ASYNCHRONOUS CIRCUIT DESIGN

A CASE STUDY OF A FRAMEWORK CALLED ACK

by
Hans Jacobson

A thesis submitted to the faculty of
Luleå University of Technology
In partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Engineering
Luleå University of Technology
May 1996

Abstract

As design systems have grown in complexity and clock speeds are constantly increasing, several limitations to the conceptual framework of synchronous design have begun to be noticed. Some notable problems due to higher performance demand are clock skew, power dissipation, interfacing difficulties and worst case performance. It is therefore not a surprise that the area of asynchronous circuits and systems, which generally do not suffer from these problems, is experiencing a significant resurgence of interest and research activity. However, a number of important problems have to be solved before asynchronous design methods can be successfully transferred to the CAD industry. First, asynchronous circuit design should be based on high level synthesis methods that are based on standard HDLs, the same basis as used for synchronous circuits. Second, tools to synthesize asynchronous circuits should be capable of handling and generating efficient implementations for reasonably large designs, such as the ones found in high-level synthesis benchmarks. This requires a method that offers flexibility to use different signaling protocols, to decompose large centralized controllers and to take advantage of advances in standard logic synthesis. Third, where efficiency is critical, it should be possible to obtain customized complex-gate based circuits. Finally, in order to appeal to current VLSI CAD tool users, asynchronous high level synthesis tools should be available as part of existing CAD frameworks. In this thesis, a framework called ACK incorporating all these features is presented.

Acknowledgements

Many persons have contributed to make my master project at the University of Utah a memorable experience.

I would like to thank Glenn Jennings, my advisor at Luleå University of Technology, who has taught me most of what I know about digital design. His enthusiasm and willingness to work closely with his students has been an invaluable source of inspiration during my years of study.

I would like to thank Ganesh Gopalakrishnan for the great opportunity he has given me of doing research in asynchronous design. His guidance and enthusiasm for research has greatly helped me to gain knowledge in asynchronous design methodologies.

I would like to thank Prabhakar Kudva who besides being a great friend has taught me most of what I know of research in asynchronous design. His ability to sort out important ideas together with his patient explanations of the intricacies of asynchronous design has been a constant source of encouragement and has helped me greatly during the period of my project.

I would like to give special thanks to the great people at the Computer Science front office, and especially Colleen Hoopes, for all their help.

Erik Brunvand, Jens Sparsø , Chris Myers, Lüli Josephson, Bill Richardson, Ratan Namulasu, Robert Thacker, Wendy Belluomini, and Marshall Soares have all been great sources for interesting and fun discussions.

I would also like to thank Prabhakar Kudva and Sophia Kartsonis for being such good friends and for all the fun events they brought me to. Last but not least I would like to thank David and Laura Richins for being best friends and a family to me away from home.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The digital design methodologies of today are dominated by the synchronous style, where execution of functions in a machine are kept in lock-step by a central timing generator. This has not always been the case. In the early days of digital design a variety of design styles flourished. One of the dominant research areas during this time was in a particular style called asynchronous design. Asynchronous circuits are sequential circuits that do not require any central timing to coordinate their internal operations. During the 1950's and 60's, many computers and systems were built using this type of circuit. However, during the 70's, the interest in asynchronous design started to decline and had all but disappeared in the early 80's. The reason for this was the rapidly growing complexity of digital systems. Synchronous circuits offered simplicity in their discrete and deterministic behavior. Designers only had to make sure that the clock period was large enough for the system to reach a stable state before the next clock tick. Asynchronous circuits, on the other hand, required very detailed examination to ensure a proper behavior, a task that became too hard as system complexity increased.

However, with the advance of modern technology, system complexity and the demand for higher performance has revealed several inherent problems with the synchronous design style. Some of the more notable problems are clock skew due to high frequency operation, power dissipation due to clock distribution and high speed interfacing with the environment. Asynchronous circuits do not suffer from these problems and have therefore lately received renewed attention from researchers and designers.

1.1 Synchronous Design

Synchronous systems operates in what is called the *time domain*. The time in a synchronous circuit is quantified and discrete. Information is hindered to flow freely by state holding elements which let information propagate through only at the boundaries of a discrete time interval.

A system based on this approach consists of one or more subsystems which are surrounded by an environment with which they communicate. The information and state of the system is usually held in storage elements and is allowed to flow between those during the duration of a discrete time interval. In synchronous design the boundaries of this time interval are physically represented by the rising and falling edges of a *clock* signal generated by a global clock generator. These edges are used to trigger the storage elements to store the new information on their inputs. Since the information must be stable before storing it, the length of the time interval is determined by the worst case performance of the slowest operation of the system. Unfortunately this forces operations that are faster and thus complete early to idly wait for the next occuring clock edge.

Each subsystem is divided into a control part and datapath. The control part uses a finite state machine to describe the behavior of the subsystem and the datapath contains arithmetic operations and storage elements. The storage elements are often represented by registers in the form of flip-flops or transparent latches.

The new states and output information of the controllers are calculated from the current state and input signals by combinatorial logic residing between the registers. The idea is illustrated in Figure 1.1 where the synchronous finite state machine's current state is stored in the flip-flop waiting for the next clock tick to propagate the new state. The input and output signals also have Flip-flops (not shown in the figure) holding their values stable until the next clock tick when the new values are copied through.

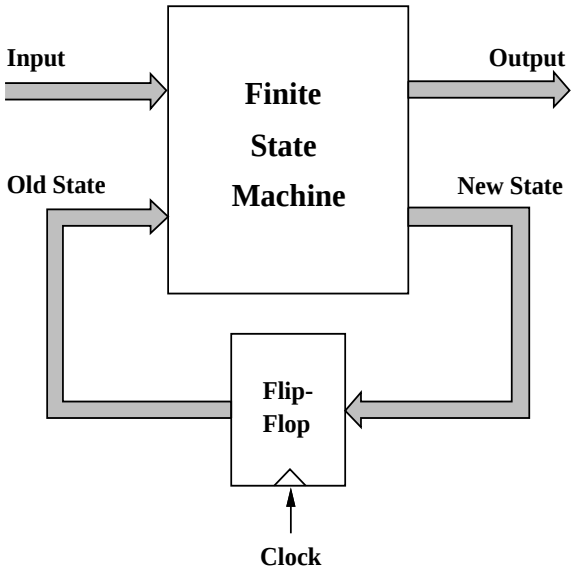


Figure 1.1: Example of synchronous controller implementation

As mentioned, the major reason for synchronous design being so popular stems from that it behaves in a discrete and deterministic way provided that some rules are met. These rules

include meeting setup and hold time for the registers, which requires matching the propagation delay of information flowing through the combinatorial logic accordingly. However, special techniques are required when information is passed between domains ruled by different clocks. Efficient and safe communication between such domains is in general a hard problem to solve with a synchronous design methodology.

1.2 Synchronous Limitations

When system complexity and clock speed increase, synchronous design has some inherent problems due to its way of keeping operations in lock-step execution. Some of the more obvious problems are presented below.

Clock Skew. Since all operations in a synchronous design are required to be kept in lock-step with each other, distribution of a global clock becomes a problem for high frequency systems. As circuits grow in complexity and size, the time for the clock signal to propagate to different parts of the system takes different amount of time resulting in a skew of the clock, which if too severe, may cause the circuit to malfunction. This problem can be solved by either slowing down the clock or building carefully balanced clock trees to minimize the skew but at the cost of performance degradation or increased area and design effort.

Power Consumption has become an important issue with the increasing portability of digital systems. Clock distribution over a whole circuit is a large source of power consumption since it is constantly driving the clock buffers, latches and combinational logic although no useful computation is done. This problem can be partly reduced by using gated latches and clock gating to locally power down the circuit, but results in increase of area and design effort.

Worst Case Performance. Since a synchronous circuit is driven with a constant clock rate, the clock period must be large enough to always comply with the worst case computation delay under worst case process, voltage and temperature conditions. The performance degradation of the system because of these restrictions is severe when compared to operation under nominal conditions [15].

External Inputs. Synchronous systems pose a problem when synchronizing with external inputs arriving at arbitrary times. If such a signal is sampled during a transition, we risk synchronization failure [43] leaving the circuit in a metastable state during which its outputs are undefined. Although metastability is usually resolved quickly, there is no bounded time for its duration. This is becoming a notable problem with increasing clock speeds. There is no known method to eliminate the problem of metastability although some methods to lower the probability exist.

Modularity. Exchanging a component in a synchronous environment ruled by the same clock require global changes to the environment in order to comply with the new component's worst case behavior. If the component is slower than the currently slowest part of the system, then the whole system must be slowed down accordingly. If the component is faster, other components in the system may still be slower and thus limit us from exploiting the full performance potential of the new component. Modularity simplifies system organization and increase the lifetime of a system. Unfortunately the modular design methodology is not easily incorporated in synchronous systems.

Composability. One solution to increase the throughput of a synchronous system is to use locally clocked components that execute at different clock speeds. However, synchronous systems have limited composability in that efficient lock-step communication, although feasible for heterochronous subsystems if the local clocks are derived from the same global clock (polyrythmic clocking [31]), can be hard to achieve.

In contrast, asynchronous design and circuits do not suffer from the above stated problems. Although asynchronous circuits have some problem of their own, their advantages make them an interesting and viable alternative and complement to synchronous circuits.

1.3 Asynchronous Design

Asynchronous systems operates in what is called the *event domain.* Unlike the synchronous style time has no direct meaning in asynchronous circuits in that it has no effect on when operations are executed. Instead, operations are invoked by *event signals* generated by the control logic.

By an event on a physical wire we mean that the signal on that wire is changing value. Events are an abstract representation since we do not care about the actual value of the signal (logic 1 or 0), only that it is changing. The information flows in an asynchronous system are controlled by these events when they actually occur, meaning we do not have to wait for any clock tick to occur before proceeding. This gives us the potential to exploit *average case delay* of the circuit, rather than worst case delay as for synchronous circuits.

Asynchronous circuits communicate via handshakes. A handshake consists of a series of signal events sent back and forth between the communicating elements. We can divide the communicating elements into a *sender* and a *receiver* part. The sender is the element that initiates the handshake sequence. If the sender wants the receiver to perform a certain task, it makes a *request* to the receiver. When the receiver has finished executing the task it make an *acknowledge* to the sender that the task has been completed. This is the way sequencing of actions is handled in asynchronous circuits - by handshake communications.

If two elements are to communicate in this fashion, they must be able to understand each other. It is therefore necessary to follow established handshake protocols. The most common of these are the two phase and four phase protocols which will be further discussed in Section 2.3.1.

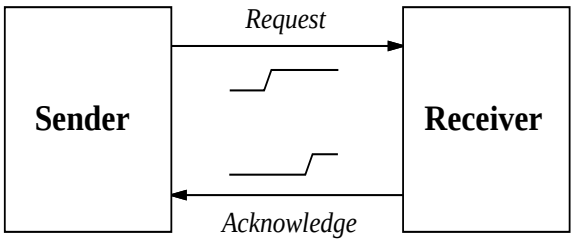


Figure 1.2: Example of a sender/receiver handshake

When elements are using the two phase protocol for communication, the sender starts by sending a request event to the receiver as illustrated in Figure 1.2. When the receiver has finished what it was requested to do, it sends an acknowledge event to the receiver. This completes the two phase handshake sequence. Note that if the request and acknowledge wires were initially set to zero, they will both be high after the handshake. The next time a handshake is made between the sender and receiver the wires will be set to zero again.

Unlike synchronous design where the control part of the circuit is always modeled as a finite state machine, the control can take on many different forms in asynchronous design. Some methods use syntax translation targeting restricted macro module libraries, some use a series of program transformations while others targets asynchronous finite state machines. These methods also differ in how delays in the circuit are viewed. Some methods allow unbounded but finite delays giving very robust circuits, while others impose timing constraints on the delays making them sensitive to process and runtime variations just as synchronous circuits.

1.4 Asynchronous Limitations

Although asynchronous circuits do not suffer from many of the problems found in synchronous circuits, they do have some problems of their own. The most important of these are discussed in the following paragraphs.

Hazards. Since asynchronous circuits rely on events on wires to communicate and sequence their order of execution, they are susceptible to glitches and hazards. Therefore special care must be taken during synthesis to eliminate the possibility of function and logic hazards. However, the resulting circuit is still sensitive to glitches caused by noise, ground bounces etc.

Handshake Latency. Due to their way of communicating via handshakes, asynchronous circuits have a handshake overhead that reduces performance. This penalty can be reduced by

placing communicating elements close to each other during place and routing.

Different Design Methodologies. There exist a wide variety of asynchronous design methodologies. Unfortunately, this results in inconsistent specification and implementation styles, making it difficult to make fair comparisons between systems. This also makes it hard to leverage off on existing research and algorithms.

Immature Synthesis Methodologies. For asynchronous design to be accepted as a viable option by synchronous designers and industry, there is need for mature synthesis methodologies. Unfortunately many proposed methods are still in their early stages of research and have not yet been demonstrated on large industrial designs.

Despite these problems asynchronous design is a viable complement to synchronous design. It is especially useful for applications requiring low latency operations and applications that can take advantage of average case delay. Low power applications is also an area where asynchronous circuits have an advantage. Many designs have been effectively implemented as asynchronous circuits, yielding better performance or power efficiency than their synchronous counterparts. Examples of large scale asynchronous designs are the post-office chip [14], an infrared computer communication chip [63], a SCSI interface chip [74], a high-performance cache controller [49], a DCC error corrector [48], a number of RISC processors [40, 15, 25], and a low power hearing aid just to name a few.

1.5 Scope of the Thesis

This thesis has two main objectives. (1) To introduce the reader to the basics of asynchronous design. (2) To present a new approach to high level synthesis of asynchronous circuits. There is a vast area of research covering widely different approaches to asynchronous design. This work will therefore be constrained to the following issues:

In the first part of this thesis the reader will be introduced to the most rudimentary and fundamental issues involved in asynchronous design. The thesis will present the basic communication principles used by asynchronous circuits. The main advantages and problems of asynchronous design will also be briefly discussed. The thesis will also make a short presentation of some of the methodologies most often used to represent control as well as datapath in asynchronous design and also the advantages and disadvantages of these. There are many subtleties of these methods that will not be discussed here since a thorough description of these would take up a whole thesis by itself. This first part should only be seen as a survey of asynchronous design. References are given locally in the text for the interested reader. The second part of this thesis will focus on presenting a methodology of asynchronous design and a framework developed at the University of Utah based on some of the techniques presented in the

first part. The techniques will mainly focus on methods for generating the control part of the circuit. This will include presentation of a behavioral Petri net representation for asynchronous design descriptions, a method for high level synthesis into refined controllers and allocation of datapath, partitioning of the controllers, translation of the controllers into burst mode state machines, synthesis of these state machines and a method for technology mapping of these to a network of customized complex gates. These techniques used in this second part will be described in more detail than those in the first part but the reader should be aware that it is not possible to cover all details of the methods due to limited space. References are therefore made where applicable.

1.6 Contribution of the Thesis

The contribution underlying this thesis is a new methodology for high level synthesis of asynchronous circuits. This work was to a major part carried out by Dr. Prabhakar Kudva, which the author joined at the later stages of implementation of the framework incorporating this methodology. Most of the techniques are the work of Dr. Kudva, and was carried out during his doctorate studies at the University of Utah.

The master project carried out by the author has included the following issues:

1. To learn the basics of asynchronous design methodologies. Some of the knowledge gained by this part of the work is presented in section 2 of this thesis.

2. To help with the further development of the synthesis framework. This included:

   (a) Gain full understanding of methodologies used in the system

   (b) Help develope new techniques and improve already existing techniques. This included participating in the development of a new technique for complex gate synthesis, a new technique allowing four phase refinement, a new technique for automating synthesis and interconnection, a new technique for contraction of controllers, improvement and formalization of a technique for partitioning controllers and a new approach of signal sharing between incompletely specified machines. These methods (except for the interconnection and contraction) will all be presented in this thesis.

   (c) Help deciding future extensions to the framework. Some of these are briefly presented in section 9.

   (d) Implement a number of benchmark circuits to evaluate the efficiency of the synthesis methodologies. This included taking the designs from high level specification and simulation to layout and simulation at the transistor level. Results for some of these circuits are given locally in the chapters of this thesis.

Additional parts of the framework, not directly a part of the authors work, will also be described to give the reader a better understanding of the complete synthesis flow. This include the burst mode synthesis section 7). The parts of the tool representing interconnection and contraction are left out to keep the size of the thesis managable.

The new methodologies of this work are presented in this thesis and include a new technique for representing behavioral design descriptions, a high level synthesis method for targeting partitioned asynchronous finite state machines that are incompletely specified and a state machine synthesis method targeting customized complex gates. These methodologies have been incorporated into a synthesis framework called ACK.

Chapter 2

Introduction to Asynchronous Design

Asynchronous circuits work very differently from synchronous circuits in their way of communicating and determining what parts of the circuit should compute. The conceptual framework in synchronous design is that of a global clock driving all activity in the circuit in a lock step fashion. As mentioned in section 1, the complexity and clock speeds in today's circuits is starting to show the inherent limitations of this concept. To overcome this problem, we need a new concept that does not suffer from these limitations. One such conceptual framework, that of asynchronous design, has lately shown promise in the design of high performance as well as low power circuits. This concept has promising features and advantages that motivates further research and development of its design techniques.

Asynchronous design features a wide variety of methodologies. Some of the main approaches will be briefly presented in this section, but the reader should be aware that there are many subtleties accompanying each individual style that will not be mentioned here since that would be beyond the scope of this thesis.

Subsection 2.1 will first discuss some of the advantages that motivate asynchronous design. Subsection 2.2 then briefly discusses the dividing of a design into separate control and datapath. Subsection 2.3 will thereafter present the communication style used by asynchronous circuits. The model for a complete circuit will then be discussed in subsection 2.4 including a presentation of the different timing models existing in asynchronous design. Subsection 2.5 will take up the issue of control and datapath interaction and is followed by a presentation of different asynchronous design specification and synthesis styles for control implementation in subsection 2.6.

2.1 Advantages of Asynchronous Design

Asynchronous circuits exhibits many interesting possibilities for effective implementations and ease of design.

The fact that made digital designers go synchronous in the 1970's, namely the overwhelming complexity of ensuring a proper behavior of asynchronous circuits laid on the designers back, is a thing of the past. In fact, ease of design may come to be one of the main advantages of asynchronous design. With the help of modern computers, much of the tedious and complex work of ensuring correct behaviors has been automated. Several asynchronous tools can generate a complete circuit layout from a behavioral specification, correct by construction, ready for fabrication without any intervention from the designer, e.g. [32]. Some tools can also mathematically prove that a design is correct before it is built, a capability that is very important as designs grow in complexity.

The modularity of a system will also play an increasingly important role in digital design. With the rapid progress in VLSI design comes accelerated research and development costs. Cost can be reduced by increasing the lifespan of the products by making incremental improvements, something asynchronous circuits automatically can take advantage of by running as fast as the individual modules allow it to. Such an approach would also result in a tremendous win in terms of time to market.

By removing the global clock and taking advantage of the low latency and average case delay, asynchronous circuits also show promise in performance compared to similar synchronous solutions. Removing the global clock also results in power savings as there is no clock tree or latches drawing power although they do no useful work. Instead, the distributed control of operation in an asynchronus circuit works as an automatic fine grained power down of inactive parts. If a circuit runs faster than it needs to, large power savings can also be made by dynamically scaling down the supply voltage [48, 11].

Some styles of asynchronous circuit design also generate very robust circuits. Such circuits do not depend on any delay constraints on wires or gates for correct operation. They are therefore well suited for operation under high variations in temperature and supply voltage.

2.2 Datapath and Control

As in synchronous design, the circuit is often divided into separate controllers with associated datapaths for modeling and efficiency reasons.

The datapath represents the circuitry implementing specified operators, such as adder or multiplication units. Registers and muxes to store data and direct the data flow also belong

to the datapath. Such circuitry is called a datapath resource. The datapath resources are usually allocated and synthesized into a gate netlist separately from the controllers since they use different synthesis methods.

The controllers represent the circuitry that controls the sequence of actions as specified in the design specification. The controllers act on the datapath resources and by communication let the resources know when they should perform their designated task. There are many ways of implementing the control circuitry which will be discussed later.

Unlike synchronous circuits where the datapath resources are passive entities, asynchronous datapath resources must contain some portion of control not only to be able to know when to perform their task but also to let the requesting controller know when they have completed it. This logic can range from a simple delay to a sophisticated completion detection mechanism. The synthesis method used for the datapath elements therefore also depends on which of the many styles of control implementation that is used.

Since asynchronous circuits communicate by passing event signals between each other, special circuitry is needed to translate level signals to event signals when we need to know a boolean value from the datapath. This function can be implemented by a *select* element which takes a level signal (from the datapath) and an event signal from a controller and generates an event signal on one of two outputs depending on if the level signal was high or low.

Once all datapath resources and controllers have been synthesized to gate level models they are connected to each other forming the whole specified design.

2.3 The Communication Style

Synchronous and asynchronous design are built on two completely different conceptual frameworks. Because of that, the communication styles, the way circuits communicate information between themselves, are quite different. In synchronous design, information propagates with each clock tick in a steady lock step fashion, whether it is needed or not. In asynchronous design, the information propagates through the circuit only when and where it is needed. Much like in synchronous circuits controllers decide *where* the information is needed, but unlike synchronous circuits there is no clock deciding *when* it is propagated. Instead, each controller decides for itself, through interaction with other controllers and the environment, when information under its control is supposed to propagate to other parts of the circuit. In synchronous design this could to some extent be seen as if each separate controller generates a clock tick *locally* to an individual part of the circuit when needed. Of course, the difficulty here would be to synchronize the information flow when controllers want to interact or more than one controller act on the same part. This is due to the clocks in this case being asynchronous with respect to each

other.

2.3.1 The Handshake Concept

The idea with asynchronous circuits is that different parts of the circuit are allowed to carry out their tasks at their own pace. However, in order to fulfill a useful purpose, the information flow between parts of the circuit must occur in a certain sequence. We therefore need to *synchronize* the actions to meet this sequence ordering. Since we have no global clock doing the job for us, we must introduce other means of synchronizing actions. The concept used in asynchronous design is that of *handshaking*.

The idea of handshaking is that the parts (controllers and datapath resources) of the circuit should be viewed as independent and self managing entities which through communication between themselves control the flow of information in the circuit. The communication between such entities are conducted via handshakes.

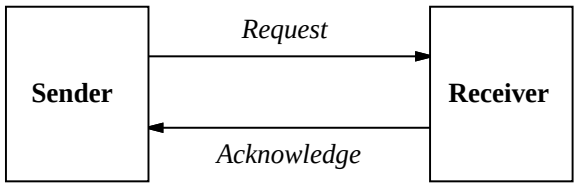
For instance, let's take the example illustrated in Figure 2.1 where an entity called the *sender* wants another entity, called the *receiver*, to carry out a computation. Underlined signals represent input signals. The handshake between these entities is then carried out in the following fashion. The sender initiates the handshake by sending a request to the receiver. When the receiver sees the request it carries out the requested computation. When the computation has finished the receiver sends an acknowledge to the sender. When the sender sees the acknowledge signal, it knows that the computation has completed and it can then carry on with the next task.

These request and acknowledge signals sent between the sender and receiver are physically represented by transitions, also referred to as *events*, on wires. Since all asynchronous circuits rely on events for communication, an important requirement is that transitions on wires change *monotonically*. If there is a glitch (if a signal goes $0 \rightarrow 1 \rightarrow 0$) on a wire, then that glitch may be seen as an event which may cause the circuit to enter a wrong state resulting in malfunction.

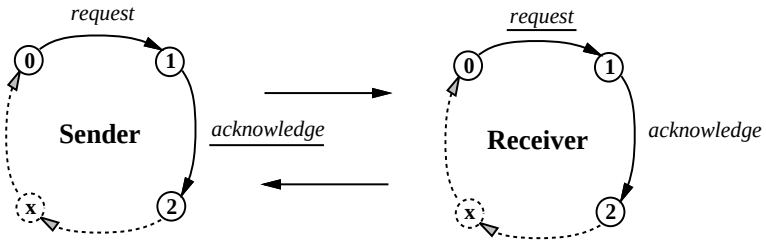
There are two main styles of signaling, *transition signaling* and *level signaling*. These styles are presented next.

2.3.2 Transition and Level Signaling

In transition signaling any transition on a physical wire, either rising or falling, has the same meaning, as can be seen in Figure 2.2. For a synchronous circuit there is a distinct difference between the levels and the rising and falling edges of a signal. In an asynchronous circuit using transition signaling, there is no distinction between different levels or the edges of the signals.



(a) Handshake at the structural level



(b) Handshake at the state machine level

Figure 2.1: Example of Handshaking Concept

The only thing that is important is that there is an event on the signal, that it changes value, not what it changes value to.

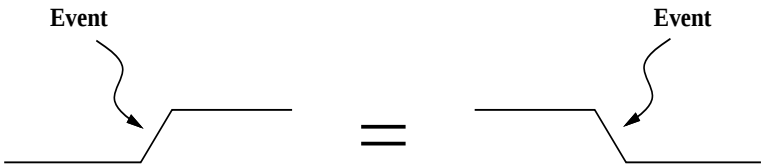


Figure 2.2: Two equivalent transitions

Since transition signaling is not concerned about the value of the signal, it is an abstract representation of the circuit as it will finally be implemented in hardware. Transition signaling is often referred to as the *two phase signaling protocol*.

In level signaling, as in synchronous circuits, we distinguish between the different transitions on a wire. When performing a handshake using level signaling the resources used, such as flip-flops, are often only sensitive to the high going transition (positive edge) or one value of the signal. It is therefore necessary to reset the wires to zero again after conducting the first phase of the handshake (when the signals go up to high values). Level signaling is often referred to as the *four phase signaling protocol*.

Figure 2.3 shows the two different signaling styles. For the two phase protocol only one event on each of the request and acknowledge wires is needed for a complete handshake. For the four phase protocol, we need two events on each wire. Since all computations are usually

finished after the first half of the handshake, the returning of the wires to zero results in an overhead.

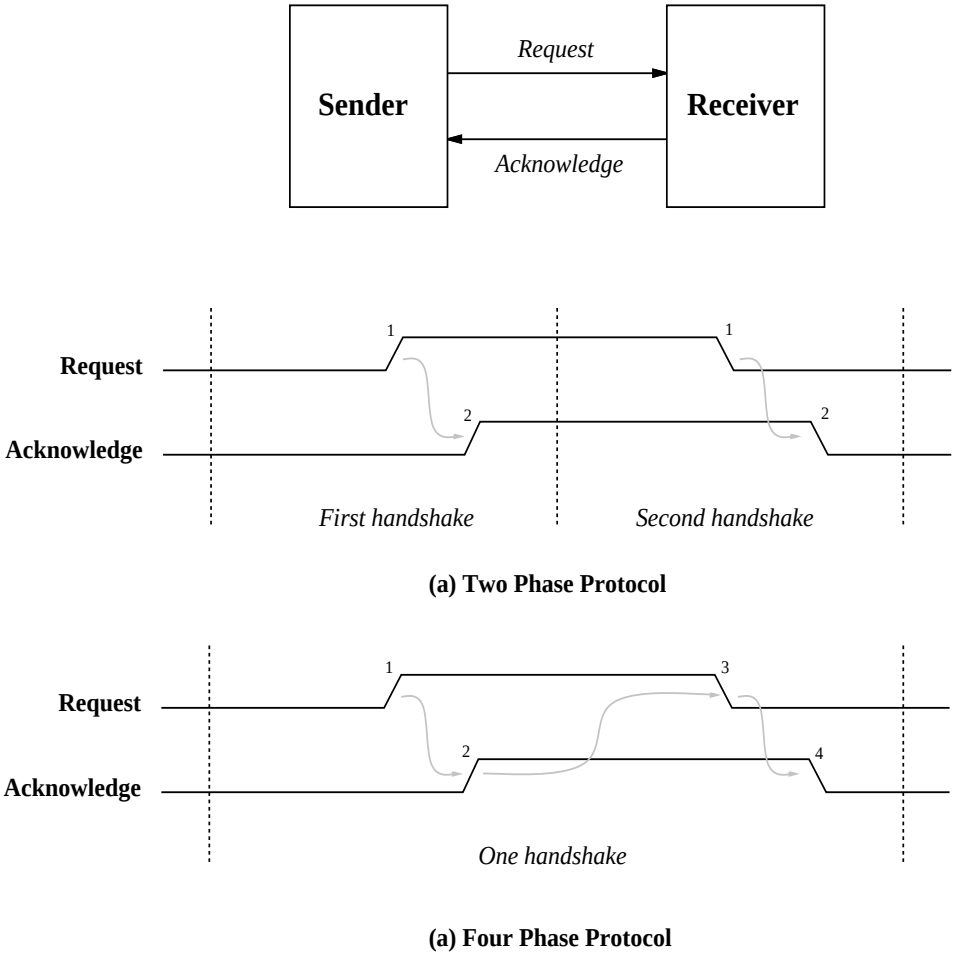


Figure 2.3: Two and four phase handshake protocols

The reason for having different signaling protocols is that different designs have different characteristics due to their interaction with the environment and its datapath resources. These differences will be discussed further in subsection 2.5.

2.4 The Complete Circuit Model

A *complete circuit* is a closed system and consists of two main parts, the *circuit* and its *environment*. A circuit model describes how delays in gates and wires are viewed. An environment model describes how the environment is allowed to interact with the circuit.

2.4.1 Delay Models

Central to circuit implementation is the notion of delay [71]. It is important to distinguish between *stray delays* which are an inherent physical property of any circuit, and *delay elements* that are explicitly added by the designer.

Stray delay occurs in gates and wires as a result of their physical properties such as resistance and stray capacitance in gates and propagation delays in wires. Delay elements are delays explicitly added by the designer to ensure certain functional properties of the circuit. Delay elements can be divided into two main categories, the pure delay and the inertial delay. Pure delays simply propagate the input signal to the output after a fixed time d, as can be seen in Figure 2.4(a). A pure delay is often implemented as a chain of inverters. Inertial delays only propagate signals that have persisted for a fixed amount of time, delaying them a fixed time d. Such delays are often used to filter out undesired glitches, as illustrated in Figure 2.4(b)(the M-gate in this figure is a Majority gate). However, inertial delays are not reliable since they require time to "recover", are sensitive to process variation and produce slow transitions that are sensitive to noise. There is also a third category that is often used in asynchronous circuits, namely that of *asymmetric delay*. An asymmetric delay has different delay times depending on the level of the signal propagating through. Signals going through the asymmetric delay shown in Figure 2.4(c) have a propagation delay d_h when the signal is high and d_l when the signal goes low. The gate in this figure is an AND gate which quickly resets the output to a logic zero. For all circuits in this thesis we assume the use of pure delays.

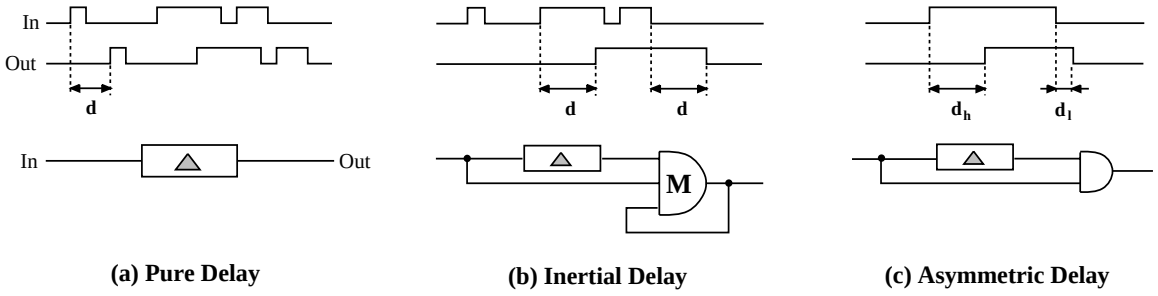
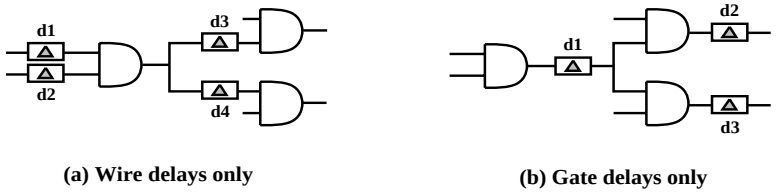


Figure 2.4: Pure, Inertial and Asymmetric Delay Models

There are different ways to view the effects of the stray delays and delay elements in a circuit model. The wire delay through a network of gates is usually modeled as delay elements residing on the input wires of the gates. The gate delays are modeled as delays on the output wires of the gates, while the gate in itself is viewed as a function evaluator with zero delay. Some circuit models use delays on both in and outputs meaning they assume delays in both wires and gates, while others only use the delays on either inputs or outputs. The model using delays on both in and outputs is the most general, but is also the most difficult one to implement. Figure 2.5 shows how delays on input and output wires are viewed. These two models only

differ when we have multiple fanouts from a gate. If a wire delay model is used, the forked signals due to a multiple fanout can arrive to the destination gates at arbitrary times. If a gate delay model is used, since the wire delay is assumed to be zero, the forked signals will reach the destination gates at the same time. The fanout points are then seen as *isochronic* with respect to each other.



(a) Wire delays only **(b) Gate delays only**

Figure 2.5: Delay models for wire and gate delays

2.4.2 Environment and Circuit Models

The way a circuit and its environment interact can be classified into two categories. If the environment is allowed to respond to the outputs from a circuit without any timing restrictions, they are said to operate in *input/output* mode. Otherwise they operate in a mode where there are timing constraints assumed. Two of the most commonly used timing constraint methods are those of *fundamental mode* and *timed* circuits.

Circuit models are used to define how the circuit is affected by stray delays and delay elements. There are several models that can be divided into two main categories, those that can handle *unbounded* or *bounded* delays. In an unbounded delay model, a delay can assume an arbitrary finite value for the time it takes a signal to propagate through. In a bounded model, the delay can assume any value within a given time interval. Pure delays are assumed in these models.

The circuits that use input/output mode for its interaction with the environment are also those that use the concept of unbounded gate and wire delays. Circuits that have a set of restrictions for interacting with its environment use the bounded delay model. The classes of circuits that use these models will be discussed next.

Input/Output Mode Circuits

Common to all circuits operating in I/O mode is that they must rely completely on input and output signals to know when actions are started or have completed. For example, to be sure that all receivers have absorbed an output signal from a controller they are all required to generate acknowledge signals going back to the controller. Similarly, the environment must receive an output signal response from a controller in order to know that a set of input changes

to it have been absorbed. To guarantee a correct circuit behavior under unbounded delay assumptions it is therefore also necessary to use special completion detection circuitry for the datapath resources. Circuits operating in I/O mode can be further categorized into four main styles of asynchronous circuits. The categories are divided by the extent to which they assume unbounded delays in gates and wires.

In the *delay insensitive* circuit model both gate and wire delays are assumed to be unbounded. That means that the circuit has to function properly regardless of the delays and that signal transitions can occur at arbitrary times. The main difficulty in this type of circuit is that an acknowledge has to be generated for every wire fork, often resulting in complex circuitry. Very few circuits can be built out of basic gates using this model [39], however, useful circuits can be built using more complex components [20].

Quasi delay insensitive circuits resemble the model for delay insensitive circuits very closely by assuming arbitrary delays in both gates and wires. However, it relaxes the constraints for multiple fanouts by allowing isochronic forks [38]. This way of modeling wire forks can simplify the design process significantly, but requires that the difference in delay of the wire forks are negligible compared to gate delays and that the destination gates have similar threshold values.

Speed independent circuits assume arbitrary gate delays but zero or negligible wire delays and therefore in all important aspects essentially work as quasi delay insensitive circuits. The assumption that wire delays are negligible may allow less complex logic and solutions to a larger set of problems. However, the number of problems that can be implemented using this style is still quite limited.

A *Self timed* circuit is a legal interconnection of self timed elements [43]. Each self timed element is assumed to be contained in a small area called an equipotential region. Within such a region one may treat a signal as identical on all points of a wire, that is, wire delays are assumed to be negligible. A self timed element may itself be implemented as a speed independent circuit or may contain other types of logic using careful timing analysis. For communication between these self timed elements unbounded delays on wires is assumed. Delay insensitive signaling conventions must therefore be used between elements.

Fundamental Mode and Timed Circuits

Unlike for input/output mode circuits, fundamental mode (defined shortly) and timed circuits expect the environment to behave in a certain manner or their implementations will not work properly. A circuit working under such assumptions does not need to generate an output as response to a set of input changes. The environment can assume that the circuit has absorbed the input changes after a certain bounded amount of time. It does not necessarily need an

output response even if that is often the preferred and safest way to do it.

