THE FLOW-TABLE TECHNIQUE FOR THE SYNTHESIS OF ASYNCHRONOUS SEQUENTIAL CIRCUITS

by

Avaneendra Gupta

Memorandum No. UCB/ERL M92/83

11 August 1992

COVER PAGE

THE FLOW-TABLE TECHNIQUE FOR THE SYNTHESIS OF ASYNCHRONOUS SEQUENTIAL CIRCUITS

by

Avaneendra Gupta

Memorandum No. UCB/ERL M92/83

11 August 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

THE FLOW-TABLE TECHNIQUE FOR THE SYNTHESIS OF ASYNCHRONOUS SEQUENTIAL CIRCUITS

by

Avaneendra Gupta

Memorandum No. UCB/ERL M92/83

11 August 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

THE FLOW-TABLE TECHNIQUE FOR THE SYNTHESIS OF ASYNCHRONOUS SEQUENTIAL CIRCUITS

Avaneendra Gupta

Department Of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

Abstract

Synthesis of asynchronous sequential circuits has been studied extensively from a theoretical perspective. However, due to the inherent difficulties encountered in the various stages of the synthesis procedure, many of the techniques have not been automated for use in practical applications. In this report, the detailed stages involved in the synthesis of asynchronous circuits using the traditional flow-table based technique are described. The synthesis procedure derives the flow-table representation of the circuit from a verbal description or a signal transition graph specification. State reduction is then invoked, identifying and merging equivalent states. The reduced flow-table is subsequently subjected to state assignment techniques. The excitation functions derived from the flow-table are modified to guarantee the absence of any combinational hazards. Under the assumptions of bounded stray delays, fundamental mode of operation, and the restriction that successive input changes can be between adjacent input combinations only, the synthesis procedure yields an optimized hazard-free realization of the asynchronous circuit.

A. Richard Newton

Professor A. Richard Newton
Research Advisor

Acknowledgements

Being amidst the wealth of opportunity and academic enrichment at UC Berkeley, I have been most fortunate to have received the astute and invaluable guidance from my professors and colleagues who have inspired and shaped my research on the synthesis of asynchronous circuits. I sincerely thank my esteemed professor and research advisor, Prof. A. Richard Newton, whose consistent support and inspiration helped me pursue my study with purpose and determination, and without his guidance, this work would not have been possible.

I also wish to thank Prof. Robert Brayton and Prof. Alberto Sangiovanni-Vincentelli for their stimulating lectures which helped me build a strong foundation in computer-aided design. My sincere thanks to Prof. Robert Brodersen for consenting to be the reader for this report and for his constructive criticism which helped make this report complete.

My heartfelt gratitude to my friend and mentor, Bill Lin, who has been the guiding-light throughout this research as well as other class projects which I undertook at Berkeley. Besides being an unending source of inspiration, guidance and stimulating discussions, he has been a very helpful and patient mentor and is a wonderful friend.

Last, but not the least, my fond and heartfelt thanks to my fiancee, Neha and to both our parents. Without their enthusiasm and encouragement, this work would never have seen the light of day.

This research was sponsored in part by the Defense Advanced Research Projects Agency under contract N00039-88-C-0292 and JFBI90-073, and Digital Equipment Corporation. I would like to extend my grateful thanks to them for their invaluable support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The need for asynchronous systems

The operation of clocked synchronous networks is synchronized with a central system clock whose time period is greater than the delay of the *critical path*. Since the critical path is the slowest combinational path that would be used at least once during the circuit operation, the system-level latency is dictated by the worst-case stray delays. In such circuits, a state change occurs only in response to a clock pulse and all storage elements which must change state do so simultaneously at the clock pulse. Further, in response to an input change, the next state does not change immediately but only at the next clock pulse. These characteristics of clocked synchronous circuits eliminate any further timing considerations and, provided that the clock period is carefully chosen, eliminate problems caused by hazards. They also simplify the process of circuit synthesis from its initial description of the network as a state table or a state diagram [1].

However, the presence of a central fixed-period clock has its own inherent limitations and problems. First, the determination of the fixed clock period necessitates an exact evaluation of the worst case system delay which is defined by the delay of the critical path. This evaluation may not always be feasible due to the absence of exact values for all types of delays which occur in the circuit. If estimated values for the stray delays are used to determine the clock period, a very pessimistic estimate of clock frequency must be used to guarantee the proper operation of the circuit under all possible input combinations.

Added to this problem is the problem of clock-skew. In the case of large circuits, line delays could add up to be significant enough so that the time taken by the signals to travel down the wires could be significantly large. In such a situation, the clock pulse could arrive at different memory elements at different times, causing possible erroneous operation.

Perhaps the most important practical limitation of a central fixed-period clock to synchronize the circuit operation is in the design of networks with inputs which could change at any time and cannot be synchronized by a central clock. For such circuits, it is not possible to determine a fixed period for the system clock since the rate of arrival of inputs is both variable and unknown.

Asynchronous systems, which are designed to operate correctly in the absence of synchronizing clock pulses, offer an attractive alternative to the limitations and problems posed by synchronous designs. They do not have their computational rates constrained by a central fixed-period clock. Thus the system-level latency is dominated by the average-delay only and not by the worst-case delay. For example, an asynchronous or a self-timed adder can signal when the result on its outputs is valid and the absence of any synchronization makes asynchronous design the only choice in applications where the rate of arrival of inputs is variable. In an asynchronous design, the speed of operation of the system is governed by the rate of arrival of external inputs into the system and the action dependencies within the system and is independent of the element (gates) and line (wire) delays.

1.2 Limitations of asynchronous designs

However, attractive as it may seem, the synthesis of asynchronous systems poses many timing problems. A correctly designed synchronous circuit waits long enough for all the memory elements to reach a stable condition before allowing the next input change. This design procedure eliminates errors due to circuit timing in the synthesis process. However, in the synthesis of asynchronous circuits, the absence of a synchronizing clock and the presence of unequal delays through the various paths in the circuit necessitate special design techniques to eliminate the inherent timing problems.

Therefore, in order to simplify the design and synthesis of asynchronous circuits, a few assumptions must be made about the external environment as well as the operation of the circuit. The most important of these is the assumption that the circuit operates in *fundamental mode*, ie., after a change in external inputs, the next input change occurs only after the circuit reaches a stable (unchanging) condition. This means that the time interval between two successive input changes is sufficiently large so as to allow the circuit to reach its stable state after the first input change. This assumption, along with the others made on the external and internal environments to guarantee the proper operation of the circuit, is described in detail in subsequent chapters.

1.3 Survey of asynchronous design methodologies

The theoretical aspects of the design of asynchronous digital systems have been studied quite extensively in literature, although the practical application of these techniques has only recently gained momentum. This recent revival of interest in asynchronous design is mainly due to the development of mathematical theory which has enabled a deeper understanding of asynchronous behavior [2, 10, 20].

The oldest and the traditional approach to asynchronous design is based on the representation of the initial circuit in the form of a *flow-table* (or state-table). The flow-table is subsequently synthesized to yield excitation equations for the non-input signals (internal state variables and output signals) [1, 2, 7, 8, 9]. The problem of hazards has also been studied in detail and the necessary conditions required to guarantee hazard-free operation of the synthesized circuit have been formulated. The major objective of this work was to develop an automated system for the synthesis of hazard-free asynchronous circuits using the flow-table synthesis technique as described in the above references and to compare it with some of the more recent approaches, as described below.

Chu [10] introduced the concept of Signal Transition Graphs (STGs) as a restricted class of interpreted Petri nets. STG's have recently been used as an efficient specification for asynchronous circuits. In [10] a procedure to synthesize asynchronous circuits guaranteed to function correctly under no assumption of gate-output delays is also presented. This synthesis procedure first modifies the initial STG description of the circuit, if necessary to guarantee that the STG satisfies certain properties of *liveness* and *persistency*. It then converts the modified STG to its equivalent finite automaton called the *state graph*, which is then transformed, if necessary, to ensure that every state in the graph is assigned a unique coding. Excitation functions for the next-state and output signals are then derived directly from the state graph.

Lavagno [14], under the more realistic assumption of unbounded gate-input delays describes a synthesis procedure which guarantees that most of the synthesized circuits are hazard-free. He also outlines an algorithm to design delay tests which could be employed to check the synthesized circuit for the presence of hazards. Lavagno's approach, however, assumes that the given STG specification of the asynchronous circuit satisfies the *Unique State Coding* (USC) property [14].

Vanbekbergen [17] presents a technique to satisfy the STG requirements of the USC property and persistency by introducing the concept of *Generalized Lock Classes* and *lock graphs*. These transformations are performed at the STG level before its conversion to the equivalent state graph.

In another follow-up to the above paper [18], he suggests a technique to satisfy the USC property at the state graph level by the addition of new variables (internal signals) to the initial STG. The internal signals are added in a way that the logic of one particular non-input signal is minimized, while maintaining the original concurrency in the STG.

Ebergen [19] has developed a new synthesis technique for asynchronous circuits, both *speed independent* and *delay-insensitive* circuits. This technique is based on a mathematical formalism called the *trace theory*, and uses the model of a *directed trace structure* to specify the initial circuit.

Ilana, et al [23, 24] describe a technique for realizing speed-independent combinational logic, and present the design of a simple asynchronous RISC processor. Other successful designs of asynchronous processors have been reported by Martin, et al [21, 22], Ilana, et al [23], and Meng, et al [13].

1.4 Organization of the report

In this report, a synthesis procedure is presented which guarantees that the synthesized circuit is free of any critical races as well as any static and dynamic hazards under the assumption of single-input changes. The input to this procedure is the flow-table representation of the asynchronous circuit. The flow-table could be derived from the circuit's verbal description or a signal transition graph specification. The table is then reduced by identifying and merging equivalent states, and the reduced flow-table is subjected to state assignment techniques. The derivation of the excitation Y-matrix is dependent on the state assignment technique employed. Two-level logic equations for the internal state variables and the output signals are generated from the Y-matrix. The excitation functions are modified, if necessary, to eliminate any combinational hazards. The final logic equations synthesized guarantee that the circuit will operate correctly under the assumptions of bounded stray delays and fundamental mode of operation, and with the restriction that successive input changes are between adjacent input combinations only.

In Chapter 2, preliminaries regarding sequential machines and the finite-state model are presented and the generic hardware models of synchronous and asynchronous networks are described. An overview of the complete synthesis procedure for asynchronous circuits is presented in Chapter 3. This procedure is then described in detail in the following chapters. In Chapter 4, the design example of the 4-phase handshake protocol is introduced, and the process of derivation of the initial flow-table from a verbal description of the circuit is described.

Subsequently, the algorithms employed for the reduction of the flow-table are described in Chapter 5. In Chapter 6, different state assignment techniques for asynchronous circuits are summarized and the unicode state assignment procedure is described in detail. The final stage of the synthesis procedure which involves the realization of a hazard-free combinational logic from the reduced flow-table is presented in Chapter 7.

A summary of the signal transition graph based synthesis procedure for asynchronous circuits is presented in Chapter 8. This chapter is aimed at understanding the pros and cons of the flow-table and the STG synthesis techniques which are presented in Chapter 9. Finally, the implementation details of the synthesis package async which has been implemented are described in Appendix A.

Chapter 2

The Finite-State Model

2.1 Sequential machines

In a *combinational circuit*, the outputs are *combinational functions* and depend only on the present inputs to the circuit. No information or data is stored in the circuit. In contrast, a function whose value depends not only on the present external inputs but also on the previous inputs is called a *sequential function*. *Sequential Circuits* are realizations of sequential functions and the mathematical model used to describe a sequential function is called a *sequential machine* [2].

2.1.1 Finite-state machines

In a sequential machine, the output at a particular time t depends on the external inputs applied to the circuit at that time, as well as the inputs applied at previous time points. This dependency on inputs at previous time points requires that the information regarding the previous inputs be stored in the machine in some form. The *input-history* of a circuit at a particular time t is the sequence of inputs applied to the machine at previous time points.

However, any particular sequential machine can have infinite types of previous histories. This would require infinite memory capacity for storing them. However, practically speaking, it is not possible to implement machines with infinite storage capacities. Thus a restriction on the types of machines which could be studied and implemented is inevitable. Implementable machines should therefore have their behavior affected by past histories in only a finite number of ways.

The different unique internal histories of a machine are each represented by an *internal state*. Each state thus corresponds to a particular history of past inputs. Then, the fact that the outputs

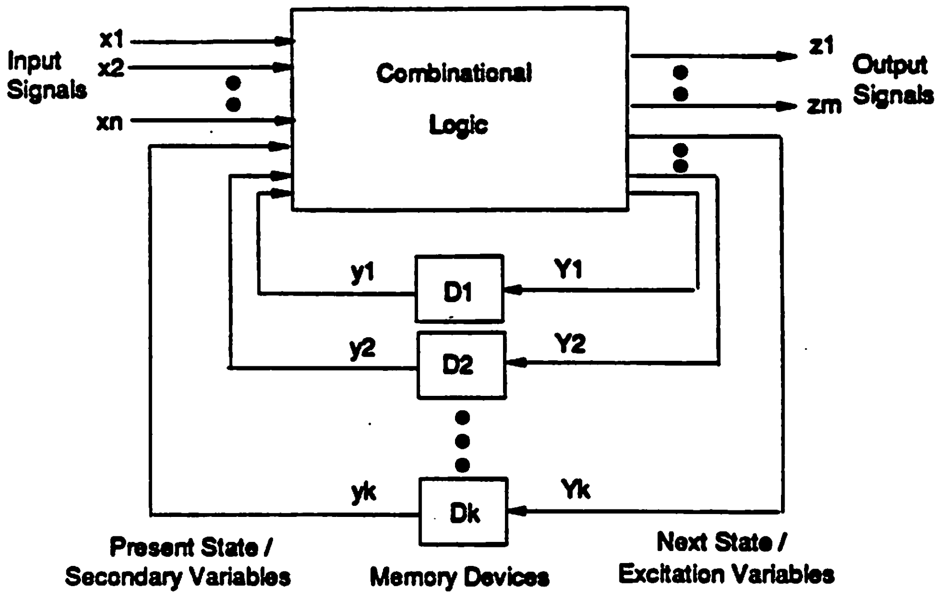

Figure 2.1: *Schematic representation of a Sequential Machine*

depend on the present external inputs as well as the past inputs can be expressed as a function of the present inputs and the state of the machine. Since states are stored using memory devices, only a finite number of which can be used in practice, analysis of sequential machines is restricted to only those machines which have a finite number of internal states.

2.1.2 The basic model of a sequential machine

A sequential machine can be schematically represented as shown in Figure 2.1. The input to the machine is from a finite set of input symbols. This set is called the input alphabet I. The signals which constitute the input alphabet are called *input variables* and may take values from the set $(0, 1)$. An *input configuration I* is defined as an ordered tuple of 0's and 1's, where each member of the tuple represents the value of the particular input variable in I. Similarly, the output is produced from a finite set of output symbols called the *output alphabet O*. The signals constituting the output alphabet are represented by binary-valued *output variables*. The ordered tuple of 0's and 1's identifying the values of the variables in a particular output symbol is called the *output configuration.*

In Figure 2.1, the set of variables $(x_1, x_2, ..., x_n)$ represents the set of input variables while the set $(z_1, z_2, ..., z_m)$ represents the set of output variables.

The signal value at the output of each memory element represents the *state (secondary) variable.* The set $(y_1, y_2, ..., y_k)$ in Figure 2.1 represents the set of state variables and the k-tuple of 0's and 1's defines the *present internal state* of the machine at any time t. The signal value at the input to each memory element (which is an output from the combinational logic) is identical to the respective value of the state variable at the next time point $t + 1$ and is therefore termed the *excitation (next-state) variable.* The set $(Y_1, Y_2, ..., Y_k)$ represents the set of excitation variables and the k-tuple of values for the excitation variables is termed as the *next state* of the machine.

As is evident from the schematic diagram, the output of the machine produced by the combinational logic is a function of the present inputs as well as the present state of the machine. Such machines where the output is a function of both the present inputs and the internal state are called *Mealy machines.* Machines in which the output is a function of the present state only and independent of the external inputs are called *Moore machines.* It is possible to convert any Mealy machine into its equivalent Moore machine and vice-versa, so that both the machines produce the same output sequence for any input sequence [2].

The next state of the machine also depends on the present values of the external inputs as well as the present internal state. The internal behavior of the machine is restricted to a deterministic behavior, in the sense that for every possible pair of present state and input combination there exists only one possible transition to a new next state. Such behavior is represented by the *state transition function,* which maps every $(presentstate, externalinput)$ condition to a next state. When a machine is in a particular state at any time t and an external input is applied to it, the machine temporarily goes into an *unstable state,* in which the values of the excitation variables $(Y's)$ and the state variables $(y's)$ are unequal. The machine is said to have reached a *stable state* when $Y_i = y_i$ for all i, ie., the values of signals at the inputs and outputs of each memory element are equal.

Summarizing the above in mathematical terms, a sequential machine M can be represented by a quin-tuple $M = (I, O, S, \delta, \lambda)$, where :

I = finite set of input variables

O = finite set of output variables

S = finite set of states

$\delta : I \times S \rightarrow S$ is the state transition function

$\lambda : I \times S \rightarrow O$ is the output function for Mealy machines, and

$\lambda : S \rightarrow O$ is the output function for Moore machines

2.1.3 Flow-table representation of a sequential machine

One of the methods of representing the relationship between the input, present state, output and next state variables of a sequential machine is the *State table* or the *Flow-Table*. The *behavior* of a machine on the application of a particular input sequence is defined as the succession of states through which the machine passes along with the output sequence produced. Every machine has an *initial state* which is the internal state of the machine before the application of any input sequence. The flow-table along with the initial state uniquely specify the behavior of the machine. The state which the machine reaches after the application of the input sequence is called the *final state* of the machine.

