then constructed for the clock, outputs and state variables.

The Karnaugh map for the clock is shown in Figure 3.29(b). (Blank entries in the map represent don't-care entries.) The clock function may be implemented using an arbitrary sum-of-products implementation.

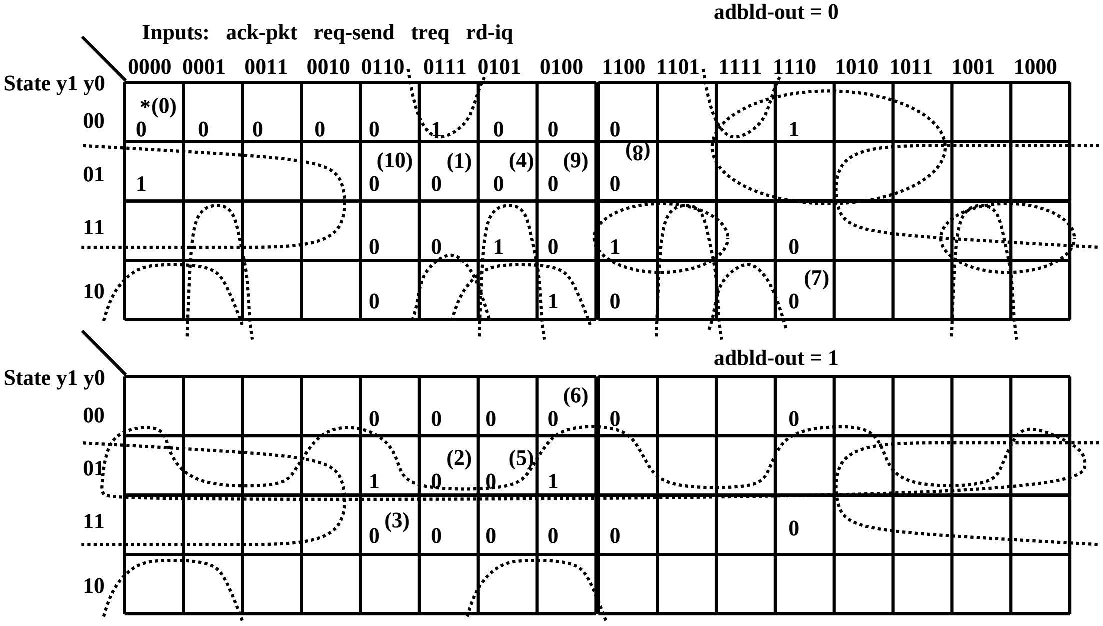
Karnaugh maps for the three outputs are shown in Figures 3.30, 3.31 and 3.32. Required cubes for 1→1 and 1→0 transitions are indicated, as well as final hazard-free covers. (Required cubes are only shown if they contain more than one minterm.)

The next-state function for state variables *y1* and *y0* is shown in Figure 3.33. Any arbitrary implementation of the next-state functions is permitted.

The final implementation is described in sum-of-products form. To satisfy the timing requirements, it may be necessary to add appropriate delay to the clock output and feedback path. To insure that the clock resets correctly, reset logic may be attached (see Figure 3.18). Alternatively, an optimized solution may be used, as discussed in Section 3.7: The inverted clock is fed back directly to each hazardous clock product. For the given example, *no* clock feedback lines are necessary, since each clock product is hazard-free for every specified state transition.

Merged States:	State Encoding y1 y0:
(0 6)	0 0
(1 2 4 5 8 9 10)	0 1
(3)	1 1
(7)	1 0

(a) State Assignment



adbld-out = 0

Inputs: ack-pkt req-send treq rd-iq

State y1 y0	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
00	*(0) 0	0	0	0	0	1	0	0	0			1				
01	1				(10) 0	(1) 0	(4) 0	(9) 0	(8) 0							
11					0	0	1	0	1			0				
10					0			1	0			(7) 0				

adbld-out = 1

State y1 y0	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
00					0	0	0	(6) 0	0			0				
01					1	(2) 0	(5) 0	1								
11					0 (3)	0	0	0	0			0				
10																

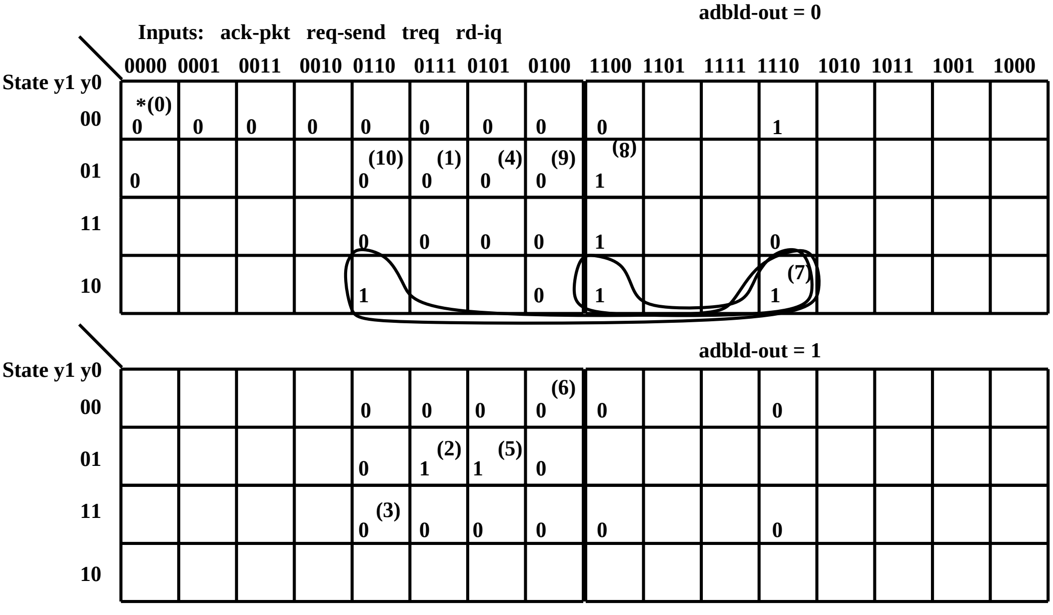
*Specification states are indicated when they are entered.

Sum-of-products implementation:

CK = y1' y0 adbld-out rd-iq' + y1' adbld-out' ack-pkt treq + y1 y0 adbld-out' ackpkt treq'
+ y0' adbld-out' req-send treq rd-iq + y0 req-send' + y1 adbld-out' treq' rd-iq
+ y1 y0' ack-pkt' treq'

(b) Clock Karnaugh Map for HP Controller.

Figure 3.29: State assignment and clock Karnaugh map for HP controller.



adbld-out = 0

Inputs: ack-pkt req-send treq rd-iq

State y1 y0	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
00	*(0) 0	0	0	0	0	0	0	0	0			1				
01	0				(10) 0	(1) 0	(4) 0	(9) 0	(8) 1							
11					0	0	0	0	1			0				
10					1			0	1			(7) 1				

adbld-out = 1

State y1 y0	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
00					0	0	0	(6) 0	0			0				
01					0	(2) 1	(5) 1	0								
11					(3) 0	0	0	0	0			0				
10																

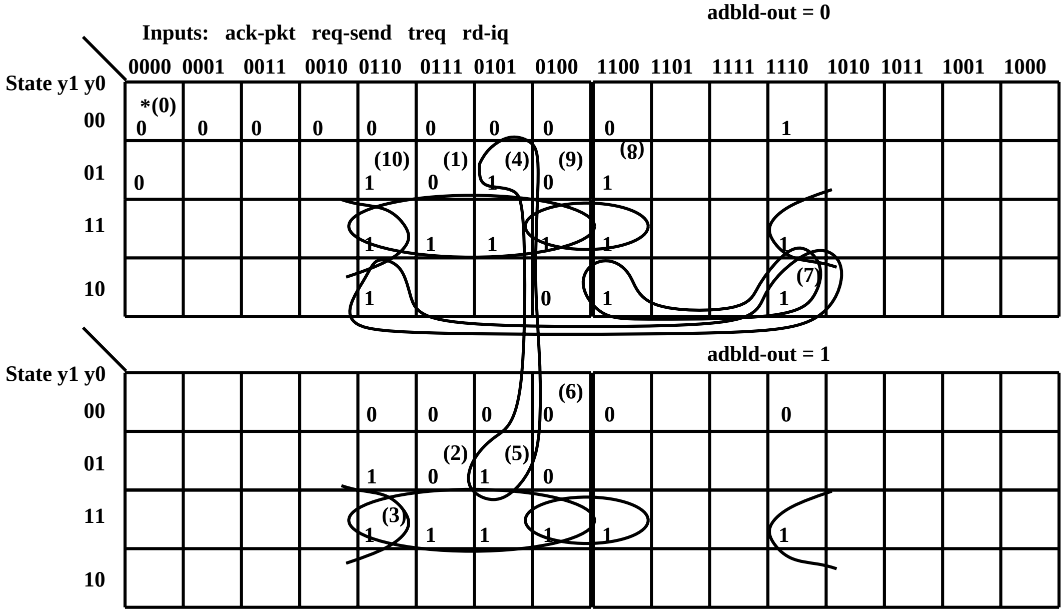
Note: only "required cubes" are shown above.

*Specification states are indicated when they are entered.

Final sum-of-products implementation:

PEACK = y1 y0’ ack-pkt + y1 y0’ treq + y1’ y0 rd-iq adbld-out + y1’ adbld-out’ ack-pkt treq
+ y0 adbld-out’ ack-pkt treq’

Figure 3.30: Karnaugh map for output *peack*.



adbld-out = 0

Inputs: ack-pkt req-send treq rd-iq

State y1 y0	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
00	*(0) 0	0	0	0	0	0	0	0	0			1				
01	0				(10) 1	(1) 0	(4) 1	(9) 0	(8) 1							
11					1	1	1	1	1			1				
10					1			0	1			(7) 1				

adbld-out = 1

State y1 y0	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
00					0	0	0	(6) 0	0			0				
01					1	(2) 0	(5) 1	0								
11					(3) 1	1	1	1	1			1				
10																

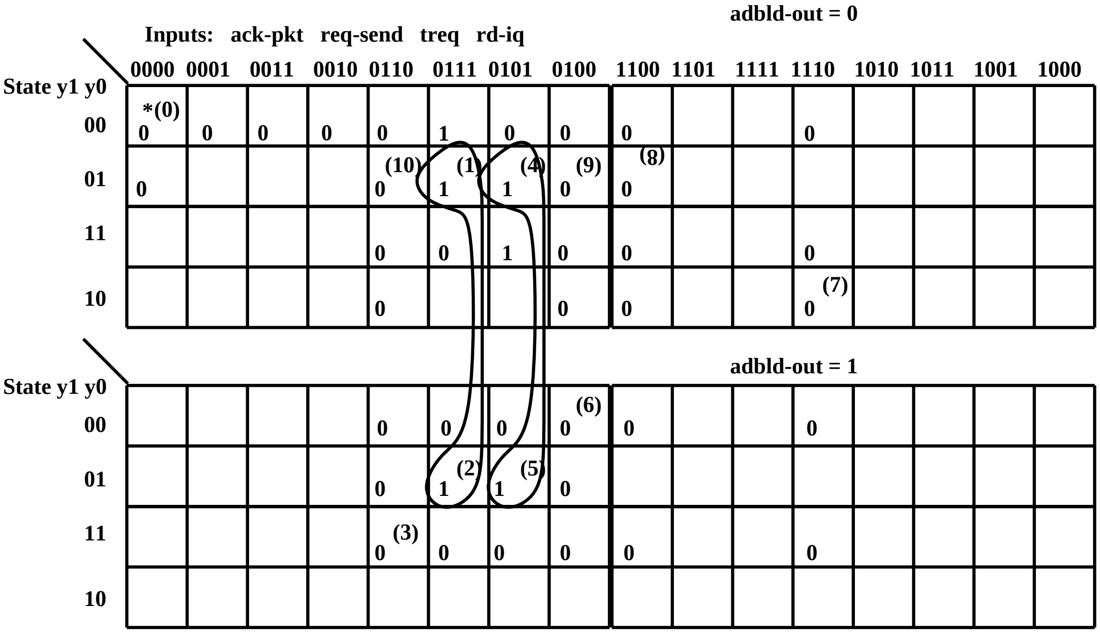
Note: only "required cubes" are shown above.

*Specification states are indicated when they are entered.

Final sum-of-products implementation:

TACK = y1 y0 + y0 treq' rd-iq + y1 ackpkt + y1 treq + y0 ack-pkt + y0 treq rd-iq' + adbld-out' ack-pkt treq

Figure 3.31: Karnaugh map for output *tack*.



Inputs: ack-pkt req-send treq rd-iq

adbld-out = 0

State y1 y0	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
00	*(0) 0	0	0	0	0	1	0	0	0			0				
01	0				(10) 0	(1) 1	(4) 1	(9) 0	(8) 0							
11					0	0	1	0	0			0				
10					0			0	0			(7) 0				

adbld-out = 1

State y1 y0	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
00					0	0	0	(6) 0	0			0				
01					0	(2) 1	(5) 1	0								
11					(3) 0	0	0	0	0			0				
10																

Note: only "required cubes" are shown above.

*Specification states are indicated when they are entered.

Final sum-of-products implementation:

ADBLD = y1' y0 rd-iq + y1 adbld-out' treq' rd-iq + y1' adbld-out' req-send treq rd-iq

Figure 3.32: Karnaugh map for output *adbld*.

Y1 Y0

Inputs: ack-pkt req-send treq rd-iq

adbld-out = 0

State y1 y0	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
00	*(0)					01						10				
01	00				(10)	(1)	(4)	(9)	(8)							
11							01		01							
10								01				(7)				

adbld-out = 1

State y1 y0	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
00								(6)								
01					11	(2)	(5)	00								
11					(3)											
10																

*Specification states are indicated when they are entered.

Final sum-of-products implementations:

Y1 = treq rd-iq
Y0 = y1 + ack-pkt' treq

Figure 3.33: Karnaugh map for state variables *y1 y0*.

3.10 State Splitting

Flow table reduction has only been considered when it results in a partition of states. In general, though, a state may be mapped to more than one reduced state after flow table reduction. This case is called "state splitting", and is formalized below.

Let $\mathrm{P}(X)$ be the *power set* of set X, and define $\mathrm{P}(X)^+ \equiv \mathrm{P}(X) - \phi$. A flow table $F' = (S', I, O, s'_0, f', n')$ is a *reduced flow table* of $F = (S, I, O, s_0, f, n)$ if there exists a function $h : S \rightarrow \mathrm{P}(S')^+$, such that $s'_0 \in h(s_0)$, and for every state $s \in S$ and input value $x \in \{0,1\}^m$, and for each state $s' \in h(s)$:

1. $f'_j(s', x) = f_j(s, x)$, whenever $f_j(s, x)$ is defined; and

2. $n'(s', x) \in h(n(s, x))$, whenever $n(s, x)$ is defined.

Two distinct states, $s, t \in S$, are said to be *merged* by mapping h if h maps them to the same state of F', that is, $h(s) \cap h(t) \neq \phi$. State s is *split* by the mapping into the states of $h(s)$.

State splitting introduces a subtle problem into the locally-clocked synthesis method.

Example

A burst-mode specification is shown in Figure 3.34 and its primitive flow table is shown in Figure 3.35. If state splitting is allowed, the minimum reduced flow table requires 4 states: $\{ACD, AE, BDG, F\}$; no other reduced 4-state reduced table exists for the specification.

However, consider the merger of states A and C into the reduced state ACD for input value $abc = 100$. State A was stable for this input value, but the reduced state is unstable and requires a state change to the state AE. That is, for the $A \rightarrow B$ transition in reduced state ACD, the clock function at input $abc = 100$ will go high, enabling a state change *before* the specified input burst is complete ($abc = 110$). As a result, the clock may glitch and the system may malfunction.

