This is to certify that the

dissertation entitled

AUTOMATING THE DESIGN OF LARGE-SCALE
ASYNCHRONOUS SEQUENTIAL LOGIC CIRCUITS

presented by

Sheng-Fu Wu

has been accepted towards fulfillment
of the requirements for

Ph. D. degree in Electrical Engineering

P. David Fisher
Major professor

Date May 6, 1991

DATE DUE	DATE DUE	DATE DUE

AUTOMATING THE DESIGN OF LARGE-SCALE ASYNCHRONOUS SEQUENTIAL LOGIC CIRCUITS

By

Sheng-Fu Wu

A DISSERTATION

submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Electrical Engineering

1991

ABSTRACT

AUTOMATING THE DESIGN OF LARGE-SCALE ASYNCHRONOUS SEQUENTIAL LOGIC CIRCUITS

By

Sheng-Fu Wu

The computer-aided design process described simplifies the task of designing large-scale asynchronous sequential logic circuits (*ASLC*'s). It provides a highly structured, interactive approach for modeling sequential logic functions and for mapping these models into *ASLC* architectures and gate-level circuits. A design automation system, which implements this process, has been developed and tested. It contains five modules: the ***behavioral descriptor***, which maps the functional design specification into a primitive flow table; the ***merger***, which minimizes the number of states needed to implement the functional model; the ***connector***, which adds cycles and states, as needed, to avoid critical races; the ***assigner***, which encodes the states and generates the state excitation table and output table; and, finally, the ***equation generator***, which eliminates static or dynamic hazards and converts the state excitation table and output table into two-level, sum-of-product expressions for the state equations and output equations. This task-oriented system provides a convenient way to describe the functional behavior of sequential logic functions. It can reduce the design cycle time and improve the reliability of the overall *ASLC* design process and can also be used to facilitate the investigation of alternative *ASLC*

implementations for the purpose of optimizing the performance of a specific sequential logic function. Moreover, it can assist the researchers and designers in developing rules which may satisfy some particular requirements or applications, such as fault-tolerant or testable *ASLC*'s designs.

ACKNOWLEDGMENTS

I would like to express my thanks to my major professor and advisor Dr. P. David Fisher for all of his assistance with my graduate research and the development of my thesis as well as the other members of my Graduate Guidance Committee: Dr. Jacob Plotkin, Dr. Michael A. Shanblatt, and Dr. Chin-Long Wey. In addition, I would like to thank my wife, my family, and my friends for their support, patience, and encouragement in my efforts to achieve a Ph. D. degree while pursuing a graduate education in the United States of America.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Sequential logic functions may be implemented in either clocked sequential logic circuit (*CSLC*) or asynchronous sequential logic circuit (*ASLC*) architectures. *ASLC*'s have several important intrinsic advantages over their *CSLC* counterparts. An *ASLC* sequential logic function is potentially faster since it does not have to wait for the arrival of a clock pulse before effecting a state transition [1-8]. In complex *CSLC*'s, clock skewing may limit overall performance [9], while this problem is not encountered in their *ASLC* counterparts. Moreover, *ASLC*'s generally require less space to implement since the basic functional primitives are gates, not gates and memory, as is the case for *CSLC*'s. However, for a given sequential logic function, the *ASLC* design process is much more complex and time consuming than that of the *CSLC* [10]. This drawback has led state machine designers to prefer *CSLC* architectures to their *ASLC* counterparts. Consequently, before the intrinsic advantages of *ASLC* architectures can be fully exploited, there exists a need for efficient *ASLC* design tools which significantly simplify the process of designing this class of sequential logic functions.

1.1 Motivation

For a given sequential logic function, the process of designing *ASLC*'s is significantly more complex than that of their *CSLC* counterparts. This is due in part to critical-race and hazard problems that are associated with *ASLC* architectures. But, through appropriate state assignments, the race conditions can be avoided [11-13]. Various types of hazards and the design of hazard-free sequential logic circuits have been widely studied, and methods have been developed to identify and eliminate them [14-16]. Although it can be a very tedious process, the *ASLC* designer can verify that a particular design is functionally correct, as well as race and hazard free [17,18]. Some *CAD* tools can be used to simulate the designed circuits to assist in the design verification process, e.g., *VHDL*, *CALCAD*, or Schematic Editor [19]. But, because of the complexity and difficulity of the overall *ASLC* design process, only the simplest sequential logic functions have been implemented using *ASLC* architectures. An automated set of design tools would enable state machine designers to evaluate alternative sequential logic function architectures more thoroughly before committing the design to a specific architectural implementation.

The complexity and difficulity of designing *ASLC*'s involves the following:

1. The table size of a primitive flow table (*PFT*): The *PFT* describes the transitions between all the possible allowed states. Many inputs and outputs can make the flow table unmanageably large.
2. The generation of a merged flow table (*MFT*): In the merging process, finding the strongly connected subsets from a *PFT* (especially from a large *PFT*) is a very difficult task. This involves some relative techniques and concepts in graph theory.
3. The elimination of race conditions: The difficulties of making a proper state assignment to avoid race conditions are well known. Generating race-free state assignments and modifying the *MFT* are very difficult tasks especially for a large *MFT*.

4. The elimination of hazards: In order to generate hazard-free state equations and output equations, consensus terms must be added into those equations. For large excitation (or output) tables, finding consensus terms may become a more difficult and complex task.

The primary objective of the work reported here is to map human (expert) knowledge into the implementation of an *ASLC* design system to speedup the overall *ASLC* design process. This system would lessen the burden on sequential logic function circuit designers by greatly reducing the chances that errors would creep into the design and by greatly reducing the overall design cycle time. This increase in designer productivity could be used in part to explore alternative implementations for purposes of optimizing the overall circuit's performance. By providing the opportunities of investigating many different implementations, researchers and designers may identify some rules for determining a better *ASLC* design to fit some particular requirements or applications, for example fault tolerance design, testable design, or the asynchronous control parts in the *ASIC* design.

An *ASLC* design automation system (*MSUASLC*) that achieves this objective has been developed and tested. Software was written in the C programming language with approximately 20,000 source code; and while the current version of this software runs on Sun workstations, it is readily transportable to other platforms.

1.2 System Overview

The *ASLC* design automation system (*MSUASLC*) is illustrated in Figure 1-1. It provides a view of the design hierarchy traversed by the *ASLC* design system. It consists of five modules, each of which can accept data files from either an up-stream module or interactively from the circuit designer [20,21]. These modules perform the following functions:

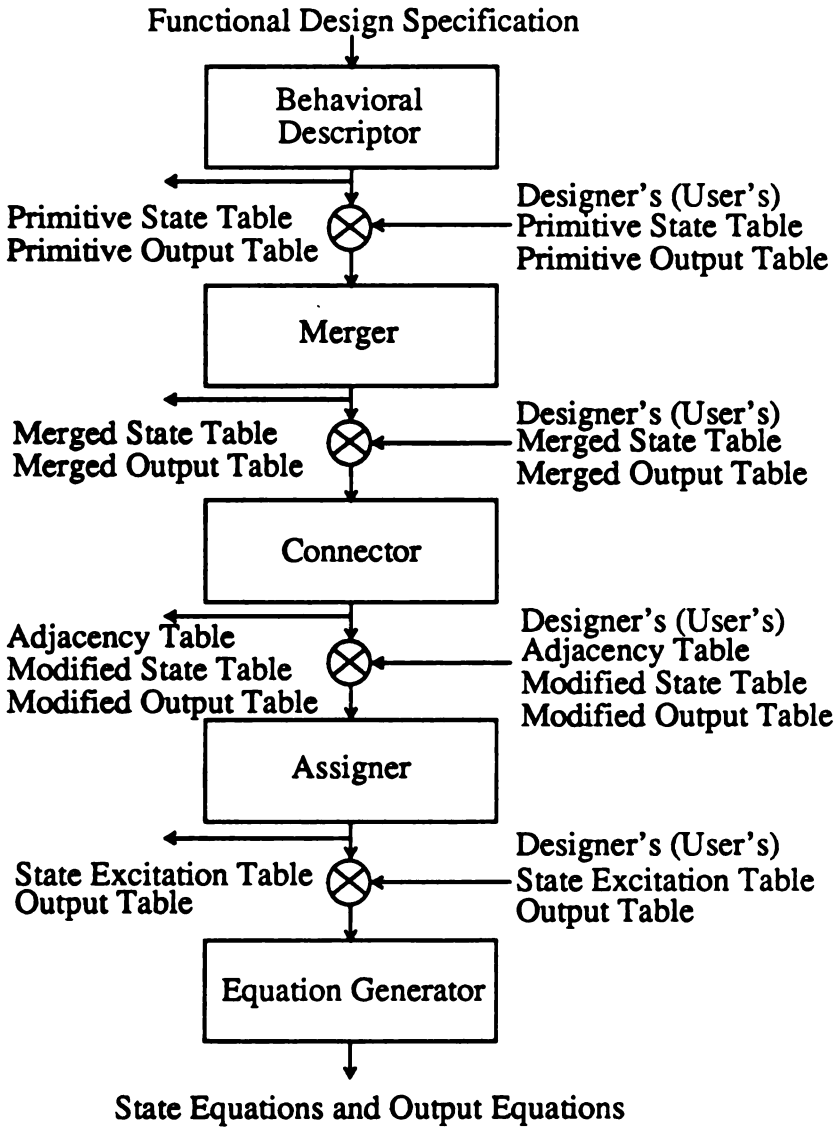


Figure 1-1. Configuration of the *ASLC* design system.