A *Fundamental mode* circuit [30] is in its construction very similar to a synchronous state machine in that it is arranged as a combinational block with state signals fed back to hold the state. Associated with the combinational block are restrictions on when signal changes from state feedback and environment may occur. These restrictions require that the logic in the combinational block must have stabilized, i.e., all internal activity must have ceased, before new inputs may arrive. This restriction is usually fulfilled without intervention from the designer since the environment is often slow compared to the state change and settling time for the logic. Otherwise, delay elements must be inserted.

Timed circuits [47] take advantage of special timing knowledge of the environment. The circuit is therefore specially designed to meet the timing behavior of the environment. This approach does incapacitate the modularity of a system to some degree, since the environment may change its timing characteristics when elements are replaced. However, explicit knowledge of the environment's timing often result in more efficient circuit implementations.

2.5 Controller and Datapath Interaction

2.5.1 Completion Detection

All asynchronous circuits must have a way to determine the completion of actions. With completion detection of actions we usually mean a way to determine when the actions taking place in the datapath elements have finished. The control part of the circuit already fulfill this requirement by its guaranteed output response to input signals. The methods to detect completion are many and some are only usable by certain styles of circuit realizations.

There exist two main styles of signal representation on wires, *single rail* and *dual rail* [43]. When using single rail a signal is represented by one wire that can assume a high or low state (logic 1 or 0). A ternary signaling scheme using only valid logic 1 and 0 signals can be implemented when using dual rail. A signal is then represented by two wires which are encoded to allow representation of three states, 00 for undefined, 10 for a logic 0, and 01 for a logic 1.

Since we know the state of each individual signal when using dual rail, we can build completion detection trees to determine when an action has finished. Before the action is started, all wires are set to zero, meaning that the signals are undefined. Any gate that has one or more of its inputs in an undefined state also has an undefined output. The input signals can then arrive at arbitrary times, and we can tell when the action has completed by making sure all outputs from the element are in a valid state. This is achieved by a completion tree connected to all outputs of the element. The advantage of this method is that it is possible

to exploit average case delay of the action. The disadvantage is that it requires twice as many wires as single rail, resulting in increased area, power consumption and routing difficulties. The delays through the completion tree also add an overhead. The fanout of this completion tree together with the request signal of the handshake forms the acknowledge signal.

An often used method for single rail circuits is that of simply modeling the worst case delay of the action by a matching delay on the request wire. The output of the matching delay then forms the acknowledge signal. This method has a disadvantage since it does not exploit average case delay and exact layout simulations are needed to determine the value of the delays. The advantage is that it is very easy to implement.

Another detection mechanism for single rail circuits is that of completion detection through current sensing [16]. When all activity in the element requested for the action has ceased, the element no longer draws any current, which together with the request signal triggers a current sensitive device to generate the acknowledge signal. This method exploits average case delay but also adds an overhead and can be difficult to implement.

2.5.2 Efficiency and Robustness

Several issues such as robustness, performance and power consumption needs to be taken under consideration when choosing which style, in terms of delay modeling, completion detection and handshake protocol, to use for circuit implementation. There are some important tradeoffs between the different methods and a single style may not be the one best suited for all applications.

Using delay insensitive circuits means that the whole circuit can be generated automatically without simulations to verify that no timing constraints are violated. Since the datapath signals are encoded with dual rail, there is no problem with data values arriving too late and causing metastability. The resulting circuit is also very robust and will work under very varying environmental conditions. From a performance and power consumption aspect however, a delay insensitive circuit may not be the best choice due to its many wires and complex completion detection mechanisms. Another disadvantage is that very few problems (applications) can be implemented as delay insensitive circuits. Using the quasi delay insensitive or speed independent approach gives a larger number of problems that can be implemented but also compromises the robustness of the system. These design approaches also tend to give better performance in the face of many isochronous forks. Using the self timed circuit style allows more problems to be solved, but also raises the question of how large a self timed element can be under the equipotential region assumption. This style might improve performance but will not be as robust.

As for the fundamental and timed models they can offer very good performance, but

are not as robust to environment variations. Especially the timed model which uses explicit knowledge of timing and gate delays is very sensitive to process and environment variations. These methods use *bundled data* constraints to communicate datapath signals. This means that the datapath signals must arrive to their destination before the request signal. If not, the register may go metastable or store the wrong values. The order of arrival is usually ensured by inserting delays on the request wire. As completion detection, a simple delay is often used by these methods. Using delays to model the worst case delay in a datapath resource is not very robust and requires careful timing analysis at the layout level. The inability to exploit average case delay also results in a certain performance degradation. A more robust method is to use current sensing logic to detect completion. This will add some overhead but will also allow us to exploit average case delay. Whether this overhead will justify the use of matching delays instead depends largely on the difference between the minimum and maximum delay and frequency of operation of the datapath resource.

As for which handshake protocol should be used for best performance and power consumption depends on the characteristics of the design. Two phase signaling is often used when implementing control intensive circuits or for designs which are transition oriented by nature, such as many bus protocols. Two phase signaling is also used with advantage for off-chip communication, that is, communication between separate integrated circuits. Two phase signaling also consumes less power since it only makes two wire transitions for each handshake compared to four for the four phase protocol. Using four phase signals often has an advantage when implementing datapath intensive circuits. This is because four phase signaling allows use of synchronous datapath elements such as single edge triggered flip-flops or transparent latches. Flip-flops for the two phase protocol need to be double edge triggered, which means we have to add extra circuitry to be able to latch on both positive and negative edge. The result is that two phase latches are often slow or have large area. The four phase protocol is not as well suited for off-chip communication since returning the wires to zero over long distances gives too much overhead.

2.5.3 Micropipelines

One style of design that has proven to be useful in both synchronous and asynchronous design is that of pipelines. Pipelines are often a good and relatively easy way of increasing the throughput of a system. There are two types of pipeline structures in asynchronous design. One style called *micropipeline* [66] which is an ordinary straight pipeline, and *ring* pipelines [73, 62] which are self-iterating pipeline loops.

In asynchronous design pipelines have their own form of interaction between control and datapath elements. Micropipelines work much as an ordinary synchronous pipeline. The

difference is that the control structure is a part of the pipeline structure itself. A classic structure of a micropipeline using transition signaling (two phase protocol) is illustrated in Figure 2.6.

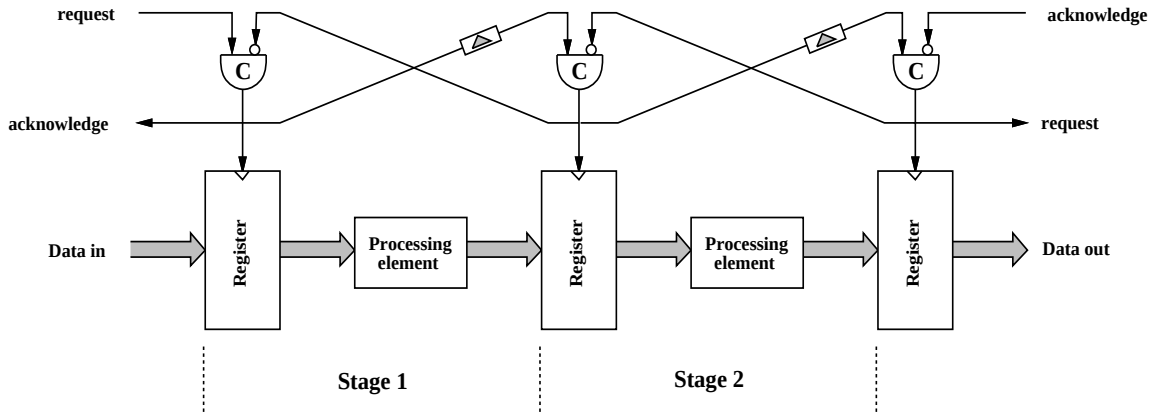


Figure 2.6: Example of a micropipeline

Such a pipeline is formed as a FIFO buffer with processing elements between each stage of registers. The control part of the pipeline consists of C-elements that work as an AND-gate for transition signaling. (When all inputs to a C-element are low, the output goes low, and when all inputs are high the output goes high, otherwise it holds it current state. A bubble means the signal is inverted.) Delay elements that model the delay of the processing elements are inserted on the request wires between the stages. The pipeline in Figure 2.6 makes use of the bundled data concept. At initialization all signal values are set to low. When the environment wants to send data through the pipeline for processing, the data signals must first stabilize at the input register of the pipeline. The environment then sends a request signal to the pipeline telling it to start computation. The request makes a high going transition on the input to the C-element which in turn generates a high going transition on its output. The register now propagates the data and the processing element in the first stage starts executing. The generated request signal goes through a delay matching the processing element's delay and ends up making a high going transition at the input of the C-element at the second stage requesting it to store the computed data. The request signal also goes back to the environment as an acknowledge signal. When the environment has received the acknowledge it is free to make a new request at any time. The data already being processed in the first stage is safe since it cannot be overwritten with new data from the environment until the second stage has stored the computed data and sent back an acknowledge signal to the C-element of the first stage. So even if the environment is fast, the new data will not be stored in the register of stage 1 until stage 2 has safely stored the computed data.

If other completion detection mechanisms such as current sensing are used, the pipeline has the advantage of using average case delay which, depending on the processing elements,

can result in significant increase of throughput.

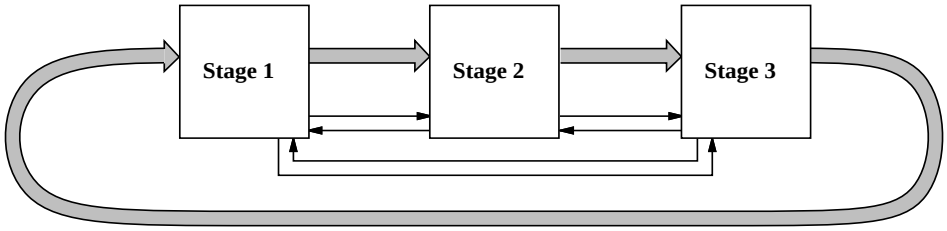


Figure 2.7: Example of a ring pipeline

Another data processing structure well suited to asynchronous circuits is that of ring pipelines. Rings are implemented as FIFO buffers with the output of the last stage fed back to the input of the first stage as can be seen in Figure 2.7. This structure is very well suited for iterative computation since the computation stages in a ring structure only need to communicate with the environment once before each sequence of iterations is started. Since communication overhead thus can be avoided during the iteration sequence and since the stages also can exploit average case delay, this type of pipeline is very efficient in many situations. Examples where asynchronous ring pipelines are especially effective is for shift and add multiplication and division units [73, 62] which can yield up to four times better performance than synchronous solutions.

2.6 Design Styles for Controllers

There are many different design styles that have been proposed for asynchronous controllers. They can be divided into roughly three main categories: translation methods, graph based methods, and asynchronous state machines.

2.6.1 Translation Methods

Translation methods offer an elegant way of specifying high level descriptions of asynchronous designs. They also offer an easy way to verify the design using standard verification tools [72]. Another advantage is that they offer a simple way of specifying and implementing arbitration for concurrent processes sharing the same hardware.

These methods synthesize a design by translating statements in the specification based on their syntactic structure. Since the synthesis is based on local transformations this style often results in less efficient circuit implementations. The two most common styles in this category is that of direct syntax translation to a limited library of standard controllers and that of syntax translation via a series of program transformations to customized controllers.

The direct translation to a library of standard controllers is often called the *macromodule* approach [7, 3]. The libraries are built up of standard control units such as C-elements, toggles, decision-waits etc. [66] which are considered to be rather slow components. Since this method is based on local transformations and targets a limited library of components, the resulting circuit implementation is not as efficient as might be desirable. A method of peephole optimization is therefore often applied to the implemented circuit after synthesis. The idea is to replace inefficient portions of the circuit with more efficient implementations. The method presented in [26] achieves this by extracting a delay insensitive behavior of local portions of the circuit by using parallel composition on trace structures [18]. This behavior is then captured in a state machine description and implemented using asynchronous state machine synthesis. Although it has been shown that better implementations can be achieved using this and similar methods, the resynthesis still can only achieve local optimizations.

The method of translation using a series of program transformations has been described in [41]. This method starts from a high level description based on the language CSP [29]. The idea here is to optimize the final circuit by performing program transformations on the design description. These transformations are done at several different levels of abstraction before the final design description can be synthesized into a circuit representation. The method targets customized controllers based on C-elements and standard gates such as AND and OR but has also used generalized C-elements with custom transistor networks. Many designs have been implemented using this method, among them the first asynchronous microprocessor [40].

2.6.2 Graph Based Methods

Another synthesis approach makes use of graph based descriptions in the form of I-Nets (a subclass of Petri nets [46]) and Signal transition graphs (STG - also a class of Petri nets). The design descriptions of these methods are usually made at the handshake level. This gives the designer greater flexibility to generate more efficient descriptions by manipulating the separate event signals. However, they do not have the power of directly describing the behavior of a system in a high level description. This makes it hard to capture the system structure of a design rather than the individual controller structures.

As illustrated in Figure 2.8 an I-Net consists of a non-empty set of places (circles) and transitions (horisontal lines) with directed arcs (arrows) between them. The input places of a transition are all places with an arc leading directly to the transition. Similarly the output places are the places at the end of all arcs going out of the transition. An I-Net is marked by tokens residing in places. When all input places to a transition are marked, the transition is enabled. The I-Net is executed by firing transitions that are enabled. When a transition has

fired, all its markings in the input places are moved to the output places. A one-safe I-Net is a graph where no more than one marking can occur in the same place at the same time. (This means no concurrency is allowed and is a requirement for the state machines used in the synthesis methodology presented in the next part of this thesis.)

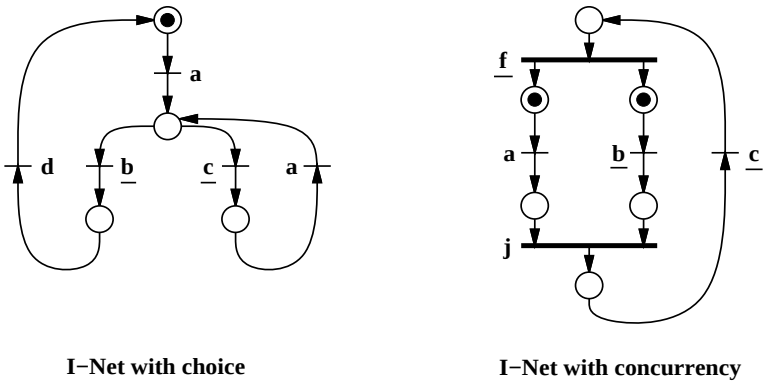


Figure 2.8: Example of I-Nets

I-Nets can be used to describe the behavior of a circuit to be implemented by assigning the names of primary in and outputs to the transitions. Every firing of a transition in the graph then represents a transition on the corresponding signal wire in the circuit.

The synthesis of such a graph to a circuit implementation is done by first generating an interface state graph (ISG) by exhaustively simulating the I-Net. For every marking a state is created, and for every enabled transition an arc (going between states) labeled with the signal name is created. An extended interface state graph (EISG) is then created by assigning initial values to the signals. In this graph the states have the same signal encoding and are separated by the different values of the signals in the different states. (When a transition occurs for a signal, that signal toggles its value giving adjacent states separate state codes.) Conflicts in state encoding are solved by introducing state variables. Since this method enumerates all markings the algorithms used can be exponential in the number of places which will hinder synthesis of larger graphs. A Karnaugh map can now be generated from the EISG. Huffman or self-synchronized style state machine circuit implementations can then be derived by using methods presented in [22, 44, 58]. However, these methods do not generate hazard-free logic and therefore make use of either inertial delays or clocked latches to filter out glitches which slow down the circuit.

Signal transition graphs are also a class of Petri nets. The basic idea of STGs is to avoid the exponential worst case complexity that is present during synthesis of I-Nets. This is achieved by introducing restrictions on the graph making it a less general Petri net than I-Nets.

An STG in its simplest form is called STG/MG and belongs to the class of Petri nets

called marked graphs. The places in such a graph are only allowed to have one input and one output transition. This means that choices are not allowed which severely limits the number of problems that can be represented by this method. More advanced forms of STGs that allow choices also exist but are associated with a number of restrictions.

When synthesizing an STG the properties of the graph are first examined. This is because if some restrictions are fulfilled then it might be possible to use more efficient synthesis algorithms, avoiding exponential worst case complexity. Unfortunately many synthesis methodologies for STGs require that the designer manually ensure these restrictions, making the specification of large designs very hard. This of course also goes against the reason for using STGs in the first place - to be able to synthesize large controllers. Still STGs are a viable alternative in graph based synthesis since at least for some type of designs it can directly synthesize larger descriptions than when using I-Nets. Unfortunately, many methods used to implement STGs have logic hazards requiring insertion of delays or flip-flops to filter out glitches thus slowing down the circuit just as for those generated with I-Nets. A method in [4] generates hazard-free logic but the gates used often have high fanin and cannot be easily decomposed. This method also makes use of algorithms having exponential worst case complexity, taking us away from the basic idea of STGs.

2.6.3 Asynchronous State Machines

An asynchronous finite state machine is a controller having specification, implementation and functionality much like that of a synchronous state machine. The specification of a controller is often described as a Mealy state machine with a set of primary inputs and outputs. The state machine consists of a combinational block implementing the output and next state functions and its state is realized through internally fed back state signals.

The synthesis approach is similar to that of synchronous state machines. Due to its way of moving from state to state, the specification of a state machine is often described as a *flow table*. The synthesis then proceeds by first performing *state minimization* to reduce the size of the flow table by merging compatible states. This is followed by a *state assignment* step that assigns binary values to the symbolic states. Boolean functions for the output and state signals are then generated from the flow table. Methods in [74, 49] use some special requirements during this step to ensure that the output signals of the combinational logic behave in a monotonic fashion. This step is called *hazard free logic minimization*. Other methods use inertial delays or constantly clocked latches. The last step is to generate combinational logic for the Boolean functions which is done by technology mapping. This synthesis methodology often targets an implementation in the form of a two-level AND/OR gate network, called a sum of products. This style of implementation has been shown to make it possible to deal with logic hazards [50]

in a straight forward way.

There is a major attraction with using finite state machines compared to other methodologies for asynchronous controller implementation. The state machine synthesis method can perform global optimizations which would be difficult or impossible to do with other synthesis methods. Because of this, state machine synthesis often results in very efficient gate level implementations. Another important advantage is the possibility to use boolean manipulation on the synthesized gate network allowing decomposition of large and slow AND and OR gates into smaller and faster ones.

The draw-back of this method is the complexity of state machine synthesis in general. Algorithms for exact solutions have exponential complexity which makes it impossible to synthesize large controllers.

Many efficient designs and synthesis methods for asynchronous finite state machines have been implemented and presented [14, 74, 49, 63]. These methods generate hazard-free logic and can therefore avoid use of inertial delays or constant clocking of latches, making the circuit fast. These methods have shown that state machine modeling and synthesis is indeed a viable option for asynchronous design.

2.6.4 Conclusions

This section has presented some of the advantages of asynchronous design. It has also explained the handshake scheme used for communication in and between asynchronous circuits. Further, different circuit models and their use of delay models and the interaction between control and datapath have been discussed. Finally, the different design styles for controller implementations were discussed.

As for which design style of those presented is preferable in terms of synthesis complexity and circuit efficiency, such a comparison is difficult to make since detailed results from synthesized examples are seldom made available. There is also a lack of standard benchmark designs for asynchronous circuits. Existing results often target different designs which adds to the difficulty to make fair comparisons.

What has been shown is that targeting finite state machines indeed gives very efficient implementations compared to a macromodule approach. It has also been made clear that taking advantage of timing information in a circuit can result in more efficient but also less robust implementations. Our approach in the synthesis system presented in the next part of this thesis will target state machines for synthesis of controllers and use the single rail approach for communication and datapath.

Chapter 3

Introduction to ACK

3.1 Motivation

The area of asynchronous circuits and systems is experiencing a significant resurgence of interest and research activity. However, a number of important problems have to be solved before asynchronous design methods can be successfully transferred to the CAD industry. First, tools to synthesize asynchronous circuits must be capable of handling and generating efficient implementations for reasonably large designs, such as the ones found in high-level synthesis benchmarks. This requires a method that offers flexibility to use different signaling protocols, decompose large centralized controllers and take advantage of advances in standard logic synthesis. Second, asynchronous circuit design should be based on high level synthesis methods that are based on standard HDLs, the same basis as used for synchronous circuits. Third, where efficiency is critical, it should be possible to obtain customized complex-gate based circuits. Finally, in order to appeal to current VLSI CAD tool users, asynchronous high level synthesis tools should be available as part of existing CAD frameworks. We present a tool called ACK that incorporates these features.

There exist a wide variety of methodologies in asynchronous high level synthesis, that target various implementation styles. Methods targeting restricted macromodule libraries for control implementation have been developed [2, 7]. A synthesis style based on a high level language called Tangram [32] and an approach called Martin-style synthesis [38] using a CSP-style high level language for specification have been presented. Our method presents an asynchronous synthesis paradigm that generates data path and asynchronous state machine controllers from a behavioral description in a standard HDL. Such a method has the advantage that it closely tracks progress in asynchronous high level and state machine synthesis techniques as well as advances in synchronous CAD such as logic synthesis algorithms, commercial tools and HDLs. We see these as important factors for asynchronous integration in industry designs.

ACK is a high level synthesis tool that can generate logic level datapaths as well as control paths starting from a standard HDL description. Although there is no consensus on the requirements for a good HDL, based on strong research some factors have emerged as important for asynchronous synthesis [2, 7, 38, 32] such as channel communication, process descriptions, and direct support of concurrency. Meanwhile, languages such as Verilog and VHDL have emerged as standards in the industry. Support for these languages is becoming an important factor in the acceptability of tools for VLSI design, for ease of use and design portability as well as behavioral simulation purposes. To balance these factors we define a language called Verilog−+, which is a synthesizable subset of Verilog that supports many of the required features in asynchronous design.

Syntax directed translation [2, 7, 38, 32] has proved to be a successful technique in high level asynchronous synthesis. Often synthesis methods that use this technique have targeted restricted module libraries, sometimes referred to as macromodules, for implementation. This method does not generate the most efficient implementations. Resynthesis techniques such as peephole optimization [26], where inefficient parts of the macromodule controllers are identified and replaced by more efficient implementations based on asynchronous finite state machines synthesis, have shown that FSM representations are a viable option for efficient control implementation. We have developed a method where the high level description is directly translated into dedicated FSM controller structures which are customized to the application.

Using a high level description for system specification that is protocol independent gives us the flexibility to choose between different handshake protocols during synthesis. Currently, *two phase and four phase* handshake protocols are supported. A synthesis tool that is flexible enough to allow support of both these protocols, starting from the same high level description and targeting similar gate-level circuits, will help in making fair comparisons and help evaluate exact tradeoffs between the two styles.

Partitioning is an important component of our high level synthesis approach. There are two main reasons to partition a controller. The first is that the complexity of finite state machine synthesis increases as the size of the centralized controllers grow. In some cases the synthesis methods fail to generate circuits for these state machines. The second reason is to improve efficiency of the implementations. Certain constructs such as loops often execute faster when partitioned, due to reduced logic overhead. Also as wire delays become significant, the ability to generate controllers local to their corresponding datapath is important. The incompletely specified nature of most asynchronous controllers makes partitioning in general a hard problem. More specifically, the steps of state assignment and logic minimization in many synthesis methods rely on the fact that the environment of the controller does not present any of the unspecified behaviors. Under this assumption, the sharing of signals between the partitioned controllers is a non-trivial problem which has not previously been addressed. This

is a particularly important issue in asynchronous synthesis and we therefore present a general technique that allows efficient user-defined partitioning of such controllers.

In order to generate high-performance controllers, ACK generates *asynchronous finite state machines* for the partitioned controllers. Many asynchronous design styles have targeted the problem of generating efficient control structures and dealt with the issues of design complexity and correctness. One class of such techniques target Huffman mode asynchronous finite state machines starting from burst mode specifications and use standard gates for implementation [14, 49, 63, 74]. We therefore target interacting burst mode controllers based on the previous control partioning and take advantage of the logic synthesis techniques available in burst mode controller synthesis to generate efficient controller circuits.

Although the interacting burst-mode controllers can be built efficiently using standard gate-level cell libraries [49, 63], even better performance can be obtained by mapping the burst-mode controllers to *custom CMOS complex-gate implementations*. Customized CMOS gate implementations have been used successfully to design a large number of burst-mode Huffman style asynchronous controllers [14, 65], however, a systematic analysis and synthesis methodology for deriving these has not been given. There are several reasons for considering custom CMOS complex-gate based circuits. As VLSI feature sizes decrease and wire delays become significant, they can provide more efficient controller implementations compared to standard-cell place and route tools. In ACK we have developed a method to derive complex-gates that relax some constraints of hazard-free synthesis of boolean functions, providing significant area and delay improvements in many cases.

If design tools are to be used by a wide audience, and also to continue to benefit from the rapid improvement in CAD tools in general, it is important to leverage off of existing CAD tools where appropriate. In addition to our algorithms, ACK makes use of many commercial and public domain tools. The Viewlogic and Cadence LAS [10] synthesis tools are used to generate datapaths and automatic synthesis of the complex gate controllers. Currently the complete ACK tool set is also encapsulated into, and can be run directly from, the Viewlogic cockpit. The Lager tool set is used for place and route of standard cells and LAS is used when targeting complex gates. Both commercial and public domain Verilog simulators have been used. The 3D [74] tool is used for synthesis of the burst mode controllers.

The rest of the thesis will examine each of these features of ACK in greater detail, providing experimental data to support our claims.

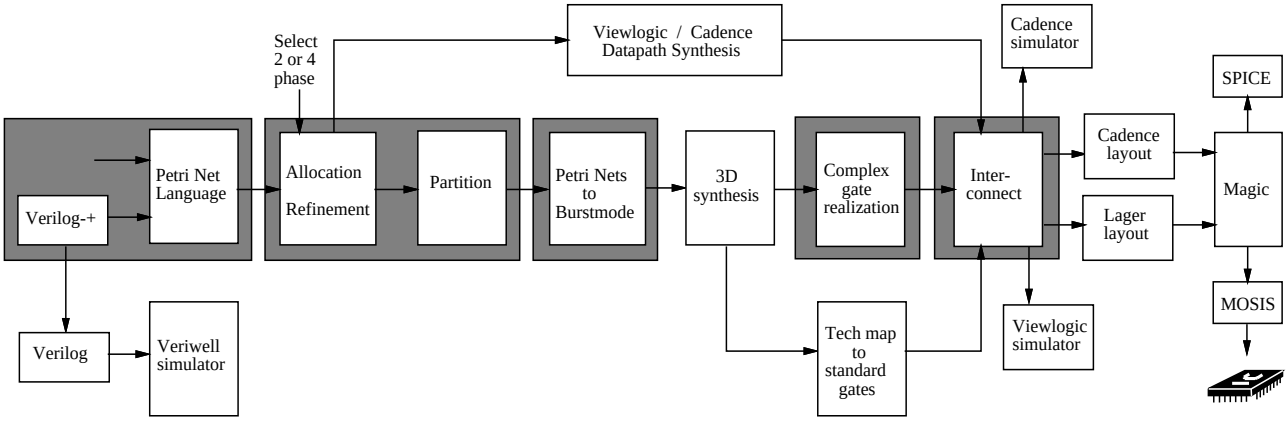


Figure 3.1: System Implementation

3.2 Synthesis Overview

A distinctive feature of ACK is its ability to derive controllers as well as datapaths starting from high-level descriptions, all the way to layout. Figure 3.1 shows a top-level description of this process. A description of a system to be synthesized consists of HDL programs describing a collection of communicating modules. Each module has process descriptions, local variables and interface ports for communication. ACK uses Verilog−+, a synthesizable subset of Verilog extended to handle channel communication. A description in this HDL can be used for high-level behavioral simulation using a standard Verilog simulator. The description is then translated into our internal Petri net format for synthesis. A HDL based on the Petri net format can also be used as input.

Once in Petri-net form, synthesis proceeds by first choosing a *protocol*, either two-phase or four-phase, for control signaling. The rest of the synthesis will assume this selected protocol. This is followed by *allocation* of datapath resources for computation, and other resources for communication. High level actions are then *refined* (macroexpanded) into a series of signal transition actions on allocated resources. The allocated datapath is automatically synthesized using standard commercial synthesis tools (we currently use Viewlogic and Cadence). Details of the high level description and synthesis aspects of the tool are provided in Section 4.

The controller generated after refinement is then *partitioned*. For each Petri net, both required (to resolve the fork-joins in each process if any) and user defined partitioning is performed. Details are given in section 5. After partitioning, the resulting Petri net controllers are converted into *burst-mode state machine* descriptions. These burst-mode descriptions are then synthesized using the 3D synthesis system [74]. Details of conversion of Petri net controllers to burst mode state machines is given in section 6. The synthesis of these controllers is then

considered in section 7.

Synthesis of the burst mode descriptions and technology mapping is then performed either to a standard gate implementation or a customized complex gate implementation. If customized complex-gate implementations are synthesized, the Cadence layout tool LAS is used to generate automatic layout. For standard cell based circuits, the Lager auto place and route tool Timberwolfe is used. Details of technology mapping is given in section 8. Finally, the layout can be extracted in Magic and simulated using SPICE or other transistor level simulators. The extracted layout can then be converted to a format supported by MOSIS and fabricated. Results for synthesized examples will be given locally in each section. Conclusions and future work will be provided in section 9.

Chapter 4

High Level Modeling and Synthesis

A description of a system being modeled is usually divided into two different specifications, one for the actual design (circuit) being modeled and another for the environment it interacts with. The environment can be specified in many ways, but basically consists of an interface together with a communication protocol and a set of timing constraints for the individually occuring signals. The design specification contains more detailed specifications about the circuit and is often divided into a hierachical structure with different levels of modeling complexity. A design in ACK is represented by a structural description consisting of a set of interacting modules communicating via shared registers, channels and event signals. Each module represents a basic functional entity of the system and consists of an interface description, a set of local variables, functions and a graph describing its behavior.

In subsection 4.1 the environment model will be briefly discussed followed by a presentation of the structural model of a design and the module entity in subsection 4.2. The high level synthesis of the description will then be presented in subsection 4.3.

4.1 The Environment

For a circuit to be able to communicate with its environment in a meaningful manner it is required that the designer must have knowledge of its behavior. This behavior is often expressed by an interface description that describes the physical properties such as input and output communication and data ports of the environment. Signaling protocols defining how communication with the interface is to proceed is also a part of the environmental specification. Although other signaling schemes exist [6, 67], these protocols are usually restricted to follow the two phase or four phase protocols for compatibility reasons. Associated with the environment is also a set of timing rules that defines how fast the circuit is allowed to respond to incoming signals from the environment and how fast the environment is in its response to signals from the circuit.

These timing restrictions can be especially important when targeting asychronous finite state machines for synthesis since they work under the fundamental mode assumption which sets a constraint on how fast the environment is allowed to respond to a set of output signals. The specification of the circuit environment is not made part of the ACK design flow but is in part implicitly defined in the circuit interface declaration.

4.2 The Design Description

The hierarchy of a design specified in ACK consists of three implicit levels. The highest hierarchy level is the structural level which is an interconnection of modules. Such a module corresponds to the second hierarchy level. The third level is represented by functions declared inside the modules.

4.2.1 Structural description

The structural description in ACK is implicit and represents the design at the architectural level as an interconnection of basic module entities. Implicit means that there is no separate description of the interconnection necessary. Instead, a naming convention similar to that in [28], which has shown to be an effective way to handle automatic interconnection of large circuits, is introduced. If several modules have the same name and type for an event wire, channel, variable or function, it is assumed to be *shared* between these modules. Event wires and channels having the same names are also connected together forming the physical interconnection of the communication interfaces between the modules internal to the design. Variables and functions not being shared are assumed to be specified locally to a certain module and can only be accessed by that module. Input or output event wires and channels of a module that do not have any corresponding output and input in any other module are assumed to be connected to the environment.

The modules are processes that execute concurrently. For such processes to share variables or functions without conflicts, a mutual exclusion scheme is necessary. If arbitration is not resolved through communication between the sharing modules before using the resources, the designer must implement arbitration circuitry to ensure the resources are used in a mutual exclusive fashion. Channels (defined shortly) are often used to ensure mutual exclusive access by obtaining synchronization between such processes.

4.2.2 Module Description

A module is the basic entity being modeled. It consists of an interface declaration which describes its in and out going event wires, channels and data ports. It also contains declarations of variables and functions. All these constructs can be shared between modules under the assumption that all access to them is mutually exclusive. Allowing several modules to share the same resources can result in less area but since it limits concurrency it may also result in a performance degradation. The decision of optimally sharing resources such as variables and functions is currently left to the designer. The types of a construct are also specified at the time of declaration. The type can be a bit, an array of bits, integers or an enumeration of integers. The declaration constructs mentioned above and what they are used for is discussed in the following paragraphs.

Event wires are mostly used for effective communication between modules executing in a sequential fashion. When an output event signal is generated by the sender it is assumed that the receiver is ready to receive the event. This is hard to ensure when using modules executing in parallel. For safe communication between such modules, channels need to be used. The value on an event wire must of course change monotonically.

Channels are used both for synchronization between processes executing in parallel and for sending data. A channel makes use of a C-element to ensure that both processes are synchronized before proceeding. For instance, if a process wants to send data to a concurrently executing process it must first synchronize the action with the other process. It does this by first asserting the data wires and then generating an event signal to one of the inputs of the C-element. It has now indicated that it is ready to send the data. When the other process is ready to receive the data, it sends an event signal to the other input of the C-element. The C-element then generates an event signal at its output that is fed back to both processes, which are now synchronized, and the receiving process can store the data.

Data ports work as in synchronous designs, that is, the values on the wires are allowed to change non-monotonically but are assumed to have stabilized by the time it is stored. Data ports are often used together with event signals using the bundled data approach to transfer data. The main disadvantage of this method is that the sender has no means of detecting if the receiver is really ready to receive the data or not.

Variables are represented as registers in hardware. They are used to store data either globally or locally depending on whether they are shared or not. Different types of registers are used depending on the chosen protocol. For two phase, double edge triggered flip-flops are used. For four phase single edge triggered flip-flops are used.

Functions can be used either as a short hand for complex expressions or a commonly occuring sequence of actions. Functions support variable passing as long as the passed variables

are of the same type as the declared parameters. The parameters can be used as local variables but do not change the value of the registers of the originally passed variables.

The Behavioral Graph

Associated with each module is also a behavioral graph describing the high level behavior of the module. The graph is a state machine with fork-joins (SFJ) and is described in the form of a Petri net. There are two types of transitions in this graph description, action transitions and fork-join transitions.

Action transitions have an in and out degree of one. Each of these transitions are associated with one or more *actions*.

Fork and join transitions start several threads executing in parallel. The in degree d_{in} (number of forking threads) and out degree d_{out} (joining threads) must be the same. The fork and join transitions themselves do not have any actions associated with them.

Each action transition in the SFJ graph has one input place and one output place. A fork transition has one input place and N output places while a join transition has N input places and one output place. Between the i:th output place of a fork transition and the i:th input place of a join transition is a single threaded subgraph (STS) which has only one input and one output place. In this section we will concentrate on the action transitions of the SFJ graph. Fork-join transitions along with SFJ and STS graphs will be further discussed in section 5.

Actions

Associated with each sequential transition T_s is an action. Such actions can be an event on a wire, a channel action, an assignment action, a choice action, a function call or a compound action. These actions will be defined in the following paragraphs.

An *event action*, E, is a transition on an input or output wire of the module. Such an action is given by $E \times \{!!, ??\}$. The "!!" indicates the generation of an output transition from a sender and "??" indicates the reception of an input transition to a receiver.

A *channel action*, C, represents a CSP style rendevouz action and is given by $C \times \{!, ?\} \times Y \in \{V, X, \epsilon\}$ where V is a variable, X is an expression and ϵ represents the case where neither variable or expression is present. The "!" and "?" corresponds to a send and receive operation respectively. A channel action can either be a pure synchronization action in which case $Y = \epsilon$. A channel can also be used to send or receive data in which case $Y = V$. A sending channel can also send an expression without needing to store it in a variable. In this case $Y = X$.

An *assignment action*, A, represents the storage of a data value and is given by $V = \{K, V, X, F\}$, where K is a constant and F is a function call. The result from the right side of the equal sign must of course be of the same type and in the range of the declared type of the left hand variable.