A flow-table is a two-dimensional array in which the rows correspond to internal states and the columns to input configurations. The entry defined by the element (i, j) corresponding to the state S_i and input configuration I_j represents the next state and the output produced if the machine is in present state S_i and gets an external input I_j. In practice, it often occurs that some combinations of states and input conditions are not possible. In other situations, although the state transitions are defined, the output values produced are not critical, and hence left unspecified. This class of sequential machines in which, for certain combinations of present state and input values, either the next state, or the output, or both are unspecified are termed as *incompletely-specified sequential machines.*

Sequential machines can be of two types : synchronous or asynchronous. This distinction is based primarily on the synchronous or asynchronous nature of the computational and communication steps of the machine [15]. A *synchronous machine* is one which is synchronous with respect to both its computational and communication steps. On the contrary, an *asynchronous machine* is synchronous with respect to communication, but asynchronous with respect to its computational steps. This distinction is explained in detail in the following sections of this chapter.

2.2 The synchronous machine model

The operation of a synchronous machine is synchronized by a central fixed-period clock. For correct operation of the machine, the time period of the clock must be greater than the critical

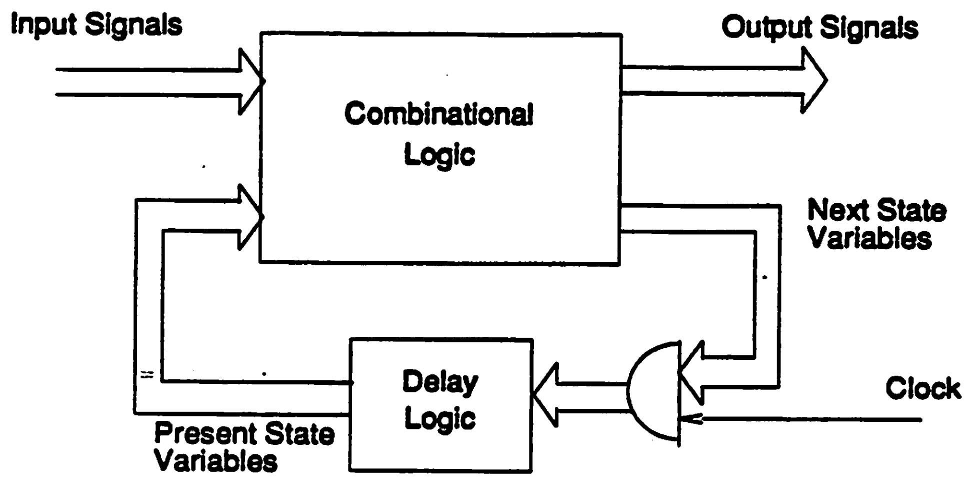

Figure 2.2: *Hardware Model of a synchronous machine*

path of the circuit. In the schematic form, a general synchronous circuit can be represented as shown in Figure 2.2. The clock pulses are used to trigger the next state of the machine after the values of the excitation variables have stabilized at the output of the combinational logic.

In a synchronous machine, change of internal state occurs only in response to a clock pulse. Thus due to a change in external inputs, the machine enters its next state only on receiving the subsequent clock pulse. The external inputs to the machine must therefore arrive in synchronization with the clock pulses. Similarly, the outputs are sampled only at the clock pulses.

2.3 Asynchronous machines

The schematic diagram of a general sequential circuit in Figure 2.1 also represents the model of an asynchronous network. The operation of the circuit is not synchronized by a central clock. Therefore, a change in inputs can directly lead to a transition to the next state without waiting for any synchronization. However, due to unequal delays in the different paths of the combinational logic, the change of internal state variables may not be simultaneous. This results in race conditions and related timing problems.

As described in the previous chapter, the design of asynchronous circuits is complicated by the fact that it is difficult to guarantee the simultaneous change of multiple signals in a single transition. Moreover, the external inputs to the circuit are not synchronized and can change at any instant. Due to these degrees of freedom, it could be possible that the input configuration changes twice in succession even before the circuit reaches stability as a result of the first change. In such a situation, the operation of the circuit would be unpredictable and erroneous. To eliminate this timing problem, certain restrictions on the external and internal environments have to be imposed.

2.3.1 Fundamental mode of operation

Under the assumption of *fundamental mode*, the external inputs to the circuit are constrained to change only after the circuit is in the stable mode, ie., the values of the excitation variables at the inputs to the delay elements and the corresponding values of the secondary variables at the delay element outputs are equal.

In addition to the above restriction, another input constraint has to be imposed. Due to the presence of stray delays and the fact that inputs can change at any time, it is necessary to prohibit the simultaneous change of two or more input signals. The restriction that only one input variable may change at any time, along with the assumption of fundamental mode is termed as the *normal fundamental mode* of operation.

For an asynchronous circuit, the external input variables along with the secondary variables define the *total state* of the circuit. Since each stable state of the circuit essentially represents a total state, the circuit can go from one stable state to another without any change in its secondary variables. This means that the two stable states are distinguished only by the states of their external inputs. This is in contrast to a synchronous circuit in which the total state is represented only by the states of the internal variables. This unique property of asynchronous circuits is employed in the process of state minimization and is described in detail in Chapter 5.

2.4 Pulse mode circuits

In fundamental mode asynchronous circuits, all input signals are assumed to be level signals. In contrast, a different class of circuits called *pulse-mode circuits* is based on inputs being pulses rather than level signals. The design of these circuits requires certain restrictions on the duration of pulses in order to guarantee deterministic circuit operation. Under the restrictions, it has been shown that the operation of pulse-mode circuits reduces to that of clocked synchronous circuits [1, 2, 9].

Chapter 3

Flow-Table Synthesis : An overview

In this chapter, the various stages involved in the synthesis of fundamental mode asynchronous circuits are outlined. The synthesis procedure begins with the initial description of the circuit in terms of a flow-table. In case the circuit is specified in terms of its graph-theoretic description as a Signal Transition Graph (STG) [10], the STG is first transformed into its equivalent flow-table representation.

3.1 Problem specification

The input to the synthesis procedure is a flow-table description or a signal transition graph representation of the asynchronous circuit. The process of deriving the flow-table from a verbal description of the circuit is very unsystematic and hence not included in the synthesis procedure. However, the process of transforming an STG specification of the circuit into a flow-table can be automated easily .

As its output, the procedure generates the excitation functions for the non-input variables, viz., the output and the internal state variables. Under the assumptions described below, the excitation functions generated guarantee the hazard-free operation of the circuit.

3.2 Assumptions

As described in previous chapters, certain constraints on the external environment have to be imposed in the design and synthesis of asynchronous circuits. Moreover, in order to guarantee the proper behavior of the synthesized circuits, a few additional assumptions on the internal environment have to be made to ensure a hazard-free operation. These assumptions are summarized below :

1. The circuit operates in fundamental mode. After a change in the input configuration, no other input change is allowed until the circuit reaches a stable state. This assumption poses the following relation between the delays of the external and internal environments : the time-difference between any two successive input transitions should be sufficiently large so as to allow the circuit to reach a steady state after the first input change. This assumption also implies that a transition from one stable state to another is only in response to a change in the input configuration.
2. All stray delays within the network are assumed to be bounded. Stray delays include both element and line delays.
3. During any state change, no critical race conditions are permitted, although non-critical races are allowed. The concept of critical and non-critical races is described in Chapter 6.
4. Although certain state assignment techniques could be employed so as to guarantee that the circuit is free of all critical races, the presence of hazards may still cause the circuit to malfunction as demonstrated by the example presented in Chapter 7. It is therefore essential to eliminate all possible combinational hazards to guarantee the correct operation of the circuit. Unger [9] has shown that any function is realizable with a circuit free of all combinational hazards involving single-input changes. However, if multiple input changes are involved, then any function with more than one prime implicant contains hazards that cannot be eliminated through logical design alone.

   It is therefore necessary to restrict the external environment so that only single-input changes are permissible. Thus successive input transitions are restricted to adjacent input combinations.

3.3 An overview of the synthesis procedure

Under the above assumptions on the external environment and the design technique, the synthesis procedure aims at deriving the excitation functions for the output signals and the internal variables of the circuit. The synthesis techniques are designed so as to ensure that the circuit behaves correctly under all input conditions, ie., no erroneous output is obtained for any sequence of applicable inputs.

Figure 3.1 outlines a diagrammatic representation of the various steps involved in the flow-table synthesis procedure. These steps are explained in brief below and studied in detail during the course of this report.

1. **Derivation of the flow-Table description :**

   From the initial representation of the circuit in terms of a verbal description or an STG specification, the flow-table is derived under the assumption of single input variable changes. Thus the entries in the flow-table corresponding to simultaneous changes of two or more input signals are unspecified. In addition, the outputs corresponding to unstable states are also unspecified and are assigned later in the synthesis process.

2. **Reduction of the primitive flow-table :**

   The initial state-table called the *Primitive Flow-Table* is modified to represent a table of an incompletely specified synchronous machine. The process of state minimization then aims at finding the minimal row machine with the same terminal characteristics as the original machine. Two types of minimization are possible in an asynchronous machine :

   - States which are redundant, in the sense that their function is accomplished by one or more other states of the machine, can be eliminated.
   - States which are distinguishable only by the values of the input variables and have the same values for the secondary variables can be merged together.

   Using standard techniques adopted for incompletely specified synchronous circuits, the reduction procedure identifies the set of states which should be included in a minimal machine.

3. **Formation of the reduced flow-table :**

   Once the set of states in the equivalent minimal machine have been identified, the initial flow-table is converted to its reduced form. In the case of asynchronous machines, certain unique characteristics of the flow-table are utilized to simplify the procedure deriving the reduced flow-table.

4. **Assignment of outputs to unstable states :**

   Since the initial primitive flow-table had the outputs of the unstable states unspecified, certain states in the reduced table may also have their outputs unassigned. These outputs are assigned based on certain design criteria like the speed of output change or complexity of the output logic.

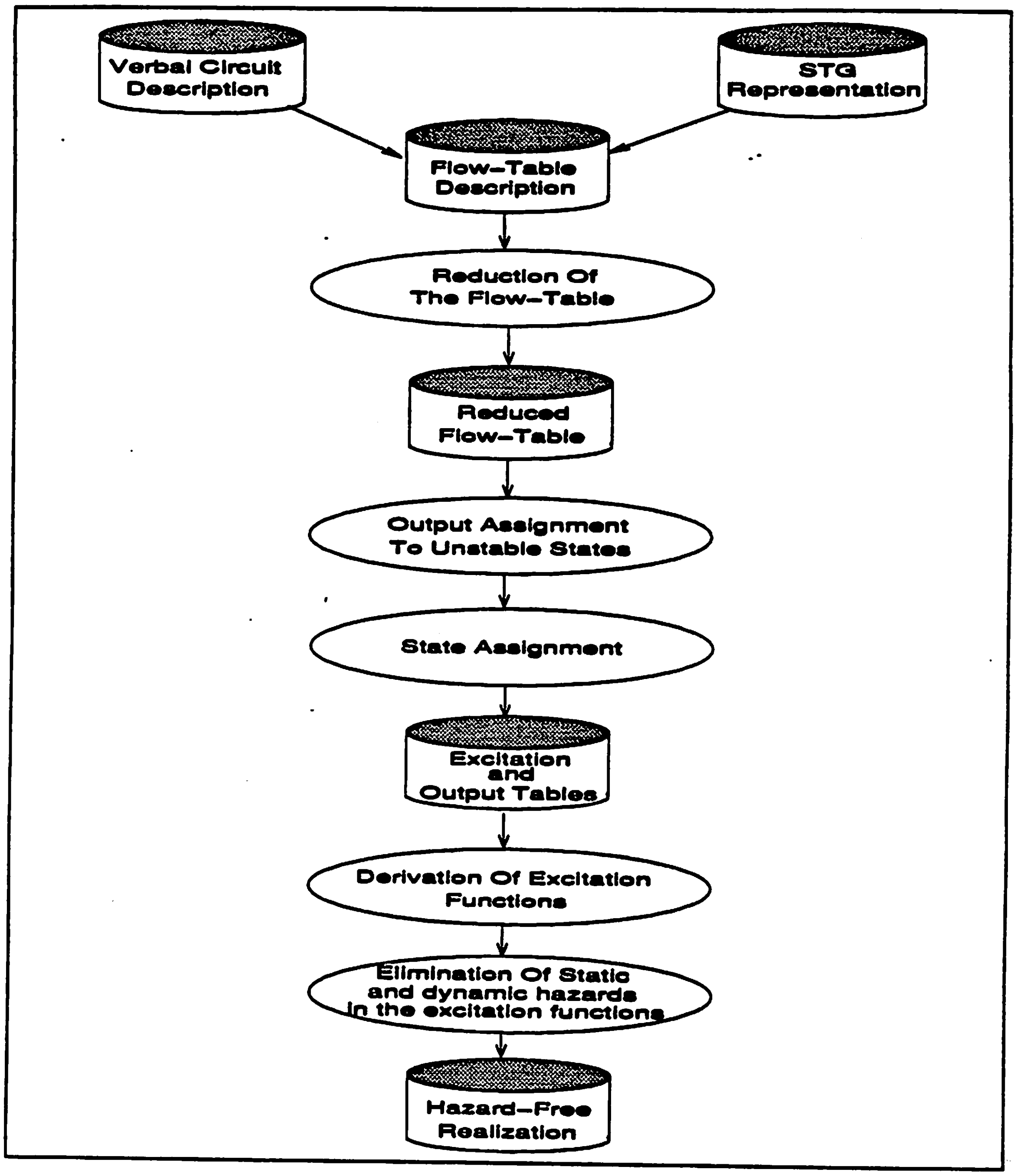
Verbal Circuit Description
STG Representation
Flow-Table Description
Reduction Of The Flow-Table
Reduced Flow-Table
Output Assignment To Unstable States
State Assignment
Excitation and Output Tables
Derivation Of Excitation Functions
Elimination Of Static and dynamic hazards in the excitation functions
Hazard-Free Realization

Figure 3.1: *Synthesis flow for fundamental-mode asynchronous circuits*

5. **State assignment :**

   State assignment techniques for synchronous circuits aim at minimizing the complexity of the combinational logic. In contrast, the assignment of the binary-valued secondary variables in asynchronous machines has a very different objective. Since the delays associated with the different state variables may be unequal, state assignment has to guarantee that the successful completion of the internal transitions is independent of the relative values of these delays. Although multiple changes of state variables are permitted in a single transition, it has to be guaranteed that when two or more state variables change in a transition, the final state of the circuit is independent of the order in which these state variables change.

6. **Derivation of excitation functions :**

   After state assignment, the excitation and output tables are derived, from which the two-level logic functions for the state variables and output signals are generated.

7. **Hazard-free implementation of the combinational logic :**

   To guarantee a proper operation of the circuit under the assumptions listed earlier, the combinational logic has to be made hazard-free. This involves the elimination of static and dynamic hazards for single-input changes.

Chapter 4

Derivation of the Flow-Table

The first step in the synthesis procedure is the representation of the asynchronous machine in the form of a flow-table. In case a verbal description of the circuit is given, the flow-table has to be derived manually. However, if the circuit is specified in terms of its graph-theoretic representation as a signal transition graph, the process of transforming it to an equivalent flow-table can be automated and thus incorporated in the synthesis procedure.

In this chapter, the special characteristics of the flow-table representation of asynchronous circuits are presented and the method of deriving the flow-table from the initial circuit description is described.

4.1 A design example

To illustrate the design process, a classical example of an asynchronous circuit is described in this section [10]. This example is referred to in subsequent chapters of this report, unless otherwise mentioned.

The asynchronous circuit considered is that of a 4-phase handshake protocol, which is an example of an interface between two circuits A and B operating independently. The signal transition graph representation of the machine is depicted in Fig. 4.1. It consists of two input signals R_{in} and A_{in} and two output signals A_{out} and R_{out} which are described below :

R_{in} : input signal from circuit A.

A_{out} : acknowledge signal to circuit A.

R_{out} : ready signal to circuit B.

A_{in} : acknowledge signal from circuit B.

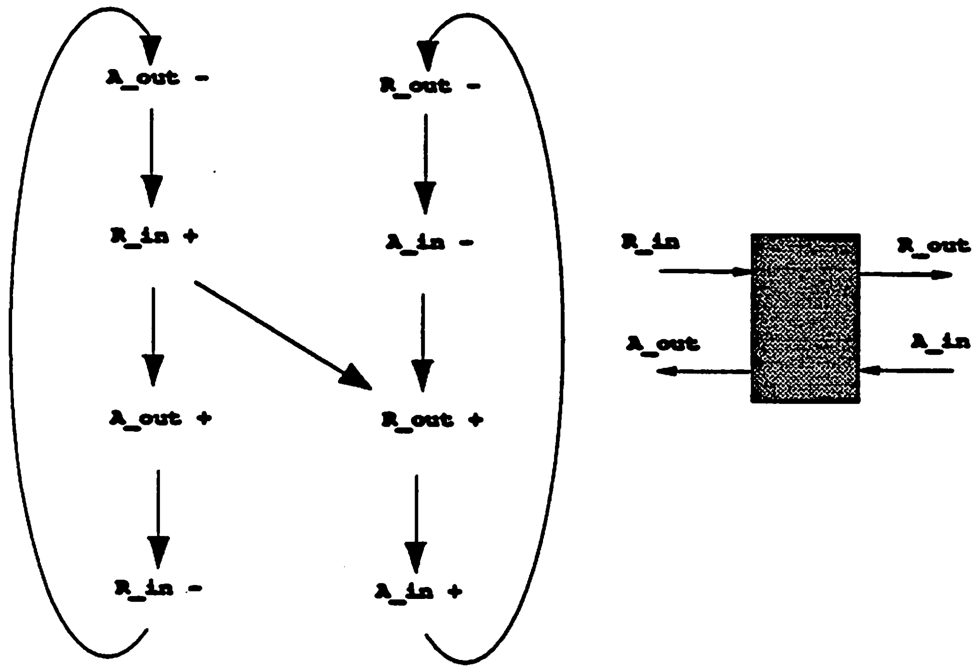

Figure 4.1: *STG Representation of the 4-phase handshake protocol [10]*

The start-state of the machine is the state in which all signals are low and is given by the initial marking on the edges $A_{out}^- \rightarrow R_{in}^+$, and $A_{in}^- \rightarrow R_{out}^+$. When circuit A wishes to signal circuit B, it does so through the 4-phase handshake in the following manner : Circuit A sends a high on signal R_{in} to the interface. Upon receiving a positive transition on R_{in}, the interface sends an acknowledge back to circuit A by a positive transition on the signal A_{out}. Simultaneously, it also signals circuit B with a high on the ready signal R_{out}. After this, the following two processes can proceed independently and in parallel :

1. After receiving the acknowledgement from the interface on A_{out}, circuit A withdraws the high on R_{in}. The interface then changes A_{out} back to low.