Behavioral Descriptor: maps the functional design specification into a primitive flow table (*PFT*), which completely captures the sequential logic function's behavioral model [22].

Merger: reduces the *PFT* into a merged flow table and merged output table, thereby minimizing the number of states by eliminating redundant primitive state assignments.

Connector: develops a state adjacency table from the merged flow table and adds cycles and states as needed to avoid critical races. It also produces a modified state table and modified output table.

Assigner: encodes the states and generates the state excitation table and corresponding output table from the information stored in the adjacency table, modified merged flow table, and previously developed modified output table.

Equation Generator: eliminates static hazards and converts the state excitation table and output table into state equations and output equations, which are placed in two-level, sum-of-products form.

This modular *CAD* system architecture has clearly defined entry and exit points and permits each module (sub-system) to be accessed independently. Therefore, concurrent execution of these modules can be achieved for different design tasks.

The *Functional Design Specification* (see Figure 1-1) includes an external input specification, an external output specification, and a sequential logic function specification Each *ASLC* input logic line can be placed into one of three categories: edge-control (trigger) input, level-control input, or data input. Edge-control inputs can cause an internal state transition when the inputs change from low to high (rising edge) or from high to low (falling edge) [23]. Level-control inputs do not cause state transitions to occur but may determine what the next internal state will be. Data inputs do not effect the internal state of the *ASLC*, only the present output state. In fact, all three types of inputs may effect the

present state of the output. The *ASLC*'s functional behavior is completely specified by the designer with the *Functional Design Specification.*

The *ASLC* design system illustrated in Figure 1-1 translates this high-level *Functional Design Specification* into a set of *State Equations and Output Equations.* Each of the five modules shown represent key steps in this translation process.

1.3 Outline

This thesis is organized as follows: Chapter 2 provides an artificial intelligence approach to model the functional behavior of *ASLC*'s. The first module (*Behavioral Descriptor*) of the *MSUASLC* is described. The algorithm for *BD* is also provided. Chapter 3 describes the second module (*Merger*) of the *MSUASLC*. Four different merging methods are developed for generating *MFT*'s. A traditional merging method (by merging largest strongly connected subgraph first), which is not included in the *Merger*, is also discussed. Some techniques related to the graph theory are also embedded into the algorithms for the *Merger*. Chapter 4 provides some new techniques for avoiding races and for generating race-free state assignments. Some new concepts dealing with the race conditions, which include **intrinsic races** (*IR*'s) and **generated races** (*GR*'s) are introduced. Several algorithms are developed and encoded into the *Connector* and *Assigner*. A graph called the **Node Weight Diagram** (*NWD*) helps ensure that the minimum or near minimum number of state variables and states are used to generate race-free state assignments. The experimental results of our approach are compared with other approaches in terms of **computation time** and **the number of state variables**. Details of the *Equation Generator* are provided elsewhere [24]. Chapter 5 contains a summary of this research work and some recommendations for future research directions.

Chapter 2

Behavioral Descriptor

with Artificial Intelligence Approach

There are several ways to describe an *ASLC*'s functional behavior. One way is to use timing diagram. However, drawing a timing diagram is not an easy way to include all the possible behavior (input/output combination), especially for any but the simplest *ASLC*. Another way is to use a state diagram. However, if you don't know how many essential states for a given design specification, you cannot draw a state diagram. In fact, it is very difficult to obtain a simplified state diagram [3-4, 6]. The most common way used to describe the *ASLC* behavior is to construct a primitive flow table.

A method has been developed which uses artificial intelligence techniques to model the functional behavior of *ASLC*'s. It provides a highly structured, interactive approach. The domain representation, production rules, and control strategies are described. The *Behavioral Descriptor* (*BD*) of the *ASLC* design automation system generates a primitive flow table which captures the *ASLC*'s functional behavior.

The application of Artificial Intelligence (*AI*) to digital system design has been widely studied by many researchers [25-30], and many *AI* applications in engineering are concentrated on *VLSI* design [31]. An artificial intelligence (*AI*) approach to the behavioral modeling of *ASLC*'s is described here. A "behavioral descriptor" (*BD*) is built to describe the whole circuit behavior with a primitive flow table (*PFT*). The *BD* accurately generates a *PFT* in a very short period of time, so that it lessens the burden on circuit designers and reduce the design cycle time. This feature becomes more important when large-scale *ASLC*'s must be designed.

In Figure 1-1, the highest level concerning *ASLC* design system is the *Behavior Descriptor* [20, 21]. At this level the input/output behavior of an *ASLC* state machine is described. This chapter deals primary with this level.

2.1 The Behavioral Descriptor

Constructing a *PFT* is the first important step in designing an *ASLC*. The inputs of this domain problem are the inputs, outputs, and function of the desired *ASLC*, and these must be provided by the designer. The output (result) of this domain problem is a primitive flow table which describes the behavior of the desired *ASLC* of the designer. Therefore, the main idea is to build a problem solver, which is named the "behavior descriptor" (*BD*). The *BD* will map the designer's design specification into a *PFT*.

The *ASLC* behavioral model includes two basic sets of parameters: *ASLC* inputs and outputs. Each primitive state corresponds to a unique allowed input/output combination. Four important parts in the *BD* are described [22]:

(1) ***Design Specifications***: consist of three high level specifications; namely, the input specification, the output specification, and the functional specification.

(2) ***Facts***: are states represented by the input/output combinations and the corresponding

state numbers which are generated by control strategies.

(3) ***Production Rules***: are in the form of "IF conditions THEN actions" clauses. There are no built-in production rules for *BD*. All the production rules are generated according to the design specification.

(4) ***Control Strategy***: controls state transitions in the *PFT* and generates the next *PFT* entries. They determine which production rule is to be fired by matching each pattern in the condition part of a rule with the changing pattern in the input signals. The control procedure looks for the facts and fills the state number into the corresponding entry in the *PFT* until all the entries have been filled with the assigned values.

2.1.1 Representation

According to Rich [32], it is often useful to divide the representation question into three subquestions:

(1) How can individual objects and facts be represented?

(2) How can the representations of individual objects be combined to form a representation of a complete problem state?

(3) How can the sequences of problem states that arise in a search process be represented efficiently?

Our specific criteria was to choose a representation that allows all of the necessary knowledge to be represented and facilitates its use in solving the problem at hand. Moreover, we want a representation that will be simple, informative, and easily used interactively on a computer system. Our specific implementation is as follows:

Input Representation for the BD

The inputs to the *BD* is the design specification of the *ASLC*. The format for describing the design specification is described as follows:

The input specification has the following format:

input_specification

in_name_1 in_name_2 in_name_3 ... in_name_n

The output specification has the following format:

output_specification

out_name_1 out_name_2 out_name_3 ... out_name_m

The sequential logic functions's functional behavior is completely specified by the designer and has the following format:

function_specification

IF in_name_1=state in_name_2=state ...

THEN out_name_1=function_1 out_name_2=function_2 ...

- ***Internal Representation of Facts for the BD***

Following is the definition for the attributes of each entry in the *PFT*.

row_no: specifies the entry's row.

column_no: specifies the entry's column.

state_no: specifies the entry's primitive state number.

stable: specifies whether the entry is in a stable state or not.

output_value: specifies the value of the output signal combination associated with a primitive state.

2.1.2 Production Rules

Many AI applications today employ some form of if-then rule-based programming; i.e., if conditions $C_1, C_2, ..., C_n$ happen, then actions $A_1, A_2, ..., A_m$ will be performed. Most of the VLSI design tools have built-in production rules which are fixed [26-28]. MYCIN provides production rules which can be modified [32]. However, there is no built-in

production rules for *BD*. All the production rules are generated according to the design specification. Therefore, it is a dynamic structure.