The problem is that standard flow table reduction algorithms do not require that stable states in the primitive flow table map to stable states in a reduced flow table. In the case of state splitting, there can be multiple instances of the same state (*e.g.*,

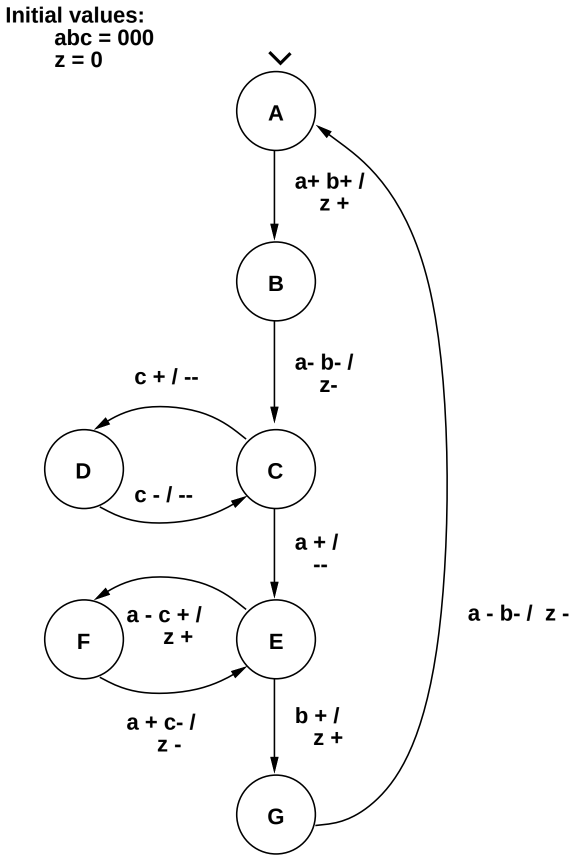
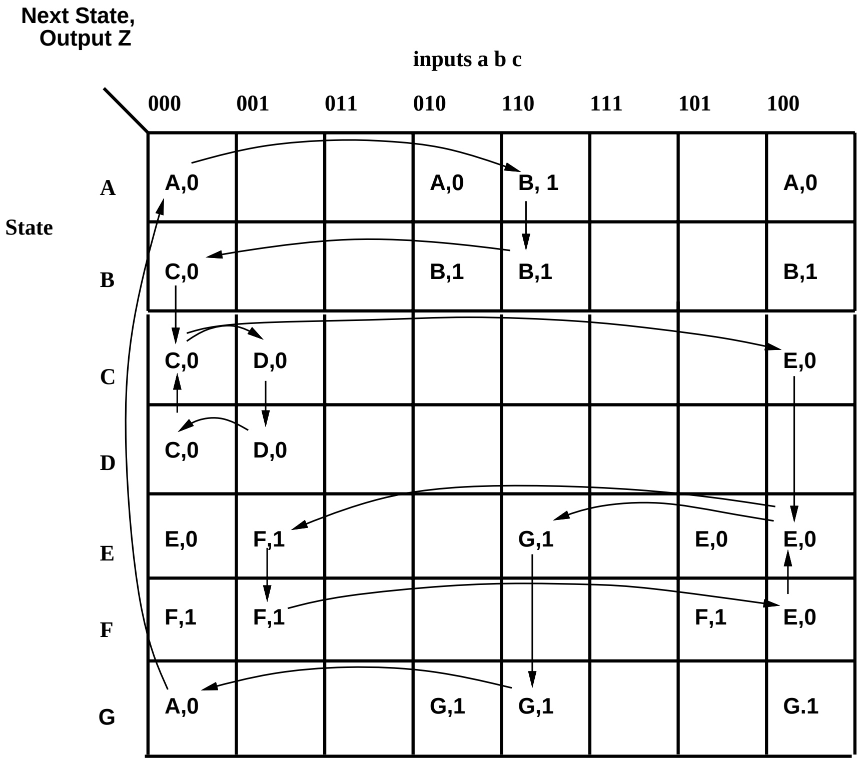


Figure 3.34: Burst-mode specification for state-splitting example.

A) in the reduced table. For the given example, state A in the primitive flow table has a stable next-state function at input value $abc = 100$. After state reduction, however, the corresponding next-state function is unstable: there is a state transition at input $abc = 100$ from one reduced state corresponding to A (*e.g.*, ACD) to another state corresponding to A (*e.g.*, AE).

It is necessary, therefore, to develop algorithms which allow state splitting but insure that stable states in the primitive flow table are always mapped to stable states after flow table reduction. Such algorithms have recently been developed by Bill Coates of HP Laboratories[23].



State	000	001	011	010	110	111	101	100
A	A,0			A,0	B, 1			A,0
B	C,0			B,1	B,1			B,1
C	C,0	D,0						E,0
D	C,0	D,0						
E	E,0	F,1			G,1		E,0	E,0
F	F,1	F,1					F,1	E,0
G	A,0			G,1	G,1			G.1

Figure 3.35: Primitive flow table for state-splitting example.

3.11 Related Work

In this section, the locally-clocked method is compared with a related approach of Chuang and Das [21].

Chuang and Das propose a self-synchronized state machine illustrated in Figure 3.36. The machine consists of combinational logic for the outputs, next-state and clock; edge-triggered flipflops; and delay elements. In the figure, the machine is "unrolled" to indicate the path of the state variables through the output logic. For simplicity, feedback loops of the clock and next-state logic are omitted.

The machine is structurally similar to a locally-clocked machine, but there are two important differences. First, an inertial delay is attached to the clock. This delay is used to filter out glitches. In the locally-clocked machine, no inertial delay is required. Second, edge-triggered flipflops are used to store outputs. In the locally-clocked machine, dynamic transparent latches are used.

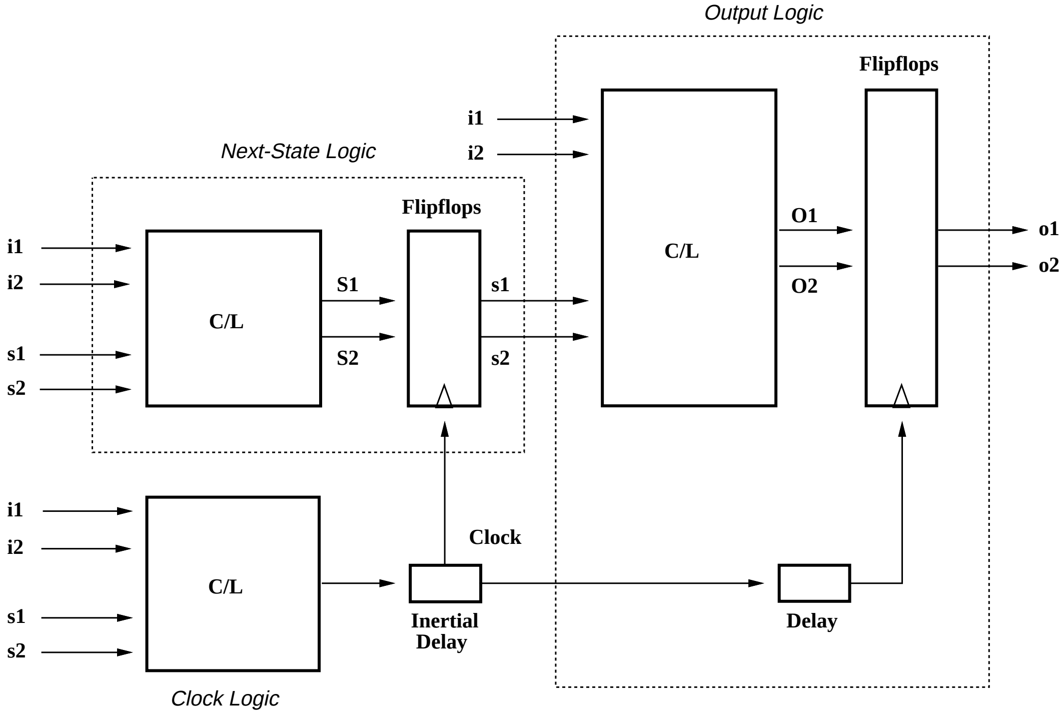


Figure 3.36: Block diagram of self-synchronized machine of Chuang and Das.

The machine operates as follows. Initially, the machine is stable and the clock is low. At some point, a multiple-input change occurs. However, inputs cannot change at arbitrary times. Instead, *all inputs must change within a fixed window of time*, *d*. That is, the machine operates in *MIC mode*, as described in Chapter 1.

If a state change occurs, the clock goes high and the next state is latched in the edge-triggered flipflops. Appropriate delay is added to the clock to insure that the next-state logic is stable before the state is latched. In particular, an inertial delay is used to filter out glitches, since the clock may have hazards. After the state changes, outputs can change. The clock is further delayed to insure that the output logic is stable before outputs are latched. The state change is also fed back to the next-state and clock logic, and the clock is eventually driven low.

There are two important features of the Chuang and Das approach which are incorporated into the locally-clocked method. First, Chuang and Das allow arbitrary state assignment. This feature simplifies the synthesis method and the resulting implementations, since critical-race free state codes are not required. Second, they use a clock function which is initially low, is driven high by an input change, and is driven low again by the resulting state change. That is, the clock function "disables itself". A similar clock function is used in the locally-clocked method.

However, the locally-clocked method has several significant advantages. First, Chuang and Das' machine operates in *MIC mode*: multiple-input changes must occur within a fixed time period. In contrast, a locally-clocked machine operates in *burst-mode*: inputs within an input burst can change at arbitrary times. The added flexibility of burst-mode is an important feature when operating in a concurrent environment.

Second, Chuang and Das rely on an inertial delay to eliminate glitches in the clock. As indicated in Chapter 1, inertial delays are difficult to build, slow down a circuit and result in slowly-changing transitions which are susceptible to noise. In contrast, the locally-clocked method does not require inertial delays. Hazards in the clock are eliminated in phase-1 using hazard-free logic and in phase-2 using simple reset circuitry.

Third, the performance of Chuang and Das' machines is poor. All output changes are "late": they occur after a state change is complete. Output changes must pass through 2 combinational logic blocks and 2 banks of flipflops, and then must wait for a clock pulse. In the locally-clocked method, output changes pass through only 1 combinational logic block and 1 bank of dynamic transparent latches. (Dynamic transparent latches typically have much smaller propagation delay than edge-triggered flipflops.) Furthermore, output changes do not wait for the clock to pulse. Instead, they pass directly through the phase-1 latches, which are transparent in phase-1. This feature considerably improves the latency of the machine.[9]

Finally, the clocking scheme in the locally-clocked method is an improvement over Chuang and Das' method. Chuang and Das use *controlled excitation*, where the clock remains low if there is no state or output change during an MIC transition. If the state or output changes, a clock pulse is generated to trigger the flipflops. The locally-clocked method uses *selective clocking*, where a clock pulse is generated *only* if there is a state change. If outputs change without a state change, no clock pulse is required. In practice, many transitions in a locally-clocked implementation are unclocked (see Sections 3.9 and

[9] There is a benefit to using output flipflops, though. Output functions may be simpler, since they are specified only when a state change occurs. However, this feature affects only one of the two combinational logic blocks on the critical path, and does not significantly affect total latency.

Number of State Changes		
	Before	After
Example	Minimization	Minimization
DME	10	4
DME-OPT	10	4
DME-FAST	10	4
DME-FAST-OPT	10	4
CHU-AD-OPT	4	2
VANBEK-AD-OPT	3	2

Table 3.2: Reduction in number of state changes after heuristic state minimization.

3.12), and the clock implementation is often simpler.

To summarize, Chuang and Das' method is sound, as long as an appropriate inertial delay can be constructed. However, the method has limited flexibility, since it requires that inputs change within a narrow window of time, *d*. Latency is poor, since the critical path is large and output changes can occur only after a clock pulse. Finally, a clock pulse is required for every output and state change, resulting in a more complex clock circuit.

3.12 Results

We now describe the results of applying the synthesis procedure to three design problems. Table 3.2 shows the effect of state minimization on the number of clocked transitions in the state diagram. Table 3.3 compares area and performance of our final state machine implementations with published solutions. In Table 3.3, our implementations are indicated by upper-case names; published designs are indicated by bold-faced type.

Our designs have not been implemented at the transistor level, so it is difficult to make a precise performance comparison. However, a logic-level comparison is possible using a rough estimate of delays. In particular, it is assumed that all simple gates and dynamic latches have a delay of 1 unit, and static storage elements have a delay of 2 units.

	Delay*		**Area**	
	Total #:		Total #:	
Implementation		static		static
	gates	storage	gates	storage
Distributed Mutex:				
Martin Design	12	4	8	4
DME	17	4	14	2
DME-OPT	12	2	10	2
DME-FAST	9	2	14	2
DME-FAST-OPT	6	0	14	2
A-D Controller:				
Chu Design	5	2	4	2
CHU-AD-OPT	7	0	8	1
Vanbekbergen				
Design #1	5	2	6	2
Design #2	2	3	3	2
VANBEK-AD-OPT	6	0	10	1

*Total delay along critical path of operation.

Table 3.3: Comparison of delay and area of published implementations with locally-clocked implementations.

1. Distributed Mutual Exclusion Element. In [54], Martin describes a self-timed controller for mutual exclusion. Controllers are placed in a ring and grant privileged access to a resource. We consider the later implementation presented by Burns [14]. As discussed in Chapter 1, Martin's implementations are *quasi-delay-insensitive*; that is, they function correctly assuming arbitrary gate delays.

(a) DME and DME-OPT. Martin's protocol is implemented directly. In implementation DME, output changes are (conservatively) set to occur late, after a state change is complete. However, the environment of each controller is largely known, since identical controllers are attached to each other in a ring. Using reasonable timing assumptions, selected output changes can be set to occur early in implementation DME-OPT. Using estimates described above for gate and storage delay, Martin's implementation is 25% faster than our unoptimized DME implementation. In particular, Martin's design has a delay of $12 + (4 * 2) = 20$ units; our design has a delay of $17 + (4 * 2) = 25$ units. However, our DME-OPT design is 25% faster than his design (16 units vs. 20 units).

(b) DME-FAST and DME-FAST-OPT. A more concurrent DME protocol is used, allowing *multiple* outputs to be generated concurrently. All output changes are conservatively set to be late in implementation DME-FAST; selected outputs are generated early in implementation DME-FAST-OPT. Our DME-FAST design is 54% faster than his design (13 units vs. 20 units), and our DME-FAST-OPT design is 233% faster than his design (6 units vs. 20 units). Since the protocol used by the DME-FAST and DME-FAST-OPT designs is more concurrent than Martin's, this is not a direct comparison. However, it indicates that locally-clocked designs may have quite good performance.

2. A-to-D Controller (Chu). In [19], Chu describes a specification and implementation of a controller for an A-to-D-converter. Again the environment of the controller is given: it is the A-to-D datapath which it controls. Using reasonable timing assumptions, we designed an optimized implementation, CHU-AD-OPT. Our design is 28% faster than Chu's implementation (7 units vs. 9 units).

3. A-to-D Controller (Vanbekbergen). In [96], Vanbekbergen specifies an A-to-D controller with a more concurrent protocol than Chu's and two implementations. His specification includes "weighted arcs"; arc weights describe delays in the environment before new inputs arrive. Using these weights and reasonable timing assumptions, we designed an optimized implementation, VANBEK-AD-OPT. Our design is 50% faster than Vanbekbergen's Design #1 (6 units vs. 9 units) and 33% faster than his Design #2 (6 units vs. 8 units).

Chapter 4

Exact Hazard-Free Two-Level Logic Minimization

4.1 Introduction

The synthesis method of the previous chapter did not provide algorithms to generate minimum-cost hazard-free combinational logic. However, it insured that a hazard-free two-level implementation always exists. In this chapter, we present a hazard-free logic minimization algorithm for locally-clocked state machines.

In fact, the problem that we solve is a more general problem; it is independent of this application. The contribution of this chapter is a solution to an open problem in logic synthesis: Given an incompletely-specified Boolean function and a set of multiple-input changes, produce an *exactly minimized two-level implementation* which is hazard-free for every specified multiple-input change, *if such a solution exists.* Our method uses a constrained version of the Quine-McCluskey algorithm [60]. The method has been automated and applied to a number of examples. Results are compared with results of a comparable non-hazard-free method (*espresso-exact* [81]). Overhead due to hazard-elimination is shown to be negligible.

This method solves a combinational synthesis problem which arises in many asynchronous sequential applications. The method has been incorporated into synthesis programs for two distinct asynchronous design styles: the locally-clocked method and the *3D method* [106].

4.1.1 Two-Level Hazard-Free Logic Minimization Problem

The two-level hazard-free logic minimization problem can be stated as follows:
Given:

> A Boolean function f, and a set, T, of *specified* function-hazard-free (static and dynamic) input transitions of f.