A *choice action*, B, is a place with two or more outgoing transitions (a branch). A choice may be based on either boolean expressions or event actions. These actions must all be of the same type (expressions or events) and must be mutually exclusive with respect to each other. That is, not more than one branch may be taken. A choice based on a boolean expression is a data dependent choice and is used to test values that are stored in the datapath.

A *function call* can either be an arithmetic expression which gives a return value or a sequence of actions which is a separately implemented subgraph. A function call is given by $F(P) \rightarrow R \in \{N, \epsilon\}$ where P is a set of parameters and R is a return value, either a non-empty bitvector/integer or an empty value. A function call with a non-empty return value can be used in assignment, channel and choice actions.

A *compound action* allows multiple actions to execute at the same time given that they are of the same type. It is given by (*action 1,...,action k*) where k is finite. Compound statements do not allow the same degree of concurrency as a fork-join but is well suited where no more than one transition in a row has concurrent actions since it avoids the overhead of invoking a fork-join.

4.2.3 Specification Languages

A language called HOP has been developed and is a textual representation of the behavioral graph discussed in the previous section. The language consists of a declaration section where names and types of events, dataports, channels, variables and functions are defined. This is followed by a behavioral description section which describes the high level behavior of the module.

The behavioral description consists of a set of current and next states representing the places in the Petri net graph. A set of statements represents the transitions with their associated actions. To ease the design specification a right arrow, $\rightarrow$, can be used as shorthand notation for the place between transitions. It can be seen as going from one transition to another in sequence with an implicit place in between. A specification of the algorithm for a factorial computation unit in HOP is shown as a graph together with its corresponding textual description in figure 4.1.

For designers not familiar with the HOP language, ACK also has an interface to allow a synthesizable subset of Verilog, called Verilog$-+$, as specification language. The reason for choosing Verilog instead of VHDL as HDL front end is that it has direct support for event

```
Module Factorial

  Event START?? : bit;
  Variable a, n : array [7:0] of bit;
  Channel nchan?, reschan! : array [7:0] of bit;

  Behavior

    <always> <= START?? -> <forever>;

        <forever> <= nchan?n
                     -> a = 1
                     -> <while>;

            <while> <= (n == 0)
                          -> <result>
                       |(n != 0)
                          -> a = a * n
                          -> n = n - 1
                          -> <while>;

            <result> <= reschan!a
                        -> <forever>
End
```

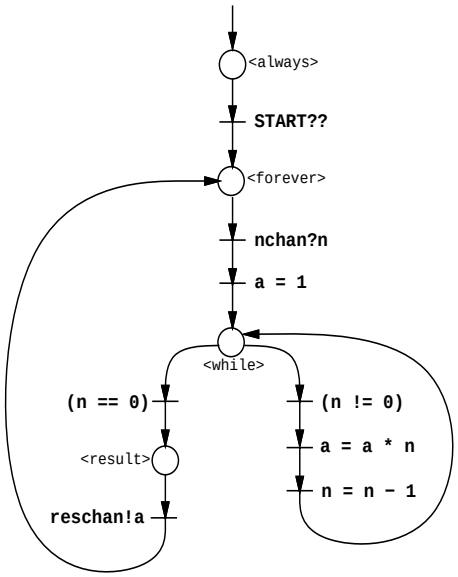


Figure 4.1: Factorial example: HOP language and corresponding graph

signals. It also has a one to one mapping of all basic constructs in HOP which makes it easy to translate. However, Verilog does not directly support channels which are important for effective implementation of concurrent systems in asynchronous design. To overcome this problem the subset has been extended with channels. The Verilog−+ language still allows behavioral simulation in a Verilog simulator since the channel construct can simply be macroexpanded to a sequence of event signals and variable assignments in Verilog. The Verilog−+ specification of the factorial unit is illustrated in Figure 4.2.

```
Module Factorial;

  reg [7:0] a, n;
  channel? [7:0] nchan;
  channel! [7:0] reschan;
  event START;

  always
    begin
      @START;
      forever
        begin
          nchan?n;
          a = 1;
          while (n != 0)
            begin
              a = a * n;
              n = n - 1;
            end
          reschan!a;
        end
    end
endmodule
```

Figure 4.2: Factorial example: Verilog−+ language

4.3 Allocation and Control Refinement

We now have a model for the specification of the system. This section will present the steps used in the high level synthesis of this description.

During synthesis, the behavioral Petri net graph is divided into separate data and control paths, both for modeling and efficiency reasons. This is done by means of *allocation* and *refinement*. Dividing datapath and control allow us to synthesize the datapath resources separately using efficient synchronous synthesis tools, and to generate customized state machine controllers for the control part. This allows us to exploit the global optimizations possible in asynchronous state machine synthesis and also track new developments in state machine synthesis methods.

Allocation is used as a step to find out what resources are needed in the datapath to implement the different actions in the behavioral specification graph. An example of a datapath resource is a register for storing the value of a variable. Refinement is used to create a control graph that acts on the allocated resources and makes sure they are used in a way corresponding to the specified actions in the behavioral graph.

As the first step in synthesis, a protocol, two or four phase, must be selected. This protocol will be used by all modules in their communication with the environment as well as between themselves and their corresponding datapaths. The synthesis method then proceeds as follows.

We will first consider the synthesis of the structural description. First we allocate the resources for event wires, channels, variables and functions that are shared between the modules. For channels, we will allocate C-elements along with data wires. For event wires, we allocate physical wires between modules. For shared variables we allocate registers and associated data wires. Handling of functions depends on whether it is defined as a subgraph or as an expression. If it is defined as a subgraph it will be divided into a separate control and datapath which will in turn be subject to allocation and refinement. If it is a simple expression, the same resources as for an assignment action will be allocated, which will be discussed later.

The interface of each module is later connected to these shared resources. In the following subsections we will consider the synthesis of each separate module.

4.3.1 Datapath Allocation and Synthesis

For each action that involves use of datapath logic, a *datapath resource* is allocated. A datapath resource is a self-timed element that communicates with its environment under bundled data constraints. A delay through the combinational parts of the element is modeled by a delay

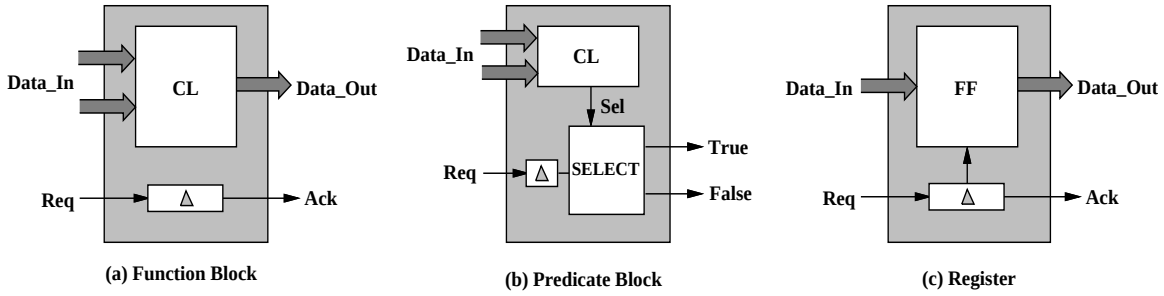


Figure 4.3: Models for datapath resources

matching its worst case performance. To comply with the selected protocol, the resources will differ in their communication with the control logic depending on which protocol has been selected. Their high level behavior however will be the same.

For *event actions* we simply allocate the physical wires that are used for the signals. No other resources are needed since event actions are only signal transitions on wires.

Expressions can occur in the context of channels, assignments, choices and functions. For an expression we will need to allocate hardware for the computation to be performed together with means of detecting the completion of the computation. For this we allocate a *function block* as can be seen in Figure 4.3. The computation part of the function block consists of combinational logic representing the Boolean function to be computed. The completion detection is modeled by a delay on the request/acknowledge wire matching the worst case delay through the combinational logic.

For *variables* we allocate registers which, depending on the selected protocol, are double edge triggered flip-flops for the two phase protocol, or single edge triggered flip-flops when using four phase. A delay on the request/acknowledge wire matches the setup and hold times of the register.

For *choice actions* based on boolean expressions we allocate a *predicate block* as illustrated in figure 4.3. A predicate block consists of a combinational logic part representing the comparison function to be implemented. The combinational block gives out a level signal, "sel", that is high or low depending on whether the expression being tested was true or false. This level signal is then fed into a select block. When the request event reaches the select block, it generates an event on its true or false output depending on the value of the "sel" signal. A delay on the request wire models the worst case delay of the combinational logic.

The functions representing the combinational parts of the allocated datapath resources are later synthesized with ordinary synchronous tools such as Viewlogic and Cadence LAS system. In the current implementation of ACK, communication and expression guards are assumed to be mutually exclusive. Techniques to relax this restriction are standard [2, 7, 38, 32]

and will be added in future versions of ACK.

Currently the delays are computed from the worst case delay of the combinational logic as given by the synthesis tools. Due to the conservative unit delay model used in Viewlogic, the accuracy is far from satisfactory. A more exact technique for modeling of the datapath delays is currently under evaluation and will be incorporated in future versions of ACK.

4.3.2 Control refinement

Once the datapath resources have been allocated we translate the behavioral Petri net into a customized control graph acting on these resources. The control graph communicates with its datapath resources and the environment via handshaking protocols under bundled data constraints. The control graph is generated by a refinement procedure that translates each high level action in the behavioral Petri net into a sequence of event signals by macro expansion. The resulting graph will thus only contain input and output event signals. The following paragraphs will discuss the refinement process on a case by case basis. Only refinement using the two phase protocol is considered. The differences when using four phase will be discussed later.

An *event action* is translated into itself since it already consists of only an event signal.

The expansion of a *channel action* depends on if it is used only as a synchronization action or if it is used for passing data. If it is used only for synchronization, then the action is expanded into a simple handshake on the allocated C-element as seen in figure 4.4(a). For a data channel a handshake on the C-element is first performed synchronizing the sender and receiver. The receiver thereafter performs a handshake on the allocated register used to store the data as seen in figure 4.4(b). If the sender wishes to send the result of an expression over the channel it first generates a handshake to the function block incorporating the expression and thereafter performs the handshake on the C-element.

An *assignment action* is expanded into one or two handshakes depending on if it is a constant or an expression or function that is assigned to the variable. If a constant or variable is being assigned to the variable, a handshake on the corresponding register is made as seen in figure 4.4(c). If the assignment is associated with an expression or function, a handshake is first performed on the resource implementing the expression or function. When the computation has finished, a handshake is then made on the register storing the data as illustrated in figure 4.4(d).

A data dependent *choice action* is expanded to a handshake on the predicate block implementing the Boolean function and the select element. A request is first generated by the controller which then waits for an input event from either the true or false output of the select element as shown in figure 4.4(e).

A *compound action* is refined exactly as its separate actions would be with the difference

that the handshakes for all the actions are generated simultaneously.

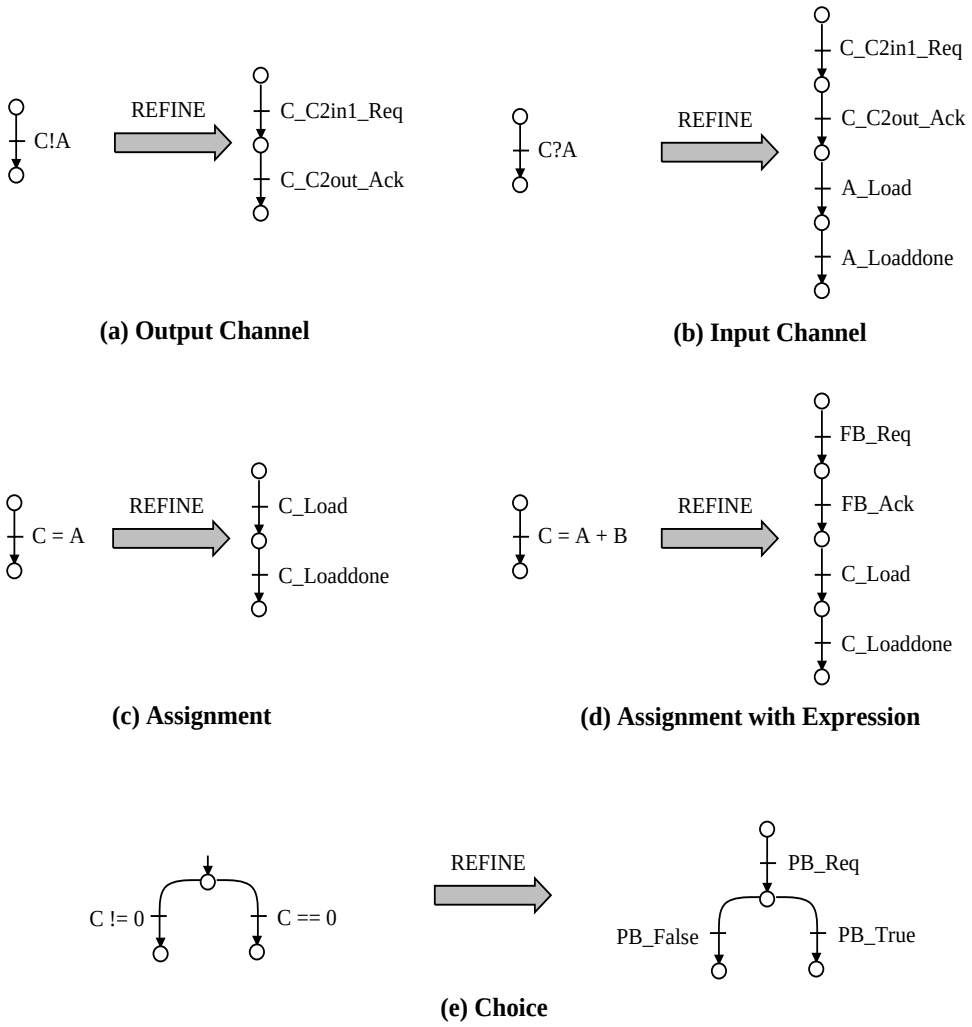


Figure 4.4: Examples of Refinement

In the case of four phase, we must do an extra handshake sequence to bring the wire values back to their initial state (zero). An example of a four phase refinement of an expression assignment action can be seen in figure 4.5. To lessen the impact of this extra handshake a procedure called *reshuffling* is later performed to hide the return to zero handshakes in the handshakes of the following action. Since all calculations are made on the first event (rising edge) when four phase protocol is used, the datapath resources use asymmetric delays to give a quick acknowledge when the signals are returned to zero.

An example of the refinement procedure is illustrated in Figure 4.5. The action $a = b + c$ consists of two subactions. First the expression $b + c$ is to be calculated, then the resulting value

is to be assigned to variable a.The datapath resources needed, an adder for the add operation and a register for the assignment operation have already been allocated. We must now generate the control graph acting on these resources. We do this by first generating a handshake sequence to the adder. The controller first generates an event on the request wire going to the adder. It then waits for an event on the acknowledge wire from the adder which indicates the computation has completed. The delay that models the worst case delay through the adder ensures that the data is stable on the output of the adder once the controller receives the event on the acknowledge wire. Once the addition has been completed, the result must be latched in the register for a.This is done in the same fashion as for the adder computation. The controller first generates a request signal to the register and then waits for the acknowledge signal indicating the data has been stored. The delay in this case models the register's setup and hold times.

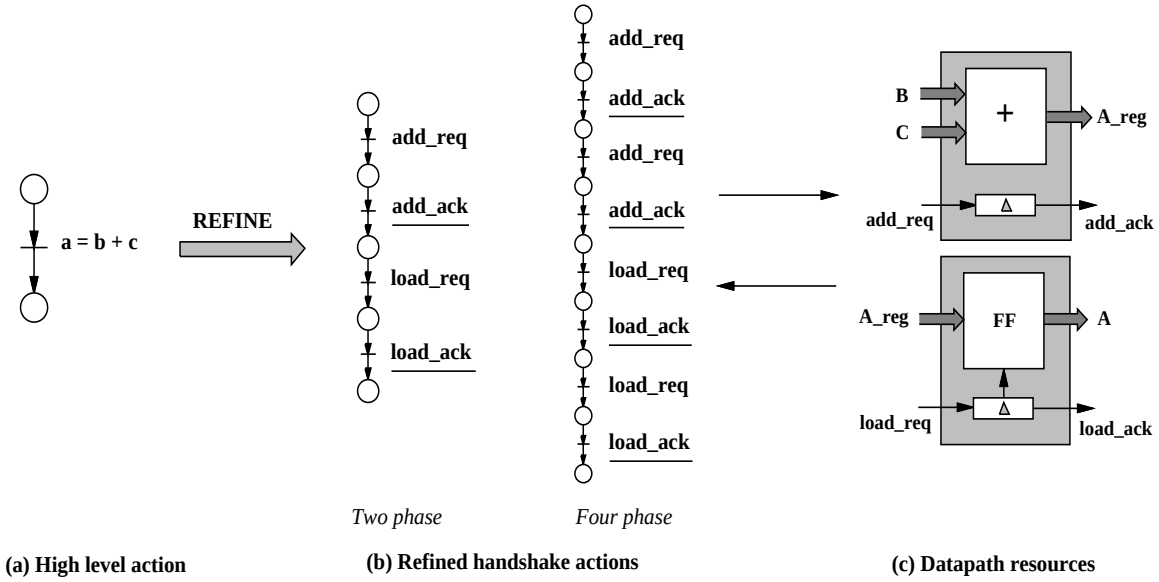


Figure 4.5: A Simple Example of Refinement

4.4 Conclusions

This section has presented the languages used for design specifications and their correspondance to the structural hierachy. The module concept has been discussed and the high level synthesis methods to create a control graph via refinement and a datapath via allocation has been presented.

As the reader may already have noticed, the handshake expansion is quite naive since for all subactions every request signal is echoed back to the controller before a request to

the next subaction can be generated. Possible improvements can be made by allowing the request to flow through several subaction stages before being sent back as an acknowledge to the controller. Such a scheme would decrease the number of in and outputs to and the complexity of the controller resulting in reduced synthesis time and less logic. When sharing resources this method can result in extra logic for the datapath completion detection but this overhead can often be hidden in the matching delays. A flow through scheme for ACK is currently under developement.

Current research also include other techniques to optimize the behavioral Petri net and the refined control graph. Standard techniques for high level optimizations frequently used in synchronous synthesis such as those presented in [17] can be directly implemented. However, optimization techniques directly related to asynchronous structures are also of importance and are the aim of our research efforts. Such optimizations require that we can share datapath resources between different actions. These techniques will be added in future versions of ACK.

The controller generated at this step may be partitioned before burst mode state machines are generated from them. In the next section, we will discuss how to partition large controllers into a set of smaller interacting controllers.

Chapter 5

Partitioning

In this section a technique to partition a centralized control-flow graph to obtain distributed control in the context of asynchronous high-level synthesis is presented. It solves the key problem of handling signals that are shared between the partitions, a problem due to the incompletely specified nature of asynchronous controllers.

Targeting asynchronous finite state machines for logic synthesis gives us several advantages in its global optimization and possibility to perform Boolean optimizations at the gate level. However, the complexity of state machine synthesis sets a limit for the size of controllers that can be synthesized. A centralized controller can also often be more complex, in terms of logic, than a collection of distributed controllers, and thus can have slower signal paths through it. Certain constructs such as loops often execute faster when partitioned due to reduced logic overhead. As feature size decrease and wire delays become significant it is also important to be able to generate controllers local to their corresponding datapath, thus reducing the wire lengths and keeping timing assumptions local. A method for partitioning centralized controllers is therefore necessary as well as desirable.

A key problem in partitioning stems from the fact that asynchronous controllers are, in general, incompletely specified. More specifically, the steps of critical race free state assigment and hazard free logic minimization in burst mode synthesis rely on the fact that the environment of the controller does not present any of the unspecified behaviors. Under this assumption, the sharing of signals between the partitions is a non-trivial problem. For example, suppose an input signal is shared between a collection of partitions. When the environment generates a change on this signal, to which of these partitions must the change be sent to? If the signal is sent to a partition that is not supposed to see it at this time then it is an unspecified input change of this partition and the circuit behavior will not be predictable. The requirements for a partitioning method are thus: (1) it must deal with control flow dependencies between parts of the original graph as well as distribute shared input and output signals in a correct fashion;

and (2) the composite behavior of the system must be the same as for the original centralized graph under the assumption that the system operates in fundamental mode. In this section we will provide a method to address this issue.

5.1 Related Work

In [38], a technique called *process decomposition* is proposed. Process decomposition does not involve signal sharing between incompletely specified machines. Signal sharing is addressed in macromodule based design systems [2, 7] by using additional macromodules such as Toggles [66] and Decision-waits [21] to steer the global input to the correct sub-controller. Since macromodule libraries contain only a limited number of macromodule types, distributed control realizations based on macromodules are often inefficient [26]. In [12, 55], a method called *contraction* has been suggested as a decomposition technique for signal transition graph (STG) specifications. Contraction preserves the global nature of the controller. It does not turn a large grain controller into many smaller grain controllers and is therefore unable to take advantage of spatial locality.

The partitioned synthesis problem addressed in this section is: given a centralized control graph and a set of partitions chosen by the designer, how do we synthesize separate controllers for each of the partitions that correctly orchestrate control flow between the partitions and correctly handle signal sharings? The identification of the partitions is not addressed here, though a few automatable heuristics, such as keeping logically unrelated iterative loops that share signals in separate partitions, usually yield good results in terms of increased performance and reduced logic complexity.

5.2 Partitioning Methodology

This section will consider the classification of graphs and subgraphs of the original control graph. Methods for partitioning sequentially as well as concurrently executing parts of the original graph along with arrangements necessary for sharing signals between these partitions will be presented.

5.2.1 Graph Classification

The centralized control graphs which form the input to the partitioning phase of ACK are single threaded state machines with a limited form of fork-join concurrency called SFJ graphs. Such a graph can be further categorized into substituent single threaded subgraphs called STS graphs. An STS does not contain fork-joins. These graphs will be defined next.

The SFJ graphs are triples $G_{SFJ} = (P, T, F)$ where $P = \{p_1, \ldots, p_n\}$ is a set of places and $T = \{t_1, \ldots, t_m\}$ is a set of transitions, where n and m are finite, and $F \subseteq (P \times T) \cup (T \times P)$ is a flow relation.

The set of transitions T is divided into sequential transitions, T_s, fork transitions, T_f and join transitions, T_j, All sequential transitions T_s have an in-degree and an out-degree of one and is annotated with a non-empty set of event signals, a burst. Transitions in T_f and T_j are labeled by an empty burst, ϵ. There is a one to one correspondance between the fork transitions and join transitions such that for each $t_f \in T_f$, there is exactly one $t_j \in T_j$ such that the out-degree of t_f is the same as the in-degree of t_j. N is said to be the degree, meaning the number of threads, of the fork-join pair (t_f, t_j). Such fork-join pairs have graphs with only one input and one output place between them such that the i:th output place of fork transition t_f is the input place of a single threaded subgraph (STS). The output place of this subgraph then is the i:th input place of t_j.

An STS, $G_{STS} = (P_{STS}, T_{STS}, F_{STS})$ is a subgraph of G_{SFJ} where $P_{STS} \subseteq P$, $T_{STS} \subseteq T_s$, and $F_{STS} \subseteq F$ is the flow-relation restricted to P_{STS} and T_{STS}. Note that T_{STS} does not include fork-join transitions, so all transitions in T_{STS} have an in-degree and an out-degree of one. An STS can have a finite set of input places p_{in} and a finite set of output places p_{out}. The number of input and output places of such an STS is indicated by the notation $\mathrm{STS}_{p_{in}p_{out}}$. With this notation each thread of a fork-join is an STS_{11} since they are required to have only one start place and one end place.

In order to generate legal burst-mode machines [49] from the partitions through burst-mode reduction [26] (discussed in section 6), the original SFJ graphs must obey the following restrictions, in that they are (1) initially quiescent, and attain quiescence infinitely often; (2) deterministic, and (3) obey the subset property [49].

5.2.2 Create Partitioned Controllers

The goal of partitioning is to generate a stand alone controller for each of the specified partitions in the original graph. Each of these partitions then support a portion of the original graph and should be invoked whenever that portion of the graph is to be executed.

There are two types of partitions in an SFJ graph. One type is *required partitions* which consists of fork-joins. Since the approach used for synthesis of finite state machines cannot handle concurrent threads in one and the same state machine, we must separate the individual threads in the fork-joins and put them in separate single threaded graphs. The other type is user defined partitions which consists of single threaded subgraphs. Since an STS describes a purely sequential flow (no fork-joins) the issue of partitioning such a controller is to split its corresponding STS into separate STS's corresponding to the specified partitions.

The problem then is to make sure that the behavior of the decomposed controllers is consistent with that of the centralized controller. This require that the decomposed controllers are able to be invoked repeatedly and that they correctly handle the flow of execution.

Partitioning of Concurrent Threads

We will first consider the partitioning of fork-joins. Each thread in a fork-join is represented by a separate STS_{11}. All STS_{11} subsiding within the same fork-join pair must be disjoint from all other STS_{11} in that fork-join pair. This means they may not share any place or transition, i.e., they are completely separate threads. To simplify the exposition, we make two assumptions. The first is that no two bursts labeling transitions contained in two different STS's of the same fork-join involve the same wire name. This means signals are not allowed to be shared between separate threads in a fork-join. The second assumption is that a signal occuring within a fork-join may not occur in a choice in any other partition.

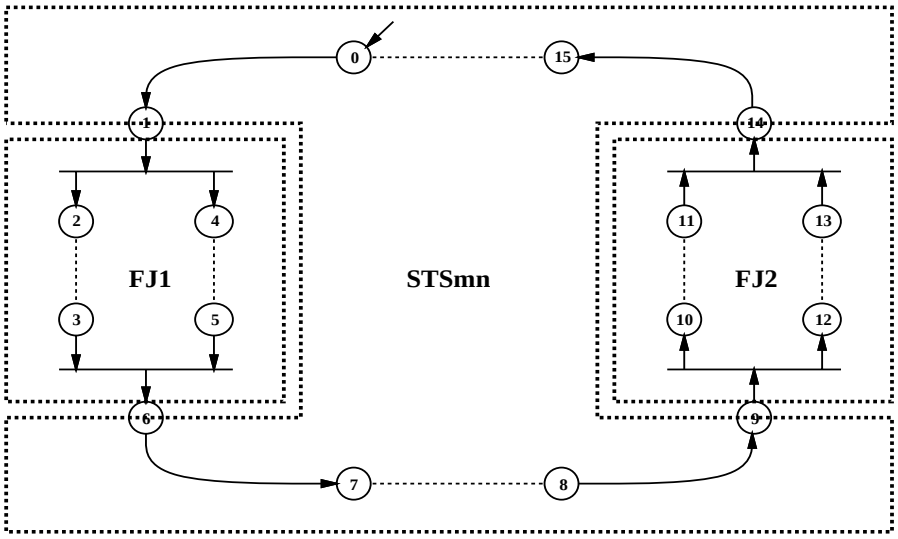
By partitioning all fork-joins we want to create the following from the original SFJ graph:

- A set of STS_{11}'s representing the separate fork-join threads. The STS_{11}'s belonging to the same fork-join will be invoked at the same time and will thus execute in parallel.
- An STS_{mn} representing the original SFJ graph but with all fork-joins removed. This is the part of the graph that will be responsible for invoking the fork-join threads.

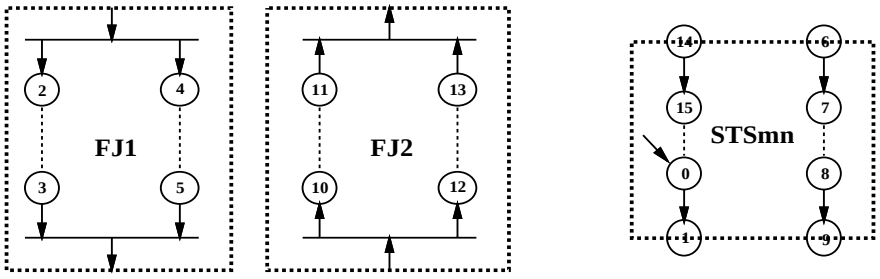
Since the STS's will be implemented as separate stand-alone controllers, we need to make them able to repeatedly execute their corresponding portion of the original graph. We must also make sure that they are only invoked when their corresponding part of the original graph is supposed to execute. To simplify the initial exposition we will assume that the SFJ contains only required partitions (fork-joins). This assumption will be relaxed momentarily.

Partition the SFJ. Assume we want to partition an SFJ graph as illustrated in figure 5.1(a). We must first divide the original graph into subgraphs corresponding to the specified partitions. We do this by first removing the two fork-join pairs from the original SFJ graph. As a result we get a set of two fork-joins along with an STS_{mn} representing the rest of the graph as can be seen in figure 5.1(b). Now we divide each of the fork-joins into their constituent single threaded subgraps, STS_{11}'s. We then assign a unique partition number i to each of these STS_{11}'s and a unique partition number p to the STS_{mn}.

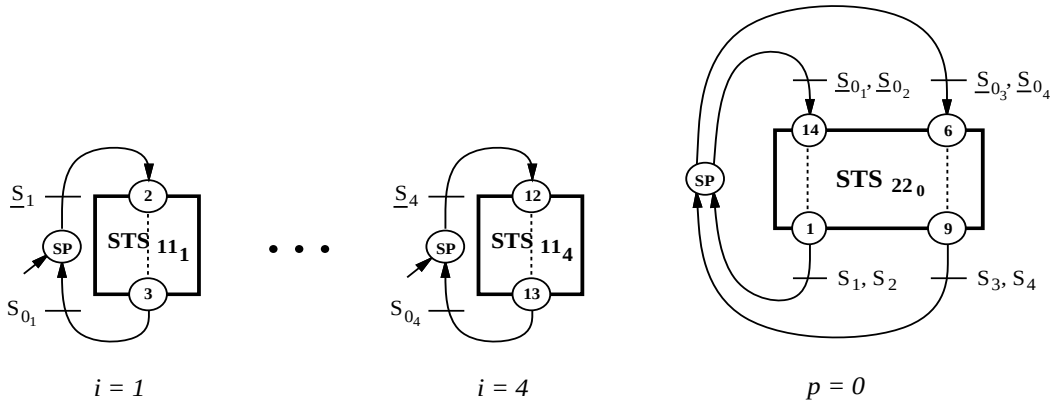
Generate stand-alone controllers. We must now alter the partitioned STS's to become stand-alone controllers and ensure that they execute when they are supposed to. This is achieved



(a) Original SFJ graph



(b) Separated fork-joins and remaning STSmn



(c) Partitioned graph containing only single threaded graphs

Figure 5.1: Example of partitioning.

by making each graph cyclic and by introducing handshake signals between them in order to control their order of execution using the following procedure. Since the STS_{11}'s execute in parallel they must each have their own handshake signals for completion detection.

We will first introduce a *start place* (SP) for each STS_{11} as well as the STS_{mn}. For each STS_{11} belonging to the fork-joins we create a transition going from the SP to the input place of the STS_{11}. This transition is then labeled with an input signal $\underline{S_i}$, where i is the partition number of the current STS_{11}. We also create a transition going from the output place of the STS_{11} to the SP. This transition is then labeled with an output signal S_{p_i} where p is the name of the partition following the fork-join, in this case the STS_{mn}. The $\underline{S_i}$ signals then corresponds to the start signals of the four STS_{11}'s while the S_{p_i} signals correspond to their done signals (the start signals for the STS_{mn} partition).

We must now introduce the same kind of handshake signals for the STS_{mn} since this is the partition responsible for invoking the fork-join threads as well as detecting their completion. In this case the STS_{mn} is a STS_{22} since it has two input and two output places. For this STS_{22} we will create two transitions, one transition going from the SP to each of the two input places. We have two fork-joins, wherefore each of the transitions therefore corresponds to the completion detection of a separate fork-join. Since each fork-join consists of two STS_{11}'s each transition is therefore labeled with a burst of two input signals $\underline{S_{p_i}}$, corresponding to the completion signals of the two STS_{11}'s. We will now create a transition from each of the two output places of the STS_{22} to the SP. These transitions then should each generate start signals to the separate STS_{11}'s in the fork-joins. We therefore label each of the transitions with an output burst of two S_i signals corresponding to the start signals of the STS_{11}'s of each fork-join.

The separate partitions of the original SFJ graph have now been made stand-alone controllers. They are now made cyclic which allow us to invoke them infinitely often and have been labeled with start and completion handshake signals so that they by themselves can determine when to execute. The graphs of the resulting stand-alone controllers are illustrated in figure 5.1(c).

By the procedure described above, the separate STS_{11}'s of the fork-joins and the STS_{22} representing the main body of the controller have now been converted into self managing cyclic controllers. To ensure a correct flow of execution, event signals have been introduced to work as start and done signals for the decomposed controllers.

Partitioning of Single Threaded Graphs

We will now relax the assumption that we only have required partitions. This means we can also have user defined partitions in the STS_{11}'s as well as in the STS_{mn}. To simplify the exposition we will only consider the case where the STS_{mn} is allowed to have user defined partitions.

However, an algorithm where the STS_{11}'s are allowed to have user defined partitions has been implemented in ACK. Now that we have a method for partitioning and how to make stand-alone controllers that communicate via handshaking for each partition, this is easily extended to cover user defined partitions as well.

Partitioning of a single threaded graph proceeds much the same as for concurrent threads. The main difference between partitioning of single threaded graphs is that each partitioned graph may have multiple input and output places and that the resulting partitions execute in a sequential fashion. The fact that these partitions are allowed to share signals introduce a subtle problem in terms of ensuring correct circuit operation.

Partition the STS A single threaded graph to be partitioned is first divided into its constituent STS_{mn} subgraphs and each of these is then given a unique partition number p. As for the partitioning of concurrent threads, a startplace is introduced for each partitioned subgraph. Transitions annotated with event signals are then added to control the flow of execution between the partitions. These start and completion event signals are labeled with the corresponding partition numbers in the same fashion as for the required partitions.

When partitioning of single threaded controllers is allowed we need to know which partition follows each fork-join in order to determine the partition number p which is included in the completion signal of each STS_{11}. We therefore need to partition the STS_{mn} after we have removed the fork-joins from the original SFJ graph. In the partitioning of the STS_{mn} each partition is assigned a unique partition number p. This number is then used when labeling the completion signals of the fork-join threads. The same holds true for each STS_{mn} which needs the partition number of each STS_{11} for the their labeling of the input and output transitions of each SP.

The procedure of partitioning is given in algorithm 5.2. The first step is to separate the SFJ into an STS_{mn} and a set of fork-joins. The STS_{mn} is then partitioned and each partition is assigned a unique partition number p. The fork-joins are then partitioned into their constituent STS_{11}'s which are each assigned a unique partition number i. To each partition that has been created an SP is added and annotated with transitions labeled with start and completion signals based on the partition numbers.

5.2.3 Signal Sharing

After the partitioning step we have a set of interacting controllers in the form of Petri net graphs. The approach later used for state machine synthesis assume the state machine description to be incompletely specified. This means that in a given state of a controller, only signals that are specified to change in the output transitions from that state are allowed to change value at

```
Type place_type = it : set of in-transitions
                  ot : set of out-transitions

Type STS_type = ip : set of input places of place_type
                tp : set of internal transitions and places
                op : set of output places of place_type

Partition_SFJ(SFJ):STSs of STS_type
   P_STS = {};
   STS_mn ∪ {FJ_1,...,FJ_k} = SFJ;
   {STS_mn1,...,STS_mnl} = STS_mn;
   foreach FJ_j ∈ {FJ_1,...,FJ_k}
      {STS_11_1,...,STS_11_g} = FJ_j;
      foreach STS_11_i ∈ {STS_11_1,...,STS_11_g}
         create SP of place_type;
         SP.ot = S_i;
         STS_11_i.ip.it = SP.ot;
         p = id number of partition following FJ_j;
         SP.it = S_p_i;
         STS_11_i.op.ot = SP.it;
      end
      P_STS = P_STS ∪ {STS_11_1,...,STS_11_g};
   end
   foreach STS_mn_p ∈ {STS_mn1,...,STS_mnl}
      create SP of place_type;
      {q_a,...,t_b} = id numbers of reachable partitions/FJs (q,...,t)
                      and their corresponding input places (a,...,b);
      SP.ot = {S_p1,...,S_pr};        // {1,...,r} = STS_mn_p.ip
      STS_mn_p.ip.it = SP.ot;
      SP.it = {S_q_a,...,S_t_b};
      STS_mn_p.op.ot = SP.it;
   end
   P_STS = P_STS ∪ {STS_mn1,...,STS_mnl};
   return(P_STS);
end
```

Figure 5.2: Algorithm for partitioning of SFJ's

that time. If this restriction is not fulfilled the synthesized circuit will malfunction.