2. When circuit B receives a 1 on R_{out}, it signals the interface with a high on its acknowledge signal A_{in}. Upon receiving this acknowledgement from circuit B, the interface withdraws its ready signal R_{out} to circuit B. Following this, circuit B withdraws its A_{in} signal.

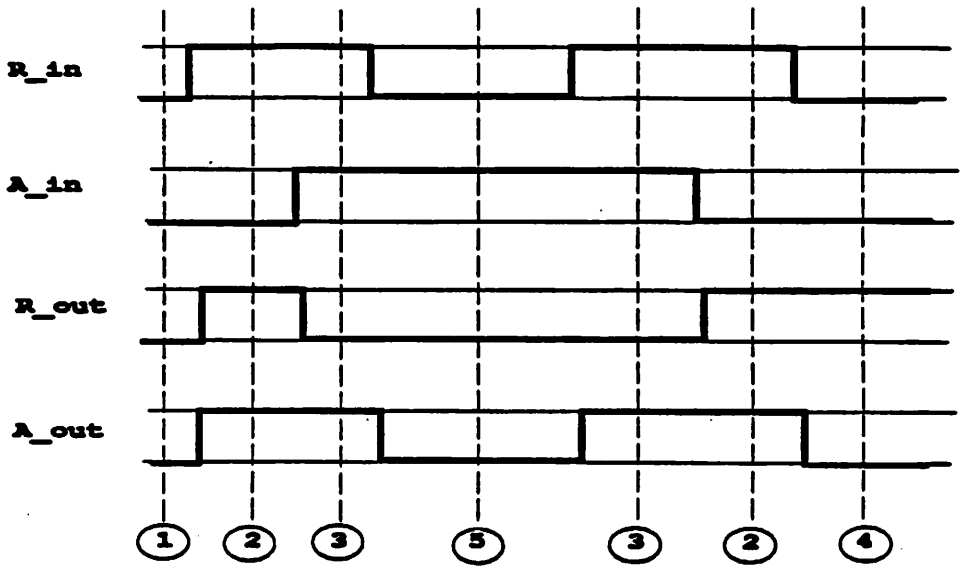

Figure 4.2: *Possible input-output sequence for the 4-phase handshake circuit*

4.2 Flow-table derivation from a verbal circuit description

In this section, the derivation of the flow-table representation from the initial verbal description of the 4-phase handshake circuit is described. Simultaneously, the properties of a flow-table for an asynchronous machine are highlighted. To illustrate the process of flow-table derivation, Figure 4.2 shows a possible input sequence and the corresponding output sequence.

As mentioned earlier, asynchronous circuits have two distinct types of states : *unstable* and *stable states*. When an input change occurs, the machine temporarily goes into an unstable states and then assumes the stable state condition. In the flow-table, stable states are distinguished from unstable ones by enclosing them in a box.

The start-state of the machine is the state where all signals are low, and is denoted by state $\boxed{1}$. Thus the stable state $\boxed{1}$ is entered in the first row under the input condition $(R_{in}, A_{in}) = (0,0)$. If R_{in} goes high at this point, then both the outputs R_{out} and A_{out} go high simultaneously. This new state is denoted by the stable state $\boxed{2}$. Thus a $\boxed{2}$ is entered in the second row under the input condition 10, with the outputs 11. Also, an unstable State 2 is entered in the first row under the input 10, signifying that the stable state $\boxed{1}$ will change to the stable state $\boxed{2}$ under the inputs 10 after passing through the transient unstable State 2.

Now, if input A_{in} also goes high, then the circuit enters a new state where the output R_{out} is low and A_{out} is high. Thus the stable state $\boxed{3}$ is entered in the third row under column 11, with

State, $R_{out}A_{out}$			
$R_{in}A_{in}$			
00	01	11	10
$\boxed{1}$,00	–	–	2
4	–	3	$\boxed{2}$,11
–	5	$\boxed{3}$,01	2
$\boxed{4}$,10	5	–	2
1	$\boxed{5}$,00	3	–

Table 4.1: *The Primitive Flow-Table for the 4-Phase Handshake Machine*

the outputs 01 and an unstable state 3 is entered in the same column in the second row. Continuing in this way, next-state and output entries are determined for every possible input combination and history of past input-values. In case a stable state $\boxed{S}$ with the same outputs under the same input condition already exists, then a transition to that state is specified by entering the unstable state *S* in the corresponding column; otherwise, a new row is augmented and a new state is introduced in the column of the input configuration. The final table derived is shown in Table 4.1. It has five rows and 4 columns, each row corresponding to a stable state and each column to a different input condition. Although states $\boxed{1}$ and $\boxed{4}$ occur in the same column with input condition 00, they are distinct since they have different outputs.

In conclusion, the main features of the primitive flow-table which completely specifies the logical behavior of the given machine, can be enumerated as follows :

1. The table has exactly one stable state in each row.

2. In the event of an input change, a horizontal move occurs from the present stable state to an unstable state in the column of the new input configuration.

3. A vertical move from an unstable state *S* to the corresponding stable state *S* represents a change in the values of the internal state variables.

4. A horizontal move can only start from a stable state, since an input change can occur only when the circuit is stable (assumption of fundamental mode of operation).

5. Since it is assumed that only one input signal can change at any time, entries corresponding to multiple input signal changes in any row are unspecified.

6. Outputs corresponding to unstable state are also unspecified.

7. Even under the same input condition, there are distinct stable states for each different output condition.

4.3 STG transformation to the equivalent flow-table

As described earlier, the derivation of the flow-table from a verbal description of the circuit is very unsystematic and prone to errors. However, signal transition graphs provide an efficient means for specifying the behavior of asynchronous circuits and their transformation to an equivalent flow-table can be easily automated and thus incorporated within the synthesis procedure.

The concept of signal transition graphs as effective means of specifying asynchronous circuits and their use in the synthesis procedure is described in Chapter 8. One method of translating STG's into flow-tables is described in [14]. This approach first converts a STG into its equivalent finite automaton, the state graph, which is then used to derive the flow-table. A direct method of transforming STG's into flow-tables can eliminate the intermediate state graph stage, thus reducing the huge space complexity associated with state graphs, at the same time making the transformation faster.

Chapter 5

Reduction of the primitive flow-table

5.1 Objectives of flow-table reduction

It is evident from the procedure for flow-table derivation that the table contains only one stable state in each row and entries corresponding to multiple-input changes are unspecified. The construction of the initial primitive table thus leads to more states than would actually be necessary to specify the behavior of the given machine. Reduction in the number of internal states may lead to a reduction in the number of state variables necessary to encode the internal states. This may not only result in a reduction in the number of memory elements required for the feedback loops, but may also lead to a reduction in the complexity associated with the combinational logic required to implement the excitation functions for the output and state variables. Thus for economical realizations, it is desirable to reduce the number of internal states in the flow-table.

In an asynchronous machine, each stable state represents a *total state* which is specified by the secondary variables as well as the external input variables. Thus an asynchronous circuit can change states due to a change in its input variables only and not necessarily involving a change in any of its secondary variables. Such states can therefore be distinguished by the values of their input signals. In addition, there may also be states which are redundant, ie., states whose function is accomplished by one or more other states of the machine. The reduction of the flow-table of an asynchronous circuit therefore has two objectives : removal of redundant states, and identification and subsequent merger of states which are distinguishable only by the values of the input signals.

Minimization of the number of internal states of an asynchronous machine has many advantages :

Present State	Next – State, $R_{out}A_{out}$			
	$R_{in}A_{in}$			
	00	01	11	10
1	$\boxed{1}$, 00	–	–	2
2	4	–	3	$\boxed{2}$, 11
3	–	5	$\boxed{3}$, 01	2
4	$\boxed{4}$, 10	5	–	2
5	1	$\boxed{5}$, 00	3	–

Table 5.1: *Modified Flow-Table for the 4-Phase Handshake Machine*

1. Since the number of memory elements necessary for the implementation of a machine is usually proportional to the number of internal states, state minimization may reduce the complexity and cost and hence the reliability of the realization.

2. Diagnosis of a machine is much easier in the absence of any redundant states.

3. State minimization reduces the length of the test-patterns needed to test the machine and thus reduces the time complexity associated with testing.

It is therefore desirable to transform the given machine into another machine with the same terminal behavior, but which is free of any unnecessary states. Flow-table reduction thus corresponds to finding the minimum-row table with the same terminal characteristics.

5.2 Analogy with flow-table reduction for synchronous machines

The primitive flow-table derived for asynchronous machines can be modified to represent a table of an incompletely specified synchronous machine. Each row of the primitive table contains only one stable state. Therefore, the stable state S occurring in a row r_i could be denoted as the present state for row r_i. Then, each entry of row r_i occurring under an input configuration I_j represents the next-state and corresponding output generated when the machine is initially in state S and gets a new input configuration I_j. The modified table for the 4-phase handshake machine is represented in Table 5.1.

Since the modified table has many unspecified entries, it is analogous to the flow-table of an incompletely specified synchronous circuit. It however has the special characteristic that in each row, there is exactly one next state entry which is the same as the present state. This means that for each present state and for exactly one input configuration, the machine does not undergo any change in its internal variables. Standard flow-table reduction techniques used for state minimization of incompletely specified synchronous circuits can now be applied to the modified table of the asynchronous circuit. In this chapter, an exact algorithm for the minimization of incompletely specified finite-state machines with special reference to asynchronous machines is presented.

5.3 Previous work in state minimization

The general theory of incompletely specified machines has been widely studied in literature [1, 2, 9, 27, 30]. The minimization process for such machines consists of finding the set of maximum compatibles and selecting the smallest closed collection of compatibles from the set. However, other than explicit enumeration, no systematic procedure for selecting a minimal closed set of compatibles has been presented. Since the set of all maximum compatibles is obviously closed, the upper bound on the number of states in any minimized table is equal to the number of maximum compatibles [9]. In contrast, the set of maximum compatibles for a *completely* specified machine is disjoint. Therefore, in the case of completely specified machines, the number of states in the minimum machine is equal to the total number of maximum compatibles.

Ginsberg [29] proves the lower bound on the number of the states in any minimal machine to be the number of states in the largest *maximal incompatible* (those incompatibles not included in any other incompatible). Grasselli, et al [27] present a minimization procedure by illustrating that only a few compatibility classes need to be considered as members of any minimal solution. Their procedure is therefore less enumerative than most known methods. They describe an integer linear program formulation for the selection of the essential maximum compatibility classes. This formulation is an extension of McCluskey's prime implicant table [4]. Rao and Biswas [30] extend the ideas of [27] by giving stricter conditions for the generation of prime classes and the elimination of those maximum compatibles which will never be included in any minimal solution.

An exact algorithm for the minimization of incompletely specified machines with special reference to asynchronous machines is presented in this chapter. This algorithm is based on the generation of maximal compatibility classes (MC-Classes). From the set of MC-classes, the set of prime classes is generated. The minimal cover consists only of prime classes, and the set of prime classes to be included in the minimal cover is formulated as an integer-linear program which is solved using a binate covering algorithm.

5.4 Compatibility classes and closed covering

In this section, a few definitions which are essential for an understanding of the state minimization algorithm are presented.

Definition 1 *[1]: In a completely specified machine, two states s_i and s_j, are said to be* **equivalent** *if and only if, for every possible input sequence applied to either state, the output sequence generated is the same.*

Definition 2 *[1]: Two completely specified machines, M_1 and M_2, are said to be* **equivalent** *if and only if, for every state in M_1, there exists an equivalent state in M_2, and vice versa.*

Definition 3 *[1]: Two machines are* **isomorphic** *if and only if they are identical except for a re-labeling of their states.*

Definition 4 *[1]: A* **minimal machine** *is a one which has no equivalent states. Disregarding isomorphism, the minimal machine of a completely specified machine is unique.*

Definition 5 *[9]: In the case of incompletely specified machines, state s_i in machine M_1 is said to* **cover** *state s_j in machine M_2 if and only if, every input sequence applicable to both the machines when they are in their respective states s_i and s_j generates the same output sequence whenever the outputs of M_2 are defined.*

Definition 6 *[9]: With the above definition of state cover, machine M_1 is said to* **cover** *machine M_2 if and only if, for every state s_i in M_1, there exists a corresponding state s_j in M_2 such that s_i covers s_j.*

Definition 7 *[1]: Two states, s_i and s_j of machine M are* **compatible** *if and only if, for every input sequence applicable to both s_i and s_j, the same output sequence is generated whenever both outputs are specified and irrespective of whether s_i or s_j is the initial state.*

Definition 8 *[1]: A* **compatibility class** *is a set of states in which all members are pairwise compatible.*

Definition 9 *[1]: A* **Maximal Compatibility Class** *(MC-class) is a compatibility class not contained in any other class.*

Definition 10 *[1]: A set of compatibility classes is* **closed** *if, for every class C in the set, all its implied compatibles are also contained in the set. A* **closed covering** *is a closed set of compatibles which contains all the states of the machine. A closed cover specifies the sets of states which are compatible and therefore may be covered by single states of the reduced machine.*

In the case of completely specified machines, the *equivalence partition* , ie., the sets of equivalent states is unique. This leads to a unique reduced machine. However, for an incompletely specified machine, two or more different reduced machines may each cover the original machine. Thus the aim of flow-table reduction is extended to the dual objective of finding a reduced machine which not only covers the original machine, but also contains a minimal number of states.

A closed cover of the states of an incompletely specified machine serves the function of an equivalence partition for a completely specified machine. However, the difference lies in the fact that while the equivalence partition consists of disjoint sets, the closed cover may contain sets which are overlapping. This leads to the realization that the closed cover in incompletely specified machines is not unique. The objective of state minimization is therefore to select a closed cover which has the minimum number of compatibles and thus defines a minimum-state machine that covers the original machine.

In conclusion, the problem of state minimization aims at finding a closed set of compatibility classes of minimal cardinality, which covers all the states of the given machine.

5.5 Exact algorithm for state-minimization

An overview of the exact state-minimization algorithm is depicted in Figure 5.1. From the set of maximal compatibility classes, the algorithm generates all possible prime classes. The problem of selecting the minimal closed set of prime compatibles is formulated as a minimal binate covering problem. The solution to the covering problem gives the list of prime compatibles to be included in a minimal cover. Since the set of states in any prime compatible are pairwise compatible, they can be merged together into a single state. Thus the list of primes obtained as a solution to the covering problem also gives the number of states in the minimal machine.

5.5.1 Generation of compatibility pairs

The first step in the minimization algorithm is the determination of the pairs of compatible states. This procedure is based on the formation of the *Merger Table* [1] which is generated from the initial flow-table. The merger table describes the pairwise compatibility of the internal states. The initial merger table is generated by the following procedure :

1. If a pair of states can be directly recognizable from the flow-table as being compatible (their next states and output entries do not conflict), a '-' is placed in the cell corresponding to the pair.

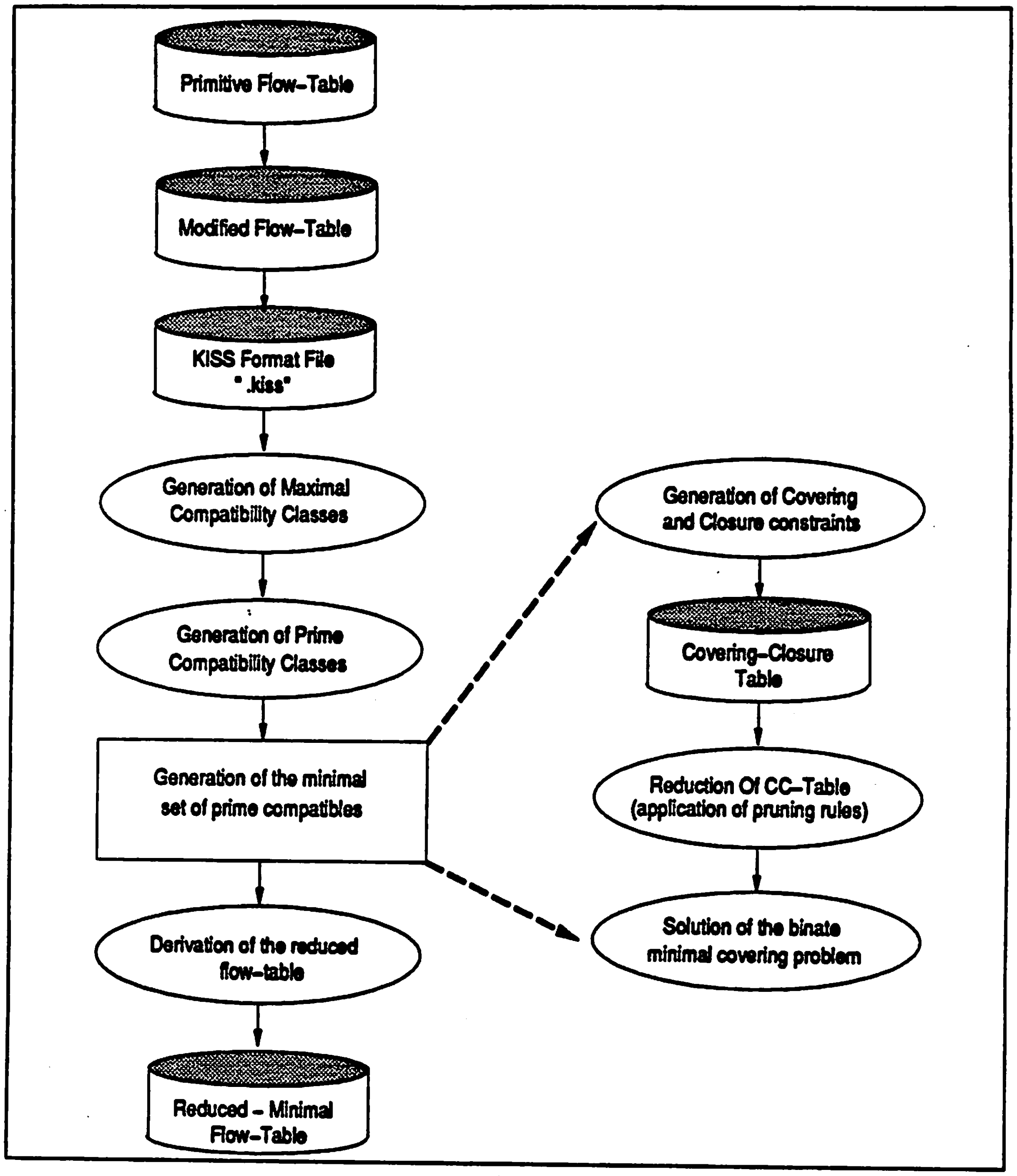
Primitive Flow-Table
Modified Flow-Table
KISS Format File
" .kiss"
Generation of Maximal Compatibility Classes
Generation of Prime Compatibility Classes
Generation of the minimal set of prime compatibles
Derivation of the reduced flow-table
Reduced - Minimal Flow-Table
Generation of Covering and Closure constraints
Covering-Closure Table
Reduction Of CC-Table (application of pruning rules)
Solution of the binate minimal covering problem

Figure 5.1: *Flow-table reduction algorithm for asynchronous machines*

2	(1,4)			
3	—	—		
4	*	—	—	
5	—	(1,4)	—	(1,4)
	1	2	3	4

Table 5.2: *The initial merger table for the 4-phase handshake machine*

2. Else, if a pair can be directly recognized as being incompatible (at least one of their output entries conflict), a '*' is entered in the cell.