Find:

> A minimum-cost cover of f whose AND-OR implementation is free of logic hazards for every input transition $t \in T$.

4.1.2 Previous Work

No general *two-level hazard-free logic minimization method* has been proposed for incompletely-specified functions allowing multiple-input changes.

McCluskey [59] presented an exact hazard-free two-level minimization algorithm limited to *single-input changes*.

Several methods have been proposed for the multiple-input change case, but each has limitations. Bredeson and Hulina [9] presented an algorithm which produces hazard-free sum-of-products implementations for multiple-input changes. However, their algorithm uses sequential storage elements to implement combinational functions, where storage elements must satisfy special timing constraints.

Bredeson [8] later presented an algorithm for hazard-free *multi-level* implementations of combinational functions with multiple-input changes requiring no storage elements. However, the algorithm does not demonstrate optimality, assumes a fully-specified function, and attempts to eliminate hazards even for unspecified transitions; in practice, results may be far from optimal. The algorithm also cannot generate certain minimum two-level solutions (if they include non-prime implicants; to be discussed later).

Closer to our work, Frackowiak [35] presented two exact hazard-free minimization algorithms for two-level implementations allowing multiple-input changes, assuming a fully-specified function. Both algorithms eliminate dynamic hazards for specified transitions. However, the first ignores static hazards while the second attempts to eliminate static hazards even for unspecified transitions. Therefore, results may be either hazardous or suboptimal.

4.2 Hazard-Free Covers

As discussed in chapter 2, a *hazard-free cover* of function f is a cover of f whose AND-OR implementation is hazard-free for a *given set* of input transitions. The following new theorem describes all hazard-free covers for function f for a set of multiple-input transitions. (It is assumed below that the set of transitions define the function; the function is undefined for all other input states.)

Theorem 4.1: Hazard-Free Covering Theorem. A sum-of-products C is a hazard-free cover for function f for a specified set of input transitions if and only if:

(a) No cube of C intersects the OFF-set of f;

(b) Each *required cube* of f is contained in some cube of the cover, C; and

(c) No cube of C intersects any *privileged cube* illegally.

Proof. The result follows immediately from Lemmas 2.1–2.3, Corollary 2.1, and the definitions of hazard-free cover, required cubes and privileged cubes. Conditions (a)-(c) insure that the function is covered correctly and hazard-free covering requirements are met for each specified input transition. □

Conditions (a) and (c) in Theorem 4.1 determine the implicants which may appear in a hazard-free cover of a Boolean function f. Condition (b) determines the covering requirement for these implicants in a hazard-free cover. Therefore, Theorem 4.1 precisely characterizes the covering problem for hazard-free two-level logic.

In general, the covering conditions of Theorem 4.1 may not be satisfiable for an arbitrary Boolean function and a set of specified input transitions (cf. [91, 6, 35]). This case occurs if conditions (b) and (c) cannot be satisfied simultaneously, and is discussed further in the previous chapter. However, as demonstrated in chapter 3, the locally-clocked method always generates functions for which a hazard-free cover exists.

4.3 Exact Hazard-Free Logic Minimization

Many exact logic minimization algorithms are based on the Quine-McCluskey algorithm [81, 60]. The Quine-McCluskey algorithm solves the two-level logic minimization problem. It makes use of a *prime implicant table*, which indicates which prime implicants cover each ON-set minterm of a Boolean function. The algorithm has three steps:

1. Generate the prime implicants of a function;
2. Construct a prime implicant table; and
3. Generate a minimum cover of this table.

Our two-level hazard-free logic minimization algorithm is based on a constrained version of the Quine-McCluskey algorithm. Only certain implicants may be included in a hazard-free cover, and covering requirements are more restrictive.

We base our approach on the Quine-McCluskey algorithm to demonstrate a simple solution to the hazard-free minimization problem. There now exist much more efficient algorithms than Quine-McCluskey [81, 82]; the hazard-elimination techniques described here can be applied to these methods as well.

Theorem 4.1(a) and (c) determine the implicants which may appear in a hazard-free cover of a Boolean function *f*. A *dynamic-hazard-free implicant* (or *dhf-implicant*) is an implicant which does not intersect any privileged cube of *f* illegally (cf. *DHA-Implikant* [35]). **Only dhf-implicants may appear in a hazard-free cover.** A *dhf-prime implicant* is a dhf-implicant contained in no other dhf-implicant. An *essential dhf-prime implicant* is a dhf-prime implicant which contains a required cube contained in no other dhf-prime implicant.

Interestingly, a prime implicant is not a dhf-prime implicant if it intersects a privileged cube illegally. A dhf-prime implicant may be a proper subcube of a prime implicant for the same reason.

Theorem 4.1(b) determines the covering requirement for a hazard-free cover of *f*: **every required cube of f must be covered**, that is, contained in some cube of the cover.

We assume a standard cost function for covers where every implicant has the same cost.[1] The *two-level hazard-free logic minimization problem* is therefore *to find a minimum cost cover of a function using only dhf-prime implicants where every required cube is covered.*

Our hazard-free Quine-McCluskey algorithm has the following steps:

1. Generate the *dhf-prime implicants* of a function;

[1] The cost function can be generalized for single-output functions to include literal-count as a secondary cost (see also discussion in [81], page 14).

2. Construct a *dhf-prime implicant table*; and
3. Generate a minimum cover of this table.

Step 0: Make Sets

Before generating dhf-prime implicants, three sets must be constructed: the *req-set*, the *off-set*, and the *priv-set*. The req-set contains the required cubes for the function *f*; it also defines the ON-set of the function. The off-set contains cubes precisely covering the OFF-set minterms. The priv-set is the set of privileged cubes along with their start points.

The sets are generated by a simple iteration through every specified transition of the given function, using Algorithm **Make-Sets** (see Table 4.1). If the function has a $0 \rightarrow 0$ change for a transition, the corresponding transition cube is added to the off-set. If the function has a $1 \rightarrow 1$ change, the transition cube is added to the req-set.

If the function has a $1 \rightarrow 0$ transition (or symmetrically, a $0 \rightarrow 1$ transition), then the maximal ON-set cubes are added to req-set and the maximal OFF-set cubes are added to off-set. In addition, the transition cube and its start point are also added to the priv-set, since this transition cube must not be intersected illegally. Through most of this chapter, a $0 \rightarrow 1$ transition from input state x to y will be considered as a $1 \rightarrow 0$ transition from input state y to x, so it has "start point" y.

Step 1: Generate DHF-Prime Implicants

The dhf-prime implicants for function *f* are generated in two steps. The first step generates the prime implicants of *f* from the req-set (which defines the on-set) and the off-set, using standard techniques [81, 82]. The second step transforms these prime implicants into dhf-prime implicants using algorithm **PI-to-DHF-PI**. This algorithm is a simpler version of Algorithm B in [35]. The algorithm iteratively refines the set of prime implicants until it generates the set of dhf-prime implicants. In practice, many prime implicants are also dhf-prime implicants (see Experimental Results). Also, there are fast existing algorithms to generate the the prime implicants of a function.

Pseudo-code for the algorithm is given in Table 4.2. tmp-set is initialized to the set of prime implicants. The algorithm iteratively removes each implicant, p, from tmp-set.

```
Algorithm Make-Sets (set T of input transitions):
    req-set = {}; off-set = {}; priv-set = {};
    for each transition t of T
      A = start point of t; B = end point of t;
      t-cube = [A,B];

      case (t)
        $0 \rightarrow 0$ transition:
          add t-cube to off-set;
        $1 \rightarrow 1$ transition:
          add t-cube to req-set;
        $1 \rightarrow 0$ (or $0 \rightarrow 1$) transition:
          add each maximal ON-set subcube to req-set;
          add each maximal OFF-set subcube to off-set;
          add t-cube and its start-point A to priv-set;
    return (req-set, off-set, priv-set).
```

Table 4.1: **Step 0:** Algorithm *Make-Sets*.

If p has no illegal intersections with any cube of priv-set, it is a dhf-implicant; it is placed in dhf-pi-set.

If p illegally intersects some privileged cube c in priv-set, then cube p is "split", or reduced, in every possible way by a single variable to avoid intersecting c. The reduced cubes are returned to tmp-set. In general, these reduced cubes may have *new* illegal intersections: a reduced cube, *p-red*, may illegally intersect a priv-set cube, c, even if p legally intersects c.

The algorithm terminates when tmp-set is empty. The resulting cubes in dhf-pi-set are all dhf-implicants. In addition, it is easily proved that the algorithm generates all dhf-prime implicants. Subcubes of other cubes in dhf-pi-set are removed; the result is the set of dhf-prime implicants.

As an optimization, we eliminate implicants that *contain no required cube*. If a dhf-implicant contains no required cubes, it can always be removed from a cover to yield a lower-cost solution. (Note that a dhf-prime implicant may intersect the ON-set and yet contain no required cube; see Experimental Results, Section 4.9.)

```
Algorithm PI-to-DHF-PI (pi-set, priv-set)
    tmp-set = pi-set; dhf-pi-set = {};

    while (not empty (tmp-set))
      remove a cube p from tmp-set;
      if (p has no illegal intersections with any cube of priv-set)
        add p to dhf-pi-set;
      else
                /* p illegally intersects a priv-set cube; */
                /* reduce p to avoid intersection */
        c = any cube of priv-set which p intersects illegally;
        for (each input variable v which appears as a don't-care
          literal in p and as literal $v$ or $v'$ in c)
          p-red = the maximal subcube of p where v is set
                to the complement of its value in c;
          add p-red to tmp-set;
    delete all cubes in dhf-pi-set contained in other cubes;
    return (dhf-pi-set).
```

Table 4.2: **Step 1:** Algorithm *PI-to-DHF-PI.*

Step 2: Generate DHF-Prime Implicant Table

A *dhf-prime implicant table* is constructed for the given function. The rows of the table are labelled with the *dhf-prime implicants* used to cover the columns. The columns are labelled with the *required cubes* which must be covered. The table sets up the two-level hazard-free logic minimization problem.

Step 3: Generate a Minimum Cover

The dhf-prime implicant table is solved in three steps, using simple standard techniques. More sophisticated techniques can also be applied [81, 7, 60].

First, *essential dhf-prime implicants* are extracted using standard techniques.

Second, the flow table is iteratively reduced. Rows and columns of the table may be removed using *row-dominance* and *column-dominance* operations, respectively. These operations may lead to further opportunities for *(secondary) essential dhf-prime implicant removal.* The operations are iterated until there is no further change.

Finally, if the table is still non-empty, a covering problem remains (*cyclic covering problem*). It is solved using an exhaustive algorithm called *Petrick's method.* Each column lists implicants which cover a required cube. The column is translated into a Boolean sum of rows; the covering problem for the table can be stated as a Boolean product of these sums. This product is multiplied out to generate all possible solutions. A minimum solution is then selected.

4.4 Hazard-Free Minimization Example

The Karnaugh map from figure 2.2 is reproduced in figure 4.1 (the function is slightly modified from ex. 3.4 of [35]). Set $T = \{t_1, t_2, t_3, t_4\}$ contains four *specified* function-hazard-free input transitions. Each transition t_i is described by a transition cube C_i with start point m_i:

$$\begin{array}{lll} t_1: & m_1 = ab'c'd & C_1 = ac' \\ t_2: & m_2 = ab'cd' & C_2 = ab'c \\ t_3: & m_3 = a'bc'd' & C_3 = a'c' \end{array}$$

$$t_4\text{:}\quad m_4 = a'bcd \qquad C_4 = c$$

The input transitions are indicated in figure 4.1(a). The start point of each transition is described by a dot, and its transition cube is described by a dotted circle.

Step 0: Make Sets

The req-set, off-set and priv-set are generated using Algorithm *Make-Sets*, as illustrated in figure 4.1(b).

t_1:	*req-cube-1*	=	ac'	t_4:	*req-cube-4*	=	$a'c$
t_2:	*off-cube-1*	=	$ab'c$		*req-cube-5*	=	bcd
t_3:	*req-cube-2*	=	$a'c'd'$		*off-cube-3*	=	acd'
	req-cube-3	=	$a'bc'$		*off-cube-4*	=	$ab'c$
	off-cube-2	=	$a'b'c'd$		*priv-cube-2*	=	c
	priv-cube-1	=	$a'c'$		*priv-start-2*	=	$a'bcd$
	priv-start-1	=	$a'bc'd'$				

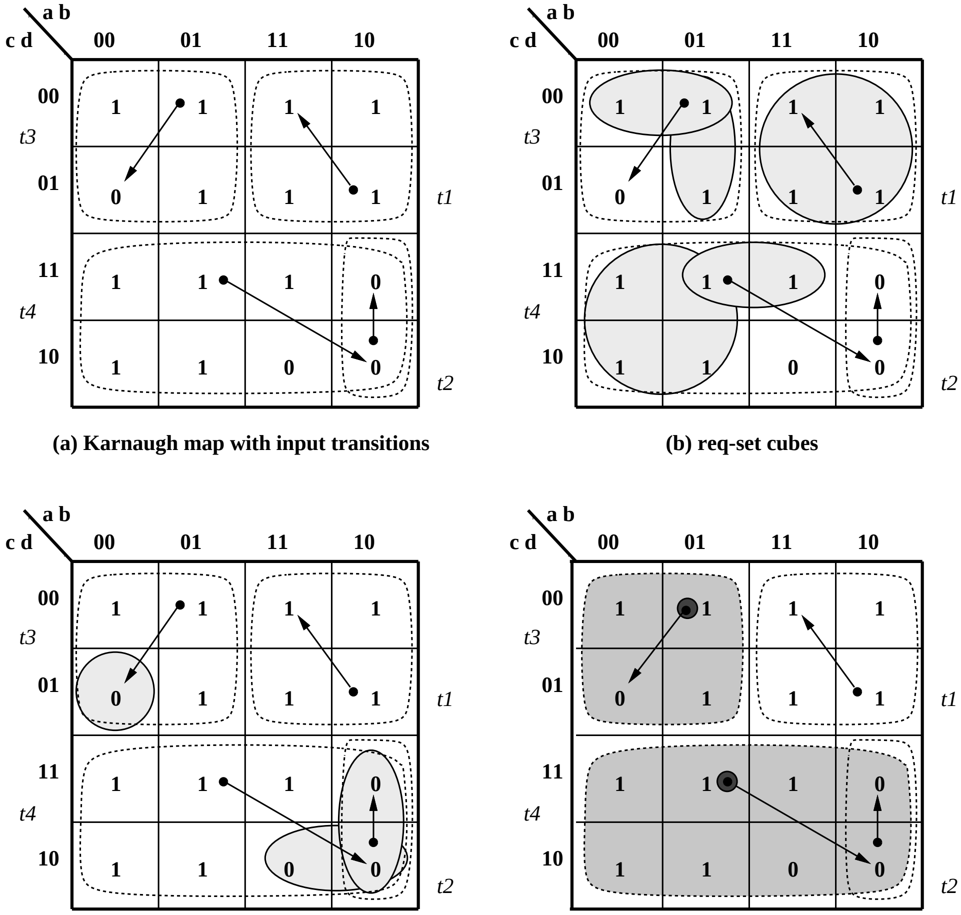
The final sets produced by the algorithm are:

$$\begin{aligned}\text{req-set} =\ & \{ac', a'c'd', a'bc', a'c, bcd\},\\ \text{off-set} =\ & \{ab'c, a'b'c'd, acd', ab'c\},\\ \text{priv-set} =\ & \{\langle a'bc'd', a'c'\rangle,\ \langle a'bcd, c\rangle\}.\end{aligned}$$

Step 1: Generate DHF-Prime Implicants

First, prime implicants are generated from the req-set and off-set:

$$\begin{aligned} p_1 &= c'd' & p_5 &= a'c\\ p_2 &= a'b & p_6 &= bd\\ p_3 &= bc' & p_7 &= a'd'\\ p_4 &= ac' \end{aligned}$$



(a) Karnaugh map with input transitions

(b) req-set cubes

(c) off-set cubes

(d) priv-set cubes

Figure 4.1: Hazard-free minimization example: **Step 0**.