The partitioning approach described above often result in signals being shared between partitions. As mentioned earlier concurrent threads are not allowed to share signals and only one sequential thread of a given controller is allowed to execute at any given time. Under these restrictions it is easy to see that a decomposition resulting in two or more partitions sharing the same signal will violate the assumptions made in the synthesis of incompletely specified state machines.

Take the simple example illustrated in figure 5.3. In this example the controller has been divided into two partitions using the approach described earlier. The decomposition of the controller resulted in that the two partitions share the input signal $\underline{a}$ and the output signal

b. This means that both partitions are sensitive to input signal $\underline{a}$, that is, $\underline{a}$ makes a transition somewhere in each of the partitions. Partition 1 and 2 in the figure start in states 0 and 5 respectively. In these states partition 1 is the active partition and is supposed to see an event on input $\underline{a}$ while partition 2 is waiting to be invoked by an event on input $\underline{S_2}$. However, when the environment generates an event on signal a both partition 1 and 2 will see it since they both have a in their input sets. Partition 1 which has $\underline{a}$ as a specified next transition will absorb the signal change and correctly generate an output event on S_2. Partition 2 however, does not have $\underline{a}$ as a specified next transition and will therefore absorb an unspecified signal transition, thereby entering an unspecified state resulting in malfunction of the circuitry implementing the partition. Sharing an output signal, such as b in the example, between partitions also results in problems since an event on either the output b from partition 1 and the output b from partition 2 somehow must be merged to the same wire.

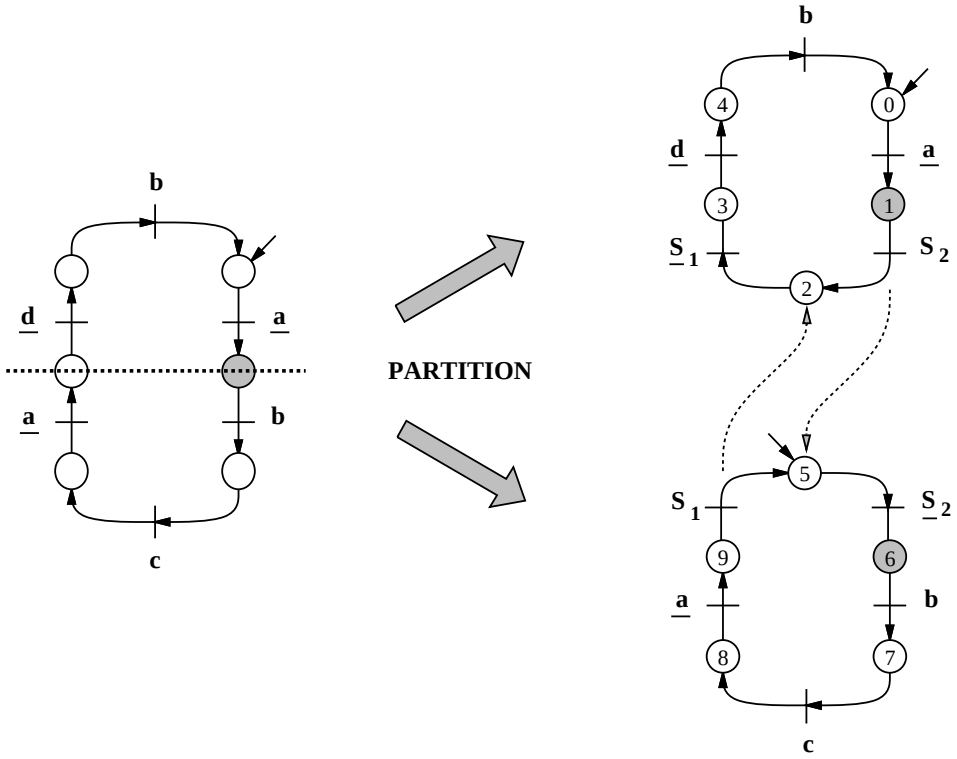


Figure 5.3: Example of signal sharing.

For the partitioning approach presented above to generate partitioned controllers that correctly interacts with the environment we must find solutions for sharing input and output signals between partitions. Two possible solutions to these problems are presented in the following subsections. The method presented first treats the case where the two phase signaling protocol has been used in the refinement step and the second method deals with the four phase protocol.

Two Phase - Resolving Output Sharing

A solution for the sharing of output signals must involve a method for merging several separate output signals onto the same wire. Merging signals having arbitrary values onto the same wire might seem to be an unsolvable problem and would be so without explicit knowledge of the signal behavior and characteristics of the outputs. Even with such knowledge, merging output signals could pose problems requiring complex circuitry to solve.

Fortunately the way our controllers are specified and the way our partitioning approach works gives us sufficient knowledge of both signal behavior and characteristics to allow an efficient solution to the output sharing problem. First, our partitioning approach guarantees that controllers sharing output signals can only generate events on these that occur in a sequential fashion. Second, we are only interested in when there is an event on a wire, not the actual value of the signal on the wire. To merge output signals having these characteristics it would suffice to have a device that for an event occurring on an arbitrary input generates an event on its output. Such an approach would be desirable since the order in which the events on the inputs are generated does not matter which would make the solution independent of the original control specification. Under the conditions outlined the problem can be solved by merging the signals with an ordinary XOR gate. The specification of an XOR gate is that for every change in value on any of its inputs the output changes value. This behavior complies with the stated requirement that for a signal event on an arbitrary input an event on the output should be generated.

The approach for solving the problem of output sharing thus becomes the following. For every occurance of a shared output signal o in partition x, rename o to o_x. In our example in figure 5.4, b would be renamed to b_1 in partition 1 and to b_2 in partition 2. In the final circuitry implementing these partitions these two outputs are then connected to the inputs of an XOR gate. The output of the XOR gate then becomes the output signal b.

Two Phase - Resolving Input Sharing

As mentioned earlier, sharing of an input signal requires that only the partition that currently is activated may see an event on that input. If other partitions were to absorb the same input event they would enter an incorrect state and thereby malfunction. A solution to sharing input signals between partitions therefore require a method for distributing the shared signal to the right controller at the right time. To solve this problem we need explicit knowledge of *when* a certain event on the shared input is to be distributed to *which* control partition.

We therefore need a device that given an event on the shared input can generate an event on an output going to the correct control partition. Since no general characteristics of the input

signals can help us discern which control partition the event is supposed to reach, a general device that is independent of the original controller description as that for shared outputs cannot be used for input signal distribution. The conclusion is therefore that without knowledge of the original controller specification we cannot predict in what fashion the input signals are supposed to be distributed to the different partitions. We therefore need a customized device for distributing the signals.

Since our partitioning approach guarantees that all partitions are sequential and that no two concurrently executing threads may share the same signal, the events on an input wire is guaranteed to occur in a sequential fashion. Because of this, the behavior of a signal is exclusively determined by the occurrance of branches and the signal in question. Our approach for generating the graphs describing the behavior of the customized signal distribution devices is therefore as follows.

For a shared input signal we first make a copy of the original controller, that is, as it was before the partitioning step. Since the shared signal of interest can only occur sequentially, we then remove all fork-join threads that do not contain it. We then remove all signals in the graph except choice signals and the signal of interest. We now have a graph that completely describes the behavior of the shared input signal. We now need to generate output responses going to the right control partition for each occurrence of these input events. An output signal is therefore added after each occurrence of the input signal of interest. This output signal is named after the input signal and the corresponding partition in which it would occur had the copy of the original graph been partitioned. The graph that describes the Input Translator State Machine (ISM), as it is called, is now completed. What remains is to also rename the input signal of interest in the partitions, as created by the partitioning step. The new names of the shared input signals of these partitions then becomes the original name concatenated with the partition name it occurs within.

The ISM for the shared input signal $\underline{a}$ in the example illustrated in figure 5.4 is thus created as follows. Since the original graph contains no branches there are no choice signals present. We therefore simply remove all signals except $\underline{a}$ from the original graph. We then add an output signal a_1 after the first occurrance of $\underline{a}$ and a_2 after the second occurrance of $\underline{a}$. We then rename the corresponding input signals in each partition such that $\underline{a}$ in partition 1 becomes $\underline{a_1}$ and $\underline{a}$ in partition 2 becomes $\underline{a_2}$.

Four Phase - Resolving Output Sharing

The same arguments as those used in the case of sharing output signals using the two phase protocol must also hold true for the four phase case to ensure correct circuit behavior.

However, since the four phase protocol gives us additional information about the state

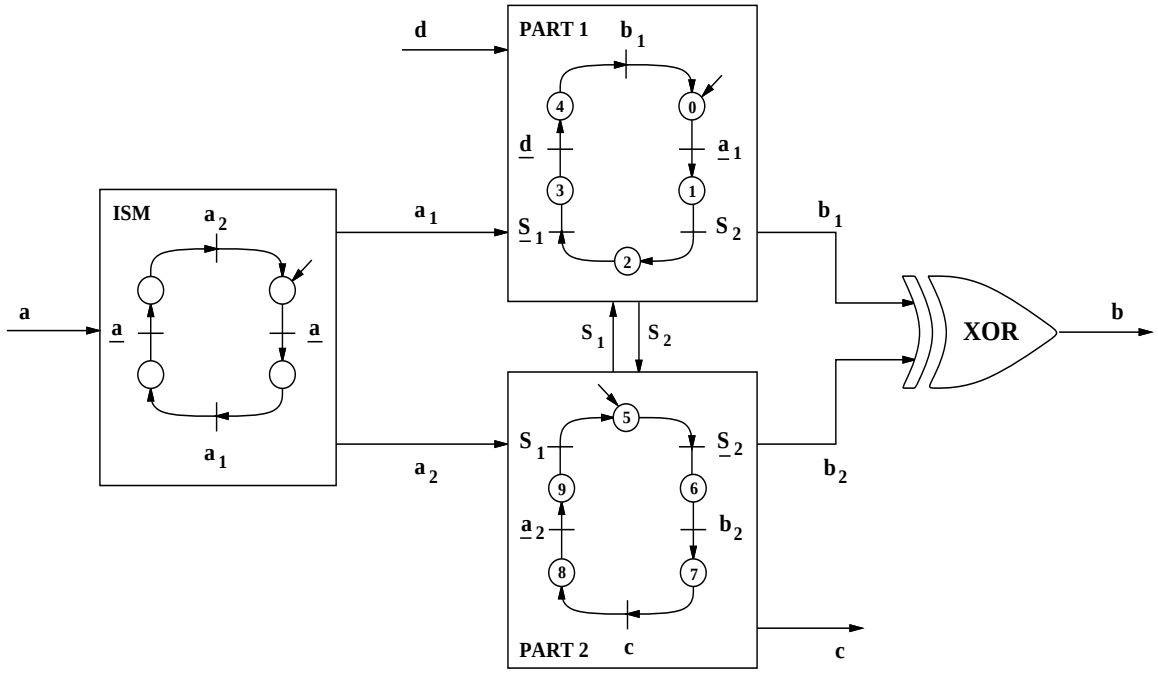


Figure 5.4: Solution to input and output signal sharing using two phase protocol.

of the signals after a completed handshake sequence, less complex logic can be used to handle output sharing. The four phase protocol ensures that after a completed handshake sequence, the signals involved will be returned to the same values they had before the handshake was initiated. Since each signal will be initialized to a logic 0 by later steps in the circuit synthesis, an ordinary OR gate will fulfill the required conditions outlined for output sharing. Using an OR gate instead of an XOR gate also offers a slight advantage in that the OR gate is a better current driver and thus is faster.

The renaming procedure is the same as for the two phase case. Thus, for every occurance of a shared output signal o in partition x, rename o to o_x. In the final circuitry implementing a set of partitions these renamed outputs are then connected to the inputs of an OR gate network. The output of the OR gate network then becomes the output signal o.

Four Phase - Resolving Input Sharing

As for the two phase case we need to generate customized controllers in order to ensure that only the currently active partition will see changes on the shared inputs. Although applicable also for four phase, the method of using ISMs to handle signal sharing can be greatly optimized to reduce the delay introduced by the ISMs when using this protocol.

When using ISMs to solve sharing of input signals a "copy" of that signal must be generated and sent to each of the partitions sharing the signal at the right time. The logic

complexity of such ISMs increase both with the number of choices and the number of occurrances of the shared signal in the original Petri-net graph. This can result in quite complex logic for many real designs. This means that much of the delay gained by reducing the complexity of the control logic by partitioning is lost when these are sequentialized with the ISMs. An important factor in designing high performance control circuitry is therefore to optimize the signal sharing logic wherever possible. In the case of using four phase signaling a much more efficient method, adding only the delay of a single two-input AND gate for each shared signal, can be used.

A signal using the four phase protocol is always guaranteed to return to its initial value after a handshake sequence is finished. We therefore know that the current value of each signal is a logic 0 as we pass control from one partition to another. Using this knowledge we can deduce that the signal sharing logic does not have to keep track of the current state of the shared signals as a new partition is entered, as was the case when using the two phase protocol.

One solution that can be used is therefore to let the original signal be directly distributed to each partition without use of ISMs and instead *block* the signal whenever a partition is not supposed to see it. Such blocking of shared signals can be done using a simple AND-gate which has an enable signal on one of its inputs and the shared input signal on the other. Whenever the partition in question is activated, the enable signal goes high and thus enables any changes on the shared input signal to propagate to the controller. Whenever control is passed to another partition the enable signal goes low, blocking any further changes on the shared input. This way, no unspecified signal changes will reach the controller during the time it is passive. The enable signal is generated by a state machine that is sensitive only to the signals involved in passing control between the partition in question and its reachable partitions.

Take the example shown in figure 5.5. The enable signal for the start partition, partition 1, generated by $SM1$ is initially set to high. The enable signal for the passive partition, partition 2, is initially set to low by $SM2$. The AND gates thus hinder any signal changes on wires c, d, and e from reaching the control logic for partition 2. When partition 1 has finished executing it passes control over to partition 2 by the handshake sequence $S2r \rightarrow S2a \rightarrow S2r \rightarrow S2a$. When $SM1$ sees the events on $S2r$ it disables the AND gates by setting the enable signal to low. At the same time, when $SM2$ sees the events on $S2r$ it sets its enable signal high, thus enabling changes on the shared signals to propagate to the control logic of partition 2. Similarly, when control is passed over to partition 1 again via the handshake sequence $S1r \rightarrow S1a \rightarrow S1r \rightarrow S1a$, $SM2$ will set its enable signal low while $SM1$ will set it high.

Since a design often contains more choices than partitions and partitions usually have very few input and output places (as defined earlier), and since we do not have to be sensitive to when the shared input signal is changing, the logic for the state machines (SM) generating the enable signals is very simple. The number of literals actually grows only linearly with

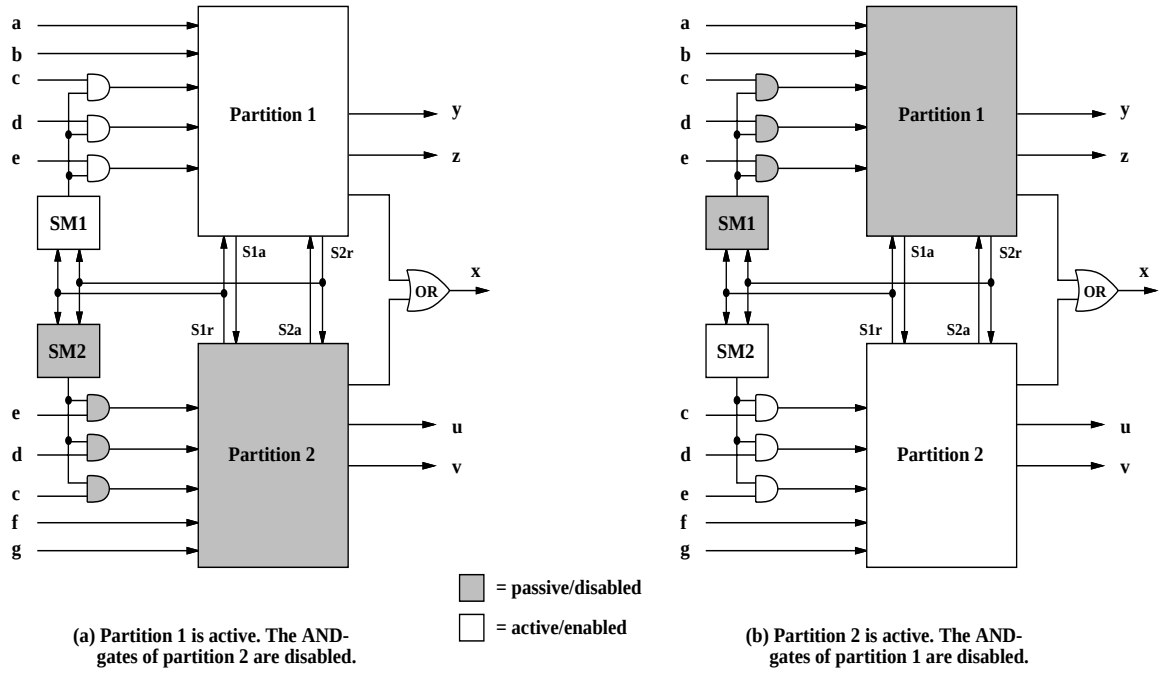


Figure 5.5: Solution to input and output signal sharing using four phase protocol.

upper bound $O(1+n)$ where n is the number of input and output places of the corresponding partition. The important thing to observe though is that the sequential delay introduced by the signal sharing logic, the AND gates, is held *constant* regardless of the complexity of the Petri-net graph and the number of partitions.

This method not only gives an advantage in reduced delay compared to the propagation through the ISMs, but also makes it possible to localize the logic for input sharing to each of the controllers. In fact, the enable signal can be included as an output in the control logic of the partition itself, making fundamental mode delay analysis easier. Since the sharing logic for a signal always consist of an AND gate it is also possible to merge it with the rest of the control logic, something that may give performance advantages when considering gate decomposition. Another advantage of the four phase signal sharing approach is that there is no restriction on sharing signals pertaining to choices since the signal sharing logic is not dependent on the actual graph flow inside a partition. The constraint that for the two phase protocol restricted us not to use a signal that occur in a fork-join in a choice in other partitions can therefore be relaxed when using four phase.

5.2.4 Partitioning Constraints

Using the partitioning method presented in this section in the scope of the refined control graphs produced by the high level synthesis procedure of ACK requires that some constraints are met

in order for the complete circuit after synthesis to work correctly after partitioning.

1. For the two phase protocol: No input signal occuring in a fork-join may be used within a choice in any other partition or fork-join.
2. It is a requirement that a partition is not introduced within the boundaries of a refined handshake sequence corresponding to a high level action from the behavioral Petri net graph.
3. The interaction between the partitioned machines is guaranteed to work correctly if and only if they also operate under fundamental mode constraints.

The reason for the first requirement comes from the ISMs requiring that all signals pertaining to a choice must be included in the graph. Assume a signal i is allowed to occur in a fork-join thread FJa_1 as well as in a choice in some other partition. Assume we have an input signal s that occur in the other thread FJa_2 of the mentioned fork-join is shared with some other partition. The ISM graph for s must then contain all occurrences of all choice signals in the original control graph. This includes signal i. Since we also must keep the signal of interest, s, the removing of all threads in a fork-join except that which contain s will result in the removal of i in FJa_1. Since the ISM will still have i in its input set, an unspecified input change will take place for i when FJa is executed. This requirement can be relaxed by allowing the ISMs that have this problem to be sensitive to selected outputs $(i_1, \ldots, i_k)$ of the ISM of the signal which's fork-join thread was removed, instead of using the original input signal (i). This procedure however, will not be described here.

There are three reasons for the second requirement. First, allowing one partition to initiate a handshake sequence and another to complete it would increase the complexity of fundamental mode analysis. It might also require insertion of extra delays on signal wires since handing control over to a partition might take longer than for the acknowledge signal to the initiated handshake to arrive to that partition. This would slow down overall circuit performance. In the case of four phase refinement, allowing partitioning of a handshake sequence would also mean that the signal values will not be returned to zero within the partitions sharing the handshake sequence. If the signals pertaining to such a handshake sequence would be allowed to be shared in this manner they have to be treated as two phase signals in the separate partitioned graphs, thus not exploiting the possible advantages of using four phase signaling protocol the designer might have had in mind.

The reason for the third requirement is inherent from the way burst mode circuits operate and will be further discussed in section 7. With a circuit operating under fundamental mode constraints we mean that after absorbing a complete input burst, the circuit must attain quiecense before the next input burst is allowed to arrive at its inputs.

5.3 Results and Conclusions

We have conducted comparisons between centralized and partitioned controllers on a large number of examples, some of which are shown in Table 5.1. Apart from making it possible to synthesize larger designs, partitioning can also decrease synthesis time by several orders of magnitude. Partitioning also often significantly decreases the number of literals in the synthesized design and often increases the overall controller performance compared to that of a centralized implementation.

In Table 5.1 we show the partitioning results for a CD Player Error Corrector from [32], a Barcode Reader from the High Level Synthesis Design benchmarks [52] adapted to asynchronous operation, an iterative implementation of the Greatest Common Divisor algorithm, a Factorial computation unit, and a synchronization Loop example. For the CD Player Error Corrector and the Barcode Reader the synthesis of the centralized controllers did not complete due to the complexity of the synthesis task. The results for these are marked with *n.a.*

In the table the *Number of BM transitions* column is a measure of controller complexity and shows the number of burst mode transitions in the specification of the controller. *I/O size* shows the size of the input and output set of the controller, *Synthesis time* shows the time in seconds for burst mode synthesis and *Number of literals* stands for the number of literals in the implementation of the controller. In the *Controller* column *Centralized* means the centralized contoller, *Part* x means partition number x and *ISM* x means Input Translator State Machine number x.

For the examples where the centralized controllers finished synthesis, a layout was generated from a two level standard gate implementation and the performance between the centralized and partitioned controllers was measured. The comparison showed that despite the extra delay introduced by adding ISMs and XORs to solve input and output signal sharing there was actually an average performance increase for the total circuit after partitioning of between 10 to 20%. This is due to the reduced complexity of the partitioned controllers compared to the centralized controller. Note that this comparison only exploited performance advantages due to temporal locality. Partitioning also gives us the possibility to take advantage of spatial locality, which as feature sizes get smaller and wire delays become significant, is an important factor for high performance designs.

In this section, we have presented a method to deal with the partitioning of incompletely specified asynchronous controllers that share signals. This work specifically provides a partitioning method in the context of asynchronous high level synthesis methods that target state machine controllers, although the basic ideas can be extended to other asynchronous partitioning problems. To make the partitioning approach easier to understand, a number of

Controller	Number of BM transitions	I/O size	Synthesis time	Number of literals
CD Player Error Corr.				
Centralized	1824	68	n.a.	n.a.
Part 1	110	18	800	94
Part 2	32	29	220	96
Part 3	28	25	140	122
ISM 1-3	81	13	230	63
ISM 4	16	6	90	14
ISM 5	7	14	80	92
Barcode Reader				
Centralized	960	49	n.a.	n.a.
Part 1	26	14	34	14
Part 2	40	8	25	3
Part 3	72	19	500	13
Part 4	26	23	53	43
ISM 1-2	56	8	22	14
ISM 3	64	9	28	53
GCD				
Centralized	126	25	33420	207
Part 1	22	15	30	33
Part 2	72	18	340	12
ISM 1-2	48	7	25	14
Factorial				
Centralized	44	20	620	88
Part 1	12	13	24	6
Part 2	28	15	36	9
ISM 1-2	16	5	20	14
Sync. Loop				
Centralized	32	5	38	199
Part 1	14	5	20	58
Part 2	12	6	16	3

Table 5.1: Results for partitioning

simplifications has been made such as not allowing threads in a fork-join to share signal wires and that a signal corresponding to a choice occuring within a fork-join may not occur in a choice in any other partition. For this reason the algorithms for generating handover between partitions and for generating the ISMs has also been kept simple. We have shown that all the simplifying assumptions made in this section can be relaxed and more efficient algorithms have also been implemented. These solutions, however, are not presented here since they would complicate the exposition.

Chapter 6

Burst Mode State Machine Generation

At this point, we have refined and partitioned state machine controllers in the form of Petri net graphs. In this section we will consider a method to translate these graphs to burst mode representations. We will first describe the concept of burst mode machines, or graphs, in subsection 6.1. We will then consider the translation of two phase Petri nets in subsection 6.2. In subsection 6.3 we will present the differences in the method used for translation of four phase Petri nets.

6.1 Burst Mode Machines

A burst mode machine is a Mealy style state machine in which every transition is labeled with pairs I/O where I is a non-empty set of polarized signals[1] signal transitions called the input burst and O is a possibly empty output burst. After reset, a burst mode machine must be quiescent and not generate any outputs until a complete input burst has been consumed. After each time an input burst has been received and a possible output burst has been generated, the machine must attain quiescens before a new input burst is allowed to occur.

A burst mode machine is a Mealy style description of a finite state machine. It consists of a set of states (circles) and directed arcs (arrows) connecting them. The arcs are annotated with a pair I/O where I is a non-empty set of input signals called an *input burst* and O is a possible empty set of output signals called an *output burst*. A burst represents a collection of signal changes on the wires of the corresponding circuit implementation. The signals are annotated with $+$ and $-$ signs to indicate whether the signal is supposed to make a rising or

[1] Annotated with + or - sign to distinguish rising and falling transitions.

falling transition. A more precise definition of a burst mode machine can be found in [49].

An example of a burst mode specification is illustrated in figure 6.1. If not specified otherwise, all inputs, outputs and state variables are initially set to zero. The state machine initially starts in state 0. The arc leading to state 1 is annotated with an input burst $a+$ and an output burst $x+$. The machine will stay in state 0 until a high going transition on wire a has occurred. After absorbing the input change the machine will generate a high-going transition on output wire x and move to state 1. State 1 is a choice state with two outgoing arcs with bursts $b+/x-y+$ and $a-/z+$ leading to states 2 and 3 respectively. The next state now depends on which input burst the choice is resolved through. Note that the input bursts of the choice are mutually exclusive, that is, they cannot both occur together. If the next signal change we see is a rising transition on wire b then we generate the output burst $x-y+$ and move to state 2, while if it is a falling transition on wire a we generate a rising transition on output signal z and move to state 3. If in state 2, we wait for the input burst $a-b-$. Note that these transitions can occur in arbitrary order with arbitrary delay in between. When both a and b have gone low we set output y low and move to the initial state 0. If in state 3, we wait for input event $a+$ and generate output event $z-$ and move to state 1 again. Note that a burst mode machine containing no empty output bursts describes a delay insensitive behavior with fundamental mode restriction. (Places in the burst mode graphs will later be omitted for figures that otherwise would be too large to show.)

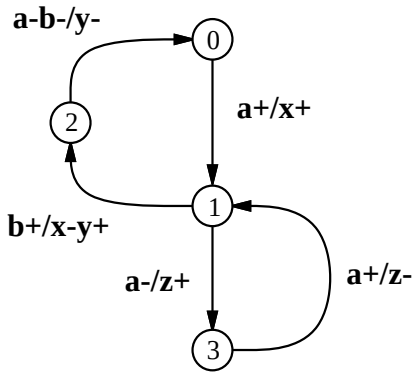


Figure 6.1: Example of burst mode specification

Beside the restriction of non-empty input bursts, there are a number of other restrictions that a burst mode machine must comply with in order for the synthesis procedure and implementation to work correctly. A burst mode circuit must comply with the requirement of quiescens. This means that after reset, a burst mode machine must be stable and not generate any outputs until a complete input burst has been consumed. After each time an input burst has been received and a possible output burst has been generated, the machine must attain quiescens again before a new input burst is allowed to occur. Since the synthesis procedure only

takes the specified signal transitions into consideration when generating the Boolean functions for the state machine, unspecified input changes are not allowed to occur or else the circuit will malfunction. The unique entry point requirement makes sure that one and the same state cannot be entered by more than one unique set of values of the signals. This means that all entry points from every predecessor state must be identical. This is not really a limitation since every burst mode machine can be translated into an equivalent specification complying with this restriction by duplicating states. A burst mode machine must also comply with the maximal set property which means that for any two arcs going out of the same state (a choice) none of their input bursts may be a proper subset of the other's input burst. Otherwise the specification would be ambiguos since the controller cannot possibly know if any more signal changes will arrive after a full subset burst has already occurred.

6.2 Conversion of Two Phase Petri Nets

The Petri net controllers are abstract representations of the final implemented controller behavior since they make no assumptions about initial signal values or polarities of each signal transition. However, in order to synthesize Boolean functions representing the controller logic at the gate level, burst mode machines must have explicit knowledge of initial signal conditions and the polarities of the different signal transitions. The translation process for generating burst mode controllers must therefore deal with assigning specific values to the signals of the controller.

After the refinement and partitioning of our behavioral Petri net, the resulting single threaded controllers are a subset of I-Nets. The difference is that I-Nets allow concurrent threads, while our controllers only have signal concurrency in the form of bursts, a requirement of the burst mode machines. That is, in any one control graph only one transition can fire at a time. A transition however can contain several input or output signals (a burst). This is a restriction from the definition of burst mode controllers which cannot handle concurrent threads. Compliance with this restriction is ensured by the partitioning procedure in our high level synthesis method where all concurrent threads are divided into separate control graphs. For an I-Net with only input and output burst concurrency corresponding to a deterministic and delay insensitive machine obeying the quiescens restriction, we can obtain a burst mode machine with the exact same behavior provided that the fundamental mode timing constraint is met by the environment.

The translation method proceeds by traversing the Petri net graph in a depth first search (DFS) manner while collecting signal bursts and building the burst mode graph. The method is described by the algorithm in figure 6.2.

```
type place = record
                out_trans : array [1..#OT] of trans;  // reachable transitions
                num : integer;  // each place in the graph has a unique number
             end

type trans = record
                out_place : place;  // reachable place
                signals : string;     // string of signal names representing a burst
                signal_type : {input,output}  // are the signals in or outputs?
             end

var state_vec : array [1..#I/O] of bit;  // state vector for I/O signals
var burst, in_burst, out_burst : string;  // burst signal names
var bm_state : integer;                   // global burst mode next state

begin
 G_PN =  parse(input file);
 root_place = G_PN.root_place;
 state_vec = 000...000;
 table_store(root_place.num, state_vec);
 bm_state = 0;
 recurr(root_place);

 Procedure recurr(place)
  var local_bm_state;
  var local_state_vec;
  begin
     local_bm_state = bm_state;
     local_state_vec = state_vec;
     foreach trans ∈ place.out_trans[1,...,n]
        out_burst = "";
        in_burst = trans.signals;
        curr_place = trans.out_place;
        while (curr_place.out_trans[1].signal_type == output)
           out_burst = out_burst ∪ curr_place.out_trans[1].signals;
           curr_place = curr_place.out_trans[1].out_place;
        end
        burst = polarize(in_burst, out_burst);
        state_vec = calc_new_statevec(local_state_vec, burst);
        if (table_lookup(curr_place.num, state_vec, var old_bm_state))
           G_BM = add_burst(G_BM, local_bm_state, old_bm_state, burst);
        else
           bm_state = bm_state + 1;
           table_store(curr_place.num, state_vec, bm_state);
           G_BM = add_burst(G_BM, local_bm_state, bm_state, burst);
           recurr(curr_place);
        end
     end
  end
end
```

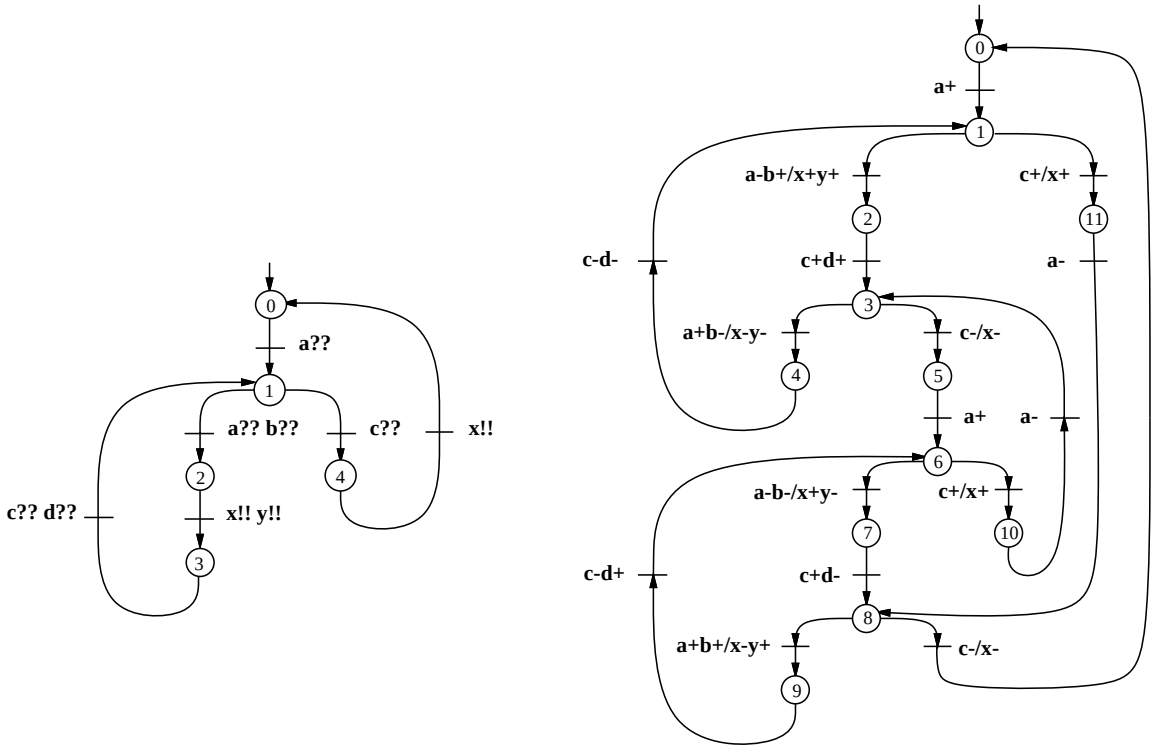
Figure 6.2: Algorithm for Petri net to burstmode conversion.

This algorithm is handed a Petri net from the high level refinement and partitioning step described earlier. This graph is then traversed recursively using the procedure **recurr**. We keep track of the state of each input and output signal by generating a state vector in which each position is filled with a certain signals current state (logic high or low). We traverse the graph by collecting input and output signals on the out-transitions of the places into corresponding input and output bursts. For each out-transition from a place we do the following: First the input signals occuring on the currently selected out-transition of the current place is collected and placed in the input burst which is now complete. We then traverse the graph following the current out-transition. The output burst is then generated by collecting output signals until a place with an out-transition labeled with input signals is reached. For each such place (that has an out-transition labeled with input signals) the current state vector is updated by altering the value of the signals that has been collected in the input and output burst. If this state vector does not already exist for the current place, it is stored in a hash-table together with information about the current place and the current burst mode state. The burst mode graph is then updated with the new burst mode transition. If a state vector for a certain place already exists the next state of the burst mode transition is set to the burst mode state previously stored in the hash-table. Otherwise a new next state is created for the burst mode graph. We keep track of the state vector and burst mode state of the current place by making local copies of these during the recursion. That way we will have the correct burst mode state and state vector when the recursion return to a previously visited place. Figure 6.3 illustrates a simple example of a Petri net and its corresponding burst mode graph. The algorithm traverses the Petri net graph in the number order given for its places and the burst mode graph is created in the number order given for its states. The input signals in the Petri net are annotated with ?? and the output signals with !!. A burst in the burst mode graph is described by an input burst followed by a slash followed by an output burst e.g. a+b+/x-y-. The slash is omitted when only an input burst is present. As can be seen in figure 6.3(b) the burst mode controller gets quite complex even when there is only one choice present in the corresponding Petri net graph. The handshake based Petri net graph can therefore offer advantages in specifying state machines because of its abstract, simple and compact representation.

The algorithm presented above can be optimized by only storing and checking the state vector for a more restricted subset of places. State vector information only needs to be stored for places that has out-transitions labeled with input signals and fulfills one of the following criteria.

The current place

- has more than one out-transition (choice place).
- has more than one in-transition.



(a) Handshake based Petri-net graph using two phase protocol.

(b) Corresponding burst mode graph.

Figure 6.3: Example of two phase burst mode translation

- has a preceding place, followed only by places labeled with output signals on their out-transition, with output signals on its out-transition and that has multiple in-transitions.

This optimization reduces the run-time quite significantly for large graphs although it does not remove the exponential worst case complexity. The implemented algorithm has also been improved to handle directed don't cares and non-monotonic level signals offered by extended burst mode machines.