3. Else, although the next state entries of a pair of states may conflict, but their corresponding outputs do not, then the states cannot be directly identified as either compatible or incompatible. In this case, the compatibility of the two states depends on the compatibility of the state-pairs which appear as conflicting next state entries. For such a pair of states, the dependency conditions for their compatibility are entered in the cells corresponding to the pairs. These dependency conditions are called as *implied conditions.*

The initial merger table for the 4-phase handshake protocol is shown in Table 5.2. The pair of states $(1,4)$ have conflicting outputs under the input condition 00 and hence the cell corresponding to them is marked with a '*'. Pairs $(1,3)$, $(1,5)$, $(2,3)$, $(2,4)$, $(3,4)$, and $(3,5)$ can be directly recognized as compatible pairs. However, the pairs $(1,2)$, $(2,5)$, and $(4,5)$ cannot be directly recognized as either compatible or incompatible. All these three pairs have states 1 and 4 as conflicting next-state entries and thus the implied condition $(1,4)$ is written in the cells corresponding to them. It is to be noted that for purposes of flow-table reduction, both the stable and unstable states are treated alike and hence no distinction is made while writing them.

The next step in the generation of compatibility pairs is to update the initial merger table so as to remove any inconsistencies. The table is updated step-by-step until it cannot be updated any further. This step is based on the implied conditions entered in some cells of the merger table. If a pair of states (p_i, p_j) are incompatible and are included in the implied list of another pair (p_m, p_n), then the pair (p_m, p_n) is obviously incompatible too. This rule is applied to the merger table in a systematic manner until no further incompatibilities can be determined. In the 4-phase handshake example, it is evident from the initial merger table that the pair $(1,4)$ is incompatible. This pair is included in the implied list of pairs $(1,2)$, $(2,5)$, and $(4,5)$. Thus the merger table is modified to denote the pairs $(1,2)$, $(2,5)$, and $(4,5)$ as incompatible by placing a '*' in their respective cells. The final merger table for the example is obtained as in Table 5.3.

2	*			
3	—	—		
4	*	—	—	
5	—	*	—	*
	1	2	3	4

Table 5.3: *The final merger table*

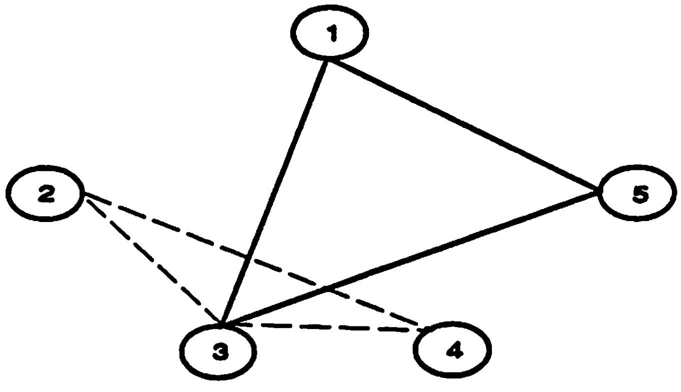

Figure 5.2: *Merger Graph for the handshake example*

The final list of compatibility pairs are the pairs of states which do not have a '*' in their cells. Thus the pairs $(1,3)$, $(2,3)$, $(3,4)$, $(1,5)$, $(2,4)$ and $(3,5)$ form the list of compatibility pairs for the 4-phase handshake sequential machine.

5.5.2 Derivation of maximal compatibility classes

From the list of compatibility pairs determined in the previous step, the maximal compatibility classes (MC-classes) can be generated by the formation of the *Merger Graph* [1] and the largest complete polygons in the graph can be identified. The Merger Graph has a node for each state of the original machine and an edge between two nodes if they form a compatible pair. A complete polygon of the graph identifies a set of states which are pairwise compatible. Thus the largest complete polygons determine the maximal compatible sets. The merger graph for the 4-phase handshake example is shown in Figure 5.2. The largest complete polygons in the graph can be identified as the set of states $(2,3,4)$ and $(1,3,5)$, which form the maximal compatibles.

However, a different technique is used to determine the MC-Classes. This technique is a dual-procedure based on the use of *incompatibility pairs* to recursively break down large sets of states into smaller sets until the sets finally remaining are the MC-Classes [26]. To illustrate the basic operation, consider the initial set of states $(1,2,3,4,5)$. Now, if the pair $(1,4)$ is identified as an incompatible pair, then the original set of states is split into two sets : a set $(2,3,4,5)$ with the state 1 omitted and a set $(1,2,3,5)$ with the state 4 deleted. This operation is applied recursively on the two sets generated. The procedure for the generation of the MC-Classes is enumerated below :

1. Start with the initial set of all states.
2. Assume the procedure begins with the first state, S_1. Split the set into two sets : one containing all states except S_1, and the other containing state S_1 along with all states which are compatible with it. This second set is easily identifiable from the merger table by selecting the states with non-'*' entries in the column corresponding to S_1.
3. Move to the next column which contains at least one '*' entry. Let this column be j. For each set on the list containing the state S_j and having at least one state which is incompatible with S_j, replace it with two sets : one without S_j and the other without states which are incompatible with S_j.
4. Repeat the procedure for every column in succession. At each stage, eliminate those sets on the list which are contained in other sets on the list.

This procedure when applied to the handshake example yields the following steps :

[a].	$(1,2,3,4,5)$	the initial set of states
[b].	$(2,3,4,5)\ (1,3,5)$	splitting with respect to state 1
[c].	$(3,4,5)\ (2,3,4)\ (1,3,5)$	splitting set $(2,3,4,5)$ with respect to state 2
[d].	$(3,5)\ (3,4)\ (2,3,4)\ (1,3,5)$	splitting set $(3,4,5)$ with respect to state 4
[e].	$(2,3,4)\ (1,3,5)$	deleting sets $(3,5)$ and $(3,4)$ contained within $(1,3,5)$ and $(2,3,4)$ respectively

5.5.3 Generation of prime compatibility classes

Since the potential number of MC-Classes for large machines could be very large, the complexity of the covering problem could be reduced by eliminating those classes which would not be included in any minimal cover. This stage of the minimization procedure therefore aims at selecting those MC-Classes which are potential candidates for inclusion in any minimal solution.

A compatibility class C_j is said to be *excluded* by a class C_i if C_i contains all the internal states of C_j ($C_i \supseteq C_j$), and all the classes implied by C_i are also implied by C_j. A *Prime Compatibility* class is one that is not excluded by any other compatibility class. By this definition and by the procedure for the generation of MC-Classes, all the maximal compatibility classes generated classify as prime compatibility classes. It has also been proved by Grasselli [27] that prime compatibility classes are the only ones which need to be retained for a minimal cover.

However, the MC-Classes generated are not the only prime compatibles. Other prime compatibles have to be generated from the list of maximal compatibles. This procedure is based on the decomposition of each maximal compatible into sub-classes and subsequent elimination of those sub-classes which are excluded by other classes already generated. The algorithm begins with the initial list of MC-Classes and selects the maximal compatible of largest size (n). It then generates all sub-classes of size $(n-1)$. Only those sub-classes which are not excluded by classes already on the list are added to the initial list of MC-Classes. This process is applied recursively to the newly-determined prime classes of size $(n-1)$, to generate prime classes of all sizes. This procedure guarantees that only prime classes are added to the list, ie., no class already on the list is ever excluded by a newly generated class.

To further reduce the number of prime classes generated, thus reducing the complexity of the covering problem, additional pruning rules can be employed as described in [30]. These are based on an extension of the concept of implied compatibles called *transitivity of implication.*

5.5.4 Formulation of the minimal covering problem

The objective of the covering algorithm is to select the smallest closed set of compatibles from the set of prime compatibles generated in the previous stage, so as to satisfy the following constraints :

- Each state of the machine is covered by at least one selected prime class. These constraints form the *Covering constraints.*
- The set of selected classes is closed with respect of implication. This constraint implies that if a prime class C is selected, then all its implied compatibility pairs must be included in at least one prime class from the list of selected prime classes. This forms the set of *Closure constraints.*
- The cardinality of the set which satisfies the above covering and closure constraints is minimal.

The third condition forms the objective function which is to be minimized under the covering and closure constraints [27]. To illustrate the formulation of the covering problem, assume that the total number of states in the machine is s and the number of prime classes generated is n. Let the number of classes with a non-empty set of implied compatibles be p. Corresponding to each compatible class C_i, let c_i be defined as the variable such that $c_i = 1$ if C_i is included in the minimal cover and 0 otherwise.

Cost function

The objective of the covering problem is to assign values from the set $(0, 1)$ to each c_i, $i = 1, ..., n$, so as to satisfy the following cost function subject to the covering and closure constraints :

$$minimize \sum_{i=1}^{n} c_i$$

Covering constraints

If state s_i is included in compatibles $C_{i1}, C_{i2}, ..., C_{ir}$, then the covering constraint for state s_i specifies that at least one of these compatibles must be included in the minimal cover. This is expressed in mathematical terms in the form of a boolean equation :

$$c_{i1} + c_{i2} + ... + c_{ir} \geq 1 \qquad i = 1, .., s$$

Closure constraints

If a compatible class C_i is selected, then each member of its implied set of compatibility pairs must be contained in at least one of the classes included in the minimal cover. Let the set of implied classes for C_i be $(C_\alpha, C_\beta, ..., C_\gamma)$. Let C_α be contained in compatibles $C_{\alpha_1}, C_{\alpha_2}, ..., C_{\alpha_a}$, C_β in compatibles $C_{\beta_1}, C_{\beta_2}, ..., C_{\beta_b}$ and so forth. Thus for each compatible, C_i, the closure constraints can be formalized as :

$$\bar{c}_i + (c_{\alpha_1} + c_{\alpha_2} + ... + c_{\alpha_a})(c_{\beta_1} + c_{\beta_2} + ... + c_{\beta_b}).....(c_{\gamma_1} + c_{\gamma_2} + ... + c_{\gamma_t}) = 1$$

Constraint inequalities

The above set of covering and closure constraints can be rewritten as mathematical inequalities as shown below :

$$c_{i1} + c_{i2} + ... + c_{ir} > 0, \qquad i = 1, ..., s$$
$$-\bar{c}_i + (c_{\alpha_1} + c_{\alpha_2} + ... + c_{\alpha_a}) > -1$$

$$-\bar{c}_i + (c_{\beta_1} + c_{\beta_2} + \ldots + c_{\beta_b}) > -1$$

......

$$-\bar{c}_i + (c_{\gamma_1} + c_{\gamma_2} + \ldots + c_{\gamma_n}) > -1$$

Formation of the Covering-Closure Table

The above constraints can be represented by means of a table called the *Covering-Closure Table* or the *CC Table*. This table has a row for each compatible class and a column for each constraint. The table has two sections, one for the covering constraints and the other for representing the closure constraints. For each covering constraint corresponding to state s_i, a 1 is placed in the rows of the prime compatibles which include s_i. For each closure constraint of the form

$$-\bar{c}_i + (c_{\alpha_1} + c_{\alpha_2} + \ldots + c_{\alpha_a}) > -1$$

there is a column having a 0 in row corresponding to compatible C_i, and a 1 in rows corresponding to compatibles $C_{\alpha_1}, C_{\alpha_2}, ..., C_{\alpha_a}$.

5.5.5 Solution to the covering problem

Before applying a binate covering algorithm to the above representation of the covering problem, the size of the CC-Table and hence the complexity of the covering algorithm can be greatly reduced by the application of pruning rules to the CC-Table. These rules are based on the concept of essential rows and row and column dominance and are described in [27]. Since the covering problem is NP-complete, reduction of the size of the CC-Table may considerably reduce the time needed to find a minimal cover.

Once the CC-Table is reduced, the covering problem is solved using a standard binate covering package called "sm_mat_bin_minimum_cover()" which takes as its input the CC-Table and generates the list of prime compatibles which are included in a minimal cover.

5.6 Formation of the reduced flow-table

Once the prime compatibles which are included in the minimal cover are determined, the original primitive flow-table has to be reduced. The reduction procedure aims at merging the states which belong to the same prime compatible. However, the compatibles selected to form the minimal cover may not necessarily be disjoint, ie., a state of the original machine may be included in two or more selected prime compatibles. It is therefore necessary to make the cover disjoint before deriving the reduced table.

Once the selected prime compatibles form a *partition* of the original set of states, the initial primitive flow-table can be reduced based on the minimal cover. This reduction process is simplified compared to that of synchronous flow-tables due to the intrinsic nature of the primitive flow-table for an asynchronous machine, ie:, the presence of exactly one stable next-state in each row which is the same as the present state for that row, and the outputs for unstable states being unspecified.

The basic step in the reduction of the primitive flow-table is to merge rows which have their corresponding present states included in the same compatible. The merger process aims at combining the next-state and output entries of the rows being merged to conform to the compatibles included in the minimal cover. In case a stable state is merged with an unstable state, the merged state in the reduced table is marked as stable. Also, in such a case, the output of the stable state represents the output of the merged state.

From the process of flow-table derivation, it is observed that two stable states which occur under the same input configuration must have conflicting output entries. If this was not true then only one of them would appear as a stable state under that input condition. This leads to the fact that in the procedure for the generation of prime compatibles no two states in the same compatible can have their next-state entries under any particular input condition as both stable states. It is also true that unstable states have their outputs unspecified. Thus while merging any two rows of the primitive table, it is only possible to merge a stable state with an unstable state, or to merge two unstable states together. In either of these cases, the states being merged are the same; only their status may differ as being stable or unstable. Thus a situation where two stable states in any column are to be merged can never occur and therefore, in any merger, conflicting output entries are never encountered. This fact is utilized in simplifying the process of flow-table reduction.

The reduced flow-table for the 4-phase handshake machine is shown in Table 5.4. The reduced table is constructed by merging the rows corresponding to the states in the compatibles $(2,3,4)$ and $(1,5)$.

5.7 Output assignment for unstable states

In the primitive flow-table, the outputs for the unstable states are left unspecified. In the process of deriving the reduced table, there could be situations when two unstable states in two rows

Present State	*Next – State*, $R_{out}A_{out}$			
	$R_{in}A_{in}$			
	00	01	11	10
1	$\boxed{4}$, 10	5, –	$\boxed{3}$, 01	$\boxed{2}$, 11
2	$\boxed{1}$, 00	$\boxed{5}$, 00	3, –	2, –

Table 5.4: *Reduced table derived from the partition* {(2,3,4) (1,5)}

are merged together, as a result of which the outputs of the merged next-states remain unspecified. These unspecified output entries have to be assigned values before the reduced flow-table can be used to derive the excitation functions.

In certain situations, the assignment of outputs is dictated by the necessity to guarantee that no momentary false output occurs in the transition. In the case of synchronous circuits, momentary false outputs pose no problems since outputs are sampled at the clock pulse. However, transient false outputs in asynchronous circuits can cause problems if the output is used as an input for another asynchronous circuit. Thus in case of a transition from one stable state to another, both with the same outputs, the output of the intermediate unstable state has to be assigned the same value to prevent any transient output values.

However, in the event of a change from one stable state to another, both with different output values, the assignment of output to the unstable state could be based on different design criteria like the speed of output change desired (fast or slow output transition), or the complexity of the output logic. For a transition from stable state s_i with output 0 to stable state s_j with output 1, the intermediate unstable state s_j can be assigned the output 1 if a fast output change is desired or the value 0 if a slow output transition is required.

However, a particular unstable state S_i may be an intermediate state for more than one state transition. This means that the stable state $\boxed{S_i}$ may be reached from more than one previous stable state. In such a situation, the output assignment procedure has to take all such transitions into consideration. Let us assume that an unstable state Si is the intermediate state for transitions from states $\boxed{S_1}$, ..., $\boxed{S_k}$. Let the output of the state $\boxed{S_i}$ be Z. If either of the states $\boxed{S_1}$, ..., $\boxed{S_k}$ have outputs equal to Z, then the output assigned to unstable state Si must also be Z in order to prevent any false transitions. However, if the outputs of all these states are different from that of $\boxed{S_i}$, then the unstable state Si can be assigned an output based on different design criteria.

Another criteria for output assignment can be based on the objective of minimizing the complexity of the output logic. The heuristic used to estimate the complexity of the output logic is based on the concept of *distance*. The distance between two binary values is defined as the number of bit-positions in which the two values differ. Let n be the number of outputs of a given circuit and for a particular unstable state S, let k be the number of stable states which have a transition to $\boxed{S}$. Then the output of the unstable state S is assigned so as to minimize the total sum of the distances of the assigned output from the output of the stable state $\boxed{S}$ and the outputs of the k stable states which have a transition to S.