Some properties for the rules in *BD* are:

1. Each rule should be unique. There should not be same rules existing in the production rules of *BD*.

2. Rules should not conflict each other.

3. One rule should not cover another rule.

Violating property 2 or property 3 may cause an unexpected result. Violating property 1 does not effect the result, but it takes unnecessary space in the system. In order to have a desired design, we must pay much more attention on our functional design specification at the beginning of the design.

You may have many "IF ... THEN ..." statements in your functional-specification. However, one must be aware that there is no conflict or inclusion between each of them. For example, assume you have two "IF ... THEN ..." statements which have same "IF conditions" and different "THEN actions". These two statements conflict each other. Suppose in the two "IF ... THEN ..." statements, one "IF conditions" is covered by another "IF conditions". This violates the property 3 and should be avoided in order to generate a desired *PFT*.

2.1.3 Control Strategy

The control strategy manipulates the representation by matching one of the production rules. Each time it looks for one signal changing in the input state of the *ASLC* and searches for the matched production rule. The policy for firing a rule is to take the first matched rule and fire it. Each time only one rule is active. If a match occurs, it takes the appropriate action and generates the next value of output signals. At this time, the

combination of input signals and output signals forms a new primitive state. The control procedure decides whether this state currently exists. If it is a new state, the control procedure assigns it a new value. Next, the control procedure places the value in the corresponding row and column in the *PFT*. This process continues until all rows and columns have been filled with the assigned primitive state values or "don't cares" values, where "don't cares" indicate that the particular primitive state transition does not occur.

2.2 Algorithm for the BD

The algorithm for the *BD* is provided in the following.

Step 1. Specify input:

1. circuit input names,
2. circuit output names,
3. circuit function (behavior).

Step 2. Generate rules.

Step 3. Initialize I/O combination table and primitive flow table (*PFT*).

Step 4. Give a possible I/O combination value as the starting primitive state and mark it stable to start the operation for filling the *PFT*.

Step 5. Determine the next "move" position which is in the same row as the primitive stable state.

Step 6. Search the rules:

If matched then fire the rule to compute the outputs else keep outputs unchanged.

Step 7. Determine the primitive state number and do the following things.

1. Put the primitive state number into the "move" position in the *PFT* and mark it unstable.

2. If it is a new primitive state then find a blank row in *PFT* and put it into the same column as the "move" position, and mark it as a stable state in the new position.

Step 8. Repeat step (5) to (6) for all the adjacent "moves" in the same row.

Step 9. Repeat step (5) to (8) for all the primitive stable states until all the rows have been checked.

2.3 Design Examples and Discussion

The *BD* has been applied on many *ASLC* design examples. Some examples have quite large *PFT*'s, which cannot be shown here. In order to explain how the *BD* works, a simple example (example 2.1) is first used to illustrate it.

Example 2-1: *P-SLF*

This example deals with a specific sequential logic function (*SLF*) illustrated in Figure 2-1. This logic element, referred to as a *P-SLF*, is a multi-stable sequential logic function and has the following set of *Functional Design Specifications*:

(1) There are two input logic lines, which are labeled A and B.

(2) There are two output logic lines. They are labeled D or d and E or e, where the upper-case and lower-case letters are used to denote next-output and current-output state, respectively.

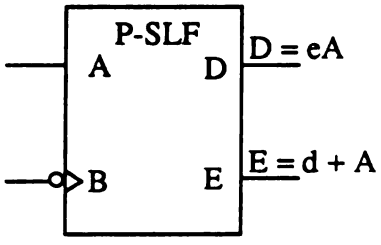


Figure 2-1. Graphic symbol for the *P-SLF*

(3) The output logic lines change state only on the falling edge of B, and the next output states are defined as follows:

D = eA; E = d + A.

The ***Behavioral Descriptor*** *maps the Functional Design Specification* into the *Primitive Flow Table*. The *Functional Design Specification* must first be read from a file or entered interactively by the circuit designer. The interactive approach would be as follows:

1. Enter the sequential logic function's input variable names.

input_specification

in_name_1 in_name_2 in_name_3 ... in_name_n

B A

2. Enter the sequential logic function's output variable names.

ouput_specification

out_name_1 out_name_2 out_name_3 ... out_name_m

D E

3. Enter the sequential logic function's functional behavior, which describes the relationship between the next output state and the present output state and present and past input states.

function_specification

IF in_name_1=state in_name_2=state ...

IF B=3 (Here, we use 3 to represent 1-to-0)

THEN out_name_1=function_1 out_name_2 =function_2 ...

THEN D=E&A E=D+A.

Once the *Functional Design Specification* has been entered into the *MSUASLC*, the *BD* generates primitive state transition rules. For example, if B in the above example makes a

transition from 1-to-0, then the next output state is updated as indicated above; otherwise, there is no change.

After these rules have been generated, the primitive flow table (*PFT*) is constructed and completed. If the sequential logic function has m input lines and n output lines, then 2^{m+n} primitive states are possible with each of these primitive states being assigned a row in the *PFT*. There would be 2^m possible distinct input states with each of these states being assigned a unique column in the *PFT*. For the *P-SLF*, m = 2 and n = 2; so, the *PFT* would be expected to contain up to sixteen rows and four columns.

The *PFT* is initially empty. The process used to complete the *PFT* is as follows: One of the allowed I/O combinations is arbitrarily selected as the starting primitive state and marked stable to initiate the process of filling the *PFT*. For the *P-SLF* example, BADE = 0000 satisfies the Functional Design Specification and represents a stable primitive state; so, this primitive state can be selected as a starting point. From this starting point, the logic state of one input line is changed and the rules applied to compute the "next primitive state". All allowed input transitions are explored. Not all combinations of the input and output logic states may be allowed by the Functional Design Specification. For example, for the *P-SLF*, BADE = 1010 is not allowed since E = d + A. Moreover, only those primitive state transitions are allowed for which one and only one input logic line changes state. For example, for the *P-SLF*, BADE = 0000 $\Rightarrow$ 1101 is not allowed because input logic lines A and B were changed simultaneously. The *PFT* is completed after each allowed primitive state has been visited and its row in the *PFT* completed.

Once the *Functional Design Specification* has been entered (or read), the *BD* automatically generates the *PFT* (see Figure 2-2). For the *P-SLF*, twelve primitive states are delineated with stable primitive states being "marked". (For illustrative purposes, we have circled the stable states in Figure 2-2). The computer (*BD*) generated primitive state table and primitive output table are also listed in Table 2-1 and Table 2-2, respectively. The

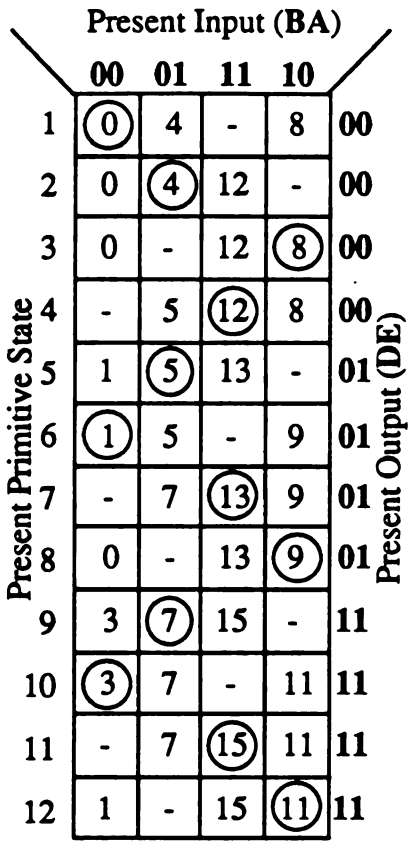


Figure 2-2. Primitive flow table for the *P-SLF*

"Y, "N", and "-" entries in the primitive state table are abbreviations that stand for "stable state", "unstable state", and "don't care" respectively. From the *PFT*, the circuit designer can readily identify all of the primitive state assignments. For example, primitive state 12 is defined as having present-input and present-output states of BA = 11 and DE = 00, respectively. From this figure we also see that the *ASLC* design system imposed the fundamental-mode rule on the input states; i.e., only one input logic line may change state at a time. The circuit designer may use this *PFT* output from the *BD* module to provide documentation of the design or to verify manually the correctness of the result. But the latter is not generally necessary since the *BD* in the *ASLC* design system guarantees the correctness of the *PFT* according to the given design specification.