Although the algorithm presented has an exponential worst case complexity in the number of out-transitions emenating from choice places in the Petri net graph we have successfully generated very large burst mode graphs in matter of seconds as can be seen in table 6.1. Since the largest burst mode graph we so far have been able to synthesize all the way to a netlist of gates using the 3D synthesis method [74], (without using partitioning) is the *GCD* circuit, the Petri net to burst mode conversion does not set the limit to what circuits can be generated by the ACK synthesis system.

Controller	Number of BM states	I/O size	Conversion time (sec)	Number of PN places	Number of choice places
Extended Barcode	10752	49	30	50	8
Barcode Reader II	1216	49	3	45	5
CD Player Error Corr. II	609	68	2	46	4
GCD II	88	25	< 1	20	2
Factorial II	68	20	< 1	23	1

Table 6.1: Results for burst mode generation.

6.3 Conversion of Four Phase Petri Nets

For the four phase protocol we use the same algorithm to derive burst mode machines from the refined Petri net graphs. Since all signals in a handshake are reset to zero before the handshake is started we will never get the problem of covering all possible combinations of signal polarities which causes the state explosion for the two phase case when choices are present. The conversion time will therefore be linear to the number of places in the Petri net graph.

However, the four phase protocol has a performance problem in that it uses twice as many transitions for a handshake as the two phase protocol. Since all computations and storage of data have been performed on the rising edge of the handshake signals, the extra transitions required to reset the signals to zero again will not result in any work being done. Since valuable time is lost while resetting the wires we will use a method that tries to hide the passive phase of the handshake (falling transitions) in the active phase of the next handshake (rising transitions). The use of burst mode machines allow us to do this easily by simply merging the bursts of these handshakes. The procedure is called *reshuffling* and proceeds as follows.

For any current node (except the start node which has no predecessor), we check the predecessor nodes and the successor nodes. If the output burst of the predecessor node contains a transition on any of the signals in the output burst of the current node or if any of the successor nodes contain any signal in their input bursts that is also contained in the input burst of the current node, no reshuffling is done. Similarly, no reshuffling for signals passing control between partitions or synchronization signals between concurrently executing processes may be done. This is because reshuffling of such signals could result in the assertion (high) of a signal that has not yet been deasserted (low) which would result in deadlock. Otherwise, the current input burst is merged into all successor node's input bursts and the current output burst is merged into all predecessor node's output bursts. The current place is now empty and can be removed.

An example of reshuffling using the algorithm is given in figure 6.4. The application of the algorithm on a factorial example is shown in figure 6.5 where the refinement step is shown followed by burst-mode reduction and reshuffling of the burst mode graphs. Note that most

states in the figures have been omitted to save space.

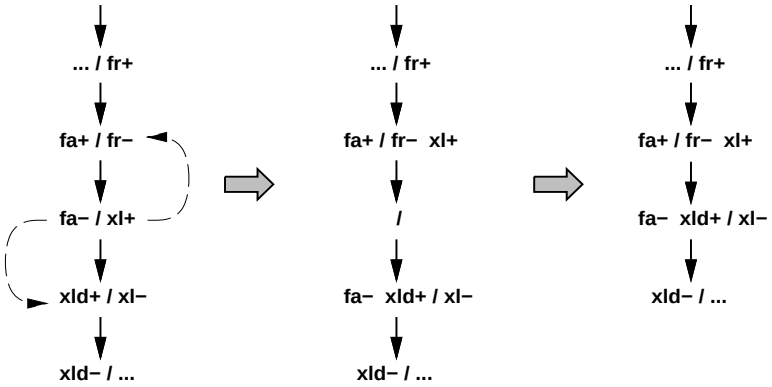


Figure 6.4: Example of reshuffling

The reshuffling method used in ACK ensures that the order of high level action execution is maintained. Other methods, such as late or collective return to zero (where the order in which the handshakes are returned to zero is not guaranteed to be maintained) can sometimes give better results. We are currently studying the problem of more efficient reshuffling for future versions of ACK.

6.4 Conclusions

This section has presented a method for generating correct burst mode machines from a Petri net specification with event signal notation. A problem with this method is the limitation to the size of graphs that can be translated due to state explosion when using two phase Petri nets. A method to improve the performance of four phase burst mode machines via reshuffling was also presented.

From the burst mode graphs that have been created using the method presented in this section the next step is to generate Boolean functions for the controllers they describe. This is done using existing burst mode synthesis tools [74, 49] and will be discussed in the next section.

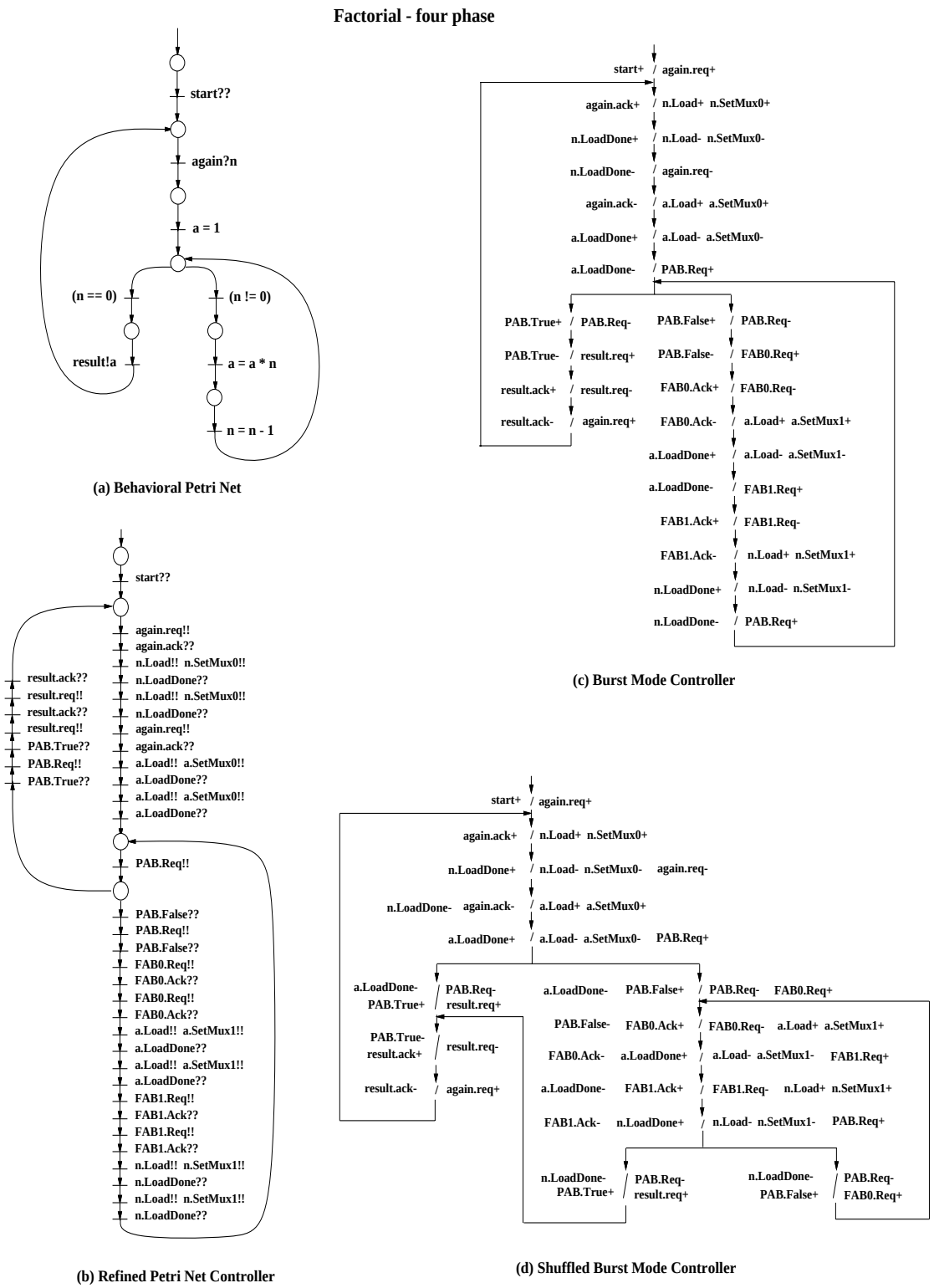


Figure 6.5: Factorial: (a) Original behavioral specification, (b) refined four-phase handshake based Petri net, (c) burst mode graph, (d) reshuffled burst mode graph

Chapter 7

Burst Mode State Machine Synthesis

At this point we have generated a collection of asynchronous finite state machines in the form of burst mode controllers that now need to be synthesized to Boolean function representations. This section will describe synthesis of such state machines and discuss two different styles of controller implementation that can both be used in ACK.

7.1 Fundamental Mode Asynchronous Finite State Machines

An asynchronous finite state machine (AFSM) is a controller having specification, implementation and functionality much like that of a synchronous state machine. The specification of a controller is often described as a Mealy state machine with an interface consisting of a set of primary inputs and outputs. The state of the controller is represented by a set of internal state signals that are fed back to the input side of the controller. The Boolean functions representing the behavior of the output and state signals are implemented as a block of combinational logic. So far the asynchronous and synchronous styles are very similar. However, there are some notable features that separate them.

The main difference between the two styles is that the input and output signals of an asynchronous state machine must change in a monotonic fashion. In a synchronous implementation, neither input or output signals need to change monotonically since the worst case requirements ensure that the logic has settled and the inputs and outputs stabilized before the next clock tick lets them propagate through the registers. Since asynchronous systems rely on event signaling, input and output signals of the state machines must change monotonically for the handshake communication to work reliably. This puts restrictions on the combinational

logic that implements the state machine.

An important difference is the way output signals respond to state and input signal changes. Unlike the synchronous style, asynchronous state machines have no registers that need to wait for the next global clock tick before propagating the output signals to the rest of the system. Asynchronous state machines therefore often have the advantage of average case delay compared to the worst case delay of synchronous state machines.

Another difference is the handling of state variables. In a synchronous state machine the state signals are always stored in a register before being fed back to the inputs of the machine. In Huffman asynchronous state machines however, the signals are fed back directly. Such direct feedback requires some special handling of the state signals.

7.1.1 Specification and Synthesis Methodology

The computation in a state machine is based on its state. From the current state and inputs to the machine it generates signals representing the outputs and its next state. The states are then fed back as inputs to the machine, moving it to the new state. This way of moving from state to state makes it possible to describe the specification of a state machine as a *flow table*.

The synthesis approach is similar to that of synchronous state machines. *State minimization* is first performed to reduce the size of the flow table by merging compatible states. This is followed by a *state assignment* step that assigns binary values to the symbolic states. Boolean functions for the output and state signals are then generated from the flow table. The last step is to generate combinational logic for the Boolean functions. Some special requirements during this step are necessary to ensure that the output signals of the combinational logic behave in a monotonic fashion. This step is called *hazard free logic minimization*. The concept of hazards will be explained in subsection 7.3. In the meantime it is sufficient to view hazards as the possibility of glitches occurring at the outputs of a combinational logic block. After the concepts of hazard free gate networks and other neccessary definitions have been presented the complete synthesis method will be presented in more detail in sections 7.4, 7.5, 7.6.

There is a major attraction with using finite state machines compared to other methodologies for asynchronous controller implementation. The state machine synthesis method can perform global optimizations which would be difficult or impossible to do with local transformations as done in most other synthesis methodologies that use syntax based decomposition into gate level implementations. State machine synthesis therefore often results in very efficient gate level implementations. The draw-back of this method is the complexity of state machine synthesis in general. Algorithms for exact solutions have exponential complexity which makes it impossible to synthesize large controllers. Methods for partitioning such large controller specifications (as presented in section 5) without losing too much of the global optimization

possibilities during synthesis are therefore necessary.

7.1.2 Input Constraints

There is a large number of asynchronous state machine methods. These methods can be hierarchically categorized based on their constraints on input changes [71].

SIC - single input change allows only one input to change at a time. After an input change has occurred a minimal time interval d must pass before any new input change is allowed.

MIC - multiple input changes allow a set of input signals to change during a time interval d_c. These signals are treated as if they were occurring simultaneously. A minimal time interval d_n must pass before a new set of input signals are allowed to change.

UIC - unrestricted input changes allow any signal to change at any time as long as no signal changes more than once in a time interval of d.

An asynchronous controller in its simplest form is called a Huffman machine. A Huffman machine in its simplest form allows only SIC mode operation. A common restriction is also to require the environment to wait until the circuit has stabilized before a new input change is allowed. This is called *fundamental mode* restriction and also applies to the fed back state variables. Unfortunately, the requirement that only one input at a time may occur after which the machine has to stabilize introduces severe restrictions on concurrency and makes this style impractical for real designs.

The MIC style has not been successfully used in real designs either. The reason is that a number of complications, both for input and state variable changes, arise when MICs are allowed. The types of complications that can occur are function and logic hazards for input changes and critical races when several state variables change simultaneously. These hazards will be described in subsection 7.3. Solutions to solve these problems have been presented [71] but the method makes use of inertial delays which have questionable reliability and slow down circuit operation. The general solution has therefore been to adopt the SIC mode of operation.

A special form of input change called *data driven* mode was first introduced by Davis et al. [14]. This style is an extension to the MIC mode of operation since it does not have the restriction that inputs must occur within a limited time interval. Instead a set of inputs is specified as a *burst* and the signals may change in any order with arbitrary delay in between. After a full burst has been received outputs and next state signals are generated. This method require fundamental mode operation. In [49] a constrained and formalized version of Davis data driven mode called *burst mode* (BM) was introduced. This style uses specifications in the form of Mealy style finite state machines and imposes no timing restrictions on multiple input

changes. This style was later extended with non-monotonic level input signals and directed don't care signals in [74] called *extended burst mode* (XBM). A more thorough description of this type of state machine has been given in section 6. Since its introduction this has been the most successfully used style of asynchronous state machine synthesis and has shown to generate efficient solutions to a large set of real designs.

A method has also been presented to allow UIC mode operation [71]. However, as the MIC style, this method never saw wide use due to its use of inertial delays. It is also unclear how real life designs can be efficiently represented as a system of "ordered chaos".

7.2 Two Implementation Methods for AFSMs

There are two main styles for implementing asynchronous finite state machines, the self synchronized style and the Huffman only style. The self synchronized style uses a locally generated clock to store output and state signals while the Huffman only style does not. The self synchronized style will be discussed next followed by the Huffman only style.

7.2.1 The Self Synchronizing Style

A method using a *locally clocked* approach for hazard free synthesis of burst mode machines was developed in [49]. This approach and others [27, 68, 1, 13, 57] make use of a method called self-synchronization. This method use Huffman machines along with a locally generated aperiodic clock acting on internal latches. The clock in many of the earlier methods is used to propagate the outputs just as for synchronous machines which results in worst case performance. Still there is an advantage of this method compared to the synchronous style since the worst case constraint is local to each controller, meaning that controllers of different complexities still can execute at different speeds. Although this is a restricted way of average case delay it can still offer an advantage over synchronous controllers. The clock signal generation in these methods have been based on XOR-trees or combinational logic using inertial delays to remove glitches. These methods have resulted in implementations having rather poor performance due to the worst case design and added delays.

The method presented in [49] deals with these performance problems by only using the clock for state changes. The outputs freely flow through the internal latches that are always transparent except during a state change. This method also uses a new method for making hazard free (glitch free) minimization [50] of the combinational logic. This method thereby exploits the average case delay properties of the individual controllers as well as reduces the latching delay compared to other methods.

It is important to notice the difference between the self-synchronized and synchronous approach. The clock in a self-synchronized controller is generated locally to each controller and is sometimes used to eliminate hazards on the outputs of the combinational logic. Unlike synchronous design a clock tick is generated only when it is needed, when a new input burst arrives. The clock is therefore aperiodic, meaning it does not have a fixed cycle time.

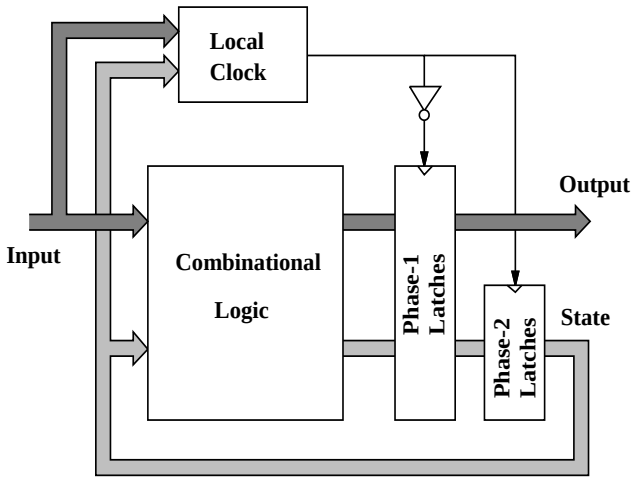
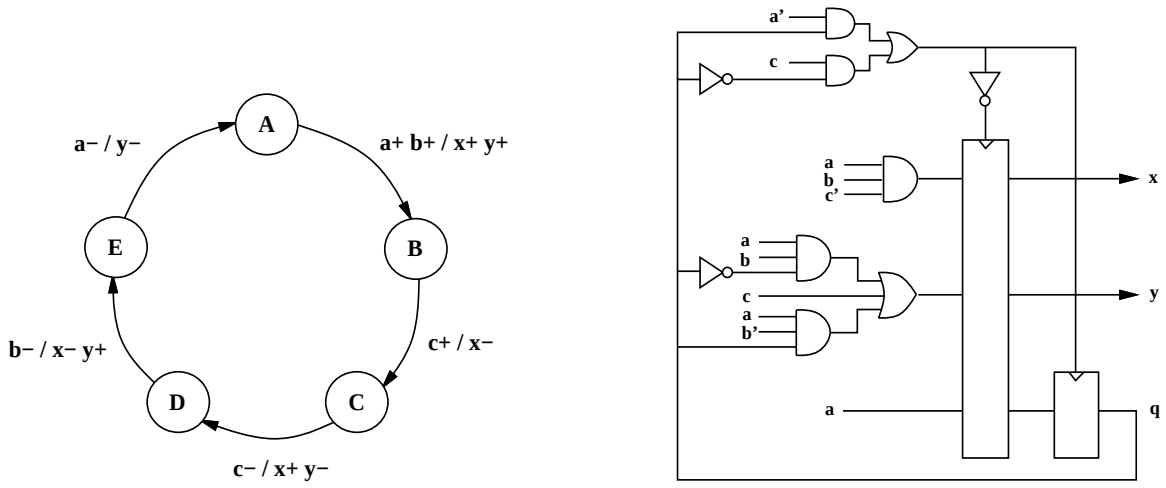


Figure 7.1: Locally clocked controller structure

The general structure of the locally clocked method used in [49] is illustrated in Figure 7.1. It consists of combinational logic for the clock, output and state signal generation, storage elements in the form of latches, and a set of primary inputs and outputs. The phase-1 latches are dynamic while the phase-2 latches are static. The reason for having dynamic latches is that signals can propagate through very quickly compared to a static latch. The clock in this method can be seen as having two phases and is generated by hazard-free combinational logic. As long as no state change is needed the clock will remain low. Any output change of the combinational logic (C_L) can therefore propagate freely through the phase-1 latches. This requires C_L to be free of hazards for the input signal changes. When a state change is needed the new output and state signals are first generated. When they have propagated through the phase-1 latches, a clock tick is generated first opening the phase-2 latches and then closing the phase-1 latches. As long as the phase-1 latches remain closed they will not propagate any glitches on the output or state signals. Therefore, C_L does not need to be hazard free for the state signal changes resulting in less complex logic. When C_L has stabilized from the state change, the C_{clk} logic lowers the clock signal, first closing the phase-2 latches and then opening the phase-1 latches. The advantage of this selective clocking method is that it tends to require less complex logic for the clock generation and that many transitions will not have the clock-cycle overhead.

An example taken from [49] is illustrated in figure 7.2. The burst mode specification of the controller is shown in figure 7.2(a), and the resulting implementation after synthesis is



(a) Burst Mode Specification **(b) Hazard Free Implementation**

Figure 7.2: Example of locally clocked controller implementation

viewed in figure 7.2(b). A step-by-step analysis of the first two transitions, from state A to B and then from state B to C, is shown in Figure 7.3.

The symbolic states A and B have been merged into a single state, encoded *q=0*, while states C, D and E have been merged into encoded state *q=1*. This means that as we go from state B to C, a change in state is necessary. The primary input signals for this machine are a, b and c. The primary outputs are x and y. The state signal is q. Initially the circuit is quiescent and the clock, input, output, and state signals are low. This means that the phase-1 latches are transparent (open) and the phase-2 latch is disabled (closed). When events on both signals a and b have occurred (they may arrive at arbitrary times) the hazard free combinational logic will generate events on outputs x and y making them go high. The outputs can propagate freely through the phase-1 latches thereby making the output changes very fast. Once the outputs have been generated we require that the circuit have time to stabilize (attain quiescens) before the environment responds with a new set of input changes. When an event occurs on input c, an event is generated on output x making it go low. The machine must now change state, state signal q must go from low to high. State signal q is represented by input signal a, which went high in the previous transition. To propagate the new value of the state signal the circuit must generate a clock tick. A high going transition on the clock is generated, opening the phase-2 latch and then closing the phase-1 latch. The new value of q is now fed back to the combinational logic for the clock as well as outputs. The output logic is now allowed to glitch since the phase-1 latches are closed, which means we do not need hazard free logic for state changes. After the output logic has attained quiescens a low going transition is generated by the clock, first closing the phase-2 latch and then opening the phase-1 latches. The state changing

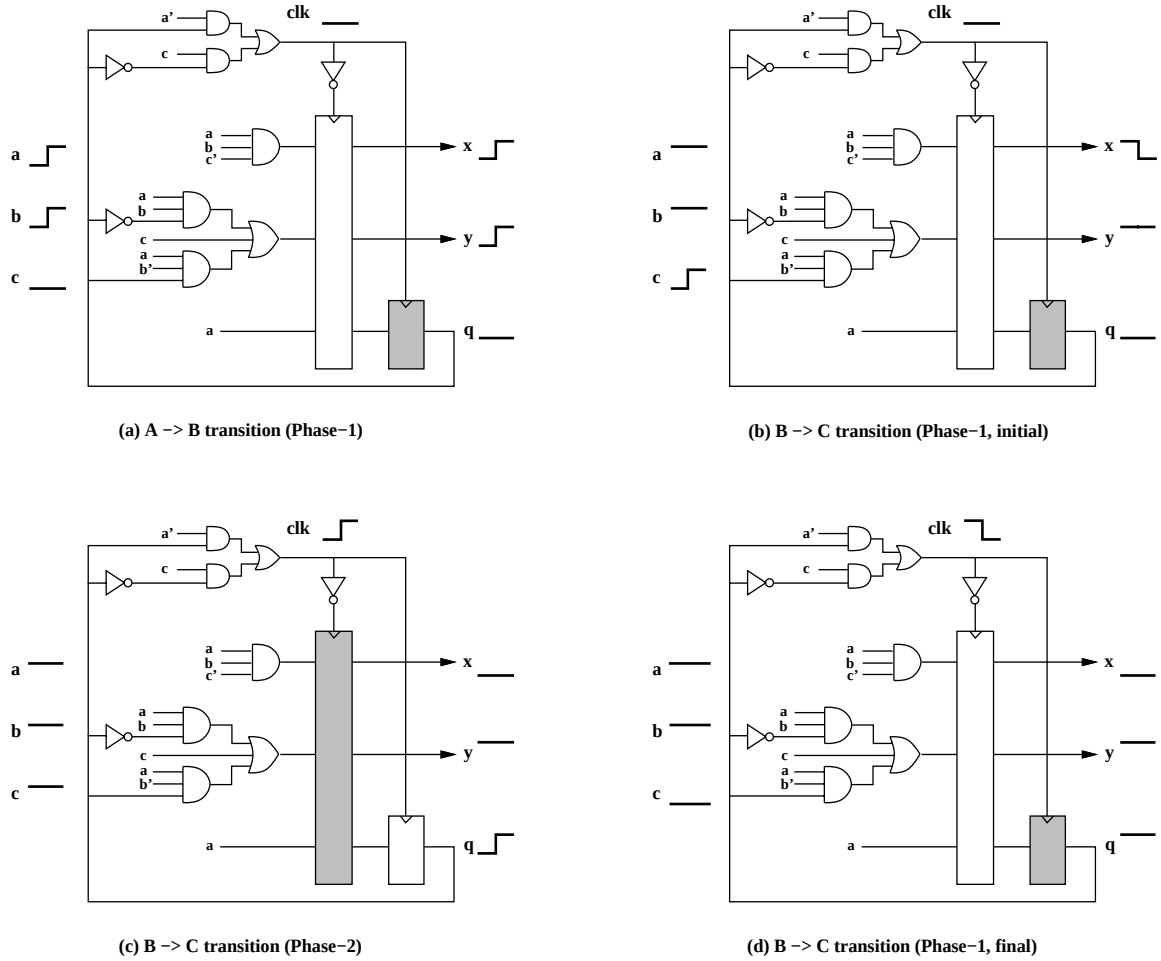


Figure 7.3: Example of locally clocked controller implementation

cycle is now complete and the circuit has settled into state C ($q = 1$) waiting for an event on input c.

As the reader may have noticed, there are timing constraints associated with the generation of the clock signal. If the rising (falling) edge of the clock signal reaches the latches too early, the output or state signals may not have had time to change before they are latched. The same is also true for the combinational logic's response to the state feedback. If the logic is hazardous for change of the state signals it must have time to settle before the falling (rising) edge of the clock reaches the latches. Otherwise glitches on the outputs may propagate through the phase-1 latches. This one-sided timing constraint on the clock can be met by inserting delays on the clock signal wires. The logic must also have time to settle before any change of the state signals are allowed to feedback. This is also a one-sided constraint that can be met by inserting delays on the feedback wires.

7.2.2 The Huffman Machine Style

A method called *3D* for hazard free synthesis of burst mode machines was developed in [74]. This method uses the Huffman only machine approach and also extends the burst mode style to include non-monotonic level input signals and directed don't care signals and is called extended burst mode.

Methods for implementing Huffman only machines for SIC and MIC have been presented [71] using inertial delays and [24, 37] that make use of delays, large flow tables, careful timing requirements and specialized state codes. All these methods slow down circuit operation.

The 3D method deals with these performance problems by generating hazard free combinational logic instead of introducing delays to ensure correct circuit behavior. Since it has no local clock and does not make use of latches, there is a potential reduction in area. Since the delay through the dynamic latches can be avoided there is also a potential performance gain compared to the locally clocked method. However, the way state changes are handled may incur more complex logic for the state variables. Also, for state changes it may require to go through two feedback loops to complete a state change and thereby increase the fundamental mode delay.

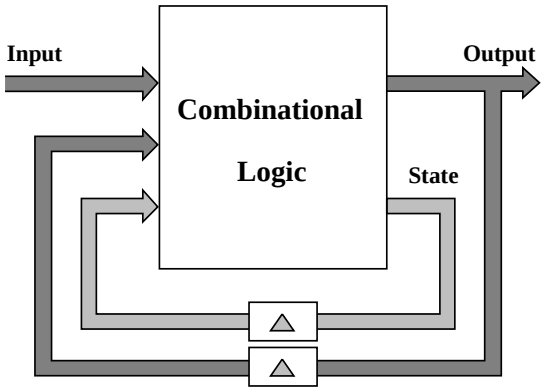


Figure 7.4: 3D controller structure

The general structure of the Huffman only machine method used in [74] is illustrated in Figure 7.4. It consists of combinational logic for the output and state signal generation, a set of primary inputs and outputs, and delays on the feedback paths. Since both outputs and state variables may be used as state feedbacks the combinational logic in this method can be seen as working in three phases. During the first phase the combinational logic is excited by a set of input changes and generates a set of output changes. In the second phase the logic is excited by the fed back outputs which together with the earlier input changes generates a set of state signal changes. Since the logic must attain quiescens between each of these phases, the total fundamental mode delay - the time from the last changing input until the logic has

stabilized after phase three, may become significant. In practise however, the environment is often sufficiently slow to accommodate this problem. If not, extra delays have to be added in the environment feedback paths.

The 3D implementation of the example from subsection 7.2.1 is shown in figure 7.5. Notice that extra logic has been added to implement a function for the state variable. Still, a large area saving is made since we do not have to generate logic for the local clock and we have no latches. The absence of latches will give us a slight performance gain in this example. A step-by-step analysis of the first two transitions, from state A to B and then from state B to C, is shown in Figure 7.5.

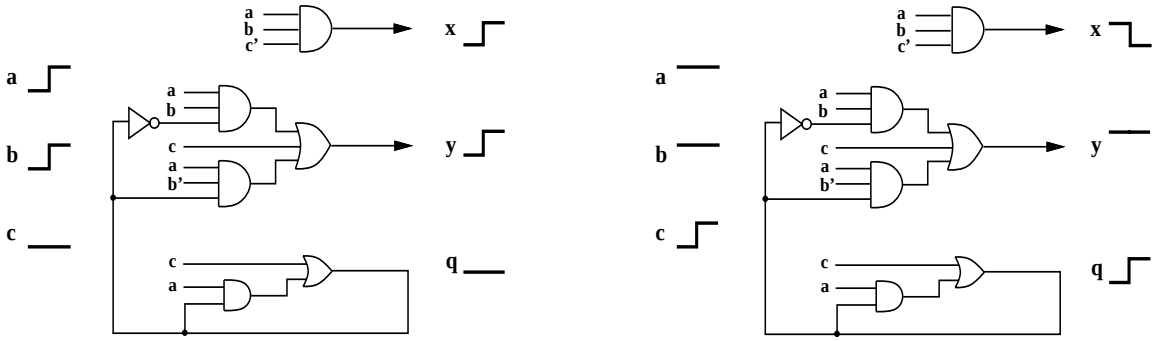


Figure 7.5: Example of 3D controller implementation

As for the self-synchronized implementation, the states A and B have been merged into a single state *q=0* while states C, D and E have been merged into state *q=1*. This means that as we go from state B to C, a change in state is necessary. The primary input signals for this machine are a, b and c. The primary outputs are x and y. The state signal is q. Initially the circuit is quiescent and the input, output, and state signals are low. When events on both signals a and b have occurred, the hazard free combinational logic will generate events on outputs x and y making them go high. Once the outputs have been generated we require that the circuit have time to stabilize before a new input change arrives. When an event occurs on input c, an event is generated on output x making it go low. The machine will now also change state, and state signal q subsequently goes from low to high. For the circuit to comply with the fundamental mode assumption it must now have time to stabilize before the transition on the fed back state signal is allowed to reach the inputs. This can be ensured by adding delays on the feedback path. Once the state signal arrives at the inputs the circuit will respond by entering a new state. The logic for the state variable and for the outputs must be hazard free even for state changes since we have no latches that filter out glitches. This requires that a critical race free state encoding is used during synthesis. The state changing cycle is now complete and the

circuit has settled into state C (q=*1*) waiting for an event on input c. Notice that this example does not suffer from the double state change mentioned earlier since no outputs are used for state feedback.

As mentioned in the example, to ensure a correct circuit behavior there is a one-sided timing constraint that requires the feedback paths of the output and state signals to be sufficiently slow for the combinational logic to stabilize before the signal changes arrive at the inputs.

7.3 Hazards

The notion of hazards is a central problem in asynchronous design. It is a fundamental requirement for asynchronous communication that signals change monotonically. Many implementation styles make use of explicit delays and careful timing constraints to ensure hazard free outputs, while others introduce constraints during state minimization, state assignment and logic minimization.

Since adding delays to ensure correct behavior slows down the circuit, a large body of work has been done in hazard free synthesis of such circuits. Many solutions for hazard free synthesis have been presented in the past [42, 8, 9, 23]. However, [42] only solves the problem under single input changes, [9] uses sequential storage elements and [8, 23] assumes fully specified functions and tries to eliminate hazards even for unspecified transitions resulting in suboptimal solutions.

Work in [50] presents a new method for hazard free minimization that, for problems where a solution can be found, eliminates all hazards. This method only eliminates hazards for specified transitions using exact algorithms, making the solution optimal. Note that since this method targets an implementation that is restricted to two level AND/OR logic not all problems have a solution.

In the discussions in previous sections it has been sufficient to view a hazard as a glitch on an output from a combinational logic block. These glitches can be caused by several different reasons which we will now take a closer look at. The concepts of three hazard considerations present in both the Huffman only and self synchronized implementation styles will be discussed. Extra hazard considerations necessary for implementing extended burst mode specifications will not be addressed here but can be found in [74]. Note that all definitions give here are based on a two level sum of products implementation.

7.3.1 Terminology and Definitions

A *logic function* f is defined as a mapping from $\{0,1\}^n \mapsto \{0,1,*\}$ where "$*$" represents a don't care value in the function.

Each element in $\{0,1\}^n$ of F is called a *minterm*.

The ON-set of a function f is the set of minterms for which $f = 1$. The OFF-set is the set of minterms for which $f = 0$ and the DC-set for which $f =$ "$*$".

A variable, v_i, has two corresponding *literals*, an uncomplemented literal v_i and a complemented literal $\overline{v_i}$.

A *product term* is a Boolean product of literals that contains a minterm in the ON-set if all literals in the product evaluates to 1.

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

A *sum of products* consists of a set of products P_s. A minterm m is included in a sum of products if some product p contains it and $p \in P_s$.

A product A contains a product B if the cube for B is a subset of the cube for A. The intersection of two products A and B is the set of minterms contained in the intersection of the corresponding cubes.

A *transition cube* $T[A,B]$, also written $[A,B]$, contains all possible minterms that can be reached during a transition from start state A to end state B.

A transition from state A to B for function f is a static transition if $f(A) = f(B)$ and a dynamic transition if $f(A) \neq f(B)$.

For a *static* transition the transition cube $T[A,B]$ contains only minterms from either the ON or the OFF-set but not both.

If for a *dynamic* transition the start state A of transition cube $T[A,B]$ contains an ON-set minterm then the end state B must contain an OFF-set minterm. If start state A contains an OFF-set minterm then B must contain an ON-set minterm.

A *cover* for a function f is the sum of products containing all minterms of the ON-set, none of the OFF-set and possibly some of the DC-set.

A *conflict* is said to occur when a minterm is assigned two values by the flow table. A conflict is resolved by placing the different values in separate *state layers*. A transition between state layers occur when a state variable changes value.

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

contains a minterm not contained in any other prime implicant.

7.3.2 Essential Hazards

The notion of *essential hazards* is linked to the state feedback of a controller. This type of hazard can only occur for controllers that are implemented as sequential machines. If the controller is purely combinational, which means we have no feedback, essential hazards cannot occur. The assumption made for asynchronous sequential state machine controllers is that the combinational logic is quiescent before any new changes of input or state feedback signals arrive.

A feed forward path is the path an input signal must take in order to generate a change on an output or state signal. If the difference in delay between the minimum and maximum feed forward paths is large compared to the feedback delay a *critical race* may occur. The race can be seen as the fed back state signal, which is generated by the minimum feed forward path, "catching up" with the input signal in the maximum feed forward path. Figure 7.6 illustrates this risk for input signal a and state signal s. All input, output and state signals are initially high. In this case, when a goes low, the logic has not attained quiescens before the state signal change is fed back, resulting in a glitch on output x. As a result of this, the logic may generate a glitch on outputs or get stuck in the wrong state.

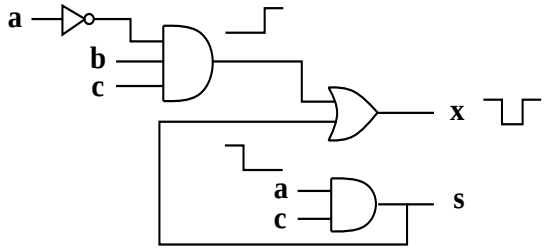


Figure 7.6: Example of Essential Hazard

The problem of essential hazards can be avoided by inserting sufficiently large delays on the feedback paths.

7.3.3 Function Hazards

A combinational function has a *function hazard* if the output changes value more than once during a specified multiple input change, that is, the output does not change monotonically. There are two types of function hazards, static function hazard can occur when we do a static transition, and dynamic function hazard that can occur when we do a dynamic transition.

Definition. A *static function hazard* is present in a Boolean function f for the static transition $A \rightarrow C$ if and only if there exists some intermediate input state $B \in [A, C]$ such that

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

An example of a static function hazard is illustrated in the Karnaugh map of Figure 7.7(a). When we go from state A to state C we may briefly pass through state B (dotted arrow) which has a different value than that of state A. We may therefore get a static $1 \rightarrow 0 \rightarrow 1$ hazard (glitch) at the output of the logic implementing the function.