Definition 11 *[9] : Flow-tables in which any input change produces at most one output change are called* **Single-Output Change** *(SOC) flow-tables, while tables which produce a sequence of output changes for a single input change are called* **Multiple-Output Change** *(MOC) tables.*

Definition 12 *[9] : SOC tables in which every state transition leads directly to a stable state without passing through a sequence of transient unstable states are called* **normal-mode** *flow-tables.*

The flow-table derivation procedure described in the previous chapter produces tables in which every transition leads directly to a stable state. Therefore, in order to make the tables normal-mode flow-tables, the outputs of the unstable states should be assigned such that every transition involves only a single-output change. This can be accomplished by the following procedure : for every transition from stable state s_i to stable state s_j, assign the output of the intermediate unstable state equal to either the output of s_i or the output of s_j.

As presented in Chapter 6, the state assignment technique is easy to automate in the case of a normal-mode flow-table.

Chapter 6

State Assignment in Asynchronous Circuits

6.1 Objectives of state assignment in asynchronous circuits

State assignment is the process of representing the internal states of a machine by combinations of values of binary state variables. In the case of a synchronous machine with n states, $\lceil \log n \rceil$ state variables are necessary and sufficient for representing the n states and the goal of state assignment is to assign a unique coding to each internal state so as to minimize the complexity of the combinational logic. However, in asynchronous machines, owing to the different delays associated with each feedback path, multiple changes of secondary state variables may lead to race conditions. Thus the primary goal of state assignment is to guarantee that the successful completion of any state transition is independent of the relative values of the delays associated with each secondary variable. Moreover, in order to guarantee the proper operation of the circuit, $\lceil \log n \rceil$ state variables may no longer be sufficient to represent all the internal states and it may be necessary to induce redundancy in the state assignment.

Against the background of the above primary objective, the secondary objectives of state assignment in asynchronous circuits could be one or more of the following :

1. Use a minimum number of state variables.
2. Minimize transition time : Each transition from one stable state to another may be routed through a number of intermediate transitions, each of which involves change of a single state variable. *Transition time* of an assignment is defined as the maximum number of steps re-

quired for the completion of any state transition. For fundamental-mode circuits which allow only single input changes, transition time of an assignment affects the minimum period between any two successive input changes and thus the speed of operation of the circuit. Thus a secondary objective of state assignment could be to minimize the transition time. The concept of transition time is discussed later on in this chapter under the connected row-set method of state assignment.

3. A modification of the above objective could be to have a single transition time state assignment so as to allow multiple state variable changes in a single transition. This technique of state assignment is called *single transition time* (STT) assignment and is described in later sections of this chapter. It is one of few state assignments techniques which can be automated with an exact algorithm.

The asynchronous state-assignment problem is different from the synchronous one due to the additional constraints introduced by the primary objective : if two or more state variables change during any transition, the final state of the circuit should be guaranteed to be independent of the relative order of the changes of those state variables. A change of multiple state variables during a single transition is called a *race condition*. A race could be either *critical* if the final state reached depends on the order in which the state variables change, or *non-critical* otherwise. Obviously, since we cannot guarantee that all state variables which have to change during a state transition will change simultaneously, any state assignment with race conditions must have only non-critical races. Thus a *race-free* assignment is sufficient but not necessary for proper operation of the circuit [9].

In the light of the above difficulties, the design of race-free circuits may not always be possible with *uni-code state assignments* such that all state transitions involve only single state variables changes. It therefore becomes necessary to relax some of the conditions of traditional state assignment techniques. We may need to assume that each state can be assigned more than one encoding, and/or that each transition can take place in more than one time-step.

Definition 13 *[9] : Assignments which have the property that a single state transition may take place in multiple time-steps are called* **Multiple Transition Time (MTT)** *assignments while those which require only a single time-step are termed* **Single Transition Time (STT)** *assignments.*

Thus in STT assignments, all state variables which have to change in a state transition are allowed to change simultaneously. While MTT assignments increase the minimum time-period

Present State	$Next-State, Z_0Z_1$			
	x_0x_1			
	00	01	11	10
1	$\boxed{1}$, 00	2	–	6
2	1	$\boxed{2}$, 11	4	–
3	1	$\boxed{3}$, 01	5	–
4	–	3	$\boxed{4}$, 10	7
5	–	2	$\boxed{5}$, 11	8
6	1	–	4	$\boxed{6}$, 00
7	1	–	4	$\boxed{7}$, 10
8	1	–	5	$\boxed{8}$, 11

Table 6.1: *Machine M1*

between two successive input changes, STT assignments do not hamper the speed of operation in this respect.

6.2 MTT state assignment techniques

6.2.1 Adjacency graph

The two different MTT methods of state assignment which will be discussed ahead are based on the formation of an *adjacency graph*. This graph has a vertex for each state (row) of the flow-table and edges between any two states which need to be assigned adjacent codes. States which need to be assigned adjacent codes are those which have a transition between them. Thus, if for any pair of states s_i and s_j and for any input condition I_k, if the next-state entry $N(s_i, I_k)$ is the unstable state s_m and the next-state entry $N(s_j, I_k)$ is the stable state s_m, then it is essential that states s_i and s_j be assigned adjacent codes and hence have an edge between them in the adjacency graph. This edge is labeled by the input condition I_k to signify the input under which the two states have a transition.

In effect, the adjacency graph lists all the adjacency constraints which must be satisfied by any race-free assignment. The state assignment problem is then to assign codings to each state so that all pairs of states which are adjacent in the adjacency graph are assigned codes differing in only a single state variable. However, if s_m happens to be the only state in the column I_k, then the race is a non-critical race and can be allowed since the circuit cannot reach any other intermediate state instead of s_m.

Present State	*Next – State*, Z_0Z_1			
	x_0x_1			
	00	01	11	10
A	$\boxed{A}$, 00	$\boxed{A}$, 11	B, 00	$\boxed{A}$, 00
B	A, 10	C, 10	$\boxed{B}$, 10	$\boxed{B}$, 10
C	A, 01	$\boxed{C}$, 01	D, 11	$\boxed{C}$, 11
D	–	A, 11	$\boxed{D}$, 11	C, 11

Table 6.2: ***Reduced table for machine M1***

To illustrate the concept of adjacency graph and the different methods of MTT assignments, the asynchronous machine $M1$ shown in Table 6.1 is used. After state minimization and assignment of outputs to unstable states, the reduced machine is as obtained as shown in Table 6.2. The final list of disjoint compatibles which are used to derive the reduced table are $(1,2,6)$, $(4,7)$, $(3,8)$, and (5). The reduced machine therefore has 4 states labeled A, B, C and D. State D has its transition on input 00 unspecified.

The adjacency graph derived from the above reduced flow-table for machine $M1$ is depicted in Figure 6.1. As is evident from the graph, state A has to be assigned a code adjacent to B. Similarly, the code of B should be adjacent to D and the code assigned to D has to be adjacent to A. Thus the set of states (A,B,D) form a cycle, as do the sets (A,C,D), (A,B,C), and (B,C,D). It is clear that the minimum number of state variables (2) are not sufficient to implement a state assignment for which all state transitions involve only a single state-variable change. Thus in this case there does not exist any unicode assignment such that all transitions occur between adjacent states.

This example illustrates that redundancy may be necessary for successful state assignment, in the sense that more than the minimum number of state variables may be required to guarantee the proper operation of the circuit.

6.2.2 Connected row-set assignment

This is a method of state assignment where each state transition is allowed to take place in more than one time-step so that each step involves only a single variable change. Each state is assigned a set of codings, which constitutes the *row-set* of the state [2]. The row-sets of every state are constructed so that each code in the set can be reached from every other code in the same set through a sequence of intermediate codes, each belonging to the same set and each transition

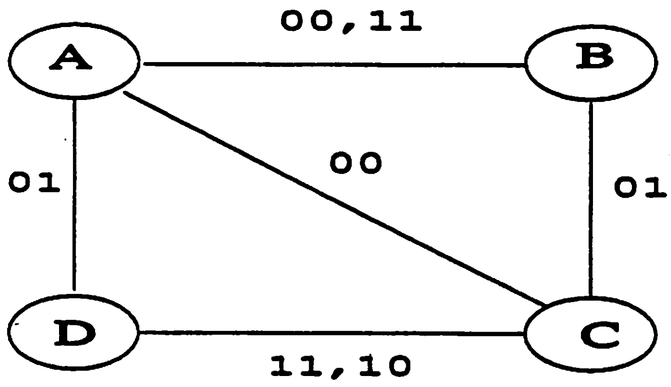

Figure 6.1: *Adjacency graph for machine M1*

involving only a single variable change. Such row-sets are termed *connected row-sets*.

For successful state assignment, the connected row-sets of each state have to be constructed so that the following condition is satisfied : for any pair of states which are adjacent in the adjacency graph, the corresponding row-sets of the two states must be *intermeshed*, ie., a code in the row-set of one of the states must be adjacent to a code in the row-set of the other state. With this construction procedure, any transition from state s_i to s_j can be accomplished by a series of transitions within the row-set of s_i to a code which is adjacent to some code in the row-set of s_j and then a final transition to that adjacent code. Thus every state transition in the flow-table can then be executed by a sequence of single-variable changes. However, there is no known non-enumerative algorithm for the construction of connected row-sets.

Criteria for constructing row-sets :

A simple criteria which could aid in the construction of connected row-sets is to order the vertices in descending order of their degrees and start with the vertex with the highest degree (which is the state with the maximum number of adjacent states). Let the degree of the vertex with the highest number of adjacencies be k. If the number of states in the machine is n, then the lower bound on the number of state variables required for state assignment is $m = \lceil logn \rceil$. However, if k is greater than m, then two there could be two possibilities : the number of state variables could be increased, or the state corresponding to the vertex with the highest degree can be assigned multiple codes so that the required number-of adjacent codes are available to be assigned to its adjacent states.

In the adjacency graph for the reduced machine of $M1$, vertices A and C have the highest degrees. Starting with vertex A, its adjacent vertices are B, C, and D. Thus two state variables are not sufficient for this state assignment. We therefore increase the number of state variables to 3 and assign the following codes : $A(000), B(001), C(010)$, and $D(100)$. The next vertex to be considered is C with adjacent vertices A, B, and D. The adjacency of C with A is already satisfied in the codes assigned so far. However, to satisfy the adjacency of C with B and D, an additional code has to be assigned to C so that it is adjacent to codes assigned to B and D. This additional code also has to be adjacent to the code 010 already assigned to C, in addition to being adjacent to the codes 001 of B, and 100 of D, Obviously, of the remaining three-variable codes, there is no code which satisfies all these conditions. We therefore assign the code 101 to C, which is adjacent to the codes of both B and D. However, to make the row-set of C connected, additional codes 110 and 111 have to be assigned to state C. This satisfies all the required adjacencies and thus is a feasible state assignment.

With this assignment, if the circuit is initially in state C represented by the code 010 under the input condition 01, and the input changes to 11, then the circuit would go through the sequence of single-variable changes $010 \rightarrow 110 \rightarrow 111 \rightarrow 101 \rightarrow 100$. The above transition therefore requires four steps for its completion. Since this is also the longest transition, the above state assignment has a transition time of four.

Reducing Adjacencies :

Before beginning the state assignment procedure, it is prudent to reduce the number of adjacency constraints, if possible. This reduction can be performed at the level of the reduced flow-table [2]. If there exist two states s_i and s_j, such that their corresponding next states under an input condition I are the same (s_k, where $s_k \neq s_i, s_j$), then the next-state entries can be modified by making $N(s_i, I) = s_j$ instead of s_k. This modification has the effect that the transition $s_i \rightarrow s_k$ is replaced by the sequence of transitions $s_i \rightarrow s_j \rightarrow s_k$. This would help in reducing the number of adjacencies associated with state s_i and thus possibly the number of state-variables.

Since the connected row-set assignment technique is largely intuitive, it is very difficult to automate. However, *universal* n-state row-set assignments [2, 9], which use a fixed number of state variables for all machines with the same number of states can be easily automated. These assignments, in general utilize more number of state variables, but may reduce the transition time and hence the speed of operation of the circuit.

6.2.3 Shared-row assignment

In the connected row-set assignment technique, the sequence of intermediate states for any two different state transitions are disjoint, in the sense that no two state transitions share the same intermediate states. However, this may not always be necessary. For example, two state transitions occurring under two different input conditions may be made to share intermediate states. This may lead to a reduction in the number of state variables needed for a state assignment free of critical races.

On the contrary, in the shared-row technique of state assignment, a unique code is assigned to each state and additional codes are used as intermediate codes to route transitions between rows which are not adjacent. Codes which are used as bridging states augment the flow-table as supplementary rows and are used in different columns to bridge transitions between different pairs of rows. However, in the same column, the shared row can be used to bridge more than one transition only if all those transitions have the same final state. Unfortunately, this method is also enumerative like the connected row-set technique.

This method is illustrated on the machine M1 whose adjacency graph is shown in Figure 6.1. The first step is to determine the destination sets for each stable state. The *destination set* corresponding to a stable state s_i under the input configuration I_j is defined as the set of states which have state s_i as their next state under the input I_j. This is denoted as D_{ij}. For example, under the input combination $I_1 = 00$, the set of states (A, B, C) have their next-state entries as the state A and therefore form the destination set D_{A1}. The list of destination sets obtained for the reduced machine M1 shown in Table 6.2 is :

Input $I_1 = 00 : (A, B, C)$
Input $I_2 = 01 : (A, D), (B, C)$
Input $I_3 = 11 : (A, B), (C, D)$
Input $I_4 = 10 : (C, D)$

Obviously, a two variable solution is not possible for the machine. For a three variable solution, let us assign the states of M1 with the following codes so as to satisfy the adjacency constraints for state A : $A(000), B(001), C(010), D(100)$. The objective is to make each destination set connected. The above assignment makes the sets (A, B) and (A, D) connected. The remaining sets which are to be connected are (A, B, C), (B, C), and (C, D). The obvious choice is to start with the state C which belongs to all these three destination sets. A code is therefore assigned from the remaining unassigned codes to the shared row R_1 such that it is adjacent to the code already assigned to

Present State	*Next – State*, Z_0Z_1				Code		
	x_0x_1						
	00	01	11	10	y1	y2	y3
A	$\boxed{A}$, 00	$\boxed{A}$, 11	B, 00	$\boxed{A}$, 00	0	0	0
B	A, 10	R1, 10	$\boxed{B}$, 10	$\boxed{B}$, 10	0	0	1
C	A, 01	$\boxed{C}$, 01	R2, 11	$\boxed{C}$, 11	0	1	0
D	–	A, 11	$\boxed{D}$, 11	R2, 11	1	0	0
R1	–	C, 10	–	–	0	1	1
R2	–	–	D, 11	C, 11	1	1	0

Table 6.3: *Shared-row assignment for machine M1*

$C(010)$. The assignment of the code (011) to the shared-row R_1 connects both the destination sets (A, B, C), (B, C). In order to connect the remaining set (C, D), the code (110) is assigned to the second shared-row R_2. The next-state entries in the Table 6.2 have to be changed appropriately to set up the above indirect transitions. The modified flow-table is shown in Table 6.3. Although, the above shared-row assignment requires the same number of state variables as the earlier row-set technique for Machine M1, in certain cases this method could give a considerable reduction in the total number of state variables used.

6.3 STT assignment techniques

As explained earlier, MTT assignments may increase the minimum time required between two successive input changes if the transition time of the assignment is more than a single state change. To circumvent this problem, assignments could be selected which either have only single-variable changes or allow the simultaneous change of multiple variables in a single transition. These assignments would require only a single time-step for the completion of any transition and hence are termed as single-transition time assignments.

Assignments which have only a single variable change for any transition are termed *one-shot assignments* [9]. These assignments assign multiple codes to each state, but the row-sets of each state need not be connected. However, the set of codes are assigned such that for any pair of states s_i and s_j which are adjacent in the adjacency graph, every code assigned to any state s_i is adjacent to some code assigned to s_j. In general, one-shot assignments require a lot more variables than other state assignment techniques.

Another method of STT assignment is based on the assignment of a single unique code to each state and the codes assigned to adjacent states may not be adjacent. These assignments allow multiple state variables to change simultaneously and therefore must guarantee the absence of any critical races. Such assignments are called *Unicode STT* assignments [2]. Unlike the assignment techniques discussed earlier which are based on trial-and-error and hence very difficult to automate, unicode STT assignments are more algorithmic and probably the only asynchronous assignment techniques which can be easily automated.

6.3.1 Unicode STT assignments

As discussed, the main objective of the unicode assignment technique is to assign a unique code to each internal state so as to guarantee that the state-table is free of any critical races. In this case, the circuit can allow multiple changes of state variables in a single state transition and still guarantee the proper operation of the circuit.

If multiple state variables are allowed to change simultaneously, then the order in which the variables change cannot be prespecified. For example, a state coded as 1001 can have a transition to another state coded as 1100 through the intermediate points 1101 and 1000. Thus the set of four points 1001, 1100, 1101, and 1000 can be specified by the *transition subcube* denoted by the cube '*1-0-*', where a '-' in a bit position means that the corresponding variable can take either of the values 0 or 1. The intersection of two transition subcubes is the set of points contained in both the subcubes.

With the above definition of transition subcubes, the condition for the absence of any critical races in any column of the flow-table can be specified. In any column represented by the input condition I_k and for any pair of transitions with different destination states, the two transitions do not have a critical race between them if their transition subcubes are disjoint, ie., their intersection is the null-set [5]. This condition forms the basis for the unicode assignment technique.

Therefore, in order to eliminate critical races between any two transitions with different destination states, there must be some state variable which distinguishes between the pair of transitions. For example, for a transition from state s_i to s_j and another from state s_k to s_l, with $s_j \neq s_l$, a variable must be assigned such that it takes the value 0 for states s_i and s_j, and the value 1 for states s_k and s_l, or vice-versa. Based on this condition, a construction procedure for unicode state assignments can be formulated as described below. This procedure works only for normal mode flow-tables.

1. For every column i of the flow-table, $1 \leq i \leq m$, where m is the number of columns, form the set of stable states in the column denoted by Q_i. Each of these sets of stable states have

then to be distinguished by the appropriate number of state variables. Sets containing only a single stable state can be eliminated.