The dhf-prime implicants are now produced using Algorithm *PI-to-DHF-PI.* The steps of the algorithm are illustrated in figure 4.2. Prime implicants p_1 through p_5 do not illegally intersect priv-set cubes *priv-cube-1* or *priv-cube-2*. As shown in figure 4.2(a), prime implicant p_1 intersects *priv-cube-1* and contains its start point. p_2 intersects both *priv-cube-1* and *priv-cube-2* and contains both start points. p_4 intersects neither priv-set cube. Similarly, p_3 and p_5 have no illegal intersections. These prime implicants are therefore dhf-prime implicants.

However, prime implicant p_6 illegally intersects *priv-cube-1*, since it intersects the cube ($bd \cap a'c' \neq \phi$) but does not contain its start point ($a'bc'd' \notin bd$; see Figure 4.2(b)). The algorithm splits p_6 into two subcubes: $p_{61} = bcd$ and $p_{62} = abd$ (see figure 4.2(c)). Cube p_{61} has no illegal intersections. However, p_{62} illegally intersects *priv-cube-2* (even though p_6 legally intersects *priv-cube-2*; see Figure 4.2(b)). Cube p_{62} is again reduced to $p_{621} = abc'd$, which has no illegal intersections (see figure 4.2(d)).

Similarly, prime implicant p_7 illegally intersects *priv-cube-2*, since $a'd' \cap c \neq phi$ and $a'bcd \notin a'd'$ (see Figure 4.2(e)). Cube p_7 is reduced to $p_{71} = a'c'd'$, which has no illegal intersections (Figure 4.2(f)).

The resulting set of dhf-implicants is:

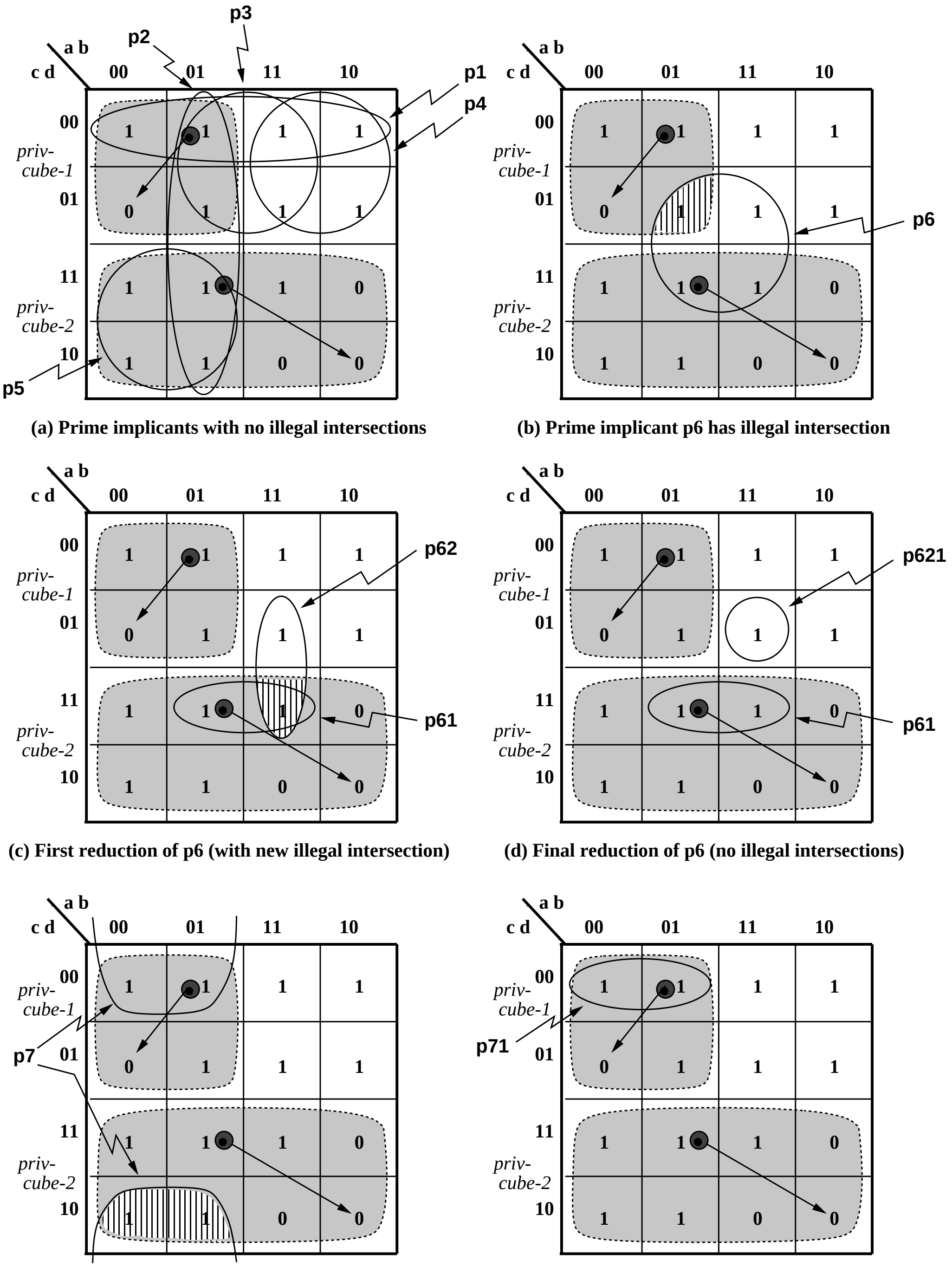
$$\{p_1, p_2, p_3, p_4, p_5, p_{61}, p_{621}, p_{71}\}.$$

After deleting cubes contained in other cubes, the final set of dhf-prime implicants is:

$$\{p_1, p_2, p_3, p_4, p_5, p_{61}\}.$$

Step 2: Generate DHF-Prime Implicant Table

The dhf-prime implicant table for the example is shown in Figure 4.3. The columns contain the required cubes generated in Step 0; the rows contain the dhf-prime implicants generated in Step 1.



(a) Prime implicants with no illegal intersections

(b) Prime implicant p6 has illegal intersection

(c) First reduction of p6 (with new illegal intersection)

(d) Final reduction of p6 (no illegal intersections)

(e) Prime implicant p7 has illegal intersection

(f) Final reduction of p7 (no illegal intersections)

Figure 4.2: Hazard-free minimization example: **Step 1**.

	required cubes				
dhf-prime implicants	ac'	$a'c'd'$	$a'bc'$	$a'c$	bcd
p_1=$c'd'$		X			
p_2=$a'b$			X		
p_3=bc'			X		
p_4=ac'	X				
p_5=$a'c$				X	
p_{61}=bcd					X

Figure 4.3: Hazard-free minimization example: **Step 2**.

Step 3: Generate a Minimum Cover

A minimum cover is generated for the dhf-prime implicant table. The essential dhf-prime implicants are: p_1, p_4, p_5, and p_{61}. Either p_2 or p_3 can be selected to cover the remaining uncovered required cube, $a'bc'$. The function therefore has two minimal hazard-free covers, each containing 5 products: $\{p_1, p_4, p_5, p_{61}, p_2\}$ and $\{p_1, p_4, p_5, p_{61}, p_3\}$.

The latter cover is shown in Figure 4.4(a). This cover is irredundant but *non-prime*, since it contains dhf-prime implicant p_{61} which is a proper subcube of prime implicant p_6.

A minimal but hazardous cover is shown in figure 4.4(b). The cover contains fewer products than the hazard-free cover, but has a logic hazard: prime implicant p_6 illegally intersects *priv-cube-1*. As a result, p_6 causes a dynamic hazard in the input transition, *t3*, corresponding to the privileged cube in Figure 4.4(b).

4.5 Existence of a Solution

For certain Boolean functions and sets of transitions, the hazard-free covering problem has no solution [91, 6]. In this case, the dhf-prime implicant table will include at least one required cube which is not covered by any dhf-prime implicant.

Example. We consider the function used in the previous section, but augment its set $T = \{t_1, t_2, t_3, t_4\}$ of specified input transitions with a new transition:

$$t_5: \quad m_5 = abc'd \qquad\qquad C_5 = abd$$

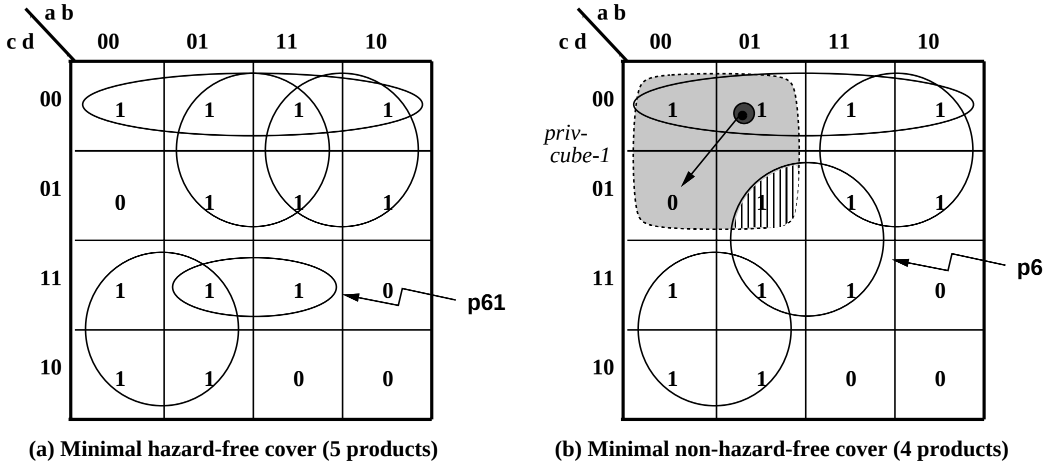

a b
c d
00
01
11
10
p61
priv-
cube-1
p6
(a) Minimal hazard-free cover (5 products)
(b) Minimal non-hazard-free cover (4 products)

Figure 4.4: Hazard-free minimization example: **Step 3**.

The input transitions are indicated in the Karnaugh map of Figure 4.5(a). The req-set now has an additional required cube: *req-cube-6* = abd. The off-set and priv-set are unchanged from the example of Section 4.4, and the function has the same dhf-prime implicants as well.

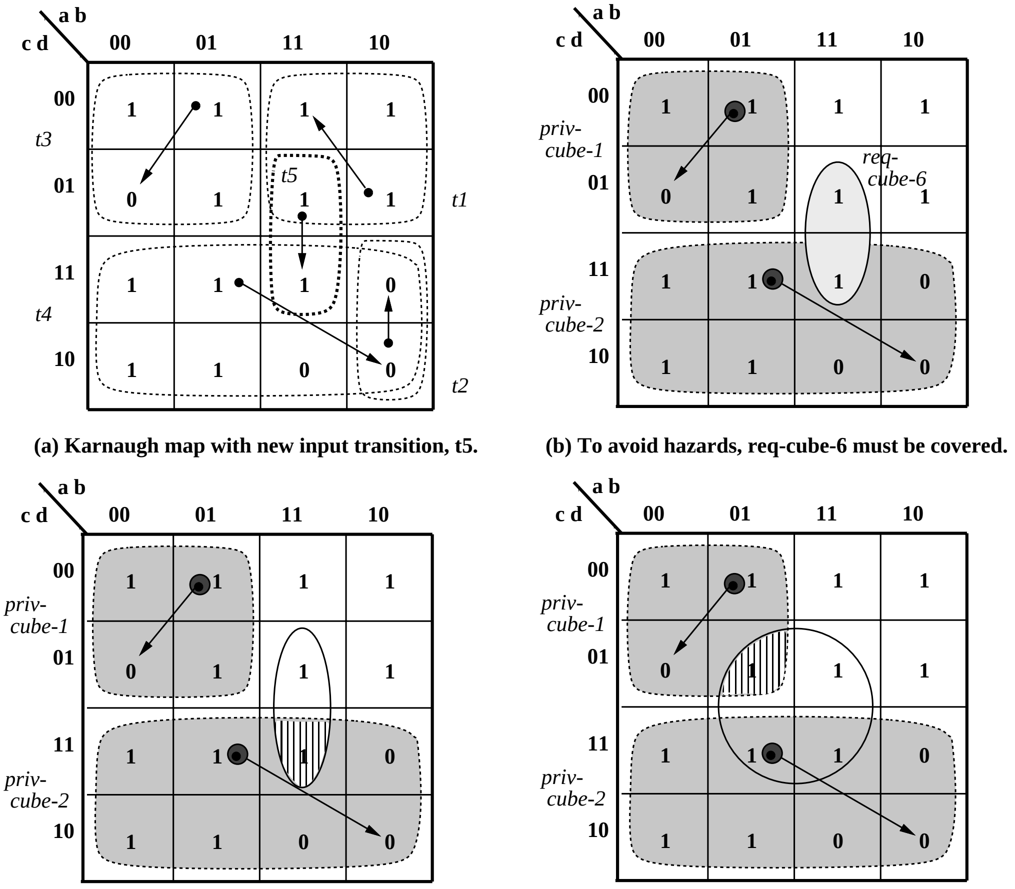
Figure 4.5(b)-(d) illustrates the covering problem. To insure no static hazard for transition *t5*, the required cube *req-cube-6* must be covered by some product. However, every product which contains *req-cube-6* also illegally intersects a privileged cube, *priv-cube-1* or *priv-cube-2*. That is, any attempt to eliminate the static hazard in transition t_5 will produce a dynamic hazard in one of the transitions, t_3 or t_4.

Table 4.3 shows the resulting dhf-prime implicant table. This table has no solution: no dhf-prime implicant contains required cube abd.

4.6 Comparison with Frackowiak's Work

It is useful to compare our approach with the related work of Frackowiak[35]. Frackowiak presents two hazard-free minimization algorithms for two-level implementations allowing multiple-input changes. The algorithms assume that functions are fully-specified.

Both algorithms eliminate dynamic hazards in specified transitions. However, the first method (*Algorithm A*) ignores static hazards. The second method (unnamed, but



(a) Karnaugh map with new input transition, t5.

(b) To avoid hazards, req-cube-6 must be covered.

(c) and (d): Every implicant which covers req-cube-6 has an illegal intersection.

Figure 4.5: Boolean function with no hazard-free cover.

	required cubes					
dhf-prime implicants	ac'	$a'c'd'$	$a'bc'$	$a'c$	bcd	abd
$p_1{=}c'd'$		X				
$p_2{=}a'b$			X			
$p_3{=}bc'$			X			
$p_4{=}ac'$	X					
$p_5{=}a'c$				X		
$p_{61}{=}bcd$					X	

Table 4.3: dhf-prime implicant table having no solution.

here called *Algorithm A'*) attempts to eliminate static hazards for *every* static transition, even if unspecified. Therefore results may be either hazardous (Algorithm A) or suboptimal (Algorithm A').

Example. The Karnaugh map of Figure 4.6(a) describes a fully-specified Boolean function. The function has four specified input transitions. Each transition t_i is described by its transition cube C_i and start point m_i:

$$\begin{array}{lll} t_1: & m_1 = a'bc'd' & C_1 = a'c' \\ t_2: & m_2 = a'b'cd' & C_2 = c \\ t_3: & m_3 = a'b'cd' & C_3 = a'd' \\ t_4: & m_4 = ab'c'd' & C_4 = ac' \end{array}$$

A minimum cover using Frackowiak's Algorithm A has 4 products (see Figure 4.6(b)). It is hazard-free for dynamic transitions t_2 and t_4, but has a static logic hazard for transition t_3.