Table 2-1. Primitive State Table for *P-SLF*

Present Primitive State	INPUT (BA) 0	1	3	2	OUTPUT (DE)
	0 Y	4 N	-	8 N	0
	0 N	4 Y	12 N	-	0
	0 N	-	12 N	8 Y	0
	-	5 N	12 Y	8 N	0
	1 N	5 Y	13 N	-	1
	1 Y	5 N	-	9 N	1
	-	7 N	13 Y	9 N	1
	0 N	-	13 N	9 Y	1
	3 N	7 Y	15 N	-	3
	3 Y	7 N	-	11 N	3
	-	7 N	15 Y	11 N	3
	1 N	-	15 N	11 Y	3

Table 2-2. Primitive Output Table for *P-SLF*

Present Primitive State	INPUT (BA) 0	1	3	2	Output (DE)
	0	0	-	0	
	0	0	0	-	
	0	-	0	0	
	-	1	0	0	
	1	1	1	-	
	1	1	-	1	
	-	3	1	1	
	0	-	1	1	
	3	3	3	-	
	3	3	-	3	
	-	3	3	3	
	1	-	3	3	

Example 2-2: *DIV-3* (alternative design specification)

The second example deals with a sequential logical function which has one input called C (Clock), one output called Q. We refer to this circuit as a divide by 3 (*DIV*-3, see Figure 2-3(a)). Every third clock transition causes the output Q to toggle. The timing diagram is shown in Figure 2-3(b).

In order to use *BD* to generate the *PFT*, we introduce two pseudo outputs G and H which can identify the first and second clock transitions so that the output Q can identify the third clock transition. The function specification is as follows:

IF C=2 (2 means 0-to-1)

THEN G=$\overline{Q}$, H=G, Q=H

IF C=3 (3 means1-to-0)

THEN G=$\overline{Q}$, H=G, Q=H

The primitive state table is shown in Table 2-3, and the primitive output table is

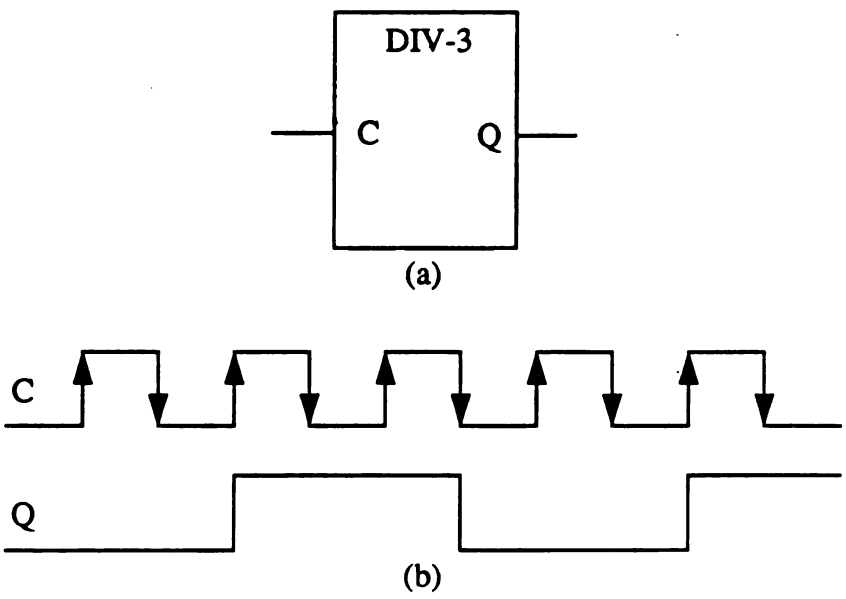


Figure 2-3. *DIV-3* (a) graphic symbol, (b) timing diagram

shown in Table 2-4. An alternative function specification, which use only one pseudo output G, is as follows:

IF C=2 (2 means 0-to-1)

THEN $Q=\overline{Q}G+Q\overline{G}$, $G=C\overline{Q}+\overline{C}Q$

IF C=(3 means1-to-0)

THEN $Q=\overline{Q}G+Q\overline{G}$, $G=C\overline{Q}+\overline{C}Q$

Table 2-3. Primitive State Table for *DIV-3*

	INPUT (C) 0	INPUT (C) 1	OUTPUT (GHQ)
Present Primitive State	0 Y	12 N	0
	0 N	9 Y	1
	3 Y	9 N	3
	6 Y	15 N	6
	3 N	15 Y	7
	6 N	12 Y	4

Table 2-4. Primitive Output Table for *DIV-3*

	INPUT (C) 0	INPUT (C) 1	
Present Primitive State	0	4	
	0	1	
	3	1	
	6	7	
	3	7	Output (GHQ)
	6	4	

The corresponding alternative primitive state table is shown in Table 2-5, and the alternative output table is shown in Table 2-6.

Table 2-5. Alternative Primitive State Table for *DIV-3*

Present Primitive State	INPUT (C) 0	INPUT (C) 1	OUTPUT (QG)
	0 Y	4 N	0
	1 N	4 Y	0
	1 Y	6 N	1
	0 N	7 Y	3
	2 N	6 Y	2
	2 Y	7 N	2

Table 2-6. Alternative Primitive Output Table for *DIV-3*

Present Primitive State	INPUT (C) 0	INPUT (C) 1	Output (QG)
	0	0	
	1	0	
	1	2	
	0	3	
	2	2	
	2	3	

Example 2-3: *Gated Oscillator*

There are many applications where we would like to be able to turn a clock on and off using a manual switch. Usually a clock consists of pulses occurring at some fixed rate. We might be tempted to solve this problem simply by ANDing the switch with the clock. The difficulty is that since the switch is not synchronized to the clock, we might turn the switch on in the middle of a pulse and in so doing produce an output pulse shorter than that required by whatever system is being driven by the clock. Similarly, our switch might turn off in the middle of a pulse. Thus, what we want to do is to design an *ASLC* that will produce an output Z which is a complete clock as long as the switch G is on, regardless of when the switch was turned on or off. This circuit is referred as gated oscillator [3]. The schematic symbol of gated oscillator is shown in Figure 2-4, and the timing diagram is shown in Figure 2-5. There are two inputs, G (switch) and C (clock), one output Z, where C is an edge control input.

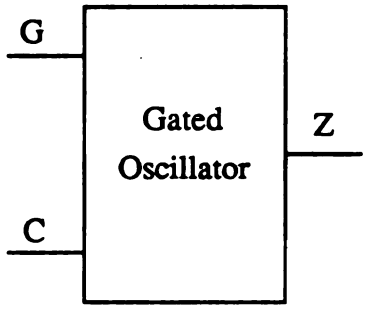


Figure 2-4. Graphic symbol for the gated oscillator

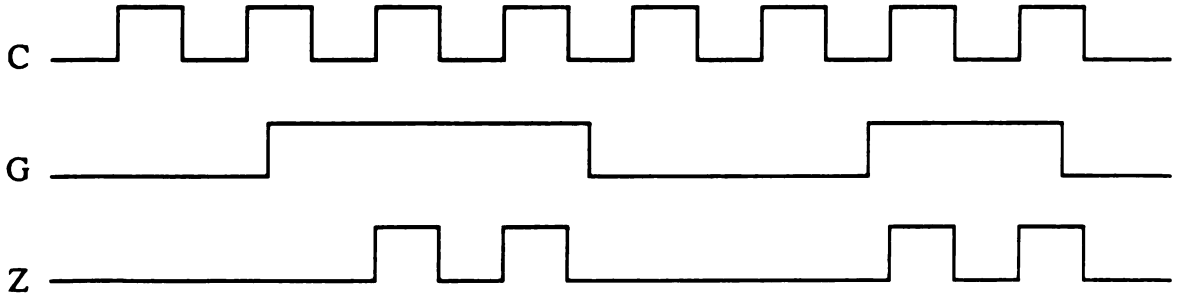


Figure 2-5. Timing diagram for the gated oscillator.

The function specification of this design description in the following.

IF C=2 (C changes state from 0 to 1)

THEN Z=G

IF C=3 (C changes state from 1 to 0)

THEN Z=0

The primitive state table is shown in Table 2-7, and the primitive output table is shown in Table 2-8.

Table 2-7. Primitive State Table for Gated Oscillator

	INPUT (CG)				OUTPUT
	0	1	3	2	(Z)
Present Primitive State	0 Y	2 N	-	4 N	0
	0 N	2 Y	7 N	-	0
	0 N	-	6 N	4 Y	0
	-	2 N	6 Y	4 N	0
	-	2 N	7 Y	5 N	1
	0 N	-	7 N	5 Y	1

Table 2-8. Primitive Output Table for Gated Oscillator

Present Primitive State	INPUT (CG) 0	1	3	2	Output (Z)
	0	0	-	0	
	0	0	1	-	
	0	-	0	0	
	-	0	0	0	
	-	0	1	1	
	0	-	1	1	

2.4 Conclusions to the BD

Design is the process of refining a representation at one abstraction level into a more detail representation at a lower abstraction level. The design of *ASLC* can be a very difficult and time consuming process because of the large number of inputs/outputs, hazards, and critical races. Many inputs/outputs can make flow tables unmanageably large. The *BD* overcomes these complexity problems.