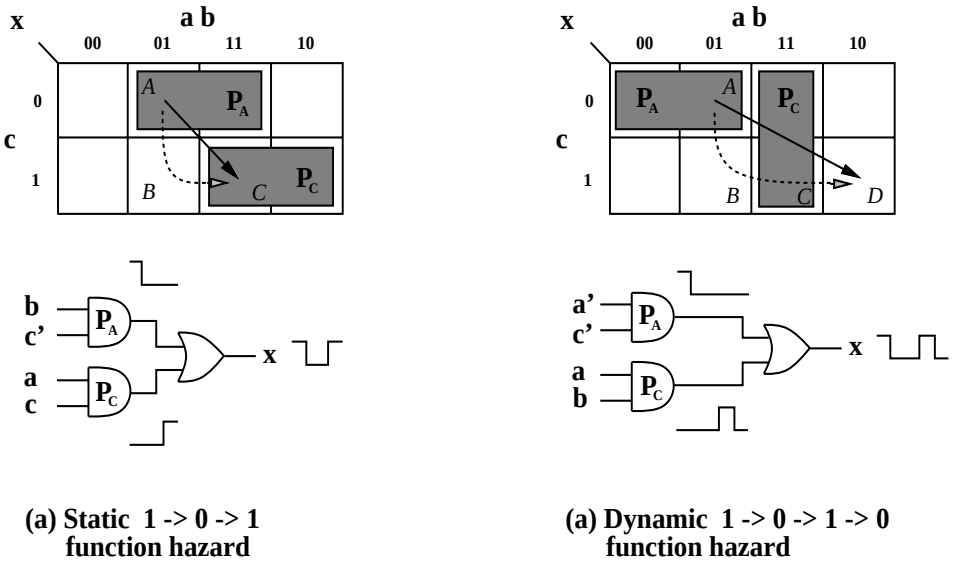


Figure 7.7: Example of Function Hazards

Definition. A *dynamic function hazard* is present in a Boolean function f for the dynamic transition $A \rightarrow D$ if and only if there exist a pair of intermediate input states B and C where ($A \neq B$, $C \neq D$) such that $B \in [A, D]$ and $C \in [B, D]$ and $f(A) \neq f(B)$ and $f(C) \neq f(D)$.

An example of a dynamic function hazard is illustrated in the Karnaugh map of Figure 7.7(b). When we go from state A to state D we may pass through states B and C (dotted arrow). B has a value different from A (we go from $1 \rightarrow 0$) and C has a different value from B (we go from $0 \rightarrow 1$) and D has a different value from C (we go from $1 \rightarrow 0$). We may therefore get a dynamic $1 \rightarrow 0 \rightarrow 1 \rightarrow 0$ hazard (glitch) at the output of the logic implementing the function.

In [9] it has been proved that there is no way of avoiding a function hazard on the output of a combinational logic block if the inputs are allowed to arrive at arbitrary times. Function hazards must therefore be avoided before the logic synthesis step can take place. This can be achieved by introducing a state feedback that takes the machine to a new state layer where the function can be implemented without a function hazard.

7.3.4 Logic Hazards

After the steps of state assignment and state minimization have been performed on the specification it is time to synthesize the flowtable to a Boolean function representation. At this stage we have already removed function hazards from the specification by placing conflicting function values in different state layers. What remains is to translate the flow table into an optimal hazard free two level sum of products function.

Hazard free logic minimization deals with this problem. This method makes use of a number of constraints to ensure that the resulting logic is free of *logic hazards*. A logic hazard is a property of the logic implementation in contrast to a function hazard which is a property of the specification. As for function hazards there are two types of logic hazards, static logic hazards and dynamic logic hazards, which can occur for static and dynamic transitions respectively. By using a two level sum of product (SOP) form for the Boolean functions to be synthesized we get a gate network structure having a strict hazard behavior which makes hazard free logic minimization easier. The synthesis methods presented in this section use the function region approach to generate a Boolean function representation using the ON-set minterms in the Karnaugh maps specifying the burst mode graph. The hazard analysis done in this subsection requires that no product term contain both a signal and its complement, otherwise additional hazards are possible [71].

Definition. A *static logic hazard* is present in the network implementing a Boolean function f, free of function hazards, for the static transition $A \rightarrow B$ if and only if during the input change from A to B a momentary pulse may be present on the output.

An example of a static logic hazard is illustrated in the Karnaugh map of Figure 7.8(a). When we go from state A to state B we will go from the product term P_A to the product term P_B. The cover of both these product terms are represented by AND-gates. If the AND gate for P_B is slow then the AND-gate for P_A goes low before the AND gate for P_B goes high, causing the output of the OR-gate to glitch. For a static logic hazard free two level AND/OR implementation, the whole transition cube $T[A, B]$ must be fully contained in a product term (AND-gate). The necessity to cover all static transitions to get a logic hazard free function may result in redundant product terms, meaning some minterms may be covered by several product terms. We will not consider static $0 \rightarrow 0$ hazards here since it is trivially realized, and has been shown in [71], that a SOP implementation covering the ON-set cannot have any such hazards.

Definition. A *dynamic logic hazard* is present in the network implementing a Boolean function f, free of function hazards, for the dynamic transition $A \rightarrow B$ if and only if during the input change from A to B a momentary 0 and a momentary 1 may be present on the output.

An example of a dynamic logic hazard is illustrated in the Karnaugh map of Figure 7.8(b). When we go from state A to state B we may go via the product term P_A (dotted arrow). If

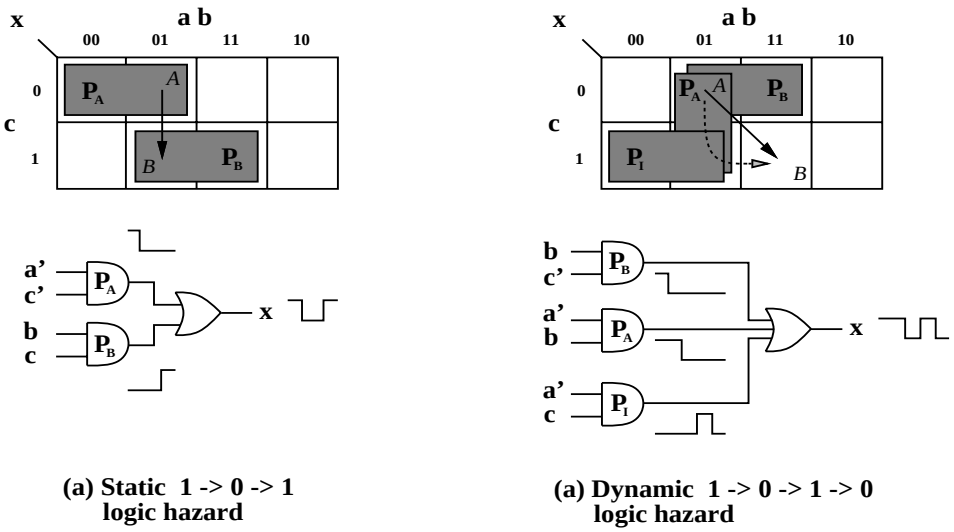


Figure 7.8: Example of Logic Hazards

the AND-gate for the product term P_I intersecting the transition cube $T[A,B]$ is slow then the AND-gate for P_A may first go low causing the output of the OR-gate to go low. The slow AND-gate for P_I now momentarily goes high and then low again, causing the OR-gate to generate a glitch at the output. The output generated in this case will go $1 \rightarrow 0 \rightarrow 1 \rightarrow 0$ due to the dynamic logic hazard caused by P_I. For a dynamic logic hazard free two level AND/OR implementation, no product term may intersect the transition cube $T[A, B]$ for $f(A) = 1$ and $f(B) = 0$ without also containing the start state A. Similarly no product term may intersect the transition cube $T[A, B]$ for $f(A) = 0$ and $f(B) = 1$ without also containing the end state B. We will not consider dynamic hazards for SIC transitions here since it is trivially realized, and has been shown in [71], that such cannot occur for a two level SOP implementation.

7.4 Hazard Free AFSM Synthesis to Two Level Logic

Now that we have discussed all the hazards that can occur in a two level AND/OR circuit implementation we can present a general synthesis method following the steps outlined in section 7.1.1. The methods presented here use exact algorithms that have exponential complexity. More time-efficient heuristic methods that generate near-optimal solutions [74, 69] exists but will not be presented here. Although the methods are presented in the context of burst mode synthesis they are general in nature and are easy to apply to other asynchronous specification styles targeting Huffman style fundamental mode state machines.

7.4.1 Conditions for Hazard Free Burst Mode Transitions

To ensure that a sum-of-products implementation corresponding to a set of specified burst mode transitions comply with the hazard free behavior requirement of asynchronous circuits, there are certain conditions that must be fulfilled. The conditions presented here have their origin in the definitions of function and logic hazards previously discussed in section 7.3 and are taken from [51, 69, 50]. The following are definitions of a transition cube and its subcubes.

Definition 1. Let A and B be two minterms. The *transition cube* $[A, B]$ has start point A and end point B and contains all minterms that can be reached during a transition from A to B.

Definition 2. A subcube $[A, B)$ of a transition cube $[A, B]$ contains all reachable minterms of $[A, B]$ except B. Likewise subcube $(A, B]$ contains all minterms except A.

As discussed previously, a sum-of-products implementation not free of function hazards does not have a hazard free solution under the unbounded gate delay model. The following definition ensures that a burst mode transition is free of such function hazards.

Definition 3. 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 $C \in [A, B)$, $f(C) = f(A)$.

Although a burst mode specification is free of function hazards, a transition from input state A to B may still have a logic hazard due to delays in the actual gate level realization. The following lemmas describe necessary and sufficient conditions to ensure that the sum-of-products implementation of a function, f, has no logic hazards for the given transition.

Lemma 1. If f has a $0 \rightarrow 0$ transition in cube $[A, B]$, then the implementation is free of logic hazards for the input change from A to B.

Lemma 2. If f has a $0 \rightarrow 1$ transition in cube $[A, B]$, then the implementation is free of logic hazards for the input change from A to B.

Lemma 3. If f has a $1 \rightarrow 1$ transition in cube $[A, B]$, 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 c of cover C.

Lemma 4. If f has a $1 \rightarrow 0$ transition in cube $[A, B]$, then the implementation is free of logic hazards for the input change from A to B if and only if no cube c in the cover C intersects $[A, B]$ unless c also contains A.

Lemma 5. If f has a $1 \rightarrow 0$ transition in cube $[A, B]$ which is hazard free in the given implementation, then, for every input state $X \in [A, B]$ where $f(X) = 1$, the transition subcube $[A, X]$ is contained in some cube c of cover C.

Lemma 3 makes sure that during a $1 \rightarrow 1$ transition from input state A to B *some* cube c of cover C holds its value constant at 1 throughout the transition. Lemma 4 ensures that *no* product may glitch during a $1 \rightarrow 0$ transition from input state A to B. Lemma 5 states that during a $1 \rightarrow 0$ transition from input state A to B every $1 \rightarrow 1$ sub transition must be free of logic hazards. The cube $[A, B]$ and maximal subcube $[A, X]$ in Lemmas 3 and 5 respectively are called *required cubes* and the transition cube $[A, B]$ in Lemma 4 is called a *privileged cube*. The following definitions state this more formally.

Definition 4. Given a function f, and a set, T, of specified input transitions of f free of function hazards, every cube $[A, B] \in T$ corresponding to a $1 \rightarrow 1$ transition, and every maximal subcube $[A, X] \subset [A, B]$ where f is 1 and $[A, B] \in T$ is a $1 \rightarrow 0$ transition, is called a *required cube*.

Definition 5. Given a function f, and a set, T, of specified input transitions of f free of function hazards, every cube $[A, B] \in T$ corresponding to a $1 \rightarrow 0$ transition is called a *privileged cube*.

A *hazard free cover*, F, of function f, is a cover of f whose two level sum-of-products implementation is hazard free *for a given set*, T, of specified input transitions where f is defined for each such transition. For a cover to be hazard free, each required cube has to be contained in some cube of cover C. Also, Lemma 4 sets a constraint as to which cubes may be included in the cover. To ensure that a cover is hazard free, a product is not allowed to glitch during a $1 \rightarrow 0$ transition. Therefore no required cube may *illegally* intersect a privileged cube. This is stated more formaly in the following definition and theorem.

Definition 6. A required cube is said to *illegally intersect* a privileged cube $[A, B]$ of a dynamic $1 \rightarrow 0$ transition if it contains any minterm in $[A, B]$ without also containing the minterm A.

Theorem 1. A sum-of-products F is a hazard free cover for function f for the set, T, of specified input transitions if and only if:
(a) No cube of F intersects the OFF-set of f;
(b) Each *required cube* of f is contained in some cube of F; and
(c) No cube of F intersects any *privileged cube* illegally.

The covering conditions outlined in Theorem 1 may not be satisfiable by an arbitrary Boolean function and a set of transitions. In addition, when all transitions are taken into consideration, the requirement that the function to be implemented is free of function hazards may not be met. To be able to realize a given burst mode specification in two level AND/OR logic it is therefore sometimes necessary to introduce internal states so that conflicting function values with the same input state as well as illegally intersecting required cubes can be placed in separate internal state *layers* where they do not conflict. For the final implementation then,

when the next transition will take us to a input state that is already defined to have another value, we generate a change in some internal state variables and move to another state layer where the correct value is defined for that input state. The same is done when the required cubes of a transition would introduce illegal intersections with some privileged cube.

The following sections will present methods that deal with introducing internal states to avoid such conflicts and how logic functions are generated for the specified outputs and the added internal state variables.

7.4.2 Primitive Flow Table Generation

As previously mentioned, the first step in the synthesis is to generate a primitive flow table from the burst mode specification. This is done by assigning each state in the burst mode machine to a unique row having a unique, stable *entry point* in the primitive flow table. Each entry in the flow table describes both an *input state* and an *internal state* which together form a *total state* of the machine. A flow table can thus be represented by letting each column in the table represent a unique state of the input signals and letting each row represent an internal symbolic state of the machine. This table then represents the functions of the outputs and symbolic next states. Symbolic states are used since the number of internal state variables and their final encoding is not known during flow table construction and minimization. Each state reachable by some specified input transition must have specified output and symbolic next state values. More formally, each input state I and internal state S in the flow table defines an output function $\lambda(I,S) = O$ and a next state function $\delta(I,S) = N$, where O is a binary vector defining the current values of the outputs and N defines the symbolic next state in total state (I,S). Unreachable entries are annotated with don't cares for outputs and symbolic next states. Figure 7.10(a,b) shows a burst mode machine and its corresponding primitive flow table.

A primitive flow table such as described above does not have any function hazards. This is because of the requirement of a burst mode specification to hold the current output and symbolic next state values until a complete input burst has been absorbed (we reach B in the transition cube $[A,B]$). Since each state in the burst mode machine is mapped to a unique row in the flow table, all transitions emenating from that state will have the same total state as start point and will thus also start with the same output and symbolic next state values. Taking this and the maximal set property (see section 6.1) of burst mode machines into account, it directly follows that no two transitions can overlap without having the same output and symbolic next state values for the total states in which they are overlapping.

It is also always possible to generate a hazard free SOP cover for each output o and state variable s from such a primitive flow table. The reason is that for a function f a sum-of-products F can always be found such that no cube of F intersect the OFF-set of f and each

required cube of f is contained in some cube of F and no cube of F intersects any privileged cube illegally. The two first requirements are trivially fulfilled and the third is fulfilled by the way the primitive flow table is generated. A cover can always be generated in which no cube of F illegally intersects any privileged cube since (as mentioned above) *all* transitions, $[A, B]$, in a row of the primitive flow table has the same total start state, A, and thus ensures that the required cubes covering these transitions will all contain the start point A. This condition is sufficient to determine that a cover where no cube illegally intersects a privileged cube can be generated.

7.4.3 Symbolic State Minimization

As mentioned in the previous subsection there always exists a hazard free solution if rows in the primitive flow table are not allowed to be merged. However, since there is a correlation between the number of internal states of a flow table and the complexity of the final logic implementation we will now try to minimize the number of states in the primitive flow table. This is done by merging compatible rows in the flow table. During merging, all outputs and symbolic next states are considered at the same time when deciding if two rows are compatible or not.

Classic Compatibility Constraints for State Merger

In classical state minimization [71] a compatibility relation defines when two states can be merged. The relation is defined in two steps as follows. Output compatibility is the initial compatibility relation. Two states, S_1 and S_2, are output incompatible if, for some input state I, outputs $\lambda(I, S_1)$ and $\lambda(I, S_2)$ are both defined, and not equal. Two states are output compatible if they are not output incompatible. The next state compatibility is the final compatibility relation and is defined recursively. Two states S_1 and S_2, are next state incompatible if, for some input state I, next states $\delta(I, S_1)$ and $\delta(I, S_2)$ are both defined, and are incompatible. Two states are next state compatible if they are not next state incompatible. Take the example of two output compatible states, S_1 and S_2, that have two different next states, S_3 and S_4 for a given input state I. For S_1 and S_2 to be next state compatible, S_3 and S_4 must be both output as well as next state compatible. For instance, if S_3 and S_4 both have the same next state S_5 for input state I (and are next state compatible for all other input states), then we can merge states S_3 and S_4 into state S_{34} which would then result in S_1 and S_2 having the same next state for I. States S_1 and S_2 can then also be merged into a single state S_{12}. To conclude, two states S_1 and S_2 are incompatible if they are (i) output incompatible or (ii) next state incompatible. Two states are compatible if they are not incompatible.

Hazard Free Compatibility Constraints for State Merger

Apart from ensuring that the two rows are compatible, the method must also ensure that no logic hazards are introduced by the merger, i.e. we are still guaranteed to find a hazard free sum-of-products cover for all outputs and encoded state variables.

As discussed earlier, input transitions in the flow table will eventually be covered by a set of required cubes. If the flow table is viewed as a Karnaugh map (which it really is - the only real difference being the use of symbolic states), such a cube will be horisontal and confined to a single row of the map. Much in the same way, the transitions between internal states will cause vertical transitions in the Karnaugh map. Since the outputs and next state values must hold their values during the state change, such a state transition introduces either a static $0 \rightarrow 0$ or a $1 \rightarrow 1$ transition cube for each output and state variable. If it is a $1 \rightarrow 1$ transition, a required cube must be introduced for this transition. Such a cube then is vertical in the Karnaugh map and is confined to a single column in the map. Note that such vertical required cubes cannot cause illegal intersections with privileged cubes in a primitive (unminimized) flow table. For a horisontal $1 \rightarrow 0$ transition the vertical cube will contain the start point of the privileged cube (the state transition leads to a unique stable entry point which is also the start point of all transitions of that state row) and thus no illegal intersection is introduced. During state minimization however, two rows must not only be output and next state compatible to be merged, the resulting merged state must also be checked to make certain that no logic hazards has been introduced by the merger.

When the merging of two states is considered, the requirement of output and next state compatibility will ensure that no function hazards will be introduced. The next state compatibility constraint also ensures that the merger will not result in an ambiguous flow table. This however, is not enough to guarantee that the outputs and next state functions can be implemented in hazard free logic. We therefore need to check that no illegal intersections (which will cause dynamic hazards in the final implementation) for either outputs or state variables are introduced by the merger. This task is complicated by the fact that the state variable encoding is not yet known. We therefore have to assume that, after state assignment, in the worst case each input transition cube $[A, B]$ in the Karnaugh map describing the function of an encoded state variable could be a $1 \rightarrow 0$ transition. This is the worst case because only such $1 \rightarrow 0$ transition cubes can be illegally intersected by other cubes which, if permitted, would cause a hazardous cover. Under this assumption each pair of transitions in the two rows must comply with the *dhf-compatibility* constraints defined as follows.
Two rows, R_1 and R_2, are dhf-compatible if and only if for each pair of transitions $[A_1, B_1]$ and $[A_2, B_2]$ belonging to R_1 and R_2 respectively, at least one of the following restrictions are fulfilled.

(a) $[A_1, B_1]$ and $[A_2, B_2]$ do not intersect
(b) $A_1 = A_2$
(c) $B_1 = B_2$ and $[A_1, B_1)$ and $[A_2, B_2)$ do not intersect
(d) $B_1 \in [A_2, B_2)$ and $B_2 \in [A_1, B_1)$
(e) $B_1 \in [A_2, B_2)$ and $A_2 \in [A_1, B_1]$ and $A_1 \notin [A_2, B_2]$
(f) $B_2 \in [A_1, B_1)$ and $A_1 \in [A_2, B_2]$ and $A_2 \notin [A_1, B_1]$

The safe intersections where state merger is allowed are shown in figure 7.9(a,b,c,d). Note that for an encoded state variable s in an internal state row, every minterm in an input transition cube $[A, B]$ will hold the value of s stable with the only possible exception if end point B has an emenating vertical transition (a state change). Because of this the merger ensures that the state variable minterm values in two overlapping transition cubes which's vertical end point transitions have been eliminated, or *nullified*, will be the same (figure 7.9(a)). That is why we can merge states resulting in overlapping transition cubes without having to worry about the values in the merged transition cubes being the same after state variable encoding. This also holds true for situations where for two transition cubes $[A, B]$ and $[C, D]$, only end point B is nullified if $[A, B]$ also contains C (figure 7.9(b)). Figure 7.9(e) shows an example of an intersection that after state encoding might not be safe. If in the final state encoding one of the state variables makes a $1 \rightarrow 0$ transition for the given input transition $[a, b]$, then some required cube for transition $[c, d]$ will illegally intersect the privileged transition cube $[a, b]$ and thus introduce a dynamic hazard.

The procedure of merging state rows then becomes the following. Two states, $S1$ and $S2$, are initially compatible if they, for all input states, I, are (i) output compatible, *and* (ii) dhf-compatible. Given this new initial compatibility relation, the final compatibility relation can then be defined as before. Two states, $S1$ and $S2$, are compatible if they, for all input states, I, are (i) initially compatible, *and* (ii) next state compatible.

Take the example in figure 7.10. At a first glance internal states A and D might seem compatible since they are output as well as next state compatible. Note however, that such a merger would violate the constraints given in figure 7.9. States A and D are *not dhf-compatible*. The state transition from internal state B to merged internal state AD in input state $ab = 01$ would introduce a required cube for output x. This required cube would then illegally intersect privileged cube $[a'b', ab]$ in state AD. If such a merger was allowed, this would introduce a dynamic hazard in the logic implementation of output x. Now let's consider internal states C and D. These states are initially compatible, i.e. they are output as well as dhf-compatible. For input state $ab = 00$ however, they have different next states E and A respectively. Since neither E and A or E and D are compatible (i.e. cannot be merged), C and D are not next state compatible.

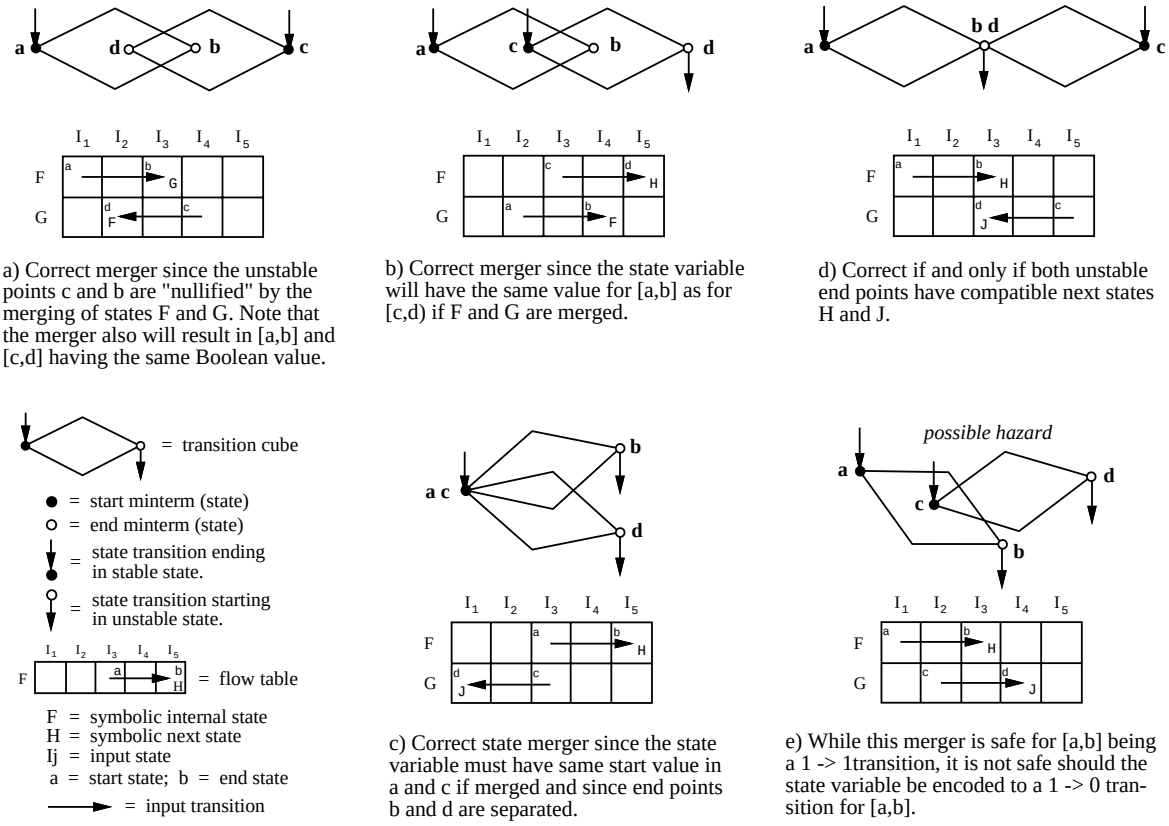


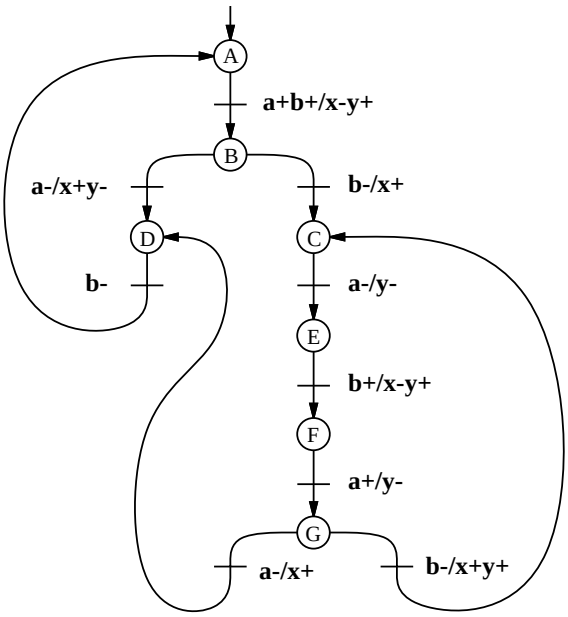
Figure 7.9: Safe state mergers for symbolic state variables.

Finding a Minimum Number of States

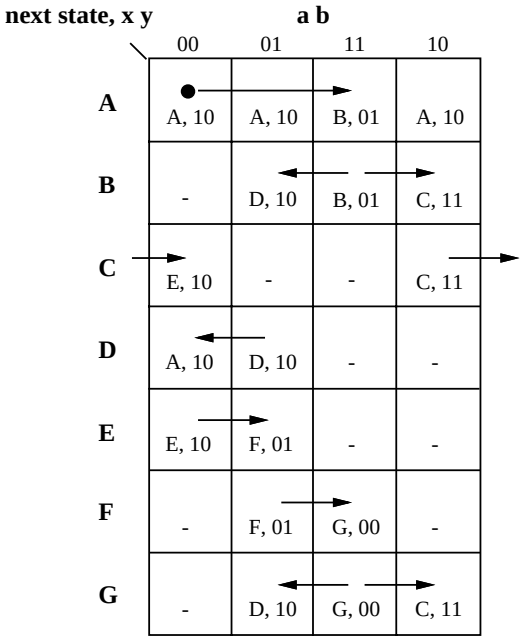
Since internal state rows can be merged in different configurations and since there is a correlation between the logic complexity of the implementation and the number of internal states, we need a method that decides which configuration will give the least number of internal states. We will briefly present such a method that is based on a method for reduction of incompletely specified Boolean matrices [42, 54, 70]. This method gives an exact solution to the problem of finding the minimal number of symbolic states for an incompletely specified Boolean matrix and is exponential in algorithmic complexity.

The method presented here is quite simple and straight forward. First we want to determine all possible combinations of internal state rows that are compatible with each other. We then want to find a minimal set of these combinations that together will cover each internal state.

The method for finding all possible combinations of internal state rows starts with forming the set of all compatible pairs of internal state rows in the primitive flow table that exist.



(a) Burst Mode Machine



(b) Corresponding Primitive Flow Table

(A)

(B, C) (B, D)

(C, E) (C, F) (C, G)

(D, G)

(E, F)

(c) Pairwise compatible states

1. (A)
2. (B, C)
3. (B, D)
4. (C, E, F)
5. (C, G)
6. (D, G)

(d) Maximal compatibles

Generate product of sums expression of maximal compatibles containing each of the internal states A to G:

1(2 + 3)*(2 + 4 + 5)*(3 + 6)*4*4*(5 + 6)*

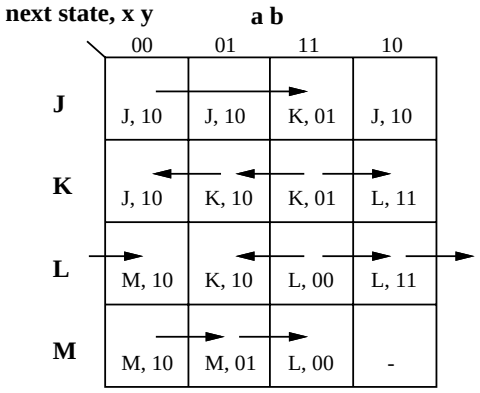
Convert to sum of products for solution:

Solutions: 1345 + 1246 + 1346 + ...

Choose 1345 as minimal solution. Maximal compatibles 4 and 5 both contain state C, erasing C from 4 gives merged states:

J = (A), K = (B, D), L = (C, G), M = (E, F)

(e) Petrick's method



(f) Resulting minimized flow table

Figure 7.10: Hazard free symbolic state minimization applied to Burst mode example.

We then expand each such pair with each possible combination of other internal state rows the pair is compatible with until no more rows can be added to it. These expanded combinations are called *maximal compatibles*. We then want to find a minimal set of such maximal compatibles that covers each internal state. This is done using Petrick's method [54]. The problem is then formulated as a product of sums expression where each sum represents all maximal intersectables that contains a certain internal state row in the flow table. This expression is then multiplied out to form a sum of products representation. The product containing the least number of maximal compatibles is then the optimal solution in terms of number of symbolic states that are needed in the minimized flow table. Since in this chosen set of maximal compatibles, these may be redundant with respect to eachother, meaning that several maximal compatibles can contain the same state, we then remove each such state from all but one of the maximal compatibles in the chosen set. The state minimization process for a simple burst mode machine is illustrated in figure 7.10.

Using this simple method we have now generated a minimized flow table with minimal number of symbolic states that is guaranteed to have a hazard free implementation. A flow table may of course be minimized with other goals in mind than just finding the minimal number of symbolic states. An example of another goal can be to minimize the flow table with least number of state changes in mind. This can be to an advantage in certain situations since each state change increase the fundamental mode delay for the transition that caused the state change. Minimizing state changes can also simplify the solution from the state assignment step.

7.5 State assignment

After we have generated a minimized flow table it is time to determine how many state variables are needed and their encoding. The main difference from synchronous state encoding is that the requirements of hazard free logic combined with the unbounded gate delay assumption requires a *critical race free* state variable encoding. Since the fed back state variables can arrive at the inputs at arbitrary times (after the output logic has stabilized), the machine may stabilize in an incorrect intermediate state instead of the specified next state in which case the circuit behavior is not predictable. There are several methods to ensure critical race free state encoding. Among them are the one-hot, one-shot, Liu or Tracey methods [70, 71].

Apart from ensuring that a state assignment is critical race free, it is also of importance to generate a direct transition from an unstable state to a stable next state since this reduces the fundamental mode delay, i.e. the time it takes for the circuit to stabilize between input bursts. Because of this it is desirable to incorporate a minimum transition time state assignment method where state variables change only once and do so concurrently. Our problem then becomes, given a minimized flow-table, generate a minimum transition time critical race free

state assignment with minimum number of state variables. In this section we will summarize such a method originally presented in [70]. Further details and proofs are presented in [70].

7.5.1 Conditions for Critical Races

A flow table, as specified earlier, that describes the behavior of a state machine is naturally divided into a set of columns representing all possible input states $\{I_1,\ldots,I_g\}$ and a set of rows representing all possible internal states $\{S_1,\ldots,S_h\}$, as seen in figure 7.11(a). In this figure a number indicates the next state and a ring identifies a stable state. Note that outputs have been omitted. In the same way the input signals control the horizontal transitions between pairs of input signal states, written $[I_a,I_b]$, fed back state variables control the vertical transitions between pairs of internal states $[S_c,S_d]$. As the internal states of a flow table are represented by symbolic names our goal now is to generate a minimum number of state variables, replacing the symbolic names, that will offer minimum transition time and then assign Boolean values to these state variables. We will now formulate some definitions on minimum transition time state assignment.

Definition 1: When the binary code of the next internal state differs from the code of the present internal state in two or more bit positions, the circuit is said to be *racing* from the present internal state to the next internal state.

Definition 2: If a race condition exist and unequal transmission delays may cause the circuit to reach a stable state other than the one intended, the race is called *critical*. All other races are non-critical.

Definition 3: A *direct* transition from internal state S_i to S_j written $[S_i,S_j]$, is a transition where all internal state variables that are to undergo a change in value are simultaneously excited.

Definition 4: A direct transition $[S_i,S_j]$ races critically with the direct transition $[S_k,S_l]$ if unequal transmission delays may cause these two transitions to share a common internal state.

Definition 5: In a minimum transition time internal state assignment, all transitions are direct.

Our problem then is to find a solution that for a given set of encoded state variables ensures that for all specified state transitions, all races are non-critical. It follows from definition 2 and 4 that two state transitions or a state transition and a stable state in the same column may not intersect eachother unless they share the same end state. We do not have to consider intersection of state transitions belonging to different columns since a unique input signal state already distinguish them.

7.5.2 Constraints for Critical Race Free Encoding

To ensure that an internal state transition does not intersect other internal state transitions under the same input signal state (column), in the final solution, at least one state variable must, for that internal state transition, be uniquely encoded with respect to all other internal state transitions or stable states present under that same input signal state. To guarantee a solution in all situations we therefore need, for each internal state transition, one state variable y_i encoded such that it separates that transition from all other transitions that can occur during the same input signal state. Since a state variable can take on two values, a logical 0 or 1, we can use it to separate two internal state transitions by assigning the variable to be 0 for one of the transitions and a 1 for the other. In this way we allocate one state variable y_i for every two internal state transitions or state transition and stable state in each column until all pairs of transitions (unless they share the same end state) or transition and stable state have been treated. This methods fits right into the theory of *partitions* which is now defined.

Definition 6: A partition π on a set S is a collection of subsets of S such that their pairwise intersection is the null set. The disjoint subsets are called the blocks of π. If the set union of these subsets is S, the partition is completely specified, otherwise it is incompletely specified. Elements of S that do not appear in π are called unspecified or optional elements with respect to that partition.

Definition 7: A flow table with the characteristic that each unstable state leads directly to a stable state is called a *normal* flow table.

Thus for two state transitions [S_a,S_b], [S_c,S_d] or transition [S_a,S_b] and stable state S_e in the same column of the flow table we create a partition π_i{S_a,S_b ; S_c,S_d} or π_i{S_a,S_b ; S_e} where S_a,S_b and S_c,S_d or S_a,S_b and S_e are the blocks of the partition (separated by ';'). A state variable y_i is then implicitly allocated for the partition. This variable will then be assigned an arbitrary 1 or 0 for the first block of the partition and the negated value for the second block. This way we have separated these two transitions by the two possible values of state variable y_i. We do this for all possible pairs of internal state transitions or state transition and stable state in the column as seen in figure 7.11(b). We can now formulate the necessary condition for a critical race free state assignment:

Theorem 1: A row assignment allotting one y-state per row can be used for direct transition realization of normal flow tables without critical races if and only if for every transition [S_i,S_j]

1. if [S_k,S_l] is another transition in the same column, then at least one y-variable partitions the pair {S_i,S_j} and {S_k,S_l} into separate blocks; and

2. if S_k is a stable state in the same column then at least one y-variable partitions the pair

$\{S_i, S_j\}$ and the state S_k into separate blocks; and

3. for $i \neq j$, S_i and S_j are in separate blocks of at least one y-variable partition.