A minimum hazard-free cover, using our method, is shown in Figure 4.6(c). The cover has 5 products and is hazard-free for every specified transition.[2]

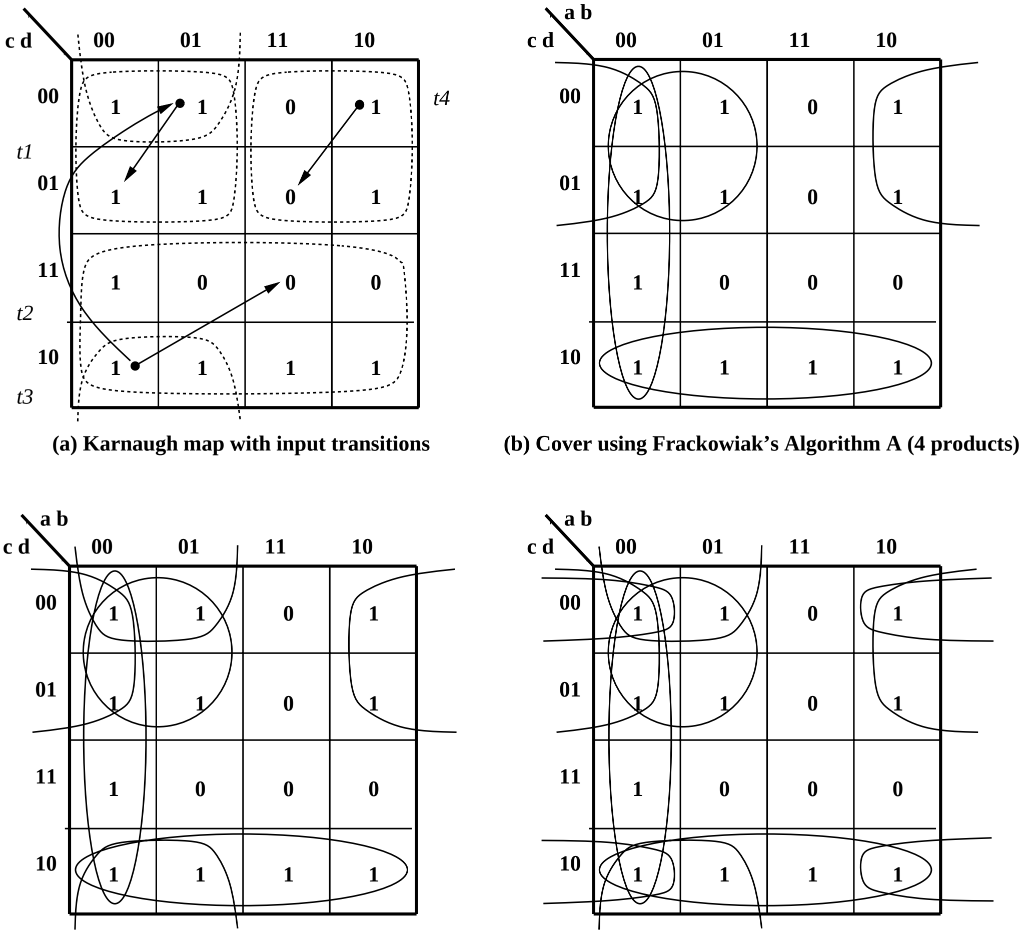
Finally, a minimum cover using Frackowiak's Algorithm A' is shown in Figure 4.6(d). The cover is hazard-free for every specified transition but has 6 products; it is therefore suboptimal. □

A final distinction between our work and Frackowiak, is that we allow incompletely-specified functions:

Example. The Karnaugh map of Figure 4.7(a) describes an incompletely-specified Boolean function. The function has six specified input transitions:

$$\begin{array}{lll} t_1: & m_1 = a'b'c'd & C_1 = a'c' \\ t_2: & m_2 = a'b'c'd & C_2 = b'c'd \\ t_3: & m_3 = a'b'cd & C_3 = a'c \\ t_4: & m_4 = abcd' & C_4 = abd' \\ t_5: & m_5 = abcd & C_5 = abc \end{array}$$

[2] Interestingly, this solution is prime but redundant, since it contains prime implicant *a'd'*. In contrast, the solution for the previous example (Figure 4.4(a)) was non-prime but irredundant.



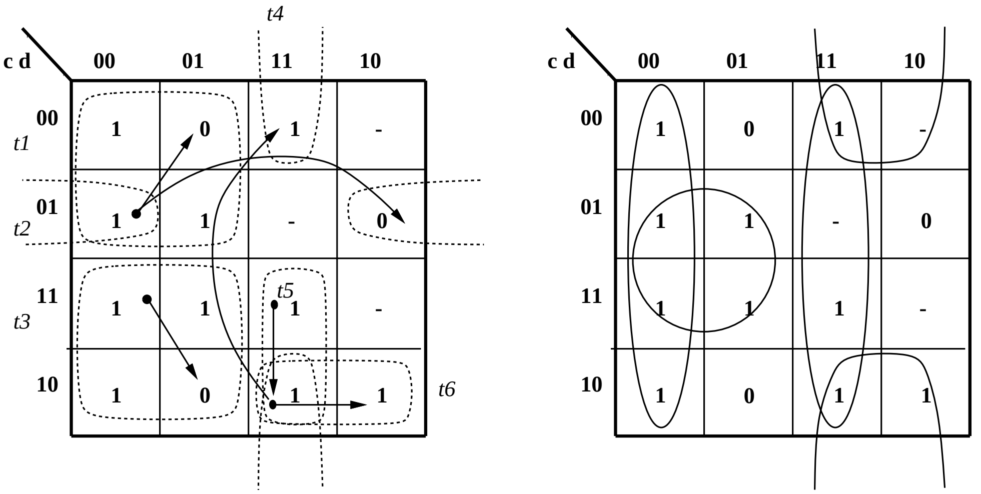
(a) Karnaugh map with input transitions

(b) Cover using Frackowiak's Algorithm A (4 products)

(c) Minimal hazard-free cover (5 products)

(d) Cover using Frackowiak's Algorithm A' (6 products)

Figure 4.6: Comparison with Frackowiak's method.



(a) Karnaugh map with input transitions

(b) Minimal hazard-free cover (4 products)

Figure 4.7: Hazard-free minimization of an incompletely-specified Boolean function.

$$t_6: \quad m_6 = abcd' \qquad C_6 = acd'$$

A minimum cover, using our method, is shown in Figure 4.7(b). The cover has 4 products and is hazard-free for every specified input transition. □

4.7 Burst-Mode Transitions

Section 2.6 introduced burst-mode transitions, which are a constrained class of function-hazard-free input transitions. In a burst-mode transition, an output value may change only at the endpoint of the transition, once all inputs have changed.

The requirements for hazard-elimination in burst-mode transitions are simpler than for arbitrary function-hazard-free transitions. In particular, Lemma 2.4 guarantees that in a sum-of-products implementation, any 0→1 burst-mode transition is always free of logic hazards. As a result, there is no need to describe a 0→1 burst-mode transition using a privileged cube.

This simplification can be incorporated into the logic minimization algorithms described in this chapter. In particular, algorithm *Make-Sets* of Section 4.3 can be simplified for the case of burst-mode transitions. By Lemma 2.4, only $1 \to 0$ transitions are now considered *privileged cubes*; remaining transitions are handled as before.

4.8 Program Implementation

We have implemented the logic minimization algorithms of Section 4.3. Our program is written in Lucid Common Lisp and is run on a DECStation 3100. However, it makes use of *espresso* [7, 81] to perform part of its computation: prime implicant generation. The advantage of this approach is that we can benefit from highly optimized existing tools.

The program generates sets for a function (*Step 0*) and writes the ON-set and OFF-set into a file in PLA format. We then use *espresso -Dprimes* to generate all prime implicants. The resulting PLA file is read in by the program, which computes the sets of dhf-prime implicants (*Step 1*). The program then constructs a dhf-prime implicant table and solves it (*Steps 2* and *3*).

This logic minimization program has been used as the the final component in the locally-clocked synthesis method for asynchronous controllers [72]. It has recently been incorporated into another synthesis method as well[106]. Both methods produce combinational functions which are guaranteed by construction to have hazard-free two-level implementations.

4.9 Experimental Results

Our hazard-free logic minimization program was run on a set of examples. The largest example is a cache controller having 20 inputs and 19 outputs (*dean-ctrl*) [71]. The program was also run on two SCSI controller designs (*oscsi-ctrl* and *scsi-ctrl*) [75].

Table 4.4 describes the results of Algorithm *PI-to-DHF-PI*. The algorithm transforms prime implicants into dhf-prime implicants. Prime implicants which contain only don't-care minterms are not included, since these implicants will never appear in an exact solution.

		prime implicants		dhf-prime implicants	
name	in/out	total	% illegal	total	% non-prime
dean-ctrl	20/19	1676	4	997	7
oscsci-ctrl	14/5	192	3	140	2
scsi-ctrl	12/5	280	1	190	2
pe-send-ifc	7/3	22	5	20	5
chu-ad-opt	4/3	6	0	4	0
vanbek-opt	4/3	7	0	6	0
dme	5/3	9	0	6	0
dme-opt	5/3	7	0	6	0
dme-fast	5/3	10	0	7	0
dme-fast-opt	5/3	15	0	14	0

Table 4.4: Results of Algorithm **PI-to-DHF-PI**.

Illegal prime implicants are those which illegally intersect some privileged cube, and therefore are not dhf-prime implicants. In every case, no more than 5% of the original prime implicants are illegal and must be further reduced.

After reduction, at most 7% of the dhf-prime implicants are not prime. It is also interesting that a number of prime implicants are discarded by the algorithm (see *dean-ctrl*). These implicants contain ON-set minterms but contain no required cubes. Since these implicants do not contribute to the hazard-free covering solution, they can be removed.

Table 4.5 presents the exact hazard-free solutions for the examples. It also gives an indication of the penalty associated with hazard elimination in our algorithms. In every case, the overhead for hazard-elimination is no more than a 6% increase in the number of products as compared with outputs synthesized using *espresso-exact* [81, 82].

Runtimes were quite reasonable for all examples tested. Even for the cache controller example, with 20 inputs and 19 outputs, total runtime was 83 seconds.

		Total Products			
name	in/out	Hazard-free Method	espresso-exact	% Over-head	Hazard-free Run-time(s)
dean-ctrl	20/19	215	202	6	83
oscsci-ctrl	14/5	59	58	2	9
scsi-ctrl	12/5	60	59	2	11
pe-send-ifc	7/3	15	15	0	1
chu-ad-opt	4/3	4	4	0	1
vanbek-opt	4/3	6	6	0	1
dme	5/3	4	4	0	1
dme-opt	5/3	4	4	0	1
dme-fast	5/3	5	5	0	1
dme-fast-opt	5/3	8	8	0	1

Table 4.5: Comparison of Hazard-Free Logic Minimization with *espresso-exact*.

4.10 Hazard-Free Multi-Level Logic

This chapter has focused on the synthesis of hazard-free two-level combinational logic. Two-level solutions are important for PLA implementations. However, for many applications, two-level realizations are inappropriate. The more general problem to be solved is to produce hazard-free *multi-level* logic for a given Boolean function and set of specified transitions.

Hazard-free multi-level logic can be generated from hazard-free two-level logic using *hazard-non-increasing* multi-level logic transformations. Multi-level transformations which introduce no hazards into a combinational network are described by Unger [91]. Recently, this set of transformations has been significantly extended by Kung [45].

The final step in logic synthesis is technology mapping, or the mapping of combinational logic to components in a given cell library. Traditional technology mapping may introduce hazardous behavior into a logic network. Algorithms must be used which insure that the mapping of the circuit into library cells does not introduce hazards. Such technology mapping algorithms have been developed by Siegel, et al. [84].

4.11 Conclusions

This chapter considers the two-level hazard-free minimization problem for several reasons: the general problem has not previously been solved; minimum two-level solutions are important for optimal PLA implementations; and these solutions serve as a good starting point for hazard-non-increasing multi-level logic transformations.

We have described the problem of implementing hazard-free two-level logic as a constrained covering problem on Karnaugh maps. We presented an automated algorithm for solving the two-level hazard-free logic minimization problem and showed its effectiveness on a set of examples.

An important feature of the algorithms is that they involve only *localized* changes to existing algorithms. As a result, we can use existing sophisticated algorithms for prime implicant generation (Step 1) and for table reduction and solution (Step 3).

With the automation of these algorithms, the basic automated locally-clocked synthesis system is complete. The algorithms have been incorporated into another synthesis system as well [106] and are applicable to a number of sequential synthesis methods.

Chapter 5

Design of a High-Performance Cache Controller

5.1 Introduction

In this chapter, we present a case study in large-scale, practical asynchronous design: the design of a high-performance asynchronous cache controller. The work integrates two distinct approaches in asynchronous system design: the design of controllers and the design of processor architectures. This chapter represents joint work with Mark E. Dean.

The goal of this chapter is to demonstrate the effectiveness of our synthesis method for the design of real-world systems. Most previous work by us [73, 72, 74] and others has focused on detailed algorithms for design methods and their application to fairly small examples. Furthermore, realistic quantitative comparisons between asynchronous and synchronous designs have been rare. For an asynchronous discipline to be widely accepted, however, it is critical to demonstrate that (a) large practical designs can be synthesized which (b) are superior, by some metric, to comparable synchronous designs.

This chapter therefore contains the following new contributions: (1) it demonstrates the feasibility of the proposed locally-clocked method for the design of a large real-world controller; (2) it demonstrates in particular how such a controller can fully support the asynchronous external interface of a new asynchronous RISC architecture; and (3) it presents a cache controller which is significantly faster than a comparable synchronous design.

Our state-machine specification and implementation are substantially more complex than other recent asynchronous examples (cf. [14, 19, 33, 47, 55, 65, 68, 96]). Our design has 16 primary inputs, 19 primary outputs, 4 state variables and 245 product terms in a sum-of-products implementation. The resulting cache performance using our controller is approximately twice as fast as an equivalent synchronous implementation.

5.1.1 Background and Previous Work

Most modern microprocessors and processor complexes use a global synchronizing clock to sequence through their operations. Global circuit synchronization simplifies the design and interfacing of digital logic structures while minimizing any pipeline sequencing overhead. But worst-case design constraints limit a synchronous system's ability to take full advantage of the available silicon performance. Synchronous operation and communication also restrict efficient data transfer between devices with differing processing rates or access methods. Although synchronous logic structures dominate the digital system industry, alternatives must be considered which extract more of the available silicon performance, and provide simple and efficient processing-rate-independent interfaces.

Several asynchronous processor designs have been proposed and/or implemented [24, 30, 58, 94, 11, 37]. Traditional self-timed design styles increase the implementation complexity and sequencing overhead for most processor-pipeline data structures. The most implementation-efficient self-timed sequencing structure for processor pipelines is *dynamic clocking* [30]. Dynamic clocking sequences the pipelined functional units in lock-step, but adjusts each cycle's period to match the present environmental conditions, process parameters and pending pipeline operations. This lock-step operation allows traditional synchronous logic structures to be used. Because of its efficiency and simplicity, we designed our processor complex around the dynamically clocked RISC processor (*STRiP*) developed at Stanford University.

In this chapter, we consider an asynchronous cache controller implementation designed to interface with a self-timed processor such as STRiP. Because of the high-performance and simplicity of locally-clocked state machines, we design a locally-clocked implementation.

The chapter is organized as follows: Section 5.2 describes the basic STRIP processor architecture and its external interface. Section 5.3 gives an overview of the protocol and datapath for a second-level cache subsystem for the STRiP architecture. Section 5.4 describes a detailed specification and implementation of the cache controller using the locally-clocked design method, as well as performance results. Section 5.5 presents conclusions.

5.2 STRiP, a Self-Timed RISC Processor

5.2.1 Basic Structure

A processor's throughput is determined by the functional units with the worst-case operational latency. A synchronous processor's performance is limited by the slowest pipeline operation and the worst-case temperature, voltage and process. This limitation holds even if the slowest operation is seldom used and the processor is operated in a nominal environment. Ideally the global synchronizing clock would adapt, from cycle to cycle, to the pending pipeline operations and the actual environmental conditions. By using an adaptive pipeline sequencing method called *dynamic clocking*, a self-timed RISC processor called STRiP was developed [30].

Dynamic clocking is a pipeline sequencing method which is best described as a self-timed, synchronous structure. All pipelined functional units sequence in lock-step via a global sequencing signal. This signal's period adapts, on a cycle-by-cycle basis, to the environmental conditions, process parameters, and pending pipeline operations. The pipeline sequencing signal also stops and waits for operations with indeterminate delays to complete (external memory and I/0 transfers). This operating characteristic supports a *fully asynchronous external interface*. The main goal of dynamic clocking is to provide a sequencing and interface method which uses synchronous logic elements while providing the interface efficiency and performance available through self-timing. Therefore, STRiP is a self-timed processor containing the *implementation simplicity* of a synchronous design and the *adaptive operation*, *robust interfaces*, and *wide operating range* of asynchronous designs.

5.2.2 Interface and System Overview

STRiP's Bus Interface Unit (BIU) connects it to the other processor complex and system devices. It is important that this interface minimize the overhead required for external data transfers. All external data transfers between the second-level cache(s), system memory, coprocessors, and main system bus use a fully asynchronous handshaking protocol. This type of interface allows devices and subsystems of different operating speeds to communicate easily and efficiently across a common bus. Often a processor's processing and data transfer rate is different from an external memory device's data access time and transfer rate. A synchronous interface must insert extra delay (a discrete number of cycles) to synchronize the communication between these two dissimilar devices. Dynamic clocking eliminates the synchronization overhead and potential metastable conditions common in synchronous interfaces. The basic organization of a single-processor subsystem is shown in Figure 5.1.