Once the *PFT* has been generated by the *BD*, the complete behavior of the designed *ASLC* is controllable and observable. Since we can easily predict the next outputs from the current state and the next inputs by using *BD*, there is no need to draw the timing diagram for knowing the *ASLC* behavior. The *BD* helps us quickly produce the *PFT* without using pen-and-paper methods. It provides the information about the circuit behavior and speedup the *ASLC* design process.

Chapter 3

Merger: The Merged Flow Table Generator

The second step in designing *ASLC*'s is to reduce the primitive flow table (*PFT*) to a table containing as few rows as possible. In order to achieve this purpose, a merged flow table generator called ***Merger*** has been developed. The reason to reduce the *PFT* is that the flow table will eventually become the excitation table, in which the number of rows determines the number of state variables and, therefore, the complexity of the implementation [3].

The ***Merger*** module reads the *PFT* and produces a merged flow table (*MFT*). In general, this translation process eliminates redundant primitive state assignments and leads to a more efficient implementation of the *Functional Design Specification*. Different merging processes are included in the ***Merger*** for generating alternative *MFT*'s. The ***Merger*** will choose the *MFT* with a minimum number of rows as the resulting *MFT* to be used by the ***Connector*** in *MSUASLC*. The resulting *MFT* will have the same number of columns as the primitive flow table since the number of unique combinations of the input states remain the same, but the number of rows may be less, indicating that some of the

primitive states were redundant. The merged state assignments are also used in conjunction with the *PFT* to produce an output table which relates the present output logic states to the present merged state and present input logic states. In order to investigate different implementations for some particular applications or requirements, designers can choose any one of the alternative *MFT*'s as the resulting *MFT*.

3.1 The Merged Flow Table

The *PFT* can be reduced to an *MFT* through the merging process. A **merger diagram** is used to show which rows in *PFT* can be merged. An example of merger diagram is illustrated in Figure 3-1 which is corresponding to the *PFT* shown in Table 3-1. In the merger diagram, the row number (state) is given inside the circle and the output corresponding to this row is shown adjacent to the circle. The circled entries are referred to as **nodes**. Therefore, each node is corresponding to a row in a *PFT*. If two rows in a *PFT* can be merged, the corresponding two nodes in a merger diagram are connected by a **line** (or **edge**). A set of nodes is said to be **"strongly connected"**, if for every node in the set is connected to all the other nodes of the same set. For example, Figure 3-2 shows the strongly connected groups for different number of states.

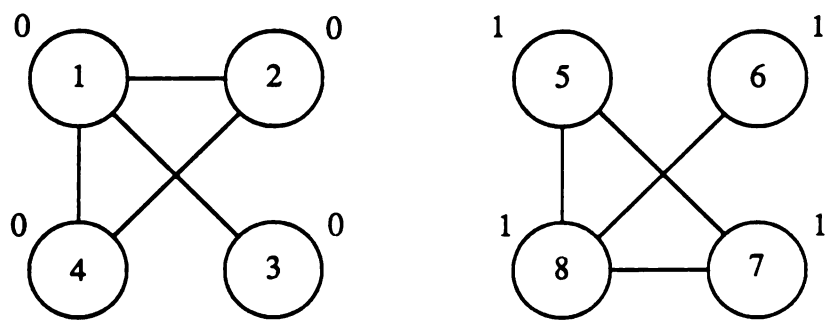


Figure 3-1. Merger diagram for the *PFT* of Table 3-1

Table 3-1. A Primitive State Table

Present Primitive State	(INPUT (TC) 0	1	3	2	OUTPUT (Q)
	0 Y	2 N	-	4 N	0
	0 N	2 Y	6 N	-	0
	0 N	-	7 N	4 Y	0
	-	2 N	6 Y	4 N	0
	-	3 N	7 Y	5 N	1
	1 N	-	6 N	5 Y	1
	1 N	3 Y	7 N	-	1
	1 Y	3 N	-	5 N	1

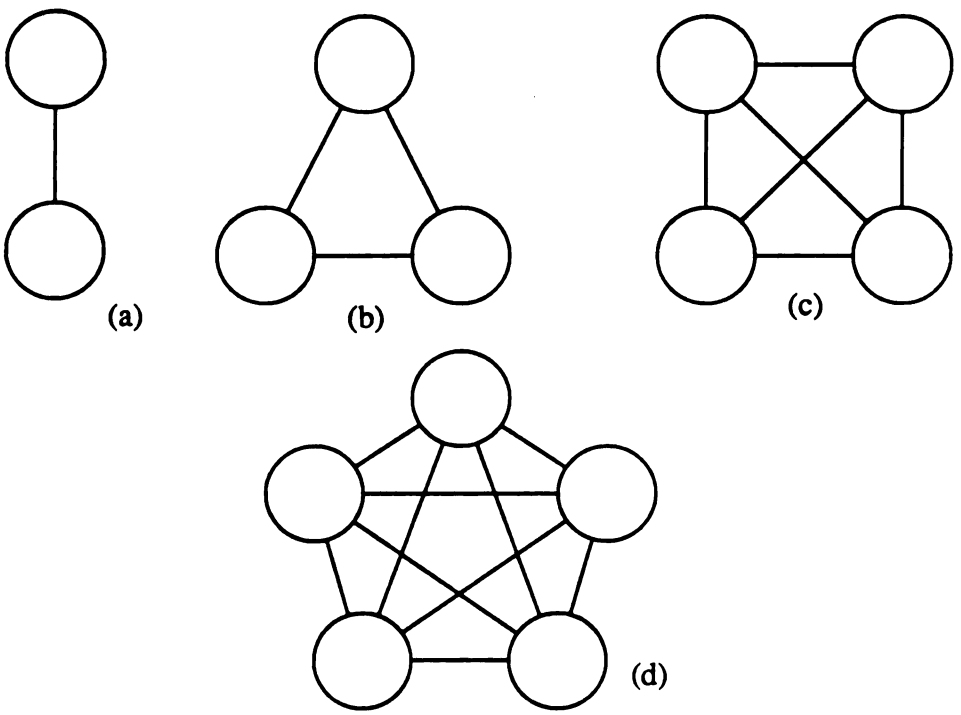


Figure 3-2. Strongly connected groups (a) 2 nodes; (b) 3 nodes; (c) 4 nodes; (d) 5 nodes.

The **rules** which are used to merge rows are described in the following:

1. Two rows can be merged if in each column either the state labels are the same, or one or both entries are "don't cares".
2. A set of rows can be merged into a single row if the set is strongly connected in the merger diagram.
3. When two rows are merged, a stable entry and an unstable entry become stable, and two unstable entries stay unstable. Continuing to compare the rows in pairs, we can merge a set of rows.

Applying the above rules to the Table 3-1, we can merge rows 1, 2, and 4 into one row based on the merger diagram shown in Figure 3-1. Similarly, rows 5, 7, and 8 can be merged into one row. The resulting merged state table and merged output table are shown in Table 3-2 and Table 3-3, respectively.

Table 3-2. A Merged State Table

	(INPUT (TC)			
	0	1	3	2
Merged State	0 Y	2 Y	6 Y	4 N
	0 N	-	7 N	4 Y
	1 Y	3 Y	7 Y	5 N
	1 N	-	6 N	5 Y

Table 3-3. A Merged Output Table

INPUT (TC) 0	1	3	2	Output (Q)
0	0	0	0	
0	-	1	0	
1	1	1	1	
1	-	0	1	

3.2 Alternative Merging Methods in the Merger

A given primitive state may be capable of being merged in more than one way, but primitive states must only be assigned to one of these merged states. Therefore, additional constraints are necessary in order to decide which grouping is best. One such constraint might be to require that only primitive states with identical output states be merged.

There are four different merging methods provided in the *Merger*. The first method (method 1) is to merge rows with only the same outputs from the first row to the last row in the *PFT*. The second method (method 2) is to merge rows with the same outputs but starting from the rows (nodes) with minimum link degree (i.e., least strongly connected subsets first). Here, the link degree is defined as the number of edges (lines) attached to a node in the merger diagram. For example, the link degree of node 1 in Figure 3-1 is 3, but the link degree of node 3 is 1. The third method (method 3) is to merge rows starting from the minimum link degree (least strongly connected subsets first) regardless if the merged

rows have the same outputs or not (merging rows with mixed outputs). The fourth method (method 4) is to first merge rows with the same outputs, then extend to different outputs if possible.