7.5.3 Row Compatibility Constraints

We can now create a Boolean matrix where each partition π_i represents a row and each internal state S_j represents a column. For each partition π_i (row) we then assign an arbitrary 1 or 0 to each of the blocks in that partition. If we let each row induce a state variable y_i we now have a solution for the state assignment problem. This solution however, may not yield a minimum number of state variables. Indeed, some of the allocated state variables may be redundant, i.e. they are the same for two or more rows in the Boolean matrix. We will remove such redundancies by a procedure similar to encoding of incompletely specified Boolean matrices [19].

Definition 8: Two rows of a Boolean matrix, R_i and R_j, have an *intersection* of R_i and R_j, written $R_i \cdot R_j$, iff R_i and R_j agree wherever both R_i and R_j are specified. The intersection is defined as a row which agrees with both R_i and R_j wherever either is specified and contains optional entries everywhere else.

Definition 9: Row R_i is said to *include* row R_j if and only if R_j agrees with R_i wherever R_i is specified.

Definition 10: Row R_i is said to *cover* row R_j if and only if R_j includes R_i or its complement $\overline{R_i}$.

7.5.4 Finding a Minimum Number of State Variables

The method of encoding a Boolean matrix is based on merging intersectable rows in the matrix. This is done by first generating a list of all pairwise intersectable rows as shown in figure 7.11(d). These pairs are then expanded to maximal intersectables as seen in figure 7.11(e). A maximal intersectable is an intersectable to which no more rows can be added. The next step is to select a minimum number of maximal intersectables such that each row of the Boolean matrix is covered by one intersectable. This is done by formulating the problem as a product of sums where a factor is a sum of all maximal intersectable containing a certain row in the Boolean matrix (Petrick's method [54]). In figure 7.11(f) the first factor consists of the sum of the maximal intersectables A and B since these both contain row 1 in the Boolean matrix and thus either of these maximal intersectables can be used to cover row 1. The same is done for all rows in the matrix giving us a product of sum expression for all possible combinations of maximal intersections that cover all rows in the matrix. This product of sum expression is then converted into a sum of products form. In this form, the term containing the fewest literals describes a least number of maximal intersectables that cover all rows of the original Boolean matrix. In

the example in figure 7.11(f) this term is BCD which then requires three state variables for a critical race free state assignment. The minimized Boolean matrix with the final state variable encoding can be seen in figure 7.11(g).

To conclude this section: To ensure a critical race free circuit the state assignment method in this example requires three state variables with the encoding given in figure 7.11(g) for each corresponding symbolic state. This information is then passed back to the flow table where the symbolic state rows are replaced by the state variables and their encoding. Note that the intermediate internal state rows must also be inserted in the flow table.

We now apply this state assignment method to our simple burst mode example originally specified in figure 7.10. Figure 7.12(a) shows the minimized flow table from figure 7.10(f). For sake of clarity outputs have been removed and stable states have been circled. From the minimized flow table we generate partitions and assign them arbitrary values in the Boolean matrix (figure 7.10(c). Pairwise intersectables are then generated and maximal compatibles are constructed from them. Finding a minimal solution in this case is trivial since there are only two maximal compatibles, P and Q, both which must be present to cover all internal states. The resulting state variables, z_0 and z_1, and their final encoding are illustrated in figure 7.12(f).

7.6 Hazard Free Two Level Logic Minimization

Now that we have the number of state variables and their final encoding we can generate a primitive cover for each of the outputs and state variables. Such a primitive cover however, contains separate cubes for each of the required cubes. This is a non-optimal cover since in many cases it is possible to *expand* required cubes which will then require less number of literals to implement. Cubes can often be further expanded by taking the don't care set into account. By expansion it is also often possible to find cubes that each covers more than one required cube which will then require less number of product terms to implement the cover. As a result, expanding cubes of the primitive cover saves area as well as reduces the delay through the circuit and is an important phase in logic synthesis.

The method for hazard free minimization presented here is based on the Quine-McCluskey algorithm for solving the two-level logic minimization problem with proper restrictions added to ensure no logic hazards are introduced by the covering procedure. This method is explained in more detail in [50]. A heuristic near optimal method is presented in [69]. The Quine-McCluskey algorithm has three basic steps:

1. Generate the prime implicants of a function.

2. Construct a prime implicant table.

	I_1	I_2	I_3	I_4
a	(1)	4	7	10
b	1	5	8	(11)
c	2	(4)	(8)	(10)
d	-	(6)	(7)	11
e	1	6	8	(12)
f	(2)	(5)	-	12

(a) Original flow table

I_1: π_1 = {a,b ; c,f}; π_2 = {a,e ; c,f}

I_2: π_3 = {a,c ; d,e}; π_4 = {a,c ; b,f}; π_5 = {b,f ; d,e}

I_3: π_6 = {a,d ; b,c}; π_7 = {a,d ; c,e}

I_4: π_8 = {a,c ; b,d}; π_9 = {a,c ; e,f}; π_{10} = {b,d ; e,f}

(b) Generate partitions

	$\pi_1 ... \pi_{10}$		*States* a	b	c	d	e	f
1	ab	cf	0	0	1	-	-	1
2	ae	cf	0	-	1	-	0	1
3	ac	de	0	-	0	1	1	-
4	ac	bf	0	1	0	-	-	1
5	bf	de	-	0	-	1	1	0
6	ad	bc	0	1	1	0	-	-
7	ad	ce	0	-	1	0	1	-
8	ac	bd	0	1	0	1	-	-
9	ac	ef	0	-	0	-	1	1
10	bd	ef	-	0	-	0	1	1

(c) Create Boolean matrix

(1,2) (1,7) (1,10) (6,7)
(2,5') (2,6) (7,10)
(3,4) (3,5) (3,8) (3,9) (8,9) (8,10')
(4,5) (4,8) (4,9) (9,10)
(5',6)

(d) Create pairwise intersectables

A (1,2)	D (3,4,8,9)	G (6,7)
B (1,7,10)	E (3,5)	H (8,10')
C (2,5',6)	F (4,5)	J (9,10)

(e) Create maximal intersectables

(A + B)(A + C)(D + E)(D + F)(C + E + F)(C + G)
(B + G)(D + H)(D + J)(B + H + J)

BCD + ABDFG + ADFGH + ADFGJ + ACDGH
+ ACDGJ + ABDEG + ADEGH + ADEGJ
+ BCEFHJ + ADFGHJ

(f) Use Petrick's method to get a minimal solution (BCD).

	a	b	c	d	e	f
B (1,7,10) : y_1	0	0	1	0	1	1
C (2,5',6) : y_2	0	1	1	0	0	1
D (3,4,8,9) : y_3	0	1	0	1	1	1

(g) Minimized Boolean matrix with final state variable encoding.

Figure 7.11: Example of minimum transition time state assignment

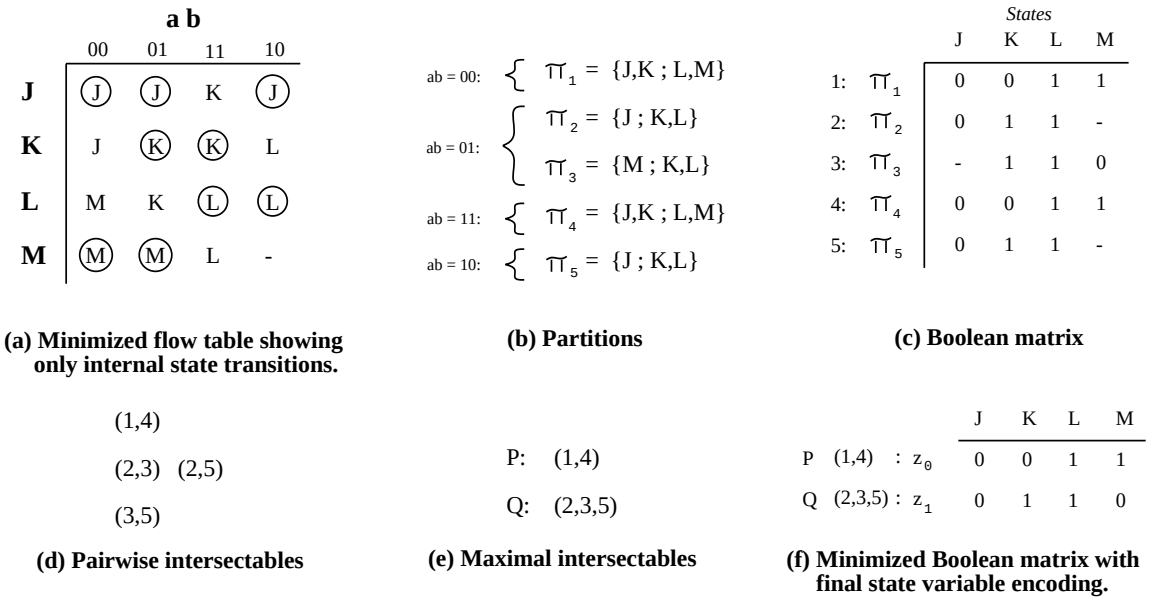


Figure 7.12: Critical race free state assignment applied to Burst mode example.

3. Generate a minimum cover of this table.

An implicant is a cube that does not contain any minterm in the off-set of a function f. Theorem 1 (a) and (c) in section 7.4.1 restricts which implicants can be used in the cover without introducing hazards. Only implicants which do not intersect any required cube illegally may be present in a cover of the function. Such implicants are called *dynamic hazard free implicants*, or *dhf-implicants*. A *dhf-prime implicant* is a dhf-implicant which is not contained by any other dhf-implicant of f, and which is not entirely contained in the don't care set. An *essential dhf-prime implicant* is a dhf-prime implicant which contains a required cube contained in no other dhf-prime implicant. Theorem 1 (b) states that each required cube must be contained in some cube of the final cover. The two level hazard free minimization problem then is, *to find a minimum cost cover of a function using only dhf-prime implicants where every required cube is covered.* The steps in this minimization problem are presented below.

Step 0. Make Sets: The privileged cubes, the on-set formed by the required cubes, and the off-set of the specified transitions have already been identified during the primitive flow table construction. They are passed along to the logic minimization step as the sets *priv-set*, *req-set*, and *off-set* respectively.

Step 1. Generate DHF-Prime Implicants: The next step in the logic minimization then is to generate the set of dhf-prime implicants for the function f. This is done by first generating the prime implicants of f from the req-set and off-set using existing techniques presented in [59, 60]. From this set of prime implicants we want to form a set of dhf-prime implicants. Prime implicants that illegally intersects any privileged cubes must therefore first be

identified. Since such a prime implicant may contain a required cube not covered by any other prime implicant, it cannot simply be removed from the set. Instead such a prime implicant is iteratively reduced into cubes that do not intersect any privileged cube. Resulting reduced cubes that are contained in other cubes are removed from the set. As an optimization, dhf-prime implicants that do not contain any required cube can also be removed from the set.

Step 2. Generate DHF-Prime Implicant Table: A dhf-prime implicant table is now constructed from the given set of required cubes and dhf-prime implicants. The rows of this table are labeled with the required cubes that must be covered by some dhf-prime implicant. The columns are labeled with the dhf-prime implicants. This table then sets up the problem of two level hazard free logic minimization as a unate covering problem.

Step 3. Generate a Minimum Cover: The dhf-prime implicant table can now be solved in three steps using simple standard methods. *Essential dhf-prime implicants* are first extracted using standard techniques. The flow table is then iteratively reduced by removing rows and columns using *row-dominance* and *column-dominance* operations [17]. These operations may lead to further possibilities of secondary removal of essential dhf-prime implicants. These operations are iterated until there is no further change. If the table is still not empty after the iterations have finished, a *cyclic covering problem* remains. Each row in the table now lists dhf-prime implicants that can be used to cover a required cube. The problem of finding the cover requiring the least number of such dhf-prime implicants can be solved using Petrick's method. For each column, the dhf-prime implicants that can be used in a cover is then expressed as a sum. The covering problem for the table can then be formulated as a Boolean product of all these sums. This expression is then multiplied out to a sum of products expression and a product containing the least number of dhf-prime implicants can then be selected as a minimal solution. Other criteria can also be used such as selecting the solution giving a minimum number of literals.

Note that while reduction of prime implicants in step 1 removes the illegal intersections with privileged cubes, it may at the same time remove prime implicants that are needed to completely cover required cubes. In such cases no hazard free solution exists for the given function. In the case of burst mode synthesis however, we have already made sure that such cases cannot occur by the constraints on state merger during state minimization of the primitive flow table.

Logic minimization applied to our simple burst mode example originally specified in figure 7.10 is illustrated in figure 7.13. Note that the symbolic next state names have been replaced by the encoded state variables z_0 and z_1 and vertical state transitions have been replaced by arrows showing the transitions to the next states. From this Karnaugh map we then generate dhf-prime implicants. In this example it was not necessary to reduce any prime implicant in order to generate dhf-prime implicants. We then construct a dhf-prime implicant

table in which each dhf-prime implicant that can cover a required cube is marked with an "X". Essential dhf-prime implicants are then identified and used to generate a minimum hazard free cover for the output or state variable in question. The minimum hazard free cover then is a Boolean expression formed as a two level sum of products that is directly implementable as a network of logic AND and OR gates.

7.7 AFSM Synthesis in ACK

The burst mode controllers created by the method of burst mode generation presented in section 6 must now be synthesized to Boolean functions for subsequent implementation as a network of logic gates.

7.7.1 Hazard Free Synthesis

ACK makes use of the 3D method [74] to synthesize hazard free Boolean functions from the burst mode specifications. There are several reasons for chosing this tool. First of all, since the output and state signals do not have to pass through any latches, this method offers a possible advantage in reduced propagation delay. Second, this method requires only standard gates (AND,OR,INV) found in virtually any library. Not having to bother with sequential components such as latches also makes technology mapping and realization easier since no consideration for setup and hold times or timing estimation of the clock signal delay to latch correct values is needed. Third, this method offers synthesis using both the function region and excitation region approach [13] as well as the extended form of burst mode machine specification. Although only the functional region approach and ordinary burst mode machines are currently supported in the high level synthesis methodology of ACK, these additional features are easy to implement and could have advantages in certain situations. The only disadvantage of the 3D method is the possibly larger fundamental mode delay the three cycle model that is used for state changes introduces compared to a minimum (single) transition time state assignment method as presented in section 7.5. On the other hand, this fundamental mode delay can be hidden in the execution time of the datapath resources which are often slow enough to allow the control logic to attain quiescens before new input changes arrive.

7.7.2 Technology Mapping

After we have synthesized our burst mode controllers we have a collection of Boolean functions that now need to be mapped to a network of gates.

A part of the technology mapping procedure is to find the best solution for a given

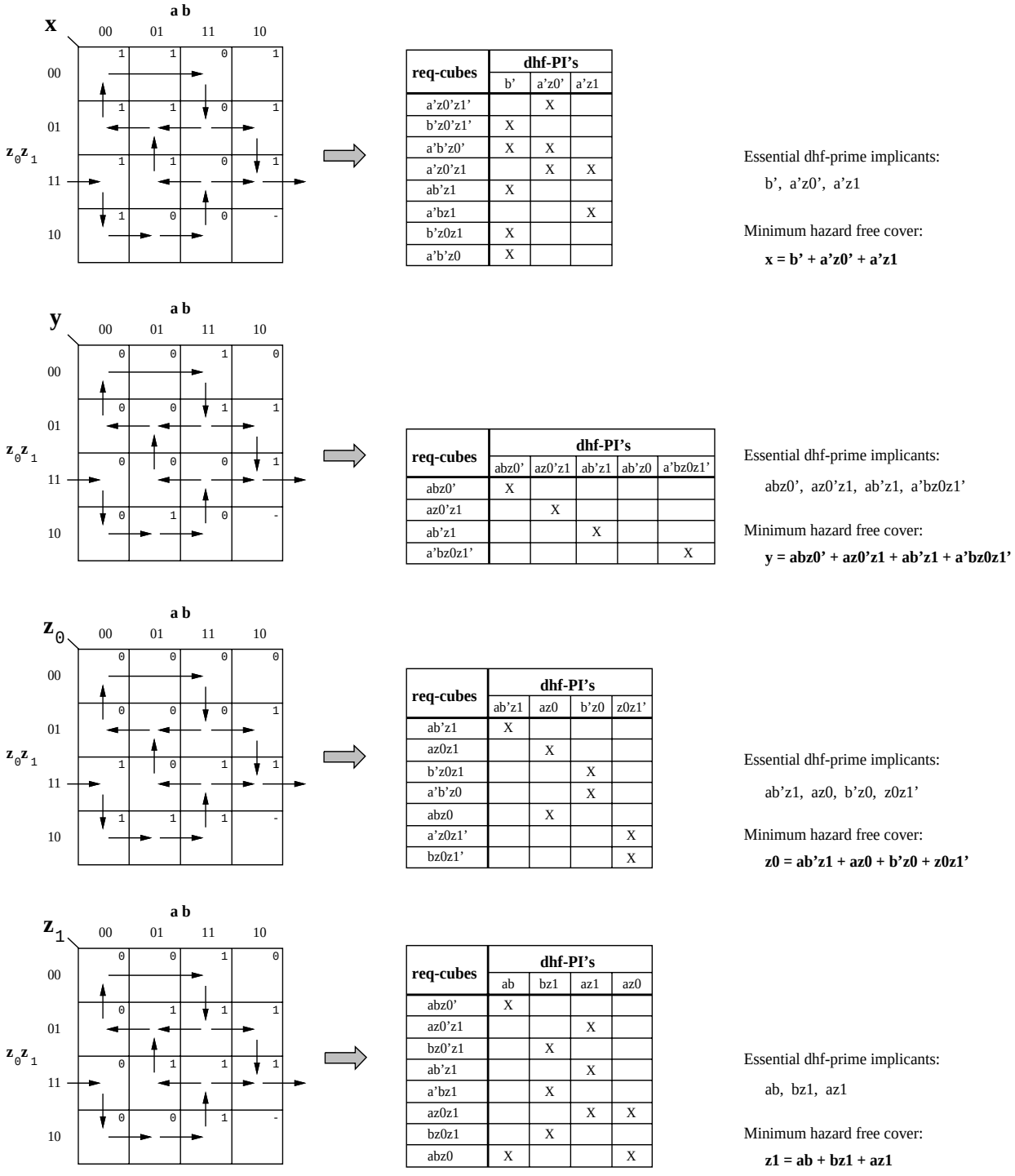


Figure 7.13: Hazard free two level logic minimization applied to Burst mode example.

area/performance requirement. Since complex burst mode state machines often give high fan-in gates which besides not being supported by all libraries are also very slow, we need a method to decompose these into smaller and faster gates. The targeting of a two level sum of products form together with the bounded delay model used gives us the possibility of decomposing such gates by means of algebraic transformations, something that is not possible for many unbounded delay models. It has been shown in [71] that many algebraic transformations including the associative, distributive and DeMorgans law do not introduce any new hazards in such circuits. An important part of the technology mapping process then is to find the combination of gate sizes that gives the best area/performance tradeoff from a given library of gates.

A method in [5] also allows decomposition based on average case delay. This method is based on assigning probability values to branches in the burst mode specification. The gate decomposition is then based on with what frequency different actions are executed. Depending on the probability distribution a performance gain of between 10-20% can be achieved using this method.

Apart from mapping to a two level sum of products implementation there are many other styles of implementation that can be targeted [33, 36, 63]. Some of these, and especially a method to target custom complex gates will be discussed in the next section.

7.8 Conclusions

This section has presented the concept of asynchronous finite state machine synthesis. Two methods used to generate hazard free implementations from burst mode specifications have been briefly presented. The important requirement of hazard free minimization and definition of hazards has also been given. An overview of the technique of representing a burst mode machine as a primitive flow table and the minimizations that can be done to that table has been presented. A method for state assignment of the symbolic states used in the flow table has also been presented. Finally a method for two level hazard free logic minimization has been described. A short discussion of the synthesis method used in ACK and of technology mapping has also been given.

Chapter 8

Synthesis and Technology Mapping to Complex Gates

Although targeting two-level simple gates can give efficient implementations the ability to generate customized complex gate based circuits is an important feature for performance critical applications.

This section addresses the problem of realizing hazard-free single-output Boolean functions through a network of *customized complex CMOS gates* for asynchronous controllers. A customized CMOS gate network can be either a single CMOS gate or a multilevel network of CMOS gates, where each CMOS gate is tailored to give the most efficient implementation for a given specification. It is shown that hazard-free requirements for such networks are less restrictive than for simple gate (AND/OR, MUX, AOI, etc.) networks. Analysis and efficient synthesis methods to generate such networks under a multiple-input change assumption (MIC) will be presented.

Customized CMOS gate implementations have been successfully used to design a large number of *burst-mode* asynchronous controllers [14, 65]. However, previous methods do not present systematic models and synthesis algorithms to take advantage of the particular hazard properties of these circuits. There are several reasons for considering customized CMOS complex-gate based circuits. As VLSI feature size decreases and wire delays become significant, customized CMOS complex-gates can provide more efficient controller implementations compared to standard-cell place and route tools. Also, the recent availability of better layout synthesis techniques that can automatically generate layouts for arbitrary transistor networks makes customized complex-gate based controllers a more viable alternative. Finally, we provide methods to derive complex-gate networks which relax some of the synthesis constraints needed for hazard-free simple-gate implementations.

8.1 Related Work

Currently, there are two main approaches to deriving hazard-free logic gate networks for asynchronous circuits. The first is a *function region* approach. In this method, one tries to find a hazard-free network for a single output Boolean function by taking into account the on-set and the off-set of the function, with respect to a specified set of multiple-input changes. The second approach deals with finding the *excitation regions* of the function. In this method, the regions of the Boolean space where the output is *enabled to change* are identified as "set" and "reset" functions. These functions are implemented and are used to control the switching of a state holding element such as a C-element or RS latch.

Various techniques for hazard-free logic minimization have been proposed for the function region approach. An exact hazard-free two-level logic minimization algorithm, based on a modified Quine-McCluskey method, is given in [50]. Hazard non-increasing transformations and algorithms for *multilevel* optimization of gate-level logic have been given in [35, 71]. A BDD-based method [36] which targets multilevel multiplexor-based networks has been developed. Technology-mapping techniques to perform hazard-non-increasing mapping of two level AND/OR networks into complex gate networks from a standard cell library have been given in [63]. Other technology mapping techniques have implemented Boolean functions as single gate hazard-free CMOS complex gate circuits [14, 65]. However, no systematic procedure to derive such CMOS gates has been outlined, which includes precise hazard-free requirements for these gates.

For the class of methods that use the excitation regions, single CMOS complex-gate circuits, called generalized C-elements [38], have been used as target implementations. These techniques usually rely on the use of state holding elements on the output of the gate.

The contribution of this section is to address the problem of deriving hazard-free customized CMOS realizations for asynchronous controllers under multiple-input changes, using the function region approach. This problem is encountered during the synthesis of burst-mode circuits [49, 74] and is a general problem in asynchronous synthesis. In particular, we present a style of CMOS gate design, called *SOP/SOP form*, that *reduces* the constraints in hazard-free synthesis of single CMOS complex gates. Second, we present a generalization of this technique to multilevel networks. This technique allows efficient solutions to a large class of asynchronous specifications. These techniques allow designers the flexibility to perform hazard-free mapping tailored to customized complex-gates, instead of being confined to a standard library.

In section 8.2, we will introduce some basic terminology. section 8.3 describes a technique to derive single CMOS complex gates. We will present techniques that address multilevel synthesis of such complex gate circuits in section 8.4. Results will be provided in section 8.5.

8.2 Terminology

We will provide definitions relating to pass transistor and CMOS logic gates we will use. We will then briefly describe some terminology on hazards.

8.2.1 Pass Transistor Networks

A model for pass transistor logic has been developed in [53, 56, 61, 63]. We will describe and extend the model presented in these works for single CMOS gates.

Definition 1. A *pass transistor* is a MOS transistor operated as a switch, where the transistor drain (source) is connected to the signal to be passed along, the transistor gate is connected to the control input, and the output signal is taken from the transistor source (drain).

At this point, we do not distinguish between a single N-type or P-type MOS transistor and a pass "gate" composed of complementary pair of transistors connected by complementary control variables.

Definition 2. A *pass network* is an interconnection of pass transistors which realizes a particular switching function $f(X)$, where $X = \{x_1, x_2, \ldots, x_n\}$ is the set of inputs to the function.

Definition 3. A *branch* of a pass network implementing the switching function $f(X)$ is a series connection of pass transistors where the drain or source of the transistor at one end of the series is connected to an input source selected from the set $\{0, 1, x_i, x_i'\}$

Definition 4. A *pass variable* is an input to a branch of the pass network. A pass variable may be chosen from the set $\{0, 1, x_i, x_i'\}$.

Definition 5. A *control variable* is an input to the gate of a transistor in the pass network. When the control variable has a value equivalent to a logic "1", the pass gate conducts.

Definition 6. A *pass implicant* is a Boolean switching function which denotes the function of a branch of a pass network, and includes information about both the pass variable and the switching function. A pass implicant is denoted $C_i[P_i]$, where C_i is a product term composed of control variables which control the pass transistors in that branch, and P_i is the pass variable which will get passed to the output if the product term is true.

This is similar to an implicant in Boolean algebra, except that instead of passing a constant "1" if the implicant is true, the value of the pass variable is passed if the implicant is true.

Definition 7. A *pass function*, $F_p : B^n \rightarrow \{0, 1, Z\}$ is a sum of pass implicants.

Definition 8. A *CMOS gate* consists of a P transistor pass network with only one pass

variable "1" and an N transistor pass network with only one pass variable "0". The outputs of the two networks are connected together to form the output of the CMOS gate. For a CMOS gate which is to implement a function F, the P pass network implements the pass function F and the N pass network implements the pass function $\overline{F}$.

Hazards and Delay Model

As discussed in section 7.3 there are two basic classes of combinational hazards: *function* and *logic* hazards. Function hazards are a property of the logic function, whereas logic hazards are purely a property of the implementation. Within the class of logic hazards, there are single-input change (SIC) hazards and multiple-input-change (MIC) hazards. Additionally, each class of hazards (function and logic) includes both *static* and *dynamic* hazards. In this section we will consider MIC logic hazards, i.e., we will assume that the given Boolean function is free of function hazards.

For the resulting complex gate implementations using the method presented in this section to be hazard free it is a requirement that only CMOS gates as defined in definition 8 are used. If not, i.e. the output of a CMOS gate is used as a pass variable in a transistor network, additional hazards are possible. For the remainder of this section it is therefore assumed that the output of a CMOS gate is only used as a control variable in other CMOS gates.

The delay model assumed in this work is that of unbounded gate and wire delays, as in previous approaches [8, 50, 36, 35]. This is a conservative model, which assumes that inputs in a MIC can arrive at any time and in any order, and that gates and wires have unknown delay. However, our model is limited by one timing constraint: on the time period for which the capacitance on a CMOS gate output holds its charge when there is no conducting path to power or ground rails through the p or n transistor networks (only leakage occurs). This time period is assumed to be much larger than the *duration of any static hazard*. This requirement is quite reasonable, since the time is related to the maximum difference in arrival of a variable and its complement to different stacks in the gate.

8.3 Hazard-free Single CMOS gates

8.3.1 Hazards in Dual Realizations

CMOS complex gate networks can be implemented in many different ways. The standard technique to implement the functions F and $\overline{F}$ is to obtain a sum of products for one pass network (p or n) and the dual of this network then becomes the complementary product of sums network. A dual of a function, f, implementing a pass network, can be obtained by applying De

Morgan's law to $\overline{f}$. Since De Morgan's law has been proven to be hazard preserving, eventual hazards in the function will also be preserved by the transformation. Because of this hazard preserving property however, we are also assured that no new hazards are introduced by the transformation.

For example, assume that function f has a static $1 \rightarrow 1$ transition $[A, B]$. A hazard during such a transition can only manifest if the cover, C, implementing f does not contain a cube completely covering $[A, B]$. This means that no single product term holds the current value throughout the transition. As mentioned in section 7.3, switching from one product term to another during a transition may result in a glitch. In a sum-of-products transistor network (p or n) however, this glitch would manifest as a brief moment when none of the transistor stacks are conducting. The dual of such a function then, would preserve the hazard of transition $[A, B]$. Since the dual function $\overline{f}$ is a negated product-of-sums of f, none of the transistor stacks implementing the dual are supposed to conduct during transition $[A, B]$. The preserved hazard however, will, in the dual implementation, manifest as a brief moment where one of the transistor stacks may briefly conduct. Let's assume that the ON-set of f is implemented as a sum-of-products network of p-transistors and the OFF-set as the dual, $\overline{f}$, implemented as a product-of-sums network of n-transistors. For static $1 \rightarrow 1$ transition $[A, B]$ then, the static hazard will result in that the p-transistor network might not conduct for a brief moment. At the same time however, the n-transistor network might conduct for a brief moment. This would result in a $1 \rightarrow 0 \rightarrow 1$ glitch on the output of the CMOS gate.

In this section we will present a more interesting realization in terms of hazard behavior, where *both F and $\overline{F}$ are implemented as sum-of-products networks*, referred to as an *SOP/SOP form* of complex gates. An example SOP/SOP complex gate implementation is shown in figure 8.1. Function F is implemented using p-transistor stacks, and function $\overline{F}$ is implemented using n-transistor stacks.

8.3.2 SOP/SOP Realization

We will first examine the hazard behavior of the SOP/SOP form of realization for CMOS complex gates. In order to do this, we will examine both SIC and MIC static and dynamic transitions on a case-by-case basis.

Case 1: Static Transitions. *For static transitions a SOP/SOP complex gate circuit is hazard-free at the output.* Both SIC and MIC static hazards occur when a given static transition causes a change from one cube of the cover to another, causing a brief period when the transistor network is not conducting. Consider F and $\overline{F}$ to be on-set and off-set covers respectively implemented as p and n transistor networks in a complex gate. It has been shown

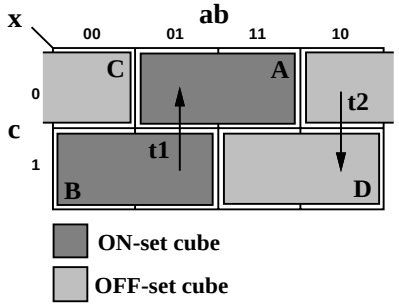
in [71] that a sum-of-products implementation of the on-set F does not have any $0 \rightarrow 0$ hazards.[1] Similarly, $\overline{F}$ does not have any $1 \rightarrow 1$ hazards. This means that in a sum of products form, a static transition over function F is always outside the cover of $\overline{F}$ and vice versa. As a result, for a SOP/SOP form of complex gates, even if the transistor network of F has a static hazard (that is, a brief moment when no p stack is conducting), the transistor network of $\overline{F}$ remains off (that is, no n stack will conduct during the transition). Therefore, *the output capacitance of a SOP/SOP complex gate holds its current charge for the duration of a static hazard* (we have no conducting path to either power or ground), and the hazard is not seen at the output.

As an example, consider the Karnaugh map and complex-gate implementation in figure 8.1. Transition t_1, from $abc : 011 \rightarrow 010$, is a static $1 \rightarrow 1$ transition. Transition t_2, from $abc : 100 \rightarrow 101$, is a static $0 \rightarrow 0$ transition. For a SOP/SOP complex gate, the on-set implements the p-transistor pass network, using cubes $A = bc'$ and $B = a'c$, and the off-set implements the n-transistor network, using cubes $C = b'c'$ and $D = ac$. For a SOP/POS complex gate the on-set implements the p-transistor pass network, using cubes $A = bc'$ and $B = a'c$, and the dual $(b'+c)(a+c')$ then implements the n-transistor network. A simple-gate AND/OR network would be implemented using the on-set cubes A and B. The hazard behaviors of these gates then become the following.

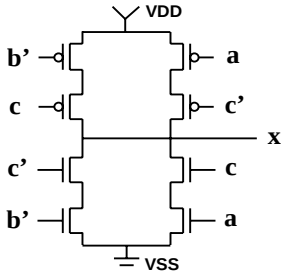
The simple-gate AND/OR implementation is free of hazards for transition t_2 but not for t_1. During transition t_1, the AND-gate for B goes low and the AND-gate for A goes high. If the AND-gate for A is slower than the AND-gate for B, the OR-gate output will generate a $1 \rightarrow 0 \rightarrow 1$ hazard. For the SOP/POS complex gate implementation, transition t_2 is hazard-free. During transition t_1 however, the n-transistor network may briefly conduct causing a $1 \rightarrow 0 \rightarrow 1$ hazard at the output. The single SOP/SOP complex-gate network however, is hazard-free for both transition t_1 and t_2. Although, for transition t_1, the p-transistor stacks for A and B can briefly be off at the same time (when c goes low), *no* n-transistor stack will conduct during the transition. As a result, the output will hold its current charge. The same holds true for transition t_2, the n-transistor network may briefly stop conducting, but *no* p-transistor stack will conduct during the transition. The output is therefore free of hazards for both of these transitions. Subsequently, there is no need to eliminate static hazards during synthesis of F and $\overline{F}$ when targeting SOP/SOP form of complex gates.

Case 2: Dynamic Transitions. For the case of SIC transitions, it has been shown in [71] that a dynamic SIC hazard cannot occur (assuming no product contains both a variable and its complement). Since F and $\overline{F}$ are in two-level AND/OR form, no hazards will occur in the complex-gate output in this case. For the case of MIC transitions, though, we will have to make the p and n pass networks hazard-free for dynamic transitions. Otherwise, even in the

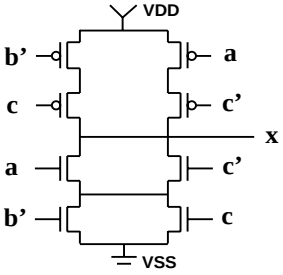
[1] Note that, throughout this section, we assume that no product contains both a variable and its complement, otherwise additional hazards are possible [71].



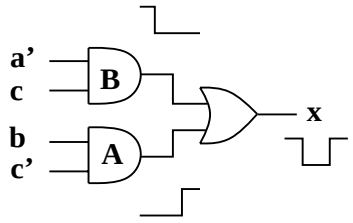
Karnaugh map



Hazard free SOP/SOP complex gate



Hazardous SOP/POS complex gate



Hazardous simple gate implementation

Figure 8.1: K-map and static hazard-free SOP/SOP complex gate

SOP/SOP form, both the p network and the n network may have dynamic hazards, creating a hazard at the output of the complex gate.

8.3.3 Algorithm For SOP/SOP Realizations

Our hazard-free algorithm for SOP/SOP complex-gate realizations is similar to an existing algorithm for hazard-free two-level simple-gate networks [50] presented in section 7.6. The key difference is that our new algorithm uses *fewer constraints*: we can ignore hazards due to static transitions in the SOP/SOP realization.

The steps after the generation of sets are common to both the complex-gate algorithm and two-level algorithm, and are summarized below. Since both p and n-transistor networks implements the ON-set and OFF-set of the given Boolean function, f, as sum-of-products networks, the procedure below is first applied to f. The function is then inverted to $\overline{f}$ (the ON-set of $\overline{f}$ then is the OFF-set of f) and applied to the procedure. This yields hazard free sum-of-products transistor networks for both ON and OFF-set of f.

In the logic synthesis procedure we first derive the req-set, priv-set and off-set for the given function. We then derive the dhf-prime implicants based on the req- and off-sets as well

as the privileged cubes. A *unate covering problem* is then formulated: the problem is to cover all the required cubes by a minimum set of dhf-prime implicants. If all of the required cubes cannot be covered by the dhf implicants, a solution does not exist.

The only difference in the synthesis approach for SOP/SOP complex gates is during construction of the req-set in the **Make-set** step. In our modified algorithm *Complex-Make-sets*, we follow the same steps as *Make-sets*, but with one key difference: we do *not* generate required cubes for $1 \rightarrow 1$ static transitions. The reason is that, for complex-gate realizations, there are no static $1 \rightarrow 1$ hazard-free requirements. Instead, to insure that the on-set of the function is still covered, we simply add the on-set *minterms* (*not* cubes) into the required set which are not already present in the required set. In summary, the required set generated by Complex-Make-sets consists of (i) all the required cubes associated with dynamic transitions, and (ii) the on-set minterms that are not already covered by other required cubes.