5.3 Second-Level Cache: Overview

5.3.1 Cache Specification

The majority of the processor's external data transfers are to and from the external cache subsystem. The second-level cache's structural characteristics are based on typical RISC system organizations. The subsystem typically contains a cache controller, tag RAMs, and data RAMs. Figure 5.2 gives a block diagram of the second-level cache subsystem and its primary interface signals. The RAMs used in most high-performance cache subsystems are SRAMs. Available SRAM densities support second-level cache sizes between 512KB to 2MB. The signals generated by the cache controller are designed to support this range of cache memory. The cache is direct-mapped with a copy-back write-allocate write policy. The line, fetch, and transfer sizes are all 128-bits. The organization shown in Figure 5.2 assumes that the processor contains the write-back buffers for the second-level cache and controls all request to the main memory subsystem.

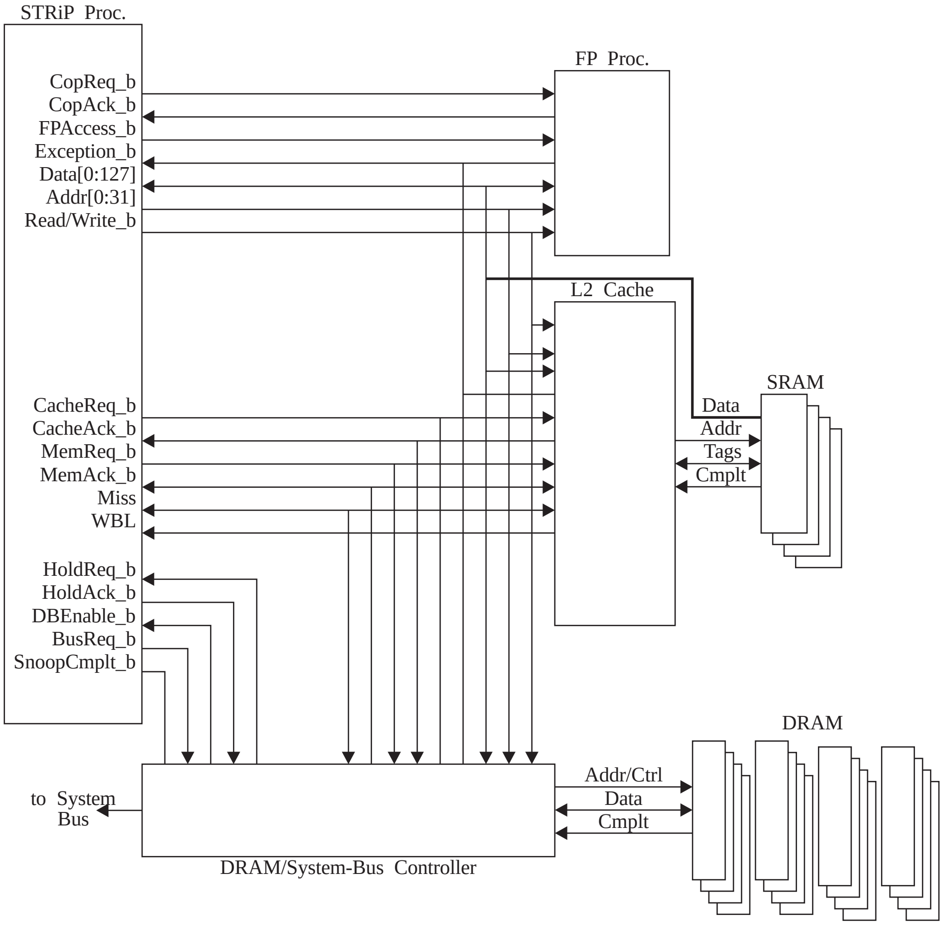


Figure 5.1: Processor complex block diagram.

5.3.2 Protocol and Signal Timings

There are several types of transfers which the second-level cache must support or monitor. These include: a cache read request, a cache write request, a main-memory read request (caused by a second-level cache miss), a main-memory write request (caused by a second-level cache write-back cycle), a read snoop, a write snoop, and a cache flush/reset request. The processor generates all requests to the external memory system, and it controls and monitors the second-level cache operation during transfers to and from main memory. Therefore, the processor's BIU generates the requests to support second-level cache miss and write-back cycles. Snoop cycles are required to support cache coherency and are generated by the main-memory subsystem. Cache flush/reset

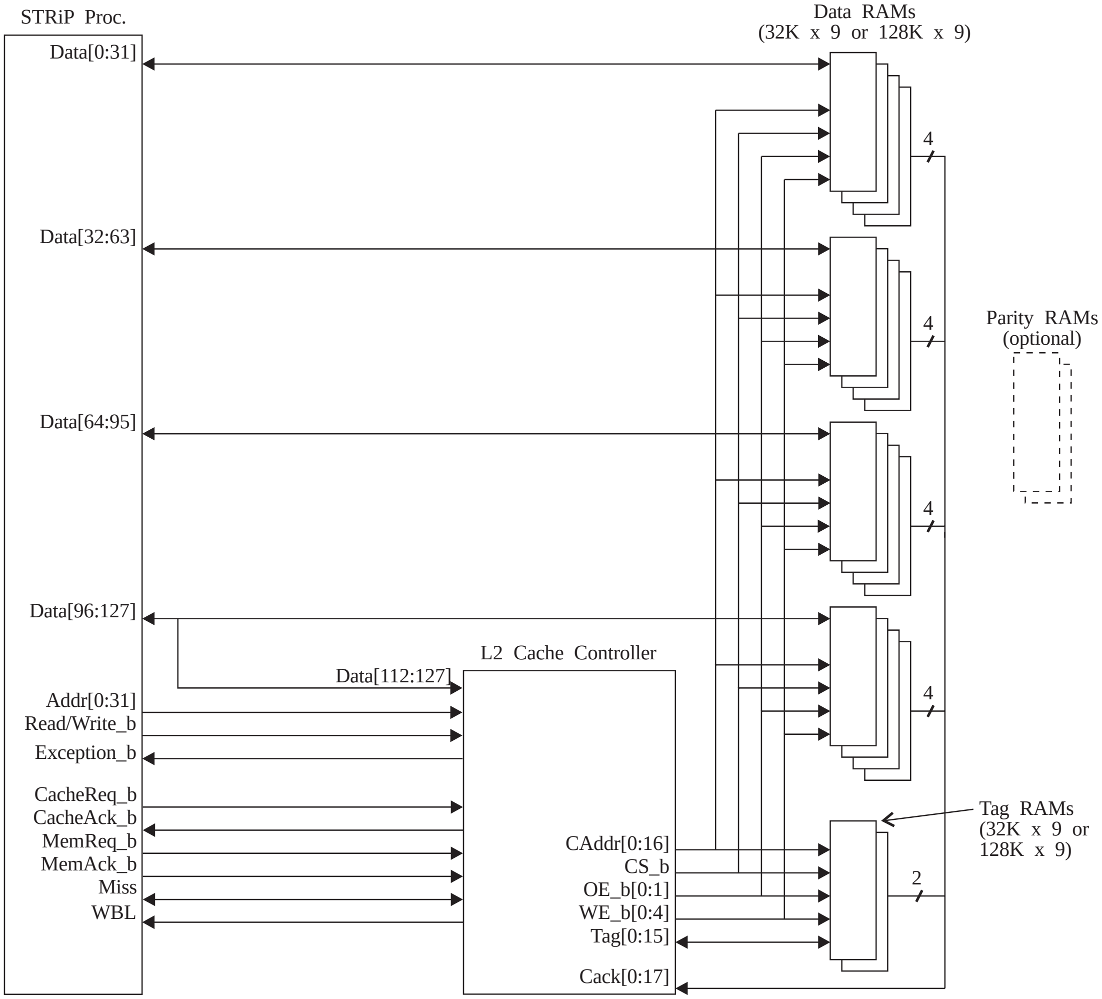


Figure 5.2: Second-level cache block diagram and primary interface signals.

cycles are controlled by the processor via an I/O port bit.

As an example of the asynchronous interface protocol and its relationship to the processor's internal dynamic clock, we describe the signalling for a second-level cache read- and write-hit transfer. Figure 5.3 illustrates the external signal timing supporting an *external read* (caused by a internal cache miss) followed by an *external write* (caused by an internal copy-back request).

The numbered transitions in Figure 5.3 are described below.

External Read Hit:

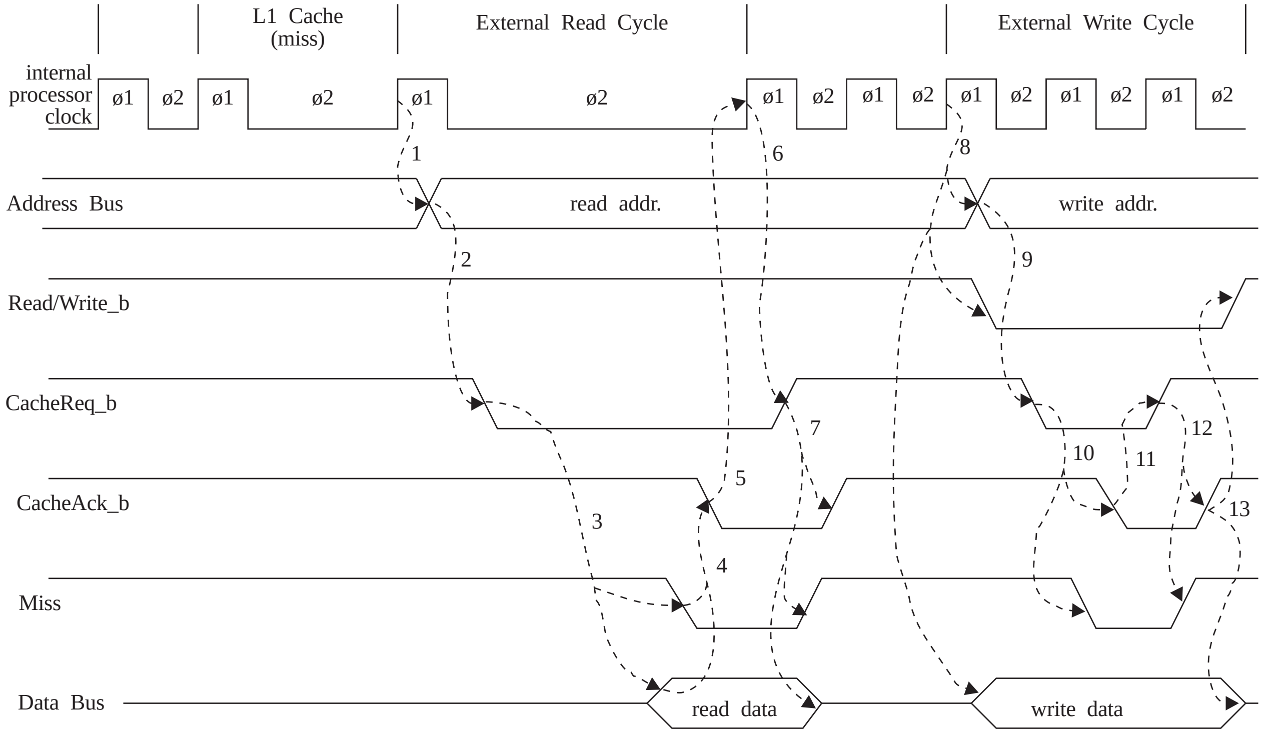


Figure 5.3: Timing diagram showing generic external read and write cycle signalling.

1. The processor's BIU drives the miss address onto the **Address Bus** at the beginning of $\phi 1$. **Read/Write_b** is also driven high to indicate a read cycle. The second-level cache controller forwards these addresses to the tag/data RAMs.

2. The BIU drives **CacheReq_b** active after the **Address Bus** and **Read/Write_b** are valid. [**Address Bus** and **Read/Write_b** setup to **Request** active is 0ns (min.)].

3. **CacheReq_b** causes a second-level cache access. At this point the cache controller generates the proper chip-select and output-enable signals to the tag/data RAMs. These RAMs in turn drive the selected tags and data back to the cache controller, using a completion signal to indicated valid data. The data is forwarded to the **Data Bus**, parity/ECC checking occurs, and the tag is compared with the target address. If the accessed location is present and error-free, **Miss** is driven inactive.

4. Once the **Data Bus** and **Miss** are driven, the cache controller drives **CacheAck_b** active. [**Data Bus** and **Miss** setup to **CacheAck_b** active is 0ns (min.)].

5. The processor latches the **Data Bus** into the internal caches and target register on the rising edge of the internal clock (caused by **CacheAck_b** going active). **Miss** is also sampled to determine if the data is valid. On a cache hit the internal pipeline continues operation, independent of the BIU's external cycling. If a second-level cache miss occurs (**Miss** active), the **Data Bus** is latched by the cache controller in case a write-back cycle is required.
6. The end of the external read cycle causes the BIU to drive **CacheReq_b** inactive.
7. The cache controller drives the RAM's chip-selects and output-enables inactive. The tag/data RAMs indicate their deselection by driving their completion signals inactive. The cache controller drives **Miss** high and **CacheAck_b** inactive, indicating that the external bus has been released. [**Data Bus** released and **Miss** setup to **CacheAck_b** inactive is 0ns (min.)].

External Write Hit:

8. Once the BIU recognizes that the external data bus has been released, a first-level cache write-back operation can occur. The write-back address and data are driven onto the bus and **Read/Write_b** is driven low. An external write-back cycle can start relative to the inactive edge of an acknowledge signal or the beginning of an internal $\phi 1$ period.
9. **CacheReq_b** is driven active, starting the external write cycle. [**Address Bus**, **Data Bus**, and **Read/Write_b** valid to **CacheReq_b** active is 0ns (min.)].
10. To execute a write operation, the cache controller drives the tag/data RAM chip-selects and the tag RAM output-enables active and performs a tag compare once tag data is valid. Once the tag compare is completed, the cache controller drives **Miss** valid and **CacheAck_b** active. [**Miss** setup to **CacheAck_b** active is 0ns (min.)].
11. The BIU drives **CacheReq_b** inactive, indicating that a write-hit was detected. The cache controller pulses the write-enable signals to the RAMs, using the RAM completion signals for write-pulse timing.
12. The external cache controller drives **Miss** and **CacheAck_b** high. [**Miss** high to **CacheAck_b** inactive is 0ns (min.)].

13. The BIU drives **Read/Write_b** high and the **Data Bus** tri-state. [**Data Bus** tri-state to **Read/Write_b** high is 0ns (min.)].

5.3.3 SRAM Completion Detection

Our goal in developing the second-level cache controller is to support fully asynchronous interfaces. To support a fully asynchronous interface between the cache controller and SRAMs, we require an SRAM configuration which provides completion detection. 9-bit wide SRAMs are assumed in our second-level cache design. This allows eight bits to be used for data/tag information with one bit used to provide access completion detection. All access completion detect bits are written with a "0" and connected to a pullup resistor. When the SRAMs are deselected, the completion signals are pulled high. When the SRAMs are accessed, the completion signals are driven low with the same access time as the other SRAM data signals. This method of completion detection for commercial SRAMs only works for read cycles. The write-enable signals must be timed via an external delay line, satisfying the minimum write-enable pulse width. However, SRAM designs typically contain internal timing signals. These signals, if externalized, would provide completion signalling for both read and write operations.

A completion detection network which combines these individual completion signals can be formed using standard techniques [65].

5.4 Second-Level Cache Controller: Design

We now describe how the above cache protocol can be specified and implemented using our locally-clocked design method.

5.4.1 Signalling Issues

The cache protocol of the previous section includes both *transitions* on control signals (e.g. **CacheReq_b**) and *sampling* of data values (e.g. **Read/Write_b**, which may not transition on every cycle). However, our burst-mode specification style requires that every system event be a *transition*: only control signals are permitted.