Another method is to identify the largest strongly connected subsets of these rows which can merge into a single row [3]. This method will be referred to as method 5. Method 5 is not used in the *Merger* because it takes too much computation time (effort), especially when the *PFT* is large. Usually, the resulting *MFT* of method 5 has the same number of merged rows as that of method 3 (identifying the least strongly connected subsets). For example, in Figure 3-3, we can have two merged groups (1, 2, 3) and (4) by using method 5 (identifying the largest strongly connected subsets). If we use method 3, we can also get two merged groups (3, 4) and (1, 2). In Figure 3-4, both method 5 and method 3 have the same two merged groups (1, 2, 4) and (3, 5). However, for the merger diagram shown in Figure 3-4, if the first selected, largest strongly connected subset is (1, 2, 3), there will be a total of three merged groups: (1, 2, 3), (4), (5) as shown in Figure 3-5. In this case, we

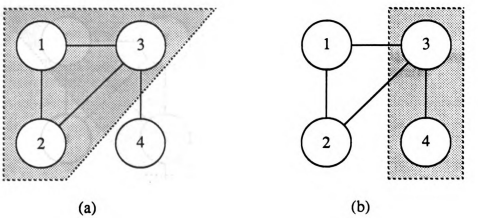


Figure 3-3. Merger diagram (4 nodes) with two different merging methods
(a) method 5 (largest strongly connected subsets first);
(b) method 3 (least strongly connected subsets first).

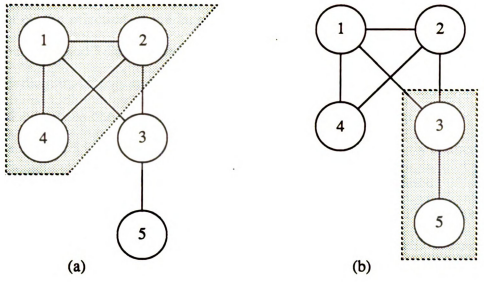
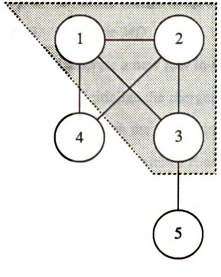


Figure 3-4. Merger diagram (5 nodes) with two different merging methods (a) method 5 (largest strongly connected subsets first); (b) method 3 (least strongly connected subsets first).



Method 5 (identifying largest strongly connected subsets) generates three merged groups:

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

Figure 3-5. A possible merging approach by using method 5 for this merger diagram.

cannot get minimum number of merged groups. Therefore, the merged results are highly dependent on the selection of the largest strongly connected subsets. Similarly, this may happen to the method 3 (least strongly connected subsets). When multiple choices exist and the merger diagram is large and complex, it becomes very difficult to decide which strongly connected subsets to choose. From the computation complexity point of view, we would like to use method 3 instead of method 5, because finding a least strongly connected subset is much easier and faster than finding a largest strongly connected subset.

In general, the merging process in the *Merger* is as follows: The merger diagram (*MD*) and the merged flow table (*MFT*) are initialized [20]. The *MD* is completed by identifying identical rows in the *PFT*. Each set of identical rows in the *PFT* becomes a new row in the *MFT* and is assigned a new symbolic state name. An algorithm for method 3 is listed in the following. Similarly, the algorithms for method 1, 2, and 4 can be generated by modifying the Step 4 of the following algorithm.

Algorithm for Method 3 in the Merger

Step 1. Read *PFT*

Step 2. Initialize merger diagram (*MD*) and merged flow table (*MFT*).

Step 3. Generate *MD*.

Step 4. Generate a new row in *MFT*:

1. Initialize the merged buffer (*MB*).
2. Search the node with minimum link degree in *MD*.
3. Merge matched rows with the node of minimum link degree.
4. Update *MD*.
5. Fill output value associated with each entry in the *MB*.
6. Link *MB* to *MFT*.

Step 5. Repeat step 4 until all the nodes in *MD* have been merged.

3.3 Examples

Example 3-1: *P-SLF*

This example shows how to merge the *PFT* shown in the Figure 2-2. The merger diagram for this *PFT* is shown in Figure 3-6. The merged groups for method 1 is shown in Figure 3-7. The merged state table and merged output table generated by method 1 are given in Table 3-4 and Table 3-5. The merged groups for method 2 is shown in Figure 3-8. The merged state table and merged output table generated by method 2 are given in Table 3-6 and Table 3-7. In this example, method 3 has just the same results as method 2, and method 4 has just the same results as method 1. This example shows that four merging methods have the same number of merged groups.

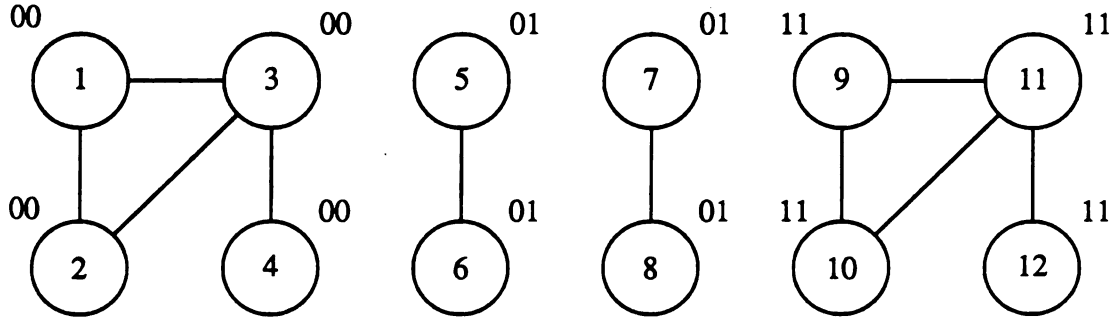


Figure 3-6. Merger diagram for the primitive flow table of Figure 2-1.

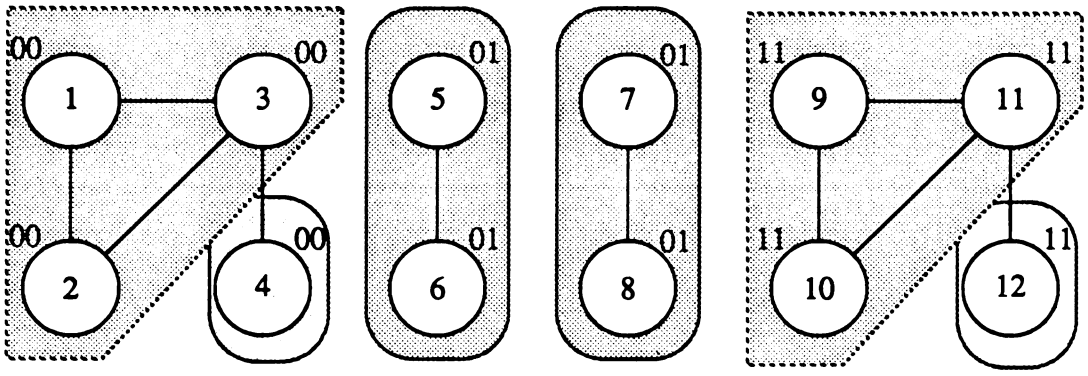


Figure 3-7. Merged groups for method 1.

Table 3-4. Merged State Table for *P-SLF* by Method 1

	INPUT (BA)			
	0	1	3	2
Merged State	0 Y	4 Y	12 N	8 Y
	-	5 N	12 Y	8 N
	1 Y	5 Y	13 N	9 N
	0 N	7 N	13 Y	9 Y
	3 Y	7 Y	15 Y	11 N
	1 N	-	15 N	11 Y

Table 3-5. Merged Output Table for *P-SLF* by Method 1

0	INPUT (BA)			
0	1	3	2	
0	0	0	0	Output (DE)
-	1	0	0	
1	1	1	1	
0	3	1	1	
3	3	3	3	
1	-	3	3	

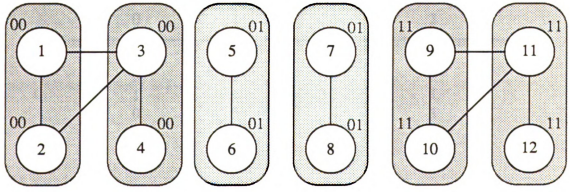


Figure 3-8. Merged groups for method 2

Table 3-6. Merged State Table for *P-SLF* by Method 2.

	INPUT (BA) 0	1	3	2
Merged State	0 N	5 N	12 Y	8 Y
	0 Y	4 Y	12 N	8 N
	1 Y	5 Y	13 N	9 N
	0 N	7 N	13 Y	9 Y
	1 N	7 N	15 Y	11 Y
	3 Y	7 Y	15 N	11 N

Table 3-7. Merged Output Table for *P-SLF* by Method 2.

INPUT (BA) 0	1	3	2	
0	1	0	0	
0	0	0	0	Output (DE)
1	1	1	1	
0	3	1	1	
1	3	3	3	
3	3	3	3	

Example 3-2: *DIV-3*

This example shows how to merge the *PFT* shown in the Table 2-3. The merger diagram for this *PFT* is shown in Figure 3-9. From this diagram, we can identify that the merged groups for method 1 are (1), (2), (3), (4), (5), and (6). There is no merging for any two rows. The merged state table and merged output table generated by method 1 are given in Table 3-8 and Table 3-9 which are the same as Table 2-3 and Table 2-4, respectively.