Note that the unate covering problem in our complex-gate algorithm is less restrictive: required cubes for static $1 \rightarrow 1$ transitions need not be covered. In fact, there are problems which have no 2-level hazard-free solution, but where a complex-gate solution exists. For instance, the example used in [50] to demonstrate the absence of a solution for hazard-free AND/OR implementation, has a solution in the SOP/SOP form of complex-gate implementation.

Figure 8.2 illustrates a small example where output x of the given burst mode machine is implemented both using a two level implementation using simple AND/OR gates and using the SOP/SOP customized complex gate method. As can be seen in figure 8.2(c) and (d) the standard gate AND/OR implementation needs 64 transistors while the complex gate solution only needs 12 transistors to implement the same function. Besides requiring less area, this also reduces the delay through the circuit.

8.4 Multi-level Implementations

8.4.1 Background and Overview

Several approaches have been used for multilevel hazard-free logic synthesis. In [8], a technique was presented to derive single-output multilevel AND/OR gate implementations. The algorithm assumes a fully-specified function and attempts to eliminate hazards even for unspecified transitions, leading to inefficient implementations.

A method using BDDs that target multilevel multiplexer based circuits is presented in [36]. The multiplexers in this method are assumed to be hazard-free.

A method to eliminate dynamic hazards from signal transition graph specifications by iterative factoring has been presented in [45]. Factoring does not eliminate static hazards.

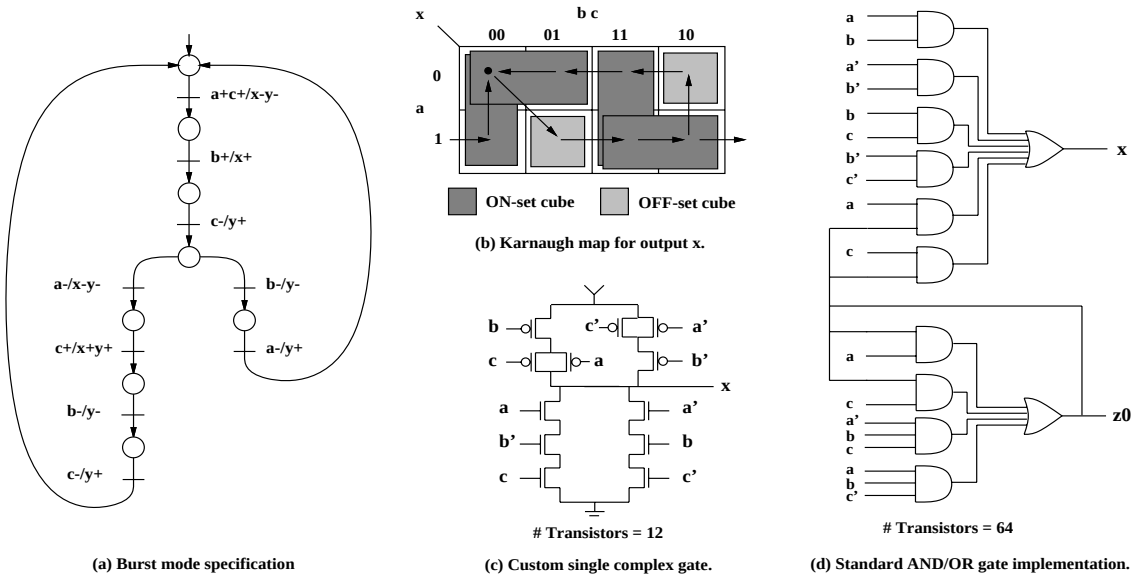


Figure 8.2: Example of single hazard-free SOP/SOP complex gate implementation

Work in [63] targets multilevel hazard-free circuits, starting from a hazard-free two-level circuit. In this method a hazard-free two-level function is decomposed into base functions using De Morgan's theorem and associative laws and then partitioned into cones which are mapped to library elements based on associated hazards. This work could be extended to use customized complex gates instead. However, since it is based on an already existing AND/OR function, it cannot take advantage of the static hazard robust behavior of the SOP/SOP form and thus cannot give solutions to a larger class of problems.

We will therefore present a new technique which is an extension of work in [8], but which deals with the special hazard requirements of SOP/SOP complex-gates. The procedure is presented in two steps. First, we will give a model for the multilevel complex gates we target. We will then outline our decomposition algorithm.

8.4.2 CMOS Multilevel Networks

The procedure outlined in the rest of this section, assumes that each CMOS gate is implemented in the SOP/SOP form discussed in the last section. A multilevel network of CMOS complex gates is defined as a single output network of multiple levels of complex gates, where the control variable for each transistor of the p and n pass networks is either an input literal or the output of another CMOS complex gate.

Consider the goal of implementing a Boolean function under a given set of MIC transitions as a single complex gate. Consider the p pass network to be implemented (the arguments for the n network are symmetric). We attempt to derive the SOP form (series/parallel) network

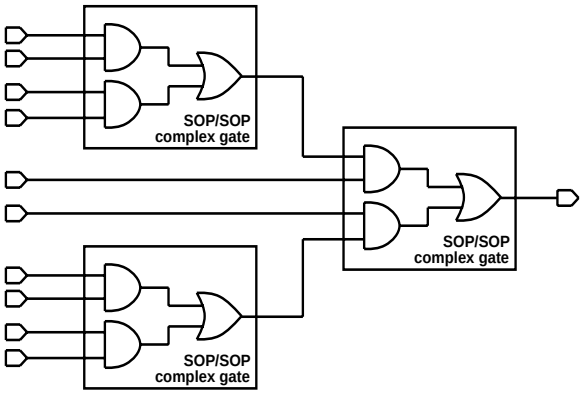


Figure 8.3: Multilevel SOP/SOP complex gates

with only input literals as control variables to the transistors. If such a solution cannot be found, we attempt to find a solution where some of the transistors have control variables which are the output of separately implemented complex gates in the SOP/SOP form. This procedure yields alternating levels of AND and OR gates starting from the output and recursively derives implementations for smaller functions until input literals are reached, as can be seen in figure 8.3. For synthesis of such multilevel circuits, one must keep in mind that the target complex gates do not require static hazard covers; therefore, we still take advantage of the static hazard-free nature of the SOP/SOP form. We will use this model to derive our complex gates taking the p pass network and n pass network separately. Note, though, that hazards now may occur due to the interaction of separate complex-gates in the network. These issues are addressed below.

We will illustrate the method with the example shown in figure 8.4. Consider the problem of generating a hazard-free single output function for the output x of a burst mode state machine represented by the Karnaugh map in figure 8.4(a). In this case, x is a simple combinational function, which must be implemented without logic hazards. Specified input transitions are given in the figure. The required cubes for dynamic transition $t1$ are bc' (A) and $a'b$, and the required cubes for dynamic transition $t2$ are $a'c$ (B) and $a'b$.

This covering problem has no hazard-free two-level solution. Each required cube must be covered by some dhf-prime implicant. Required cube A is covered only by itself (bc'), which illegally intersects dynamic transition $t2$. Similarly, required cube B is covered only by itself ($a'c$), which illegally intersects dynamic transition $t1$. Therefore, no dhf-prime exists to cover A and B.

Burst-mode sequential synthesis tools [49, 74] avoid this problem at an earlier point in synthesis: during state minimization. By using careful constraints on state merger, these methods produce Boolean functions for which a hazard-free solution exists. That is, a feedback variable would be added, making the circuit sequential rather than combinational.

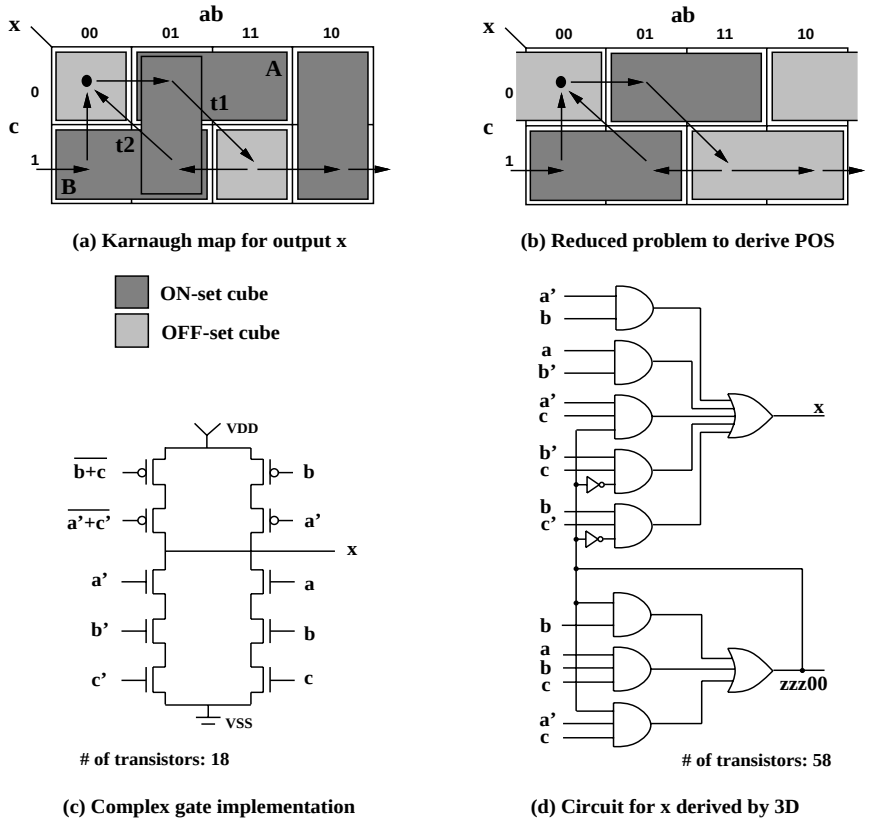


Figure 8.4: Multilevel SOP/SOP complex gate example

Since no hazard-free two level solution exists we now attempt to derive multilevel combinational logic for the output x. First we derive all required cubes and dynamic hazard-free prime implicants (DHFPI). The DHFPIs are $a'b$ and ab' for the on-set. All other implicants have illegal intersections. The two required cubes that cannot be covered in the on-set are bc' (A) and $a'c$ (B). The DHFPIs for the off-set are $a'b'c'$ and abc. The cubes for both the on and off-set that can be covered are now respectively implemented as the p and n transistor networks of a SOP/SOP complex gate.

Since no cover of required on-set cubes A and B could be found by considering the *on-set* we will now examine if it can be covered by taking the dual of a cover of the *off-set*. We will therefore try to derive a product of sums implementation of the union of the uncovered cubes in the on-set. The reduced problem for this is shown in figure 8.4(b). A hazard-free SOP cover of the off-set is $ac + b'c'$. By taking the dual of this cover we can generate a hazard-free cover of the on-set which then becomes $(a' + c')(b + c)$.

This POS cover is connected to the SOP/SOP complex gate by a series p stack of transistors as can be seen in figure 8.4(c). Since DHFPI $a'b$ is already covered by this hazard-free cover we can remove it from the final cover for x. $(b+c)$ and $(a'+c')$ can then be implemented

using simple OR gates. Note that the static hazard for transition abc:101 $\rightarrow$ 001 does not manifest in the SOP/SOP multilevel implementation. Also, no state variable is needed.

Figure 8.4(d) shows the result as generated by the 3D [74] synthesis tool, which uses *hfmin* [50] to produce a hazard-free two-level AND/OR gate implementation. A state variable has been added to eliminate the hazard problem in this case.

An algorithm for deriving multilevel complex gates is given in figure 8.5. The top level algorithm is $Derive\_CMOS\_Multi$, which calls the recursive procedure $Derive\_Multi$ and then derives complex gate implementations from the cover returned by $Derive\_Multi$. The function $Derive\_Multi$ is first discussed. Initially it is assumed that one attempts a two-level sum of products solution. Since we are dealing with alternating *levels* of sum of products and product of sums implementations, our algorithm works slightly differently for each of the levels. We derive the sum of products and the product of sums alternately. Therefore the algorithm first starts with trying to find a sum of product solution. In the absence of static hazards, such a solution may not exist when a required cube(s) for one dynamic transition is a (are) stray cube(s) for another. We will refer to such cubes as *conflicting* cubes. Sets of conflicting cubes are formed. For example if required cubes A and B are conflicting and similary B and C, the set (A, B, C) is considered the maximal set of conflicting required cubes. For each maximal set of conflicting required cubes, it attempts a product of sums solution, and then again recursively for the remaining cubes attempts a sum of products and so on. Since we have an algorithm targeted to find the minimal sum of products implementation, we also convert the problem of finding the product of sums for a function F to the problem of finding the sum of products for $\overline{F}$ and then using De Morgan's law (which has been shown to be hazard-preserving [71]) to obtain F. The inputs to the algorithm are: the level (indicating whether it is a sum of products or product of sums problem), the set of input transitions, the on-set of the function for which a hazard-free implementation is to be derived.

8.4.3 Algorithm

We will now describe all the steps outlined in the algorithm. **Step 1** is to derive the req, off and priv sets as described in the last section using algorithm Complex-make-sets if one is targeting the sum of products (say for the first level or any odd numbered level) otherwise, the algorithm Make-sets is used since we need to consider static hazards for product-of-sums (even levels). Dhf prime implicants are then derived in **Step 2**. We will call the set of dhf prime implicants as DHFPI. In **Step 3**, the covering problem is attempted. The required cubes that remain not covered by the DHFPIs are noted, we will call this set $rset_u$. Note that the only required cubes that remain not covered by the DHFPIs are due to conflicting dynamic hazard transitions during a sum of products minimization problem, i.e., a required cube of one transition becomes

a stray cube of one or more other dynamic transitions and vice versa. In the case of a product of sums implementations, the remaining required cubes could also be due to a static hazard requirement as well as due to conflicting dynamic transitions.

```
Derive_Multi(Level, Set T of input Transitions, On-set)
Step 1. if Level = odd
          Complex-make-sets(T, On-set)
        else
          Make-sets (T, On-set)
Step 2. Derive DHFPI set, req cubes rset
Step 3. Cover = MinCover(DHFPI, rset)
          if Cover return Cover
Step 4. For (rset_u = req cubes not covered by DHFPI)
          rset_u^i = set of conflicting cubes in rset_u
          Foreach rset_u^i
            on-set (on_u^i) = ∪ minterms in rset_u^i
            if (on_u^i already attempted) goto 6
            if Level = odd
             if (c_u^i = Derive_Multi(Level+1, T , off-set(on_u^i)))
               Cover = Cover ∪ ApplyDeMorgan(c_u^i)
             else goto 6
            else
             if (c_u^i = Derive_Multi(Level+1, T , off-set(on_u^i)))
               Cover = Cover ∪ c_u^i
             else goto 6
Step 5. If any DHFPI_j ∈ DHFPI is covered by a on_u^i
          DHFPI = DHFPI − DHFPI_j;
        Cover = Cover ∪ MinCover(DHFPI, rset − rset_u);
        return Cover;
Step 6. No solution. return NIL
end

Derive_CMOS_Multi(Set T of input Transitions, On-set)
  if (Cover = Derive_Multi(1, T, On-set))
     Partition each AND/OR level starting from output.
       Implement partitions as complex gates.
  else
     return NIL
  end
end
```

Figure 8.5: Algorithm

In **Step 4** we find the set of all cubes from $rset_u$ that conflict. The goal then is to derive an implementation for this on-set which is hazard-free for the original set of input transitions. If we are trying to solve a problem for the first level (AND/OR), it is clear that a sum-of-products implementation of a conflicting set of cubes A, B and C will not solve such a problem. Instead a hazard-free product of sums implementation of this on-set is attempted. Note that in the algorithm the method to derive the dual implementation is a recursive call. In order to

derive a products of sum implementation for an on-set G, the recursive call attempts to derive a sum of products implementation for the off-set $\overline{G}$ and then uses DeMorgan's law on G to obtain the product of sums implementation. A sum of products problem can be minimized ignoring static transitions since every sum of products function is implemented within a single SOP/SOP CMOS gate. A product of sums problem however, must take static transitions into account since otherwise hazards can result by the interaction of the complex gates. Therefore the product of sums problem (even levels) will use Make-sets in step 1, whereas the sum of products will use Complex-Make-sets in step 1. In **Step 5**, all DHFPIs at that level that are covered by the new cubes derived are removed. **Step 6** indicates cases where there are no multilevel solutions of this form.

The function $Derive\_CMOS\_Multi$ is the top level function that derives the cover (if there is one) using $Derive\_Multi$ and then partitions each AND/OR level starting from the output into a complex gate.

8.5 Results

To determine the benefits of the SOP/SOP form of complex gates, we have synthesized many controllers where logic hazards were present using this technique. For these examples, our method obtained a hazard-free combinational logic solution whereas the 3D tool [74] which uses *hfmin* [50] often had to add one or more state variables *just in order to prevent logic hazards.* For the comparison the Cadence schematic entry system and the LAS [10] layout syntheziser was used to generate layouts. The average critical path delays where then measured under same input slopes and output load.

Results for cases where occuring logic hazards could be resolved using only single SOP/SOP complex gates is shown in the *Single* part of Table 8.1. Results for cases where occuring logic hazards require a multilevel solution for the SOP/SOP complex gate form are given in the *Multilevel* part of Table 8.1. The area required for a hazard-free cover is shown in the *area* columns and the number of state variables that had to be added to get a solution in the two-level standard gate implementation is shown in the *statevar* column. The average delay from input event to output response under same input slopes and output load is shown in the *delay* columns.

In our experiments, the area required to get a two-level solution widely exceeds that of our SOP/SOP complex gate implementation. Due to fewer transistors and the ability to size the transistors very accurately to comply with the size of the output load, we are able to get performance gains of over 50% in many cases. The circuits produced by our method being combinational also have an advantage compared to the circuits produced by the 3D method

with regards to fundamental mode delay.

Name	Std.Gate statevar	Complex Gate area	Std.Gate area	Complex Gate delay	Std.Gate delay
Single					
comp_tr	1	136	513	0.35	0.64
tri_st	1	148	636	0.19	0.81
store_st	1	223	609	0.43	0.72
dual_si	0	96	152	0.20	0.30
for_lp	1	213	332	0.38	0.53
sim_mod	1	102	278	0.24	0.58
Multilevel					
bus_tr	2	271	1107	0.30	0.55
run_dp	1	360	1218	0.38	1.35
si_stack	1	177	312	0.20	0.32
sm_stat	2	187	1181	0.31	0.74
sm_dyn	2	222	1109	0.23	0.59
l_cov	2	173	824	0.21	0.58
comp_dp	2	395	1162	0.39	0.70

Table 8.1: Single and Multilevel Complex-gate Versus Standard Gate

8.6 Conclusions

In this section, we have presented a technique to synthesize a network of customized CMOS complex gates. We have presented a summary of properties and synthesis algorithms for a style of single customized CMOS gate and multilevel CMOS gate networks. The work was motivated by the fact that customized CMOS complex gates could provide flexibility of design, performance improvement and solutions to larger classes of problems in hazard-free asynchronous controller synthesis. During the analysis, we have also shown that a class of combinational functions which may not have a solution in the two-level AND/OR implementation form can have a solution in our CMOS gate method. Also, the multilevel method provided combinational logic solutions to cases where the two-level method could provide a solution only by adding state variables.

Our experiments has shown that using the SOP/SOP form of complex gates clearly have advantages when there are logic hazards present. Due to the reduced hazard constraints during synthesis, significant performance and area gains can be obtained.

Remaining problems that need to be solved are decomposition of the complex gate network into smaller elements and methods for transistor sizing under performance and area constraints. Decomposing complex gates is an important problem performance wise since high transistor stacks slow down the circuit. Automated transistor sizing of the whole transistor

network is important for area and performance tradeoffs. We are currently looking into these problems.

Chapter 9

Conclusions

This thesis has presented the fundamental concepts of asynchronous design. A high level synthesis framework called ACK incorporating some of these concepts has been presented.

9.1 Summary

The fundamental differences in the asynchronous and synchronous conceptual frameworks have been discussed. Advantages and disadvantages of the asynchronous and synchronous synthesis methodologies have been presented. We have presented several limitations of the synchronous concept that motivates use of the asynchronous design alternative. The most common of the many different asynchronous design styles using alternating circuit and environment delay models along with their communication styles have been presented to give the reader a basic understanding of the asynchronous concept.

A synthesis framework has been presented. This framework use a behavioral language Verilog−+ for high level specification of asynchronous designs. The Verilog−+ description is then translated into a powerful state based language, supporting important constructs for asynchronous design, called HOP which can also be used for design specification. The high level synthesis part of the framework was then described. By means of refinement and allocation the behavioral language is translated into a state machine controller using event signaling and datapath resources are allocated. The controller then goes through a partitioning procedure to resolve concurrency issues. Partitioning also allows synthesis of large controllers. The procedure presented solves the problem of dividing an incompletely specified machine into partitions that are allowed to share signals. A technique to convert the abstract event based state machines to burst mode machine with explicit knowledge of signal polarities is then presented. The concept of asynchronous state machine synthesis and different models is then explained. Using one of the methods presented, the burst mode machines are then synthesized to a Boolean function

representation. A new way of technology mapping those functions to multilevel custom complex gate networks that reduce some of the synthesis constraints is then presented.

9.2 Future Work

Many of the approaches used by the synthesis framework presented in this thesis make use of simple methods and algorithms. However, this first version of the synthesis framework is dedicated to present the basic methodology for a new conceptual synthesis methodology. Optimizations of these methods are possible but have been left for future work. Some of the possible extensions to this work are presented next.

Design Modeling A graphical based approach of the structural as well as module specification would simplify the design process as well as help the designer to more clearly see constructs suitable for partitioning or optimization. It would also remove the limitations of the naming convention when deciding which resources and wires are shared.

Language Extensions Moving towards algorithmic descriptions of the design behavior which is independent of the underlying synthesis methodology would open up the possibility of using the same specification language for synchronous as well as asynchronous design.

Refinement The refinement process used is somewhat naive in that it is only based on the syntactic structure of individual statement constructs. A method taking advantage of high level synthesis optimizations such as those presented in [17] would be desirable. Experience has also shown that careful flow analysis of the refined controller can result in optimizations at the handshaking level.

Allocation Support for user defined area/performance tradeoff should be added. This will require the ability to share resources within the same module. Methods for this are standard and can be found in [2, 7].

Partitioning Heuristics or methods for exact solutions to partitioning should be developed. Algorithms for partitioning under certain criteria such as minimal wire sharing between partitions, or resource sharing are of interest. Minimal resource sharing would mean each controller can be placed and routed in a very near vicinity of its corresponding datapath. More efficient control structures for signal sharing are also possible and currently under development.

Burst Mode Synthesis Finite state machine synthesis in general has exponential worst case complexity. Although heuristics are already partly used, more good heuristics need to be developed. At least a polynomial solution should exist and the decision to use it or not be left to the designer. A near optimal solution is better than no solution at all.

Complex Gate Implementation The method for custom complex gate implementation should be extended with a decomposition method for average case performance based on the method presented in [5]. A method for user defined area/performance tradeoff based on transistor sizing should also be incorporated.

Completion Detection Using matching delays for completion detection requires careful timing to ensure a correct circuit behavior. This step should be automated by using a timing analysis tool. Other completion detection methods such as current sensing should also be evaluated and, if shown to be efficient, incorporated into the synthesis methodology.

Bibliography

[1] Aghdasi F. Synthesis of Asynchronous Sequential Machines for VLSI Applications. In *Proceedings of the 1991 International Conference on Concurrent Engineering and Electronic Design Automation (CEEDA)*, March 1991, pp. 55-59.

[2] Akella, V. *An Integrated Framework for the Automatic Synthesis of Efficient Self-Timed Circuits from Behavioral Specifications.* PhD thesis, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, 1992.

[3] Akella V, Gopalakrishnan G. SHILPA: A High-Level Synthesis System for Self-Timed Circuits. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, 1992, pp. 587-591.

[4] Beerel P. A, Meng T. H. Automatic Gate-level Synthesis of Speed-independent Circuits. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, 1992, pp. 581-586.

[5] Beerel P. A, Yun K. Y, Chou W. Optimizing Average-case Delay in Technology Mapping of Burst-mode Circuits. In *Proceedings of the International Conference on Asynchronous Design*, 1996, pp. 244-260.

[6] Berkel v. K, Bink A. Single-track Handshake Signaling with Application to Micropipelines and Handshake Circuits. In *Proceedings of the International Conference on Asynchronous Design*, 1996, pp. 122-133.

[7] Brunvand, E. *Translating Concurrent Communicating Programs into Asynchronous Circuits.* PhD thesis, Carnegie Mellon University, 1991.

[8] Bredeson J. G. Synthesis of Multiple-input Change Hazard-free Combinational Switching Circuits Without Feedback. In *International Journal of Electronics (GB)*, Vol. 39, No. 6, December 1975, pp. 615-624.

[9] Bredeson J. G, Hulina P. T. Elimination of Static and Dynamic Hazards for Multiple Input Changes in Combinational Switching Circuits. In *Information and Control*, Vol. 20, No. 2, March 1972, pp. 114-224.

[10] Custom Layout/Virtuoso LAS User's Manual. Cadence Design Systems Inc., 1992.

[11] Chandrakasan A. P, Brodersen R. W. Minimizing Power Consumption in Digital CMOS Circuits. In *Proceedings of the IEEE*, Vol. 83, No. 4, April 1995, pp. 498-523.

[12] Chu T. A. Synthesis of Self-timed VLSI Circuits from Graph Theoretic Specifications. PhD Thesis, Department of EECS, Massachusetts Institute of Technology, September 1987.

[13] Chuang H. Y, Das S. Synthesis of Multiple-input Change Asynchronous Machines Using Controlled Excitation and Flip-flops. In *IEEE Transactions on Computers*, Vol. 22, No. 12, December 1973, pp. 1103 1109.

[14] Davis, A., Coates, B., and Stevens, K. The Post Office Experience: Designing a Large Asynchronous Chip. In *Proceedings of the 26th Annual Hawaiian International Conference on System Sciences, Volume 1* (Jan. 1993), T. Mudge, V. Milutinovic, and L. Hunter, Eds., pp. 409–418.

[15] Dean, M. E. STRiP: A Self-timed RISC Processor Architecture. Ph.D. Thesis, Stanford University, 1992.

[16] Dean, M. E., Dill, D. L., Horowitz, M. Self-timed Logic Using Current-sensing Completion Detection (CSCD). In *Proceedings of the International Conference on Computer Design (ICCD)*, IEEE Computer Society Press, October 1991, pp. 187-191.

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

[18] Dill D. L. Trace Theory for Automatic Hierarchical Verification of Speed-independent Cirucits. MIT Press, An ACM Distinguished Dissertation, 1989.

[19] Dolotta T. A, McCluskey E. J. Jr. Encoding of incompletely specified Boolean matrices. In Proceedings of Western Joint Computer Conference, Vol. 18, pp. 231-238, 1960.

[20] Ebergen, J. C. A Formal Approach to Designing Delay-insensitive Circuits. *Distributed Computing*, Vol. 5, No. 3, 1991, pp. 107-119.

[21] Ebergen J. C. Translating Programs into Delay-insensitive Ciruicts. Centrum for Wiskunde en Informatica, CWI Tract 56, Amsterdam, 1989.

[22] Fang T. P, Molnar C. E. Synthesis of Reliable Speed-Independent Circuit Modules: II. Circuit and Delay Conditions to Ensure Operation Free of Problems from Races and Hazards. Computer Systems Laboratory, Washington University Tech. Memorandum 298, 1983.

[23] Frackoviak J. Metoden der Analyse und Synthese von Hasardarmen Schaltnetzen mit Minimalen Kosten I. In *Elektronische Informationsverarbeitung und Kybernetik*, Vol. 10, No. 2/3, 1974, pp. 149-187.

[24] Friedman A. D, Menon P.R. Synthesis of Asynchronous Sequential Circuits With Multiple-input Changes In *IEEE Transactions on Computers*, Vol. 17, No. 6, June 1968, pp. 559-566.

[25] Furber S. B, Day P, Garside J. D, Paver N. C, Woods J. V. AMULET1: A Micropipelined ARM. In *Proceedings of CompCon'94*, IEEE Computer Society Press, CompCon'94, San Francisco, March 1994

[26] Gopalakrishnan G. C, Kudva P. N, Brunvand E. L. Peephole Optimization of Asynchronous Macromodule Networks. In *Proceedings of the International Conference on Computer Design (ICCD)*, 1994, pp. 442-446.

[27] Hayes, A. B. Stored state asynchronous sequential circuits. In *IEEE Transactions on Computers*, Vol. 30, No. 8, August 1981, pp.596-600.

[28] Hedberg A, Jacobson H. M, Einarsson M, Jennings G. Imposing a Unified Design Methodology on Independent Rapid Prototyping Tools. In *Proceedings of the Sixth IEEE International Workshop on Rapid Systems Prototyping (RSP95)*, June 1995, pp. 217-222.

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

[30] Huffman D. A. The Synthesis of Sequential Switching Circuits. *J. Franklin Institute*, March/April 1954.

[31] Jennings G, Jennings E. A Discrete Syntax for Level-Sensitive Latched Circuits Having n Clocks and m Phases. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 15, No. 1, January 1996, pp. 111-126.

[32] Kessels, J., van Berkel, K., Burgess, R., Roncken, M., and Schalij, F. An error decoder for the compact disc player as an example of VLSI programming. Tech. rep., Philips Research Laboratories, Eindhoven, The Netherlands, 1993.

[33] Kudva P. N, Jacobson H. M, Gopalakrishnan G. C. Synthesis of Hazard-free Customized CMOS Complex-Gate Networks Under Multiple-Input Changes. In *Proceedings of the 33rd Design Automation Conference (DAC)*, 1996, pp. 77-82.

[34] Kudva P. N, Gopalakrishnan G. C, Jacobson H. M. A Technique for Synthesizing Distributed Burst-mode Circuits. In *Proceedings of the 33rd Design Automation Conference (DAC)*, 1996, pp. 67-70.

[35] Kung D. Hazard-non-increasing Gate Level Optimization Algorithms. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, 1992.

[36] Lin B, Devadas S. Synthesis of Hazard-free Multi-level Implementations Under Multiple-input Changes From Binary Decision Diagrams. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, 1994.

[37] Mago G. Realization Methods for Asynchronous Sequential Circuits. In *IEEE Transactions on Computers*, Vol. 20, No. 3, March 1971, pp. 290-297.

[38] Martin, A. J. Programming in VLSI: From communicating processes to delay-insensitive circuits. In UT Year of Programming Institute on Concurrent Programming (1989), e. C.A.R. Hoare, Ed. MA: Addison-Wesley, 1989, pp. 1-64.

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

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

[41] Martin A. J. A Synthesis Method for Self-timed VLSI Circuits In *Proceedings of the International Conference on Computer Design (ICCD)*, October 1987, pp. 224-229.

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

[43] Mead, C. A., and Conway, L. A. Introduction to VLSI Systems. Addison-Wesley, 1980. ISBN 0-201-04358-0

[44] Molnar C. E, Fang T. P, Rosenberger F. U. Synthesis of Delay-Insensitive Modules. In *Proceedings of the 1985 Chappel Hill Conference on Advanced Research in VLSI*, 1985, pp. 67-86.

[45] Moon C. W, Brayton R. K. Elimination of Dynamic Hazards By Factoring. In *Proceedings of the 30th Design Automation Conference (DAC)*, 1993, pp. 7-13.

[46] Murata, T. Petri Nets: Properties, Analysis and Applications. In *Proceedings of the IEEE*, Vol. 77, No. 4, 1989, pp. 541-580.

[47] Myers C. J, Meng T. H. Synthesis of Timed Asynchronous Circuits. In *IEEE Transactions on VLSI Systems*, Vol. 1, No. 2, June 1993, pp. 106-119.

[48] Nielsen L.S, Niessen C, Sparsø J, Berkel van K. Low-Power Operation Using Self-Timed Circuits and Adaptive Scaling of the Supply Voltage. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 2, No. 4, December 1994, pp. 391-397.

[49] Nowick, S. M. Automatic synthesis of burst-mode asynchronous controllers. Tech. rep., Ph.D Thesis, Computer Systems Laboratory, Stanford University, 1993.

[50] Nowick, S. M, and Dill, D. L. Exact two-level minimization of hazard-free logic with multiple-input changes. *IEEE Transactions on Computer-Aided Design*, Vol. 14, No. 8, August 1995, pp. 986-997.

[51] Nowick, S. M, Coates B. UCLOCK: Automated Design of High-Performance Unclocked State Machines. In *Proceedings of the International Conference on Computer Design (ICCD)*, 1994, pp. 434-441.

[52] Panda P. R, Dutt N. 1995 High Level Synthesis Design Repository. Tech. Report 95-04, University of California, Irvine, February 1995.

[53] Pedron C, Stauffer A. Analysis and Synthesis of Combinatinal Pass Transistor Circuits. In *IEEE Transactions on CAD/CAS*, Vol. 6, No. 5, 1988, pp. 727-750.

[54] Petrick S. R. A direct determination of the irredundant forms of a Boolean function from the set of prime implicants. Air Force Cambridge research Center, Cambridge, Mass., Tech. Rep. AFCRC-TR-56-110, 1956.

[55] Puri R, Gu J. A Modular Partitioning Approach for Asynchronous Circuit Synthesis. In *Proceedings of the ACM/IEEE Design Automation Conference*, 1994, pp. 63-69.

[56] Radhakrishnan D, Whitaker S, Maki G. Formal Design Procedures for Pass Transistor Switching Circuits. In *IEEE Journal of Solid State Circuits*, Vol. 20, No. 2, 1985, pp. 531-536.

[57] Rey C. A, Vaucher J. Self-synchronized Asynchronous Sequential Machines. In *IEEE Transaction on Computers*, Vol. 23, No. 12, December 1974, pp. 1306-1311.

[58] Rosenberger F. U, Molnar C. E, Chaney T. J, Fang T. P. Q-Modules: Internally Clocked Delay-Insensitive Modules. In IEEE Transactions on Computers, Vol. 37, No. 9, 1988, pp. 1005-1018.

[59] Rudell R. Logic Synthesis for VLSI Design. Ph.D. dissertation, Department of Electrical Eng. and Computer Sci., University of California, Berkeley, 1989.

[60] Rudell R, Sangiovanni-Vincentelli A. Multiple-valued Minimization for PLA Optimization. *IEEE Trans. Computer-Aided Design*, Vol. 6, No. 5, pp. 727-750, Sept. 1987.

[61] Sasi S, Radhakrishnan D. Hazards in CMOS Circuits. In *International Journal on Electronics*, Vol. 68, No. 6, 1990, pp. 976-990

[62] Sparsø J, Staunstrup J. Delay-insensitive multi-ring structures. In *INTEGRATION, the VLSI journal*, 15, 1993, pp.313-340.

[63] Siegel P. S. Automatic Technology Mapping for Asynchronous Designs. Tech. Report, PhD Thesis, Computer Systems Laboratory, Stanford University, March 1995.

[64] Sproull R. F, Sutherland I. E. Asynchronous Systems. Sutherland, Sproull and Associates, Palo Alto, 1986, Vol. I: Introduction, Vol. II: Logical Effort and Asynchronous Modules, Vol. III: Case Studies.

[65] Stevens K. Tech. Report, PhD Thesis, Computer Systems Department, University of Calgary, 1994.

[66] Sutherland, I. Micropipelines. *Communications of the ACM*, Vol. 32, No. 6, June 1989, *The 1988 ACM Turing Award Lecture.*

[67] Sutherland I. E, Molnar C. E, Sproull R. F, Mudge J. C. The Trimosbus. CalTech Conference on VLSI, January, 1979.

[68] Tapia M. A. Synthesis of Asynchronous Sequential Systems Using Boolean Caculus. In *14th Asilomar Conference on Circuits, Systems and Computers*, November 1980, pp. 205-209.

[69] Theobald M, Nowick S. M, Wu T. Espresso-HF: A Heuristic Hazard-Free Minimizer for Two-Level Logic. In *Proceedings of the 33rd Design Automation Conference*, 1996, pp. 71-76.

[70] Tracey J. H. Internal State Assignment for Asynchronous Sequential Machines. IEEE Transactions on Electronic Computers, EC-15(4), August 1966.

[71] Unger, S. H. *Asynchronous Sequential Switching Circuits.* Wiley-Interscience, 1969. ISBN 0-89874-565-9

[72] VIS Group, the. VIS: A system for Verification and Synthesis. In *Proceedings of the Conference on Computer Aided Verification*, New Brunswick, NJ, July 1996.

[73] Williams T. E. Performance of Iterative Computation in Self-Timed Rings. In *Journal of VLSI Signal Processing*, 7, 1994, pp. 17-31.

[74] Yun, K. Y. *Synthesis of asynchronous controllers for heterogeneous systems.* PhD thesis, Stanford University, Aug. 1994.