In the case of data values, such as **Read/Write_b**, we must transform the single-rail data signal into a dual-rail pair of control signals: **RD** and **WR**. A sampling of the single-rail data signal is represented in a burst-mode specification by a transition of the appropriate dual-rail signal (which is later reset). After all data inputs have been transformed into dual-rail control signals, the cache protocol of Section 5.3.2 can be described by a burst-mode specification. (Note that many asynchronous specification styles have similar constraints, e.g. [12, 14, 19, 47, 55, 65, 67, 68, 96].)

Each data input can be implemented in hardware as a dual-rail pair of control inputs by simple gating of the input, and its complement, with a strobe signal. Such strobe signals are generated as outputs by the cache controller. (This technique is similar to one used by Martin and Burns [14, 55, 58].) To simplify the exposition, we do not include these strobe outputs in the specification described below.

5.4.2 Timing Issues

A locally-clocked implementation usually assumes that no new inputs arrive until the machine has stabilized after generating outputs [73]. That is, the machine operates correctly in fundamental mode, where the known delays in the environment are greater than the settling time of the state machine. In the case of the cache subsystem, the controller environment consists of the processor, cache SRAMs, and external memory controller. Delays of each of these components are known, and the environmental assumption is easily satisfied.

5.4.3 Controller Specification

We have described the protocols discussed in the previous section with a burst-mode specification. The specification has 16 primary inputs, 19 primary outputs, 38 states and 49 transitions. The specification supports protocols for: cache read, cache write, snoop read, and snoop write. Figure 5.4 describes the top-level specification where mode selection occurs. Figure 5.5 describes the cache read protocol and Figure 5.6 describes the cache write protocol. Both protocols include both cache hit and miss options; we describe only hit protocols in Figures 5.5 and 5.6. Snoop protocols are described by similar specifications.

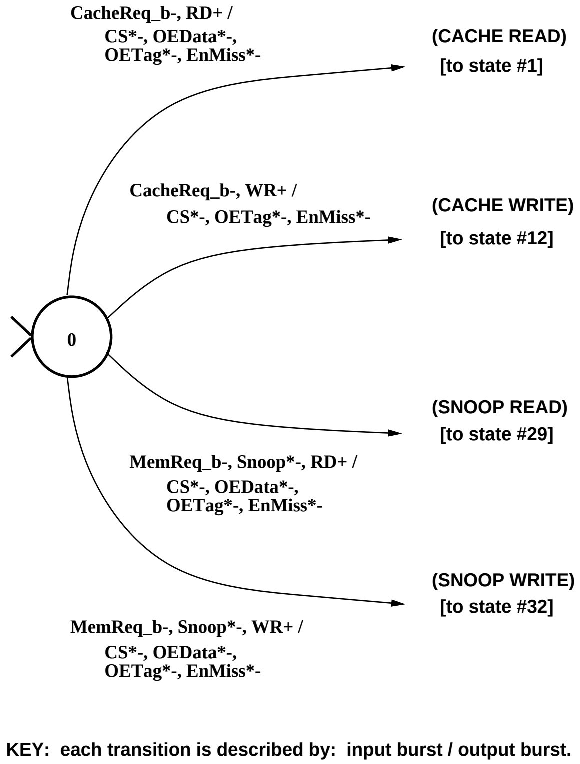


Figure 5.4: Top level of the cache-controller finite-state diagram where mode selection occurs.

Cache read and write hit protocols were described using the timing diagram in the previous section. We point out the key events in the corresponding burst-mode specification.

Cache Read Hit (Figure 5.5):

> *Transition 0→1:* The controller waits for the dual-rail read input and the cache access request from the processor (**CacheReq_b**). It then generates the appropriate outputs to initiate the cache read.
>
> *Transition 1→2:* Once the completion input signals have arrived from the cache, the controller generates acknowledge outputs to the processor. At this point, the processor may latch the read data and proceed with its operations.

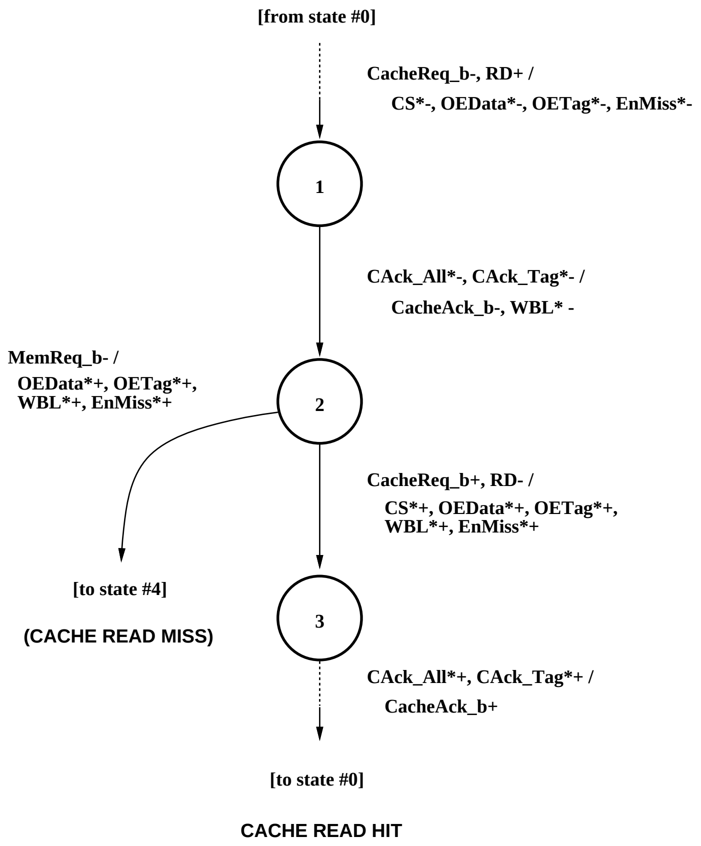


Figure 5.5: Cache-read-hit section of cache controller finite-state diagram.

Transition 2→3: The dual-rail read input is reset, and the processor read request (**CacheReq_b**) is deasserted. The controller deasserts cache (and other) signals.

Transition 3→0: The SRAM completion detection signals are deasserted; the controller then deasserts the remaining handshaking signal (**CacheAck_b**).

Note that the signalling convention of the controller on both of its interfaces is a 4-phase handshaking protocol. However, in practice the deassertion of the handshaking signals occurs *after* the processor has read the cache data. This signalling protocol effectively hides the overhead of the 4-phase handshaking and allows low-latency access to the cache. (Cf. similar hiding done by Alain Martin and Steve Burns in [14, 55, 58].)

Cache Write Hit (Figure 5.6):

Transition 0→12: This transition is similar to a read request but the controller waits for the dual-rail write input. It then generates the appropriate outputs to initiate a tag compare.

Transition 12→13: Once the completion input signal has arrived from the cache, the controller generates an acknowledge output to the processor and enables the bidirectional **Miss** signal to the processor.

Transitions 13→14,15,16,17: The processor indicates a write hit by deasserting **CacheReq_b**. Appropriate (dual-rail) address bits are used to determine the correct SRAM write address. The controller generates the appropriate cache write signals.

Transitions 14,15,16,17→18: When the SRAM completion detection signals are asserted, the write is complete. The controller then deasserts the cache write enable signals.

Transition 18→0: When the SRAM completion detection signals are deasserted, the controller deasserts the remaining cache signals as well as its acknowledge to the processor.

5.4.4 Resulting Implementation and Performance

The complete second-level cache controller for the STRiP processor has been designed using the locally-clocked synthesis method, by a combination of manual and automated techniques. Manual techniques were used for one step of the state minimization procedure. We are currently improving our automated state minimization algorithms to handle large examples. After state minimization, the machine requires 12 states; therefore four state bits are used. Sum-of-products logic equations were produced for the 19 output variables, four state variables and the local clock. The total number of products and literals for this implementation are given in Table 5.1. Our final logic-level implementation has 245 product terms, 886 literals, 23 phase-1 dynamic D-latches and four phase-2 static D-latches.

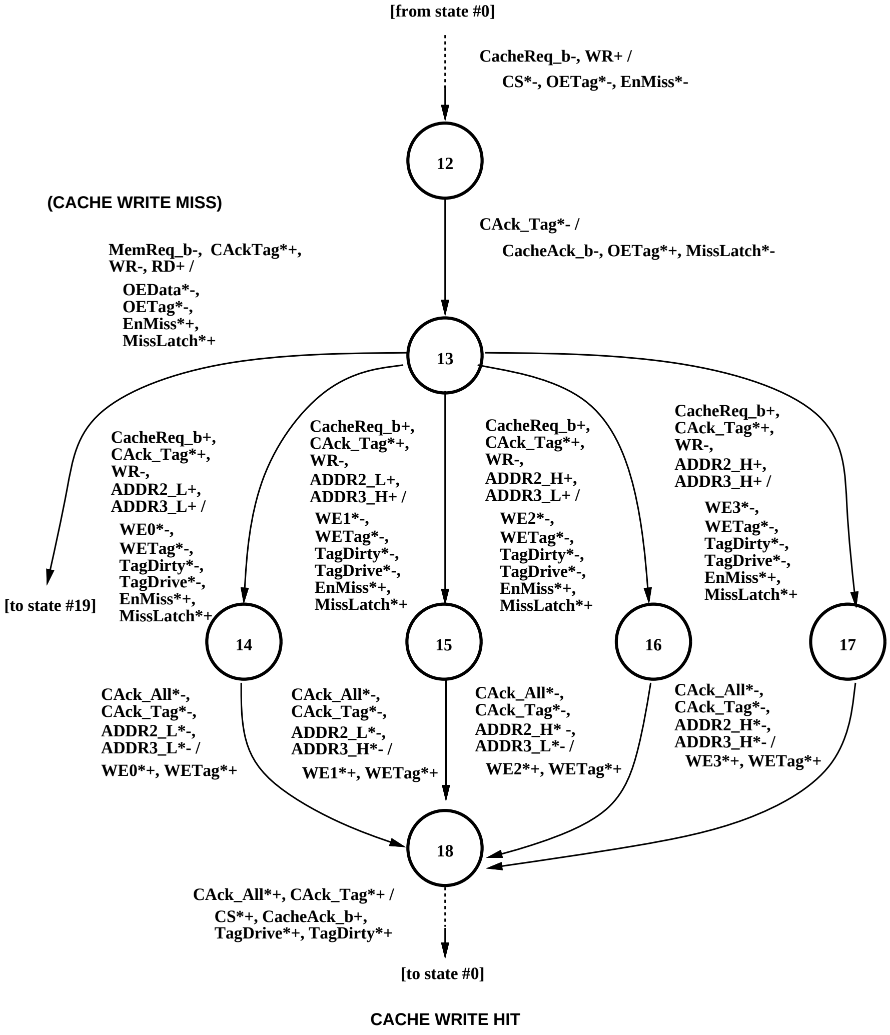


Figure 5.6: Cache-write-hit section of cache controller finite-state diagram.

An estimate of the area overhead of the asynchronous controller can be derived from Table 5.1. A comparable synchronous controller can be implemented using similar output and state logic, but requires no local clock. The local clock contains 6.9% (17/245) of the products and 12.5% (111/886) of the literals in our logic-level implementation.

Performance of the STRiP's asynchronous memory interface was analyzed assuming a 0.8μm CMOS process for the processor and second-level cache controller and an SRAM with a worst-case access time of 12ns (7ns nominal). Gate and latch delays in a STRiP processor implementation have been previously analyzed using Spice [30]. We assumed a full-custom implementation of the controller using a similar gate and process technology.

To determine the controller processing delay, we first identified the critical path for cache read access in the controller. We then manually mapped the critical logic into a simple multi-level network of full-custom gates (see Chapter 4.10) and estimated the corresponding path delay.

A breakdown of element delays controlling the second-level cache cycle time is given in Table 5.2. The internal clock's cycle period for a second-level cache read hit (including clock startup time) was 35ns, assuming nominal process, voltage, and temperature. If the second-level cache controller was included on the processor chip (additional pins required to support an external tag RAM) the cycle time is reduced to 20ns. Including the cache controller on the processor chip provides the highest performance, and advances in chip density and packaging allow this configuration to be practical for future designs.

A synchronous processor implemented in the same 0.8μm CMOS technology would have a maximum clock rate of approximately 66MHz. At this clock rate, assuming worst-case design constraints, four clock periods are required to access a synchronous cache subsystem equivalent to the proposed self-timed subsystem (three clocks if the cache controller is included on the processor chip). Therefore, the self-timed interface and controller provides approximately *twice the performance of an equivalent synchronous system* (35ns versus 60ns for an off-chip cache controller and 20ns versus 45ns for an on-chip cache controller). Much of this performance is gained through the ability of the self-timed system to take advantage of typical operating conditions and avoid discrete increments in cycle time.

Signals	**Number Required**	**Number of Products/ Literals**	**Number of Transparent Latches**
Primary Outputs	19	215/720	19
State Variables	4	13/54	8
Local Clock	1	17/111	0
Total	24	245/886	27

Table 5.1: Area Evaluation of Locally-Clocked Cache Controller.

Second-Level Cache Critical Logic Path Elements	**Async. Off-Chip Cache Typical Delay, 0.8μm CMOS (ns)**
Proc. off-chip driver	3
Cache Ctrl. on-chip driver	2*
Cache Ctrl. processing	5*
Cache Ctrl. off-chip driver	3*
SRAM typical access time	7
Cache Ctrl. on-chip driver	2*
Completion detection	2.5
Cache Ctrl. processing	2.5
Cache Ctrl. off-chip driver	3*
Proc. on-chip driver	2
Dynamic Clock Startup Time	3
Total	35

*Eliminated if second-level cache controller is implemented on processor chip (with external tag RAMs).

Table 5.2: Main elements in Second-Level Cache Critical Logic Path.

5.5 Conclusions

Our second-level cache controller design indicates the practicality of the locally-clocked synthesis method. The controller is large, compared with existing asynchronous controllers; it supports the full functionality of the second-level cache protocol; and it satisfies the interface requirements of the STRiP processor. Interestingly, the burst-mode specification style was shown to be sufficient and natural for describing a significant concurrent controller design. The features of generalized Petri nets, STGs, and parallel languages used in other competing design methods [12, 14, 19, 33, 47, 55, 65, 67, 68, 96] were not required. In addition, the design resulted in a performance improvement of approximately 100% over an equivalent synchronous implementation.

Chapter 6

Conclusions

We have presented a new method for the design of asynchronous state-machine controllers. The method is sound, has been automated and produces high-performance designs which are hazard-free at the gate-level.

The thesis has described the following results:

Burst-mode specifications. We introduced and formalized a new class of specifications for asynchronous controllers, called *burst-mode* specifications. Specifications are described by state diagrams and allow multiple-input changes. Traditional "multiple-input change" designs require inputs to change within a narrow window of time. In contrast, burst-mode specifications impose no timing constraints on inputs within a burst. The specifications typically assume a fundamental-mode of operation: no input burst can arrive until a system is stable from the previous burst.

Locally-clocked implementations. We introduced a new self-synchronized implementation style for the implementation of burst-mode specifications. Our approach was guided by two goals: correctness and performance. Unlike many existing methods, designs are hazard-free at the gate-level. In addition, realizations typically have the latency of their combinational logic.

Automated synthesis algorithms. We developed and automated a complete set of state and logic minimization algorithms. The logic minimization algorithm solves a general, previously unsolved problem, called the *two-level hazard-free logic minimization problem.* The algorithm finds a minimal cover of a Boolean function that is hazard-free for a given set of multiple-input changes, if such a solution exists. An interesting feature of our algorithms is that they require only small changes to existing synchronous algorithms. In particular, state minimization requires only a modification of the definition of "compatibility". Logic minimization generalizes the notions of "prime implicant" and "minterm" to new notions of "dhf-prime implicant" and "required cube", but otherwise sets up a standard covering problem.

Large example. Finally, we applied the method to a large, realistic example: a high-performance cache controller for a new self-timed RISC architecture. This controller is more complex than existing examples from the literature. The resulting asynchronous cache subsystem, using our design, is approximately twice as fast as a comparable synchronous subsystem.

Recent Developments

Some recent work builds on the results of the thesis, and extends it in new ways.

Yun *et al.* [106] have developed an alternative design method for the synthesis of burst-mode controllers. The method demonstrates that a burst-mode asynchronous controller can be designed without latches or a local clock. The resulting Huffman machines are called *3D machines*. These machines do not incur the area overhead of the local clock or the latches. In addition, there is a small gain in efficiency, since output changes do not need to pass through dynamic latches. However, unlike the locally-clocked method, the *3D* method requires two feedback cycles to implement a state change. Yun has developed a complete set of state minimization and assignment algorithms for the method [105]. However, he makes use of our logic minimizer, described in Chapter 4, to insure a hazard-free logic implementation. The method has been effectively applied to a number of design problems.