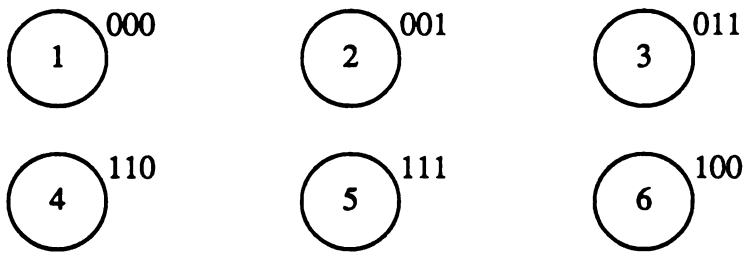


Figure 3-9. Merger diagram for the primitive flow table of Table 2-3.

Table 3-8. Merged State Table for *DIV-3* by Method 1

Merged State	INPUT (C) 0	1
	0 Y	12 N
	0 N	9 Y
	3 Y	9 N
	6 Y	15 N
	3 N	15 Y
	6 N	12 Y

Table 3-9. Merged Output Table for *DIV-3* by Method 1

	INPUT (C)	
	0	1
Output (GHQ)	0	4
	0	1
	3	1
	6	7
	3	7
	6	4

Using the other three methods will get the same merged groups, the same merged state table and the same merged output table. This example shows that four merged methods have the same merged results.

Example 3-3: *Gated Oscillator*

This example shows how to merge the *PFT* shown in the Table 2-7. The merger diagram for this *PFT* is shown in Figure 3-10. Applying method 1 to this merger diagram, we can get three merged groups: (1, 2), (3, 4), and (5, 6) (see Figure 3-11(a)). Apply method 3 to the merger diagram, two merged groups: (1, 3, 4) and (2, 5, 6) (see Figure 3-11(b)) will be obtained. The merged state table and merged output table generated by method 1 are given in Table 3-10 and Table 3-11. Applying method 2 and method 4 will get the same results as method 1. The merged state table and merged output table generated by method

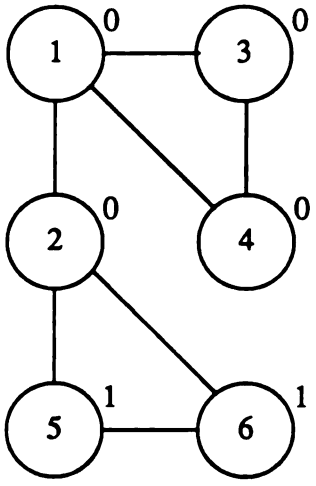


Figure 3-10. Merger diagram for the primitive flow table of Table 2-7.

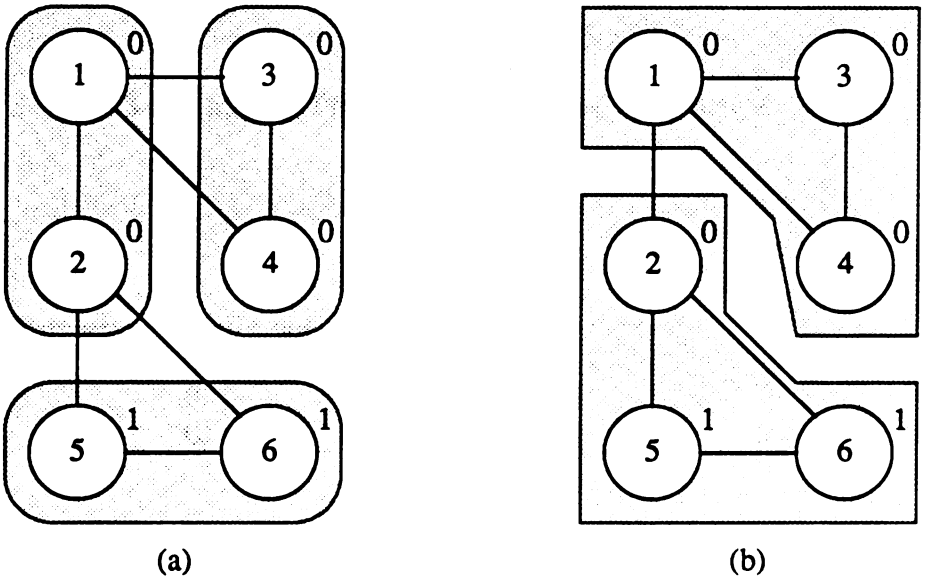


Figure 3-11. Merged groups generated by (a) method 1 (or 2, or 4); (b) method 3.

Table 3-10. Merged State Table for Gated Oscillator by Method 1

	INPUT (CG) 0	1	3	2
Merged State	0 Y	2 Y	7 N	4 N
	0 N	2 N	6 Y	4 Y
	0 N	2 N	7 Y	5 Y

Table 3-11. Merged Output Table for Gated Oscillator by Method 1

INPUT (CG) 0	1	3	2	
0	0	1	0	Output (Z)
0	0	0	0	
0	0	1	1	

3 are given in Table 3-12 and Table 3-13. This example shows that method 3 has a better merged result than the others. Here, a better merged result means that it has a fewer number of merged groups than the others.

Table 3-12. Merged State Table for Gated Oscillator by Method 3

	INPUT (CG) 0	1	3	2
Merged State	0 Y	2 N	6 Y	4 Y
	0 N	2 Y	7 Y	5 Y

Table 3-13. Merged Output Table for Gated Oscillator by Method 3

	INPUT (CG) 0	1	3	2	
	0	0	0	0	Output (Z)
	0	0	1	1	

3.4 Discussion

Four different merging methods are provided in the *Merger*. The major purpose is to give designers or researchers more opportunities to investigate the merged results. Different merging methods may give the same or different merged results, as we have seen in previous examples. The merged results depend not only on the merging method but also on the merging sequence. For example, if nodes 1, 2, and 4 in the Figure 3-4(a) are merged first, two merged groups (1, 2, 4) and (3, 5) will be obtained. However, if nodes 1, 2, and 3 are merged first, three merged groups (1, 2, 3), (4), (5) will be obtained as shown in Figure 3-5. It is possible to have many identical sizes of strongly connected subsets sharing common nodes in a given merger diagram. Thus, when using one method, it is still possible to have many merged results which are the same or different size. For example, assume we have a simple merger diagram as shown in Figure 3-12(a), Four identical sizes of merged results (see Figure 3-12(b), (c), (d), (e)) can be obtained by using either method 3, least strongly connected subsets first, or method 5, largest strongly connected subsets first.

Merging a large flow table is more time consuming than generating a large flow table. We cannot exhaustively search or generate all the possible merged results. In order to overcome the problem of combinatorial explosion, each merging method will generate one *MFT*. Four methods used in the *Merger* will generate four *MFT*'s, and the *Merger* will choose a minimum size *MFT* among the four *MFT*'s as the resulting *MFT* for the *Connector*. Designers or researchers can choose any one of the four *MFT*'s as the resulting *MFT* for the *Connector*. Designers or researchers can also merge *PFT* in some specific ways for some particular applications, requirements, or purposes.