2. For each of the remaining sets Q_i, do the following :

   (a) assign $\lceil \log m \rceil$ state variables, where m is the cardinality of set Q_i (the number of stable states in column i).

   (b) Assign a unique coding in these state variables to each of the stable states in Q_i. If the number of possible codes in these state variables is greater than the number of stable states, some of the stable states may be assigned codes in which some variables are unspecified.

   (c) The remaining unstable states in column i are assigned the same code as the stable state which is reached from them. In other words, if $N(s_j, I_i) = s_k$, with $s_k \neq s_j$, then s_j is assigned the same code in these variables as the state s_k.

3. In the final step, the unnecessary state variables can be deleted. The state assignment obtained so far can be considered as a table whose rows represent the states and columns represent the state variables. Thus each row i defines the coding assigned to state s_i. A column c_j *includes* a column c_k if c_k has the same value as c_j in the rows where c_j is specified. Column c_j *covers* column c_k if c_k includes c_j, or c_k includes $\overline{c_j}$, where $\overline{c_j}$ is the column obtained by complementing the 0's and 1's of c_j.

   With these definitions, the following steps can be taken to remove columns which are unnecessary, thus reducing the number of state variables.

   (a) Delete columns which are covered by other columns.

   (b) For any two columns, c_i and c_j, the intersection of c_i with c_j is defined as follows :

   $$c_i \cap c_j \begin{cases} = \text{null} & \text{if } c_{ki} \neq c_{kj} \text{ for some k.} \\ = c_{ki} & \text{if } c_{kj} = \text{'-'} \\ = c_{kj} & \text{if } c_{ki} = \text{'-'} \\ = c_{ki} & \text{otherwise} \end{cases}$$

   For a pair of columns c_i and c_j, if either $c_i \cap c_j$ or $c_i \cap \overline{c_j}$ exists, then the two columns can be replaced by a single column which is defined by the non-null intersection.

The above procedure is illustrated on the reduced machine $M1$ shown in Table 6.2. The sets of stable states in the 4 columns are (A), (A,C), (B,D), and (A,B,C). Of these, set (A) can be removed

State	y_0	y_1	y_2	y_3
A	0	0	0	0
B	1	0	0	1
C	1	1	1	-
D	0	1	1	-

Table 6.4: *Unicode STT state assignment for reduced machine M1*

since it is a single-state set. Thus the sets which need to be distinguished by state variables are (A,C), (B,D) and (A,B,C). We assign the state variable y_0 to the set (A,C), the variable y_1 to the set (B,D), and the variables y_2 and y_3 to the set (A,B,C).

State A is assigned $y_0 = 0$ and C is assigned $y_0 = 1$. Since state B has a transition to C in the column 01, B is assigned the same value of $y_0 = 1$ as state C. Similarly, state D has a transition to A in the same column and is therefore assigned $y_0 = 0$. In the column defined by the input configuration 11, the variable y_1 is used to distinguish states B and D by assigning $y_1 = 0$ to B and $y_1 = 1$ to state D. States A and C are assigned the values of $y_1 = 0$ and 1 respectively.

Finally, under the input condition 10, states A, B, and C have to be distinguished by variables y_2 and y_3. States A and B are assigned the values $y_2y_3 = 00$ and 01 respectively and since C is the only remaining state, it is assigned the value $y_2y_3 =$ '1–' with variable y_3 being unspecified. In this column, D has a transition to state C and thus is assigned the same coding in variables y_2y_3 as state C. The final state assignment is shown in Table 6.4 and requires four state variables. However, as is evident from the table, columns described by variables y_1 and y_2 are identical and hence can be replaced by their intersection which is equal to either of the columns itself. Also, the column specified by variable y_0 covers the column corresponding to y_3 and hence y_3 can be eliminated. Thus the final state assignment requires only two variables and is defined as A(00), B(10), C(11), and D(01).

Modifications of the above procedure have been described by Tracey [6]. This technique involves distinguishing between each pair of transitions in the same column instead of between all stable states in a column and usually produces state assignments with fewer variables than the above method.

Chapter 7

Hazard-Free realization of the flow-table

In this chapter, the techniques involved in realizing a hazard-free realization of the circuit based on the unicode STT state assignment described in the previous chapter are presented. After the state assignment stage, the next step in the synthesis procedure is the derivation of excitation tables for the next-state and output variables. In the case of asynchronous circuits, this step depends on the technique used for state assignment. From the excitation tables, the logic equations for a two-level implementation of the combinational logic are derived. The excitation functions are then modified, if necessary, to guarantee that the combinational logic is free of any static and dynamic hazards under the assumptions of fundamental-mode of operation and single-input changes described in Chapter 3.

7.1 Derivation of excitation functions

The process for the derivation of excitation functions for the output signals is the same as that for synchronous circuits. However, asynchronous circuits differ in many respects from synchronous circuits in the procedures to be followed for deriving the next-state excitation functions. In synchronous circuits, state codes which are not assigned to any state can never be reached by the circuit and hence can be ignored. Thus the specification of the Y-matrix for synchronous circuits does not specify the next-state entries for the codes which are not assigned to any internal state. On the contrary however, in asynchronous circuits with unicode STT assignments which allow multiple state variables to change during a single state transition, unassigned codes may be reached as

Present State	*Next – State*, Z_0Z_1			
	x_0x_1			
	00	01	11	10
$\boxed{A}$	$\boxed{A}$	$\boxed{A}$	B	$\boxed{A}$
$\boxed{B}$	A	C	$\boxed{B}$	$\boxed{B}$
$\boxed{C}$	A	$\boxed{C}$	D	$\boxed{C}$
$\boxed{D}$	A	A	$\boxed{D}$	E
$\boxed{E}$	A	C	D	$\boxed{E}$

Table 7.1: ***Machine M2***

State	y_0	y_1	y_2	y_3
A	0	0	0	0
B	1	0	0	1
C	1	1	1	0
D	0	1	1	1
E	1	1	1	1

Table 7.2: ***Unicode STT state assignment for machine M2***

intermediate codes in a single state transition. Thus care has to be taken to specify the next-state entries of all possible codes so as to guarantee that all transitions are executed as desired. This procedure is explained below with the help of an example machine.

Consider a transition from state s_i to state s_j in the column specified by an input condition I_k. The subcube corresponding to this transition is computed as the cube which contains all the minterms (codes) which could be reached in the $s_i \rightarrow s_j$ transition. Then for each code in the transition subcube (and not only the codes of s_i and s_j), the next-state entry under the input configuration I_k has to be assigned the code for state s_j. This ensures that the circuit will ultimately reach the destination state s_j, irrespective of the order in which the state variables change.

The above procedure for derivation of the Y-matrix is illustrated with the machine M2 whose flow-table is shown in Table 7.1. The initial unicode STT assignment is obtained by assigning variable y_0 to distinguish between states (A,C), y_1 to distinguish between (B,D), and y_2 and y_3 to distinguish between the set of states (A,B,C,E). The initial assignment is shown in Table 7.2. Since columns y_1 and y_2 are identical, they can be replaced by one column y_1. Thus the final unicode assignment which requires only three state variables y_0, y_1, and y_2 is : A(000), B(101), C(110), D(011), and E(111).

As described above, although the codes 001, 010, and 100 have not been assigned to any state, their next-state entries need to be specified in the Y-matrix if these codes could be reached

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

Table 7.3: *Y-matrix for the unicode STT assignment for machine M2*

as intermediate codes in a state transition. The final Y-matrix obtained is shown in Table 7.3. The flow-table for machine M2 contains only the state A in column 00. Thus for each of the states, the next-state entry in the input column 00 is specified as the state A, coded as 000. The rest of the codes, 001, 010, and 100 are also assigned the next-state code for state A since they are in the transition subcubes of $D \rightarrow A$ and $B \rightarrow A$ state transitions under the input condition 00. In column 01, there is a transition from state B(101) to state C(110). This transition involves the change of two state variables, y_1 and y_2. The transition subcube for this state transition is '1 – –'. Therefore, in the Y-matrix, the next-state entries for the four codes 100, 101, 110, and 111 are specified as state C (110). It should be noted that the next-state entries for codes 001, 010, and 100 under the input configuration 10 are unspecified since these codes can never be reached in any state transition under the input condition 10.

7.2 Elimination of hazards

In a synchronous circuit, the outputs are sampled at the clock pulses only after they have stabilized and hence any temporary spurious outputs do not affect the operation of the circuit to which the outputs act as the inputs. However, this is not the case in asynchronous circuits and any transient pulses at the secondary outputs may adversely affect the operation of the circuit. Moreover, since the combinational circuit controls the state variables of the complete sequential circuit, spurious pulses in the combinational logic in asynchronous circuits may take the circuit into an incorrect stable state. Even though the state assignment technique which is utilized guarantees that the circuit is free of any critical races, the presence of hazards may still cause the circuit to malfunction under particular input combinations. Hill, et al [3] illustrate this through the following example.

Present State	*Next − State*, Z_0Z_1			
	x_1x_2			
	00	01	11	10
A	$\boxed{A}$	$\boxed{A}$	$\boxed{A}$	D
B	$\boxed{B}$	$\boxed{B}$	A	$\boxed{B}$
C	A	B	$\boxed{C}$	$\boxed{C}$
D	A	$\boxed{D}$	C	$\boxed{D}$

Table 7.4: *Example to illustrate presence of hazards in a critical-race-free circuit*

Consider the asynchronous circuit given by its state transition table in Table 7.4, and the state assignment A(00), B(01), C(11), D(10). The excitation functions for the two state variables, y_1 and y_2 are given by :

$$Y_1 = \overline{y_2}\,\overline{x_2}y_1 + y_2y_1x_1 + y_2x_2x_1 + y_1x_2\overline{x_1}$$
$$Y_2 = y_2x_2 + y_2\overline{y_1}x_1 + y_1x_2\overline{x_1}$$

Assume that the circuit is initially in the stable state $\boxed{C}$ ($y_2y_1 = 11$) under the input condition $x_1x_2 = 11$. In this state, if input x_1 changes from 1 to 0, the circuit should remain at the stable state $\boxed{C}$ with both variables y_2 and y_1 remaining unchanged. However, it may happen that the product term $y_2y_1x_1$ goes to 0 faster than the product term $y_1x_2\overline{x_1}$ changes to 1 (owing to a delay in the inverter associated with x_1). This will lead to the condition in which all the product terms in the excitation function for Y_1 have the value 0, as a result of which Y_2 will momentarily change to 0. The circuit may thus make an erroneous transition to the stable state $\boxed{D}$.

It is therefore essential that the combinational logic in asynchronous circuits be designed to be hazard-free. Unger [9] has shown that for single-input changes, any function is realizable with a circuit free of all combinational hazards. However, this is not the case for multiple-input changes, and functions with more than one prime implicant contain hazards that cannot be eliminated through logical design alone. Therefore, the restriction that only single-input changes are permissible is imposed on asynchronous circuits as described in Chapter 3. The assumption of bounded stray delays is essential since the bounds have to be used to determine the minimum time difference between consecutive input changes, which is needed to guarantee that the combinational circuit reaches stability before the application of the next input change. This assumption is therefore necessary for proper operation of the complete sequential network with hazard-free combinational logic.

Different types of combinational hazards along with the conditions for their presence in two-level implementations of combinational logic for single-input changes are summarized below.

These conditions form the basis for algorithms designed to eliminate this class of combinational hazards [9, 2].

Definition 14 *[9] : Combinational hazards can be either static or dynamic hazards.* **Static hazards** *are present when the output of the circuit is required to remain constant (either 0 or 1) as a result of an input change, but the circuit produces an even number of pulses at the output before stabilizing at the initial value.* **Dynamic hazards**, *on the contrary, are present when the output is supposed to change from 0 to 1, or vice-versa, but the circuit produces three or more output changes instead of a single change.*

Definition 15 *[2] : Let I_1 and I_2 be two adjacent input conditions differing in only one variable x_i. Let f be the output function, such that $f(I_1) = f(I_2) = 0$. Then, a two-level sum-of-products realization of the function f has a* **static 0-hazard** *for the input transition $I_1 \rightarrow I_2$ if and only if there is a product term having both the literals x_i and $\overline{x}_i$ and all other literals have the value 1 in both I_1 and I_2.*

Definition 16 *[2] : A two-level sum-of-products realization of the function f has a* **static 1-hazard** *for the input transition $I_1 \rightarrow I_2$, where $f(I_1) = f(I_2) = 1$, if and only if there is no product term that has the value 1 in both I_1 and I_2.*

Definition 17 *[2] : A two-level representation of a combinational function f has a* **dynamic hazard** *for a transition between the adjacent input combinations I_1 and I_2 differing in the variable x_i and such that $f(I_1) \neq f(I_2)$, if and only if there exists a product term that contains both the literals x_i and $\overline{x}_i$ and all other literals in that product term have the value 1 in both I_1 and I_2.*

Static 0-hazards can be eliminated by deleting any product terms containing both a variable and its complement. All static 1-hazards can be eliminated by including every prime-implicant of the function in its realization. To accomplish this, not only should every point where the function has a value 1 be covered by a product term, but each adjacent pair of points with $f = 1$ should also be covered by a product term.

As an example, consider the logic equation given by $f = \overline{x}_1\overline{x}_3 + x_1x_2$. This function is graphically depicted in Figure 7.1 along with its realization. The cube $\overline{x}_1\overline{x}_3$ includes the minterms 000 and 010, while the cube x_1x_2 includes the minterms 110 and 111. This is obviously the minimum realization of the function f. However, although the cubes 010 and 110 are adjacent, the prime implicant which contains them, $x_2\overline{x}_3$ is not included in the realization of f. Consider the circuit defined by the above realization of f to be in the present state where $(x_1, x_2, x_3) = (1, 1, 0)$. Under this input condition, $a = 0$ and $b = 1$ and therefore, $f = 1$. Now, consider a single input change where x_1 changes from 1 to 0, to the new input configuration $(x_1, x_2, x_3) = (0, 1, 0)$. The value of f under this input condition should remain constant at 1, since a changes to 1 and b to 0.

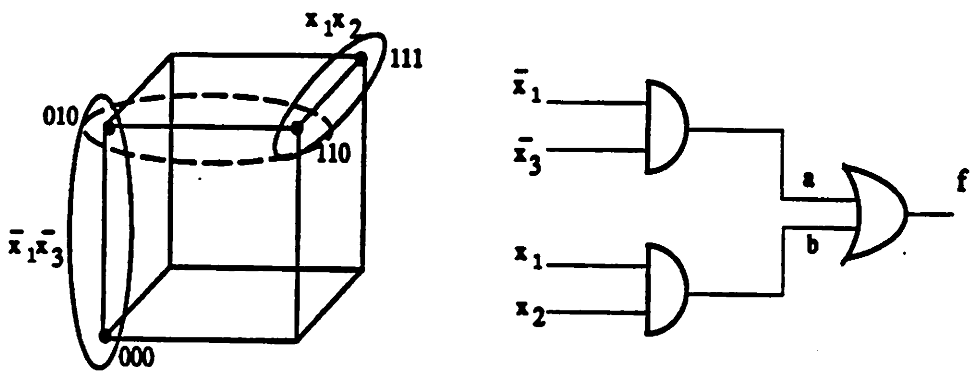

Figure 7.1: *Example to demonstrate static-1 hazard*

However, if the delay in line a is greater than that in line b, b may change to 0 before a can change to 1, thus giving a temporary false output of 0. This static-1 hazard can be eliminated by including the cube $x_2\bar{x}_3$ in the realization of f. Under the above input change, this cube has the constant value 1, thus preventing any spurious output.

From the conditions described earlier for the presence of dynamic hazards, it is evident that by the elimination of static hazards, all dynamic hazards for single-input changes are also eliminated.

As mentioned above, static and dynamic hazards are the result of unequal delays in various paths of the circuit. Another type of hazard could arise as a result of delays when, due to a change in an input signal, one state variable changes even before the input change reaches the logic circuit generating another state variable. These hazards are called *essential hazards* and occur because the change in an input signal reaches different parts of the network at different times [7, 3].

Definition 18 *[9] : A flow-table has an* **essential hazard** *starting in stable state* $\boxed{S}$ *for input variable x_i if and only if the stable state reached after one change in x_i is different from that reached after three changes in x_i.*

Such hazards are due to the basic specification of the circuit and cannot be eliminated by logical manipulations of the excitation functions. The only possible means of eliminating essential hazards is by adding appropriate delays in the feedback paths so as to ensure that the secondary state variables do not change until the input change propagates to all parts of the circuit [9, 8]. Miller [8] has proven that if an asynchronous machine contains an essential hazard, then any asynchronous network which realizes this machine and contains no steady-state hazard must contain at least one delay element. Further, if an asynchronous machine contains no essential hazards, it can be realized by an asynchronous network containing no delay elements. He has also shown that any sequential network with more than one delay element in the feedback loops can be replaced by an equivalent network containing a single delay element.

Chapter 8

Signal Transition Graph Based Synthesis Technique

In this chapter, the asynchronous design methodology based on the concept of signal transition graphs (STG's) which are used as graphical representations for specifying the behavior of asynchronous circuits.

8.1 The signal transition graph

Signal Transition Graphs (STG's) were first introduced by Chu [10] as a restricted class of live-safe free-choice Petri nets for specifying the behavior of asynchronous circuits and have recently been very efficiently used for automating the synthesis of asynchronous circuits [10, 11, 12, 14, 17, 18].

Definition 19 *[10] : An STG is a petri-net which is restricted to a free-choice net such that, if any two transitions share the same input place, then that place is the unique input place for both the transitions.*

In an STG, the transitions are interpreted as value changes on the signals (input, internal or output signals) of the circuit, and could be either positive transitions from 0 to 1 (labeled by a '+'), or negative transitions from 1 to 0 (labeled by a '-'). Further, the class of STG's used for representations of asynchronous circuits has the property that if a place has more than one transition as its successor, then all its successor transitions must be transitions on input signals.