Aghdasi [1] has developed an alternative self-synchronized design method, using "data-driven clocks". He extends our work by partitioning the local clock into several clocks. Each clock controls an individual state variable. As a result, each clock may be smaller, and clocks need not pulse for every global state change.

A different extension to our work is proposed by Yun *et al.* [107]. This work points out difficulties in using burst-mode specifications to describe certain real systems. The work proposes two practical extensions to burst-mode. First, limited concurrency between inputs and outputs is allowed. And, second, specifications are generalized to describe the sampling of level signals. These extensions to burst-mode have been implemented in the *3D* design style.

Open Problems

A number of incremental improvements to the locally-clocked method are possible. In particular, algorithms for state minimization, state assignment and logic minimization can be further optimized. Here, though, we consider more fundamental open problems that need to be addressed in the future.

Noise. In an asynchronous design, noise on wires can cause a system to fail. This phenomenon holds in a synchronous design, except that the global clock limits the window of vulnerability to noise. It is important to explore the sensitivity of asynchronous designs to noise, and to develop techniques to avoid potential problems.

High-Level System Design. A burst-mode specification is useful in describing small- and medium-sized systems. However, it would be difficult to describe a large concurrent system using a state-machine specification. Translation methods [57, 10, 3, 95] have been more effective in describing such systems. However, Davis, Coates and Stevens [29, 26] have demonstrated that a large highly-concurrent system can be constructed from a collection of interacting asynchronous state machines. State machines are specified using data-driven specifications [26], which are related to our burst-mode style. However, the decomposition of the system behavior was performed manually. It is desirable to develop formal techniques to decompose a concurrent system into a collection of communicating state machines which can be described by burst-mode specifications.

Testing. Testing of asynchronous circuits is an underdeveloped area, which is critical to the acceptance of the designs. In principle, it should not be significantly harder to test locally-clocked machines than synchronous one-phase latch machines [60]. In particular, requirements for testing include the testing of combinational logic and the delay testing of various paths through the machine to insure that timing constraints are met. However, two-level hazard-free combinational logic may contain redundant or non-prime implicants, which make testing more difficult. Therefore, it is important to develop testing and design-for-testability techniques for hazard-free combinational logic.

Bibliography

[1] F. Aghdasi. Asynchronous state machine synthesis using data driven clocks. In *Proceedings of the 1992 European Design Automation Conference*, pages 9–14.

[2] F. Aghdasi. Synthesis of asynchronous sequential machines for VLSI applications. In *Proceedings of the 1991 International Conference on Concurrent Engineering and Electronic Design Automation (CEEDA)*, pages 55–59, March 1991.

[3] 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 587–91. IEEE Computer Society Press, November 1992.

[4] D.B. Armstrong, A.D. Friedman, and P.R. Menon. Realization of asynchronous sequential circuits without inserted delay elements. *IEEE Transactions on Computers*, C-17(2):129–134, February 1968.

[5] P.A. Beerel and T. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proceedings of the 1992 IEEE/ACM International Conference on Computer-Aided Design*, pages 581–586. IEEE Computer Society Press, November 1992.

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

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

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

[9] J.G. Bredeson and P.T. Hulina. Elimination of static and dynamic hazards for multiple input changes in combinational switching circuits. *Information and Control*, 20:114–224, 1972.

[10] E. Brunvand. Translating concurrent communicating programs into asynchronous circuits. Technical Report CMU-CS-91-198, Carnegie Mellon University, 1991. Ph.D. Thesis.

[11] 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.

[12] 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*, pages 262–265. IEEE Computer Society Press, November 1989.

[13] J.A. Brzozowski and J.C. Ebergen. Recent developments in the design of asynchronous circuits. Technical Report CS-89-18, University of Waterloo, Computer Science Department, 1989.

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

[15] S.M. Burns. Performance analysis and optimization of asynchronous circuits. Technical Report Caltech-CS-TR-91-01, California Institute of Technology, 1991. Ph.D. Thesis.

[16] S.M. Burns and A.J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. In J. Allen and T.F. Leighton, editors, *Advanced Research*

in VLSI: Proceedings of the Fifth MIT Conference, pages 35–50. MIT Press, Cambridge, MA, 1988.

[17] T.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.

[18] J.-S. Chiang and D. Radhakrishnan. Hazard-free design of mixed operating mode asynchronous sequential circuits. *International Journal of Electronics*, 68(1):23–37, January 1990.

[19] T.-A. Chu. Synthesis of self-timed vlsi circuits from graph-theoretic specifications. Technical Report MIT-LCS-TR-393, Massachusetts Institute of Technology, 1987. Ph.D. Thesis.

[20] T.-A. Chu. Automatic synthesis and verification of hazard-free control circuits from asynchronous finite state machine specifications. In *Proceedings of the 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 407–413. IEEE Computer Society Press, 1992.

[21] 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.

[22] W.A. Clark. Macromodular computer systems. In *Proceedings of the Spring Joint Computer Conference, AFIPS*, April 1967.

[23] B. Coates, 1992. Private communication.

[24] I. David, R. Ginosar, and M. Yoeli. Self-timed implementation of a reduced instruction set computer. Technical Report 732, Technion and Israel Institute of Technology, October 1989.

[25] A. Davis, B. Coates, and K. Stevens, 1990. Private communication.

[26] A. Davis, B. Coates, and K. Stevens. Automatic synthesis of fast compact self-timed control circuits. In *1993 IFIP Working Conference on Asynchronous Design Methodologies (Manchester, England)*, 1993.

[27] A.L. Davis. A data-driven machine architecture suitable for VLSI implementation. In C.L. Seitz, editor, *Proceedings of the Caltech Conference on Very Large Scale Integration*, pages 479–494, January 1979.

[28] Al Davis. The mayfly parallel processing system. Technical Report HPL-SAL-89-22, Hewlett-Packard Systems Architecture Laboratory, 1989.

[29] A.L. Davis, B. Coates, and K. Stevens. The post office experience: Designing a large asynchronous chip. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 409–418. IEEE Computer Society Press, January 1993.

[30] M.E. Dean. STRiP: A self-timed RISC processor architecture. Technical report, Stanford University, 1992. Ph.D. Thesis.

[31] M.E. Dean, D.L. Dill, and M. Horowitz. Self-timed logic using current-sensing completion detection (CSCD). In *Proceedings of the 1991 IEEE International Conference on Computer Design: VLSI in Computers and Processors*. IEEE Computer Society Press, October 1991.

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

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

[34] E.B. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM Journal of Research and Development*, 9(2):90–99, 1965.

[35] J. Frackowiak. Methoden der analyse und synthese von hasardarmen schaltnetzen mit minimalen kosten I. *Elektronische Informationsverarbeitung und Kybernetik*, 10(2/3):149–187, 1974.

[36] A.D. Friedman and P.R. Menon. Synthesis of asynchronous sequential circuits with multiple-input changes. *IEEE Transactions on Computers*, C-17(6):559–566, June 1968.

[37] R. Ginosar and N. Michell. On the potential of asynchronous pipelined processors. Technical Report UUCS-90-015, VLSI Systems Research Group, University of Utah, 1990.

[38] G. Gopalakrishnan and V. Akella. Specification, simulation and synthesis of self-timed circuits. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 399–408. IEEE Computer Society Press, January 1993.

[39] A. Grasselli. Minimal closed partitions for incompletely specified flow tables. *IEEE Transactions on Electronic Computers (Short Notes)*, EC-15(2):245–249, April 1966.

[40] 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.

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

[42] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

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

[44] M.B. Josephs and J.T. Udding. An overview of D-I algebra. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 329–338. IEEE Computer Society Press, January 1993.

[45] D.S. Kung. Hazard-non-increasing gate-level optimization algorithms. In *Proceedings of the 1992 IEEE/ACM International Conference on Computer-Aided Design*, pages 631–634. IEEE Computer Society Press, November 1992.

[46] 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.

[47] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 302–308. Association for Computing Machinery, June 1991.

[48] L. Lavagno, C.W. Moon, R.K. Brayton, and A. Sangiovanni-Vincentelli. Solving the state assignment problem for signal transition graphs. In *Proceedings of the 29th IEEE/ACM Design Automation Conference*, pages 568–572. IEEE Computer Society Press, June 1992.

[49] L. Lavagno and A. Sangiovanni-Vincentelli. Linear programming for optimum hazard elimination in asynchronous circuits. In *Proceedings of the 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 275–278. IEEE Computer Society Press, October 1992.

[50] A. Liebchen and G. Gopalakrishnan. Dynamic reordering of high latency transactions using a modified micropipeline. In *Proceedings of the 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 336–340. IEEE Computer Society Press, 1992.

[51] K.-J. Lin and C.-S. Lin. Automatic synthesis of asynchronous circuits. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 296–301. Association for Computing Machinery, June 1991.

[52] C.N. Liu. A state variable assignment method for asynchronous sequential switching circuits. *Journal of the ACM*, 10:209–216, April 1963.

[53] G. Mago. Realization methods for asynchronous sequential circuits. *IEEE Transactions on Computers*, C-20(3):290–297, March 1971.

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

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

[56] A.J. Martin. The limitation to delay-insensitivity in asynchronous circuits. In W.J. Dally, editor, *Advanced Research in VLSI: Proceedings of the Sixth MIT Conference*, pages 263–278. MIT Press, Cambridge, MA, 1990.

[57] A.J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C.A.R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Institute on Concurrent Programming, pages 1–64. Addison-Wesley, Reading, MA, 1990.

[58] A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, and P.J. Hazewindus. The design of an asynchronous microprocessor. In *1989 Caltech Conference on Very Large Scale Integration*, 1989.

[59] E.J. McCluskey. *Introduction to the Theory of Switching Circuits.* McGraw-Hill, New York, NY, 1965.

[60] E.J. McCluskey. *Logic Design Principles: with emphasis on testable semicustom circuits.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[61] P.C. McGeer and R.K. Brayton. Hazard prevention in combinational circuits. In *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences*, volume I, pages 111–120. IEEE Computer Society Press, January 1990.

[62] R.B. McGhee. Some aids to the detection of hazards in combinational switching circuits. *IEEE Transactions on Computers (Short Notes)*, C-18:561–565, June 1969.

[63] C. Mead and L. Conway. *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, Reading, MA, 1980. C.L. Seitz, System Timing.

[64] T. H.-Y. Meng, R.W. Brodersen, and D.G. Messerschmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(11):1185–1205, November 1989.

[65] T.H. Meng. *Synchronization Design for Digital Systems.* Kluwer Academic Publishers, Boston, MA, 1991.

[66] R.E. Miller. *Switching Theory. Volume II: Sequential Circuits and Machines.* John Wiley and Sons, New York, NY, 1965.

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

[68] C.W. Moon, P.R. Stephan, and R.K. Brayton. Synthesis of hazard-free asynchronous circuits from graphical specifications. In *Proceedings of the 1991 IEEE International Conference on Computer-Aided Design*, pages 322–325. IEEE Computer Society Press, November 1991.

[69] C. Myers and T. Meng. Synthesis of timed asynchronous circuits. In *Proceedings of the 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 279–284. IEEE Computer Society Press, October 1992.

[70] C.D. Nielsen and A. Martin. The design of a delay-insensitive multiply-accumulate unit. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 379–388. IEEE Computer Society Press, January 1993.

[71] 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. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 419–427. IEEE Computer Society Press, January 1993.

[72] S.M. Nowick and D.L. Dill. Automatic synthesis of locally-clocked asynchronous state machines. In *Proceedings of the 1991 IEEE International Conference on Computer-Aided Design*, pages 318–321. IEEE Computer Society Press, November 1991.

[73] 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*. IEEE Computer Society Press, October 1991.

[74] 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. IEEE Computer Society Press, November 1992.

[75] 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.

[76] Suhas S. Patil. An Asynchronous Logic Array. Technical Report Technical Memorandom 62, Massachusetts Institute of Technology, Project MAC, 1975.

[77] J.L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[78] M. Rem, J.L.A. van de Snepscheut, and J.T. Udding. Trace theory and the definition of hierarchical components. In Randal Bryant, editor, *Proceedings of the Third Caltech Conference on Very Large Scale Integration*, pages 225–239. CSP, Inc., 1983.

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

[80] L.Y. Rosenblum and A.V. Yakovlev. Signal graphs: from self-timed to timed ones. In *Proceedings of International Workshop on Timed Petri Nets, Torino, Italy*, pages 199–207. IEEE Computer Society Press, July 1985.

[81] R. Rudell. Logic synthesis for VLSI design. Technical Report UCB/ERL M89/49, Berkeley, 1989. Ph.D. Thesis.

[82] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(5):727–750, September 1987.

[83] C.L. Seitz. Asynchronous machines exhibiting concurrency. In *Conference Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, 1970.

[84] P. Siegel, G. De Micheli, and D. Dill. Technology mapping for generalized fundamental-mode asynchronous designs. In *30th ACM/IEEE Design Automation Conference*, June 1993. To appear.

[85] J. Sparso and J. Staunstrup. Design and performance analysis of delay insensitive multi-ring structures. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 349–358. IEEE Computer Society Press, January 1993.

[86] K.S. Stevens, S.V. Robison, and A.L. Davis. The post office - communication support for distributed ensemble architectures. In *Sixth International Conference on Distributed Computing Systems*, 1986.

[87] I.E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[88] M.A. Tapia. Synthesis of asynchronous sequential systems using boolean calculus. In *14th Asilomar Conference on Circuits, Systems and Computers*, pages 205–209, November 1980.

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

[90] Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing*, 1(4):197–204, 1986.

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

[92] S.H. Unger. Asynchronous sequential switching circuits with unrestricted input changes. *IEEE Transactions on Computers*, C-20(12):1437–1444, December 1971.

[93] S.H. Unger. Self-synchronizing circuits and nonfundamental mode operation. *IEEE Transactions on Computers (Correspondence)*, C-26(3):278–281, March 1977.

[94] S.H. Unger. A building block approach to unclocked systems. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 339–348. IEEE Computer Society Press, January 1993.

[95] C.H. van Berkel and R.W.J.J. Saeijs. Compilation of communicating processes into delay-insensitive circuits. In *Proceedings of the 1988 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 157–162. IEEE Computer Society Press, 1988.

[96] P. Vanbekbergen, F. Catthoor, G. Goossens, and H. De Man. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. In *Proceedings of the 1990 IEEE International Conference on Computer-Aided Design*, pages 184–187. IEEE Computer Society Press, November 1990.

[97] P. Vanbekbergen, B. Lin, G. Goossens, and H. De Man. A generalized state assignment theory for transformations on signal transition graphs. In *Proceedings of the 1992 IEEE/ACM International Conference on Computer-Aided Design*, pages 112–117. IEEE Computer Society, November 1992.

[98] Tom Verhoeff. Delay-insensitive codes – an overview. *Distributed Computing*, 3(1):1–8, 1988.

[99] T. Villa and A. Sangiovanni-Vincentelli. NOVA: state assignment of finite state machines for optimal two-level logic implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(9):905–924, September 1990.

[100] T.E. Williams. Self-timed rings and their application to division. Technical Report CSL-TR-91-482, Computer Systems Laboratory, Stanford University, 1991. Ph.D. Thesis.

[101] T.E. Williams and M.A. Horowitz. A zero-overhead self-timed 54b 160ns CMOS divider. *IEEE Journal of Solid-State Circuits*, 26(11):1651–1661, November 1991.

[102] A.V. Yakovlev. On limitations and extensions of STG model for designing asynchronous control circuits. In *Proceedings of the 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 396–400. IEEE Computer Society Press, October 1992.

[103] O. Yenersoy. Synthesis of asynchronous machines using mixed-operation mode. *IEEE Transactions on Computers*, C-28(4):325–329, April 1979.

[104] M.L. Yu and P.A. Subrahmanyam. A path-oriented approach for reducing hazards in asynchronous designs. In *Proceedings of the 29th IEEE/ACM Design Automation Conference*, pages 239–244. IEEE Computer Society Press, June 1992.

[105] 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*. IEEE Computer Society Press, November 1992.

[106] 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.

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