In informal terms, an STG can be defined as a finite directed graph in which nodes represent signal transitions and the directed arcs determine the precedence constraints on the internal and the external environments. A transition is enabled when all its fanin edges have at least one token. When a transition fires, ie., the signal changes value from a 0 to 1 or vice-versa, a token is removed from every fanin edge and simultaneously, a token is added to every fanout edge of that transition. For purposes of asynchronous circuit specifications, we consider STG's with live and safe markings. A marking on an STG is *live* if every transition is or can be enabled through some sequence of firings from the marking. A marking is *safe* if no edge can be assigned more than one token, ie., once a transition has fired, it can fire again only after some other transition has fired. A STG has at least one live and safe marking if and only if it is strongly connected [10]. The STG for the 4-phase handshake protocol described in Chapter 4 is depicted in Figure 4.1.

8.2 The state graph

The equivalent finite automaton representation for an STG is called the *state graph*. A state graph is a directed graph where each state is in one-to-one correspondence with a live-safe marking of its STG. An edge from state s_1 to state s_2 means that the marking represented by s_2 can be reached from that represented by s_1 by the firing of the single transition with which the edge is labeled. The restriction to live-safe markings guarantees that all valid markings are reachable from one-another.

An STG can be converted to its equivalent state graph by an exhaustive simulation of token flow. This procedure starts with a live-safe marking and with token flow on the STG, generates the state graph. For each new live-safe marking M, it creates a new state in the state graph, and for each transition enabled in M, it fires the transition, creates a new edge from the previous state to the new state, and recursively calls the procedure on each of the new markings. The states of the graph are labeled by the values of the input and output signals in the marking corresponding to that state. Thus, in a state graph, the nodes represent the states, the node labels correspond to state assignments and the edges represent signal transitions.

8.3 Syntactic checks on the STG

Before the STG can be converted to its equivalent state graph, it has to be checked and transformed, if necessary, to satisfy a few syntactic properties of liveness and persistency [10, 11]. These properties have to be satisfied to guarantee that the STG conforms to the representation of a sequential circuit.

Definition 20 *[10] : A STG is said to be* **live** *if it is strongly connected and if in any of its simple cycles, for any signal t, transitions t^+ and t^- alternate.*

Liveness guarantees that after a signal transition, the next transition is always defined (strongly connected), and that no signal will be required to undergo two successive high or low transitions (alternation of t^+ and t^-). Obviously, this property is necessary for the STG to represent a control circuit. Therefore, a STG which does not conform to the liveness property cannot be used in the synthesis of a sequential circuit.

Another property which the STG should satisfy before it can be used in the synthesis procedure is that of *persistency*.

Definition 21 *[10] : A signal transition is said to be* **persistent** *if and only if, once the transition is enabled, only the firing of that transition can disable it. By this definition, a STG is said to be persistent if all its transitions are persistent.*

Persistency guarantees that if there is a transition on a signal t_1, due to which another transition t_2 is enabled, then the complementary transition of t_1 can only be enabled after the transition on t_2 has fired. Transitions which are not persistent can be made so by the addition of appropriate edges to the STG.

In formal terms, let $t_1 \overset{*}{\rightarrow} t_2$ denote that the firing of transition t_1, either immediately or after a sequence of other transitions, enables transition t_2. Then, the property of persistency of a STG implies that if $t_1 \overset{*}{\rightarrow} t_2$, and $t_1 \rightarrow t_3$, and if t_2 is the complementary transition of t_1, then $t_3 \overset{*}{\rightarrow} t_2$ must hold. Thus by checking this criteria for each transition, appropriate arcs can be added so as to make the STG persistent. However, input transitions in an STG are restricted to have exactly one fanin arc. Thus, if t_2 is an input transition, then the new arc is added to the immediately preceding output transition.

In the STG of Figure 4.1, after R_{in}^+ has fired, the transitions R_{out}^+ and A_{out}^+ are enabled. However, since there exists a path $R_{in}^+ \rightarrow A_{out}^+ \rightarrow R_{in}^-$, the signal R_{in} can be disabled even before R_{out}^+ has fired. This means that the transition $A_{out}^+ \rightarrow R_{in}^-$ is not persistent. This transition can be made persistent by the addition of the arc $R_{out}^+ \rightarrow A_{out}^+$. With this new arc added, the arc $R_{in}^+ \rightarrow A_{out}^+$ becomes redundant and is therefore deleted. Similarly, the arc $A_{in}^+ \rightarrow R_{out}^-$ is non-persistent and can be made so by the addition of the new arc $A_{out}^+ \rightarrow R_{out}^-$. This new arc necessitates the addition of the arc $R_{out}^- \rightarrow A_{out}^-$. The final persistent STG is shown in Figure 8.1.

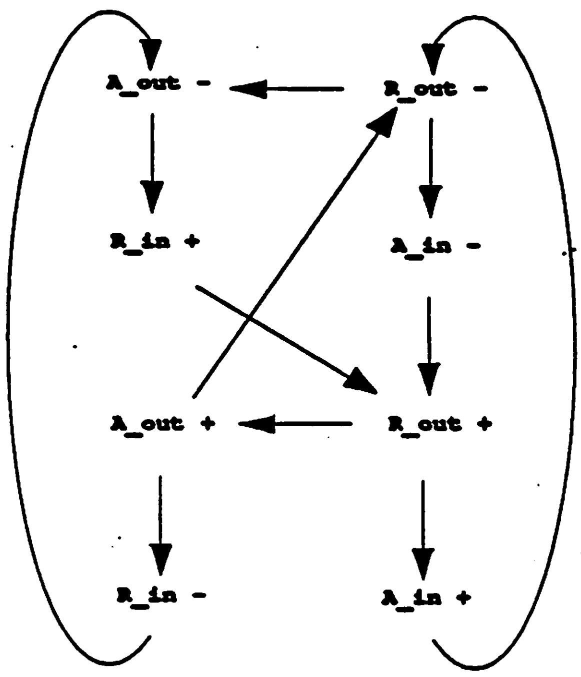

Figure 8.1: *Persistent STG for the 4-phase handshake machine*

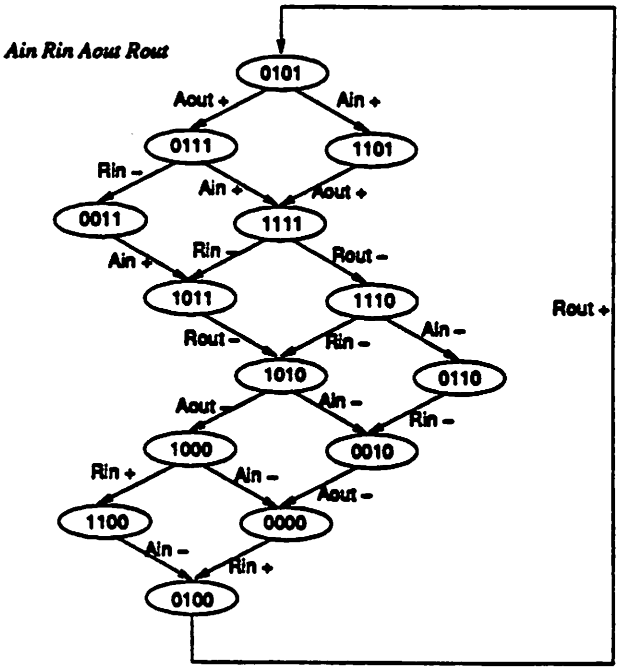

Figure 8.2: *State graph for the persistent STG of Figure 8.1*

The state graph of the persistent STG is derived by the procedure described in the previous section and is shown in Figure 8.2. The states of the graph are labeled according to the signal combination $A_{in}R_{in}A_{out}R_{out}$.

8.4 The STG-based synthesis procedure

Once the STG has been checked and modified, if necessary, to satisfy the properties of liveness and persistency, it is converted by an exhaustive token-flow simulation to its equivalent finite automaton, the state graph. Since the state graph has its states labeled by the values of the output and input signals, there may exist two or more distinct states which have the same codes assigned to them. This discrepancy may lead to erroneous behavior, since the labels on the states are used to derive the excitation functions. Such state graphs are said to have the property of *non-persistency due to state assignment*. Thus, before the circuit equations can be synthesized, the state graph has to satisfy the property of *unique state coding (USC)*, whereby each state should have a unique label assigned to it. State graphs which do not satisfy the USC property can be modified to do so by the addition of extra internal signals to distinguish the states which have the same assignments [10]. Vanbekbergen [17] also presents a technique based on the concept of *Generalized Lock Classes* to transform the initial STG so that its corresponding state graph satisfies the USC property.

After the state graph is modified such that each state has a unique code assigned to it, it can be directly used to derive the logic equations for each non-input signal. However, Chu [10] presents the technique of *net contraction* which essentially decomposes the initial STG into several STG's and maps them into their respective state graphs. Each of these state graphs is used to derive the corresponding individual circuits, from which the the final circuit is assembled. This technique has the advantage that it generally produces more efficient circuit implementations.

Chapter 9

Discussion

Although the flow-table technique for asynchronous synthesis has been studied quite extensively, the synthesis procedure has not been automated in the application world. The main problem which arises in the flow-table synthesis process is in the state assignment stage. Most of the efficient state assignment techniques like the connected row-set and shared-row assignments are essentially intuitive, trial and error procedures, and hence very difficult to automate. Other assignment procedures like the unicode STT assignments use a large number of state variables and are thus quite inefficient. The only other assignment techniques which could be easily automated are the universal n-state assignments, which have been proved to be accomplished with only two transition times [9], but have the drawback that they require twice the minimum number of state variables.

The flow-table synthesis procedure described in this report assumes bounded stray delays, a criteria which must be satisfied in order to determine the minimum time difference between any two successive input changes. For circuits in which the stray delays are unbounded, the minimum interval between two consecutive input changes so as to guarantee fundamental mode of operation cannot be determined. However, such circuits can still be synthesized by the generation of *ready* or *completion signals* which indicate to the external environment that the circuit is ready for the next input change [2].

Another drawback of the synthesis procedure described is the presence of delay elements in the feedback loops of each state variable. Moreover, the delays of these elements is assumed to be large enough to guarantee fundamental mode of operation. These delay elements hamper the speed of operation of the circuit and to an extent sacrifice the main objective of asynchronous designs : obtaining a speed-up by not restricting the circuit operation with a central fixed-period clock. As described in Chapter 7, only circuits which do not have any essential hazards can be realized without any delay elements. Circuits which demonstrate the presence of essential hazards require at least one delay element to guarantee their correct operation under all input conditions.

However, the flow-table design methodology offers many advantages over the STG-based synthesis procedure. The STG technique has to guarantee that the initial STG is live and persistent before it can be used for synthesis. This requires a large amount of pre-processing and may also sacrifice the original concurrency to some extent. The persistency requirement is necessary to guarantee that all output signals which are enabled as a result of a change in an input signal are allowed to fire before the input signal undergoes its complementary transition. However, this problem is solved in the inherent derivation of the flow-table itself. In the construction of the flow-table, all outputs which are enabled as a result of an input transition are changed simultaneously and the fundamental mode of operation guarantees the correct behavior of the circuit under all possible input conditions. Another restriction of the STG procedure is the satisfaction of the unique state coding requirement, either at the STG level, or at the state graph level. The state assignment stage in flow-table synthesis eliminates this requirement.

Against the backdrop of the above advantages, it can be concluded that if efficient techniques for state assignment are developed, the flow-table synthesis procedure could offer an attractive alternative to the signal transition graph approach to asynchronous circuit synthesis.

Bibliography

[1] Zvi Kohavi, "Switching and finite automata theory", *New York : McGraw Hill (Series: McGraw-Hill computer science series)*, 1978.

[2] Arthur D. Friedman and Premchandran R. Menon, "Theory and design of switching circuits", *Woodland Hills, California : Computer Science Press, Digital system design series*, 1975.

[3] Frederick J. Hill and Gerald R. Peterson, "Introduction to switching theory and logical design", *New York : Wiley*. 1981.

[4] E. J. McCluskey, "Introduction to the theory of switching circuits", *New York, McGraw-Hill, McGraw-Hill electrical and electronic engineering series*, 1965.

[5] C. N. Liu, "A state variable assignment procedure for asynchronous sequential switching circuits", *Journal of ACM, Vol. 10, pp. 209-216*, April 1963.

[6] J. H. Tracey, "Internal state assignments for asynchronous sequential machines", *IEEE Transactions on Electronic Computers, Vol. EC-15, pp. 551-560*, August 1966.

[7] Charles H. Roth, Jr., "Fundamentals of logic design", *Second Edition, West Publishing Company, St. Paul*, 1979.

[8] Raymond E. Miller, " Switching theory Vol. II", *New York, Wiley*, 1965.

[9] S. H. Unger, "Asynchronous sequential switching circuits", *Wiley-Interscience, New York*, 1969.

[10] Tam-Anh Chu, "Synthesis of self-timed VLSI circuits from graph-theoretic specifications", *Ph.D Dissertation, Department Of EECS, MIT*, June 1987.

[11] T. Meng, R. W. Brodersen and D. G. Messerschmitt, "A synthesis method for self-timed VLSI circuits", *Proceedings of the International Conference on Computer-Aided Design, pp. 514-517* November 1987.

[12] T. Meng, R. W. Brodersen and D. G. Messerschmitt, "Automatic synthesis of asynchronous circuits from high-level specifications", *IEEE Transactions on Computer-Aided Design, Vol. 8, No. 11, pp. 1185-1205,* November 1989.

[13] T. Meng, G. M. Jacobs, R. Brodersen and D. Messerschmitt, "Asynchronous Processor Design for Digital Signal Processing", *Proceedings of the International Conference on Acoustics Speech and Signal Processing, New York, NY, pp. 2013-2016,* April 1988.

[14] Luciano Lavagno, K. Keutzer and A. Sangiovanni Vencentelli, "Algorithms for synthesis of hazard-free asynchronous circuits", *Proceedings of the Design Automation Conference, pp. 302-308,* 1991.

[15] G. Gopalakrishnan and P. Jain, "Some recent asynchronous system design methodologies", *Technical Report No. UU-CS-TR-90-016, Department of EECS, University of Utah,* October 1990.

[16] Bill Lin and Fabio Somenzi, "Minimization of symbolic relations", *Proceedings of the International Conference on Computer-Aided Design, Santa Clara, California, pp. 88-91,* November 1990.

[17] Peter Vanbekbergen, F. Catthoor, Gert Goossens and Hugo De Man, "Optimized synthesis of asynchronous circuits from graph-theoretic specifications", *Proceedings of the International Conference on Computer-Aided Design, pp. 184-187,* 1990.

[18] Peter Vanbekbergen, Gert Goossens and Hugo De Man, "A local optimization technique for asynchronous control circuits", *Proceedings of the International Workshop on Logic Synthesis,* 1991.

[19] Jo C. Ebergen, "Translating programs into delay insensitive circuits", *Ph.D Thesis, Center for Mathematics and Computer Science, Amsterdam, CWI Tract 56,* 1989.

[20] David L. Dill, "Trace theory for automatic hierarchical verification of speed-independent circuits", *Cambridge, Mass. : MIT Press,* 1989.

[21] Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovec and Pieter J. Hazewindus, "The design of an asynchronous microprocessor", *Proceedings of the Decennial Caltech Conference on VLSI, MIT Press, pp. 351-373,* March 1989.

[22] Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovec and Pieter J. Hazewindus, "The first asynchronous microprocessor: the test results", *Technical Report CS-TR-89-06, Caltech,* 1989.

[23] David Ilana, Ran Ginosar and Michael Yoeli, "Self-timed architecture of a reduced instruction set computer", *Technical Report no. 732, Department of Electrical Engineering, Technion, Israel,* October 1989.

[24] David Ilana, Ran Ginosar and Michael Yoeli, "An efficient implementation of boolean functions and finite-state machines as self-timed circuits", *Computer Architecture News, Vol. 17, No. 6, pp. 91-104,* December 1989.

[25] Victor I. Varshavsky, "Self-timed control of concurrent processes : the design of aperiodic logical circuits in computers and discrete systems", *Kluver Academic Publishers, Boston,* 1990.

[26] M.C. Paull and S.H. Unger, "Minimizing the number of states in incompletely specified sequential switching functions", *IRE Transactions on Electronic Computers, vol EC-8, pp. 356-367,* September 1959.

[27] A. Grasselli and F. Luccio, " A method for minimizing the number of internal states in incompletely specified sequential networks", *IEEE Transactions on Electronic Computers, Vol. EC-14, No. 3, pp. 330-359,* June 1965.

[28] H.J. Mathony, "Universal logic design algorithm and its application to the synthesis of two-level switching circuits", *Proceedings of the IEE, Vol-136, Part E, No. 3, pp. 171-177,* May 1989.

[29] S. Ginsberg, "A synthesis technique for minimal state sequential machines," *IRE Transactions on Electronic Computers, Vol. EC-8, No. 1, pp. 13-24,* March 1959.

[30] C.V.S. Rao and N.N.Biswas, "Minimization of incompletely specified sequential machines," *IEEE Transactions on Computers, Vol. C-24, No. 11, pp. 1089-1100,* November 1975.

[31] R.W. House and D. W. Stevens, "A new rule for reducing CC Tables," *IEEE Transactions on Computers, Vol. C-19, No. 11, pp. 1108-1111,* November 1970.

Appendix A

Implementation details

This appendix describes the various implementation details - input / output formats, data structures and algorithms - of the synthesis package which has been developed. The package has been implemented in C programming language under the UNIX operating system. The executable version called *async*, along with the source files is in the directory */users/agupta/research/flow_table*.

A.1 Input / Output formats

The asynchronous finite-state machine is specified in the standard *KISS* format. The first three lines of the input file specify the number of inputs, outputs and internal states in the machine. These lines begin with the commands *'i', '.o', and '.s'* respectively. The file ends with the *'.e'* command. Line beginning with the '#' character act as comments and are ignored. Every other line in the input file represents a unique state transition, and is specified in the following format :

input present_state next_state output

The present_state and the next_state entries appear as symbolic values representing the names of the internal states. Unstable next_states are distinguished from their corresponding stable states by preceding them with the '*' character.

Inputs and outputs appear as strings of 0's and 1's, representing the values of the different input and output variables in a particular state transition. Since only single-output changes have been assumed, state transition entries corresponding to multiple input changes do not appear in the input file since they are unspecified, and hence serve as don't cares. Moreover, outputs of unstable states are unspecified and written as strings of '-' characters.

The output of the synthesis procedure is a reduced flow-table in *KISS format*. The output is written into a file with the name *< present_file_name > .reduced*. Since the reduced machine is subjected to state assignment before being written into the output file, the present_state and next_state entries are not symbolic entries, but actual values of binary-coded state variables.

A.2 Data structures

A.2.1 Representation of the finite state machine

The package is implemented using dynamic array structures described in the file *array.h*. Central to the implementation is the data structure for the finite state machine defined in the file *st_table.h*. The asynchronous machine is type-defined as a structure *smp_stg*, with the following fields :

```
struct smp_stg {
        char *reset_state        : start-state of the machine
        array_t *states          : array of states, each of type smp_state
        array_t *edges           : array of edges, each of type smp_edge
        int ni                   : number of inputs
        int no                   : number of outputs generated
        int no_state_variables : number of state variables needed for state assignment
}
```

Each state is type-defined as a structure *smp_state* with 5 fields, and completely represents all information regarding the particular state. This information includes its name and index number, transitions in which it is the destination, and transitions in which it is the present state. The data structure is described below :

```
struct smp_state {
        char *name       : symbolic name of the state
        int index        : index number of the state in the array of states
        array_t fanins   : array of edges which have the state as their destination state
        array_t fanouts : array of edges which have the state as their initial state
        int *code        : integer string of 0's and 1's which is the code assigned to the state
}
```

Finally, each edge represents a state transition and is type-defined as a structure *smp_edge* with the following fields :

```
struct smp_edge {
        char *input              : input for the state transition
        char *output             : output generated by the transition
        struct smp_state *src   : pointer to the present_state of the transition
        struct smp_state *sink : pointer to the destination state of the transition
        int unstable             : flag which is set to 1 if the sink state is unstable and to 0 otherwise
}
```

The above data structures efficiently model the finite state machine with minimal storage complexity. Moreover, as is evident, there is little repetition of information, since pointers are used to reference the desired element (state or edge). For example, each fanin (fanout) of a particular state is merely a pointer to the appropriate edge in the list of edges. Similarly, the source and sink states of each edge are pointers to reference the particular states of the state array. Appropriate care has also been taken to overcome the absence of random-access mechanisms in the array data structure. Internal to the package, the index values of the states are used for computation instead of the symbolic values. This has the advantage that the state can be randomly accessed with its index value, thus avoiding the time complexity involved with sequential access.

A.2.2 The flow-table reduction stage

The data structures employed for the process of flow-table reduction are defined in the file *compatible.h*. The reduction process begins with the determination of the compatible pairs. The list of compatibility pairs is type-defined as a structure *CP_LIST* with the following two fields :

```
struct CP_LIST {
        array_t *cp : array of compatible pairs, each of type CP
        int nr_pairs : number of pairs in the list
}
```

The structure *CP* is defined as follows :

```
struct CP {
        int st1, st2    : indices of the two states in the pair
        int pair_index : index number of the pair in the array
        array_t *list   : array of implied compatibility pairs
        int nr_implies : total number of implied compatibles
}
```

In the next stage of machine reduction, the list of maximal compatibles, *MAX_CP_LIST* is generated from the list of compatibility pairs. This list is type-defined as a structure *MAX_CP_LIST* with the following two fields :

```
struct MAX_CP_LIST {
        array_t *max_cp      : array of maximal compatibles, each of type MAX_CP
        int nr_max_cp        : number of maximal compatibles in the array
}
```

Each maximal compatible in turn is defined by a structure MAX_CP as shown below :

```
struct MAX_CP {
        array_t *array           : array of states which form the compatible
        int *nr_terms            : number of states in the compatible
        array_t *implied_list : array of compatibility pairs implied by the maximal compatible
        int nr_implied_pairs  : number of implied pairs in the above array
}
```

A.2.3 The state assignment stage

The state assignment stage is essentially composed of two steps. The first step involves the determination of the sets of states to be distinguished under each input condition. In the second step, appropriate number of state variables are assigned to each set and the variables are then assigned values from the set $(0, 1, 2)$. The data structures for this stage of the synthesis procedure are defined in the file *unicode_assign.h*.
The main data structure is comprised of the state assignment table which is defined as follows :

```
struct assign_table {
        array_t *variables : array of states which are to be distinguished by the same input
        int no_variables    : number of elements in the above array
}
```

Each set of states is defined as a structure *state_variable*, with the following fields :

```
struct state_variable {
        array_t *states : list of indices of states which comprise the subset
        char *input     : the input condition in which the above list of stable states occur
        int no_states   : number of states in the subset
}
```

Once the sets of states to be distinguished are determined, $\lceil log n \rceil$ number of binary valued state variables are assigned to each set S, where n is the cardinality of the set. The state variables are then assigned values from the set $(0, 1, 2)$ so as to distinguish the states in the same set by different assignments to the variables which are assigned to the set. The data structure *state_assignment* is utilized for this stage.

```
struct state_assignment {
        array_t *vars : array of state variables, each of type unicode_var
        int no_vars   : number of state variables in the above list
}
```

The structure *unicode_var* is defined as shown below :

```
struct unicode_var {
        int *states   : list of states in the machine, each of which is assigned a value from the set (0, 1, 2)
        int no_states : number of states in the list
}
```

A.2.4 Derivation of the reduced machine

The last stage in the synthesis process derives the reduced machine on the basis of the unicode STT state assignment performed in the previous step. The data structures for this stage are defined in the file *excitation.h*. Hierarchically, the central structure is the Y-matrix defined as shown below :

```
struct y_matrix {
        array_t *codes : list of rows of the flow-table, each of type y_matrix_row
        int no_rows    : number of rows in the flow-table
}
```

Each row of the above Y-matrix is defined as follows :

```
struct y_matrix_row {
        int *present_code   : code assigned to the present state of the row
        char *present_name : symbolic name of the present state
        array_t *next       : array of next state entries for the row, each of type matrix_state
}
```

Each next_state entry is defined as a *matrix_state* with the following fields :

```
struct matrix_state {
        int *code     : code assigned to the next state
        char *name    : symbolic name of the next state
        char *input   : input condition under which the next state occurs
        char *output : output produced by the transition
        int index     : index value of the next state in the list of states
}
```

A.3 An overview of the algorithm

In this section, the various steps of the overall algorithm along with the interfaces to the different procedures are presented.

1. *read_stg()* : This procedure reads the input file and initializes the data structure for the specified finite state machine.

   Input : 1. file in *KISS* format.
   Output : 1. the finite state machine of type *smp_stg*.

2. *create_comp_pairs()* : Pair-wise compatibility of states is checked and compatible pairs determined.

   Input : 1. the data structure for the machine of type *smp_stg*.
   Output : 1. list of compatibility pairs of type *CP_LIST*.

3. *find_max_compat()* : Computes the maximal compatibles based on the list of compatibility pairs generated in the previous step.

   Input : 1. the machine of type *smp_stg*.
   2. list of compatible pairs of type *CP_LIST*.
   Output : 1. list of maximal compatibles of type *MAX_CP_LIST*.

4. *attach_implied_pairs()* : Computes the implied list of compatibility pairs for each MC-Class, by finding the union of the list of implied pairs for each pairwise compatible states.

   Input : 1. list of maximal compatibles
   2. list of compatibility pairs
   3. the total number of states in the machine
   Output : 1. updates the data structures for the list of MC's with the list of implied pairs for each MC.

5. *find_prime_compats()* : Derives the list of prime compatibles (PC's) from the list of maximal compatibles generated in the previous step. This procedure is based on repeated decomposition of each MC-Class and elimination of subclasses which are excluded by other classes.

   Input : 1. the initial machine of type *smp_stg*.
   2. the list of compatibility pairs of type *CP_LIST*.
   3. the list of maximal compatibles of type *MAX_CP_LIST*.
   Output : 1. augments the original list of MC's with the prime compatibles generated.

6. *generate_matrix* : The next step in the algorithm is to find a minimal set of prime compatibles which satisfy the closure and covering constraints. This utilizes the standard binate covering package documented in the file *mincov.doc*. The data structures and the routines for handling them utilized by this package are defined in the file *sparse.doc*. This procedure represents the covering and closure constraints in the form of a sparse matrix. The data structure for the sparse matrix is essentially a doubly-linked list of rows and columns, each row and column in turn being a doubly-linked list of non-zero entries.

   Input : 1. the initial finite state machine
   2. the list of compatibility pairs
   3. the list of prime compatibles of type *MAX_CP_LIST*.
   4. a pointer to the sparse matrix generated by the procedure.
   Output : 1. the number of constraints in the sparse matrix

7. *sm_mat_bin_minimum_cover()* : This is the binate covering package defined in the file *mincov.doc.* It generates the set of rows of the covering-closure matrix which satisfy all the constraints. This set of rows defines the set of prime classes which form the minimal cover.

   Input : 1. the covering matrix of type *sm_matrix.*
   2. set of other parameters described in the file *mincov.doc.*
   Output : 1. the list of rows which form the minimal cover.

8. *find_disjoint_cover()* : Since the prime classes in the minimal cover are not disjoint, states in the original machine may be covered by more than one prime class. This routine makes the prime compatibility classes disjoint, so that they form a partition of the original set of states.

   Input : 1. the index numbers of the prime classes in the minimal cover.
   2. the original list of prime classes.
   3. the initial machine of type *smp_stg.*
   Output : 1. modified list of the selected prime classes so that the cover is disjoint.

9. *check_solution()* : Double checks that the disjoint list of selected prime classes satisfy all the closure and covering constraints.

   Input : 1. the selected list of disjoint prime classes.
   2. the original list of prime compatibles.
   3. the initial finite state machine.
   4. the list of compatibility pairs.
   Output : 1. a boolean value which if 0 (1) means that the solution is incorrect (correct).

10. *write_reduced_table()* : This procedure generates the reduced flow-table from the disjoint list of prime classes selected to form the minimal cover. The reduced table is generated from the original one by merging the states which occur in the same prime class. The states of the reduced table are given new symbolic names of the type *stk*, where *k* is the index number of the state. The new reduced machine is written in *KISS* format into a temporary file < *original_file_name* > *.reduced.*

    Input : 1. name of the temporary file into which the table is to be written.
    2. the original flow-table of type *smp_stg.*
    3. the list of disjoint prime classes in the minimal cover.
    Output : (none).

11. *read_stg()* : The data structure for the reduced machine is initialized by reading in the temporary *KISS* file.

12. *assign_outputs_single_change()* : The outputs of the unstable states in the reduced machine are assigned so as to have single output changes only. This is accomplished by assigning the output of the unstable states equal to the output of the corresponding stable states. A different variation of the output assignment procedure called *assign_outputs_min_distance()* has also been written. This procedure assigns the output of the unstable state S_i so as to minimize the sum of the distances of the assigned output from the outputs of the stable states which have a transition to S_i.

13. *unicode_state_assign()* : The state assignment stage begins with the determination of the sets of states which occur as stable states under the same input condition. These sets of states have to be distinguished by different assignments of state variables to them. This state assignment procedure is based on the unicode STT assignment technique and is described in detail in Chapter 6.

Present State	$Next - State, Z_0Z_1$			
	x_0x_1			
	00	01	11	10
1	$\boxed{1}$, 00	2	–	6
2	1	$\boxed{2}$, 11	4	–
3	1	$\boxed{3}$, 01	5	–
4	–	3	$\boxed{4}$, 10	7
5	–	2	$\boxed{5}$, 11	8
6	1	–	4	$\boxed{6}$, 00
7	1	–	9	$\boxed{7}$, 10
8	1	–	5	$\boxed{8}$, 11
9	–	3	$\boxed{9}$, 11	7

Table A.1: *Machine M3*

Input : 1. the reduced machine of type *smp_stg*.
Output : 1. the sets of states which have to be distinguished for each input condition.

14. *assign_state_variables()* : For a set of states S under the input condition I with cardinality n, $\lceil logn \rceil$ state variables are required to distinguish them. This procedure thus determines the appropriate number of state variables for each set of states and then assigns values to those variables. The final state assignment table produced is checked for row covering. Rows in the state assignment table covered which by other rows are deleted and the final minimal state assignment is thus determined.

    Input : 1. the set of states to be distinguished under each input combination.
    2. the reduced machine.
    Output : 1. the final state assignment.

15. *generate_y_matrix()* : With the above assignment to the states of the reduced machine, the excitation matrices for the next-state and output functions are to generated. This not only involves the substitution of the codes for the symbolic values of the states, but also has to determine the next-state entries for codes which are not assigned to any state.

    Input : 1. the reduced finite state machine
    Output : 1. the Y-matrix which gives the next-state and output values for each combination of the state variables.

16. *write_final_reduced_table()* : Finally, in the last stage of the synthesis process, the Y-matrix is used to generate the reduced machine in *KISS* format. The machine is written into the file $< original\_file\_name >$ *.reduced*. This file can then be used as an input to standard packages like *ESPRESSO* to generate the 2-level logic equations for the internal state variables and the output signals.

A.4 A sample execution

The execution of the program async is demonstrated on the machine M3 which is described by its state diagram in Table A.1. The machine has 2 input signals, 2 output signals, and 9 internal states. According to the assumption that successive input changes are restricted to adjacent input combinations only, the next state entries for multiple input changes are unspecified. Moreover, the outputs for the unstable states are also unspecified.

Machine M3 is synthesized using the asynchronous synthesis package, *async*. The result of the execution is described at the end of this appendix.

As is evident from the sample execution, the final reduced machine has 5 states, named *st0* through *st4*. The outputs of the unstable states in the reduced machine are assigned equal to the stable state which they lead to. This preserves the property of single output changes which is essential for unicode STT state assignment. The initial unicode assignment requires 5 state variables. However, after deletion of all rows which are covered by other rows of the state assignment table, the final state assignment requires only 4 state variables to prevent all critical races. In the final reduced flow-table, next state entries have to be assigned not only to the states of the machine, but also to the codes which are not assigned to any state. This is necessary to prevent any incorrect state transitions, since it is assumed that all state variables can change simultaneously. However, there may be certain codes which can never be reached under particular input combinations. These next state entries are unspecified and serve as don't cares.

```
reading machine m3.kiss...
read state table and computed compatible pairs...
            inputs = 2
            outputs= 2
            states = 9
computing maximal compatibles...
finding prime compatibles...
The final list of PRIME COMPATIBLES is :-
            -----------------------------------------
            0 : 1 2 6
            1 : 4
            2 : 3 8
            3 : 5 8
            4 : 7 9
            -----------------------------------------
computed covering matrix ...
            bounds: states = 9, max compatibles = 5, upper bound = 6
solving binate covering ...
total cpu time is 0.00 sec.
prime compatibles selected: 0 1 2 3 4
final disjoint list of prime compatibles selected:
            0 :1, 2, 6,
            1 :4,
            2 :3, 8,
            3 :5,
            4 :7, 9,
writing reduced table into temporary file : examples/m3.kiss.reduced...
reading reduced table from temporary file : examples/m3.kiss.reduced...
removing temporary file : examples/m3.kiss.reduced...
assigning outputs to unstable states...
the reduced flow-table is:-
            # reset state: st0
            .i          2
            .o          2
            .s          5
            .p          18
            00          st0         st0         00
            01          st0         st0         11
            10          st0         st0         00
            11          st0         *st1        10
            11          st1         st1         10
            01          st1         *st2        01
            10          st1         *st4        10
            00          st2         *st0        00
            01          st2         st2         01
            11          st2         *st3        11
            10          st2         st2         11
            00          st4         *st0        00
            01          st4         *st2        01
            10          st4         st4         10
            11          st4         st4         11
            01          st3         *st0        11
            11          st3         st3         11
            10          st3         *st2        11
            .e
assigning unicode STT assignment...
the set of states to be distinguished:-
            input 01            : st0 st2
            input 10            : st0 st2 st4
            input 11            : st1 st4 st3
```

```
initial state assignment table :-
          st0       st1       st2       st4       st3
          ---------------------------------------------
          0         1         1         1         0
          0         1         0         1         0
          0         2         1         2         1
          0         0         1         0         1
          0         0         2         1         2
          ---------------------------------------------
the final unicode state assignment is:-
          st0       st1       st2       st4       st3
          ---------------------------------------------
          0         1         1         1         0
          0         1         0         1         0
          0         0         1         0         1
          0         0         2         1         2
          ---------------------------------------------


Following codes are assigned to states :-
          st0 :     0000
          st1 :     1100
          st2 :     1010
          st4 :     1101
          st3 :     0010

Generating Y-Matrix...
          ------------------------------------------------------------
          state     code      |         00       01       10       11
          ------------------------------------------------------------
          st0       0000      |         0000     0000     0000     1100
                    0001      |         0000     —        —        —
          st3       0010      |         0000     0000     1010     0010
                    0011      |         —        —        —        —
                    0100      |         0000     —        —        1100
                    0101      |         0000     —        —        —
                    0110      |         —        —        —        —
                    0111      |         —        —        —        —
                    1000      |         0000     1010     —        1100
                    1001      |         0000     1010     —        —
          st2       1010      |         0000     1010     1010     0010
                    1011      |         —        1010     —        —
          st1       1100      |         0000     1010     1101     1100
          st4       1101      |         0000     1010     1101     1101
                    1110      |         —        1010     —        —
                    1111      |         —        1010     —        —
          ------------------------------------------------------------
writing coded reduced flow-table into file : examples/m3.kiss.reduced

Final reduced machine in KISS format :
          .i        2
          .o        2
          .s        5
          00        0000      0000      00
          01        0000      0000      11
          10        0000      0000      00
          11        0000      1100      10
          00        0001      0000      00
          00        0010      0000      00
          01        0010      0000      11
          10        0010      1010      11
          11        0010      0010      11
          00        0100      0000      00
          11        0100      1100      10
          00        0101      0000      00
```

00	1000	0000	00
01	1000	1010	01
11	1000	1100	10
00	1001	0000	00
01	1001	1010	01
00	1010	0000	00
01	1010	1010	01
10	1010	1010	11
11	1010	0010	11
01	1011	1010	01
00	1100	0000	00
01	1100	1010	01
10	1100	1101	10
11	1100	1100	10
00	1101	0000	00
01	1101	1010	01
10	1101	1101	10
11	1101	1101	11
01	1110	1010	01
01	1111	1010	01

.e