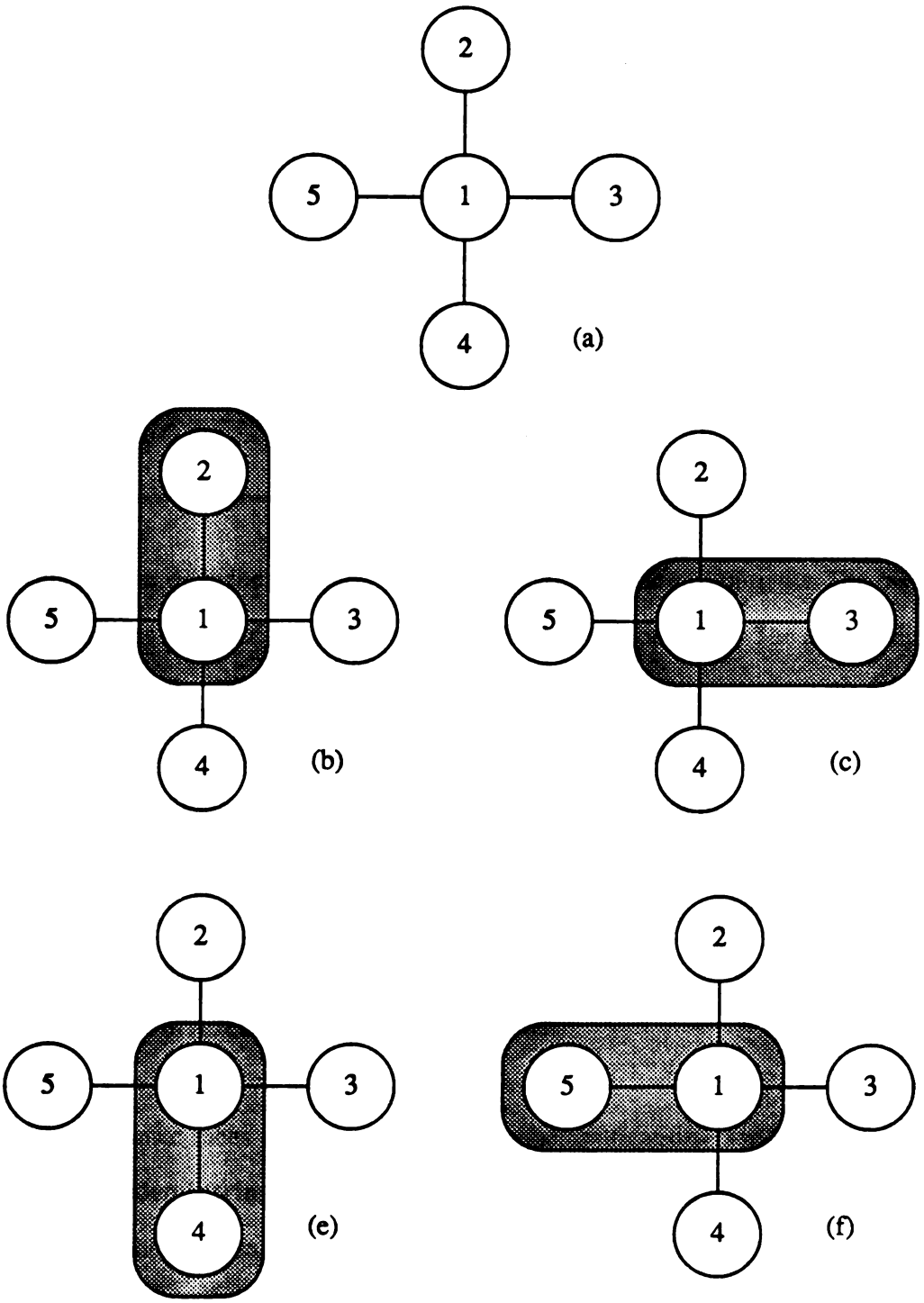


Figure 3-12. A merger diagram with four different merged results of identical size.

Chapter 4

Race-Free State Assignments

This chapter describes the process of making *ASLC* state assignments. The method of state assignments for *ASLC*'s is different from the method for *CSLC*. The state assignment procedure used in the *CSLC* design does not consider race conditions since the *CSLC* uses a clock control input to synchronize the circuit. However, the race conditions cannot be ignored in the designing of *ASLC*'s since it can cause the circuit to malfunction.

A new state assignment technique is introduced for synthesizing asynchronous sequential logic circuits (*ASLC*'s). It provides a systematic and efficient approach for generating race-free state assignments. This technique has been implemented and incorporated into an *ASLC* design automation system.

A race condition is classified as being either an **intrinsic race** (*IR*) or a **generated race** (*GR*). Intrinsic races decompose into two subclassifications: **visible intrinsic races** (*VIR*'s) and **hidden intrinsic races** (*HIR*'s). Algorithms have been developed to identify and eliminate these races. A graph, referred to as a **Node-Weight Diagram** (*NWD*), facilitates the process of making state assignments and guarantees that no races are generated. Moreover, it provides a convenient and efficient method for investigating the

implications of selecting from an allowed set of alternative race-free state assignments. The state assignment technique described adds cycles and states, as needed, to avoid *IR's* and always attempts to use the minimum or near minimum number of state variables and states.

This technique has been implemented and incorporated into the *MSUASLC* design automation system. Experimental results show that it provides significantly better results than other approaches in terms of the computation time required to make the assignments and the number of state variables required to achieve race-free *ASLC's*.

This chapter presents a systematic approach for identifying potential race conditions and for dealing with them so that they do not cause the state machine to malfunction. Specific objectives of the work reported here are four-fold: characterize and classify the different types of possible race conditions; develop efficient procedures for identifying possible race conditions; develop formal rules and strategies for eliminating race conditions that might lead the state machine to malfunction; and, finally, integrate these results into the *MSUASLC* design-automation system, and compare its performance with alternative approaches.

4.1 General Concepts on Race Conditions and Cycles

In this Section, some general concepts on race conditions and cycles are reviewed. The internal state (state) of an *ASLC* can be represented by an n-digit binary data word y. State transitions occur in response to changes in the *ASLC*'s input state (input). Let y_α and y_β represent the present and next states, respectively. The **Hamming distance** H_d between y_α and y_β is defined as the number of digit positions in which the corresponding digits of y_α and y_β are different. If $H_d > 0$, a state transition occurs since $y_\beta \neq y_\alpha$. During the time interval that the state is switching from y_α to y_β, one or more digits in y become unstable,

i.e., their actual state at any instant in time is not known. But a potential problem may occur if more than one digit in y becomes unstable at any instant in time [1]. This condition, known as a **race**, occurs when $H_d > 1$. and may cause an *ASLC* state machine to malfunction because the next state might depend upon the order in which the unstable digits in y change state. A fragment of an excitation table, shown in Figure 4-1 and Figure 4-2, illustrates the race conditions. Figure 4-1 illustrates a noncritical race. Figure 4-2 illustrates a critical race.

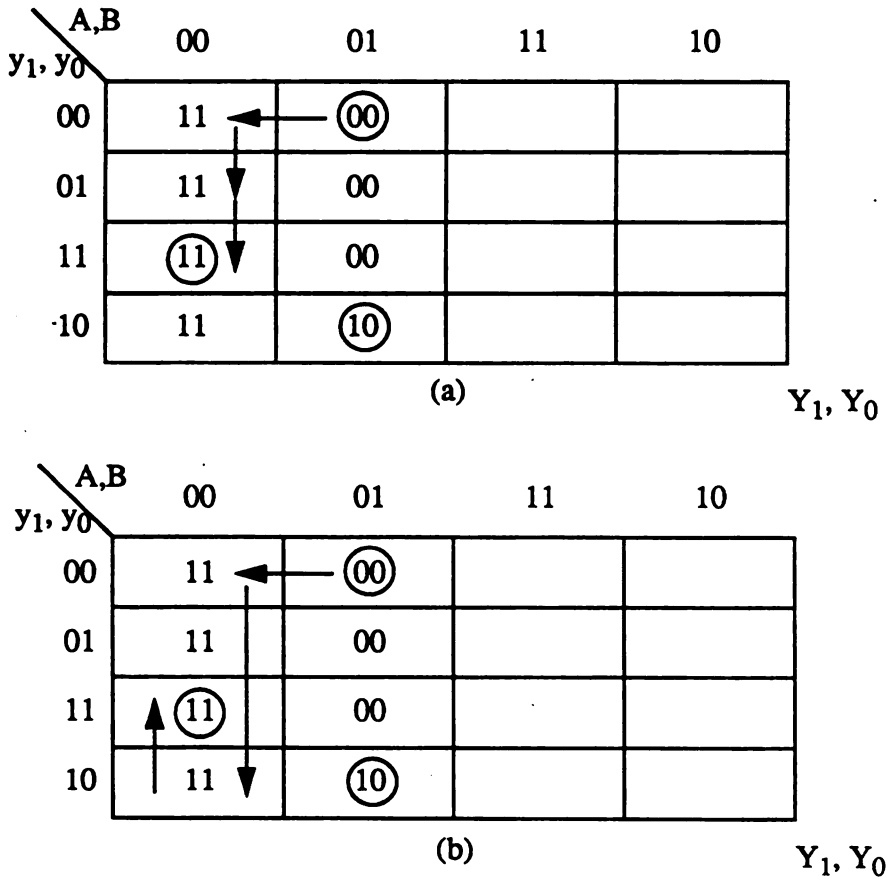


Figure 4-1. Illustration of a noncritical race (a) y_0 changes before y_1; (b) y_1 changes before y_0

Let us first focus on Figure 4-1. Suppose that the circuit corresponding to this table is sitting in the stable state $(A, B, y_1, y_0) = (0, 1, 0, 0)$ and input B changes from "1" to "0". The required transition to state $y_1y_0 = 11$ involves a change in the values of two state (secondary) variables. If these two changes occur simultaneously, the transition specified in the table will actually take place. However, from a physical point of view, the simultaneous changing of two signals in a circuit is highly unlikely -- one is bound to change slightly ahead of the other. If either y_0 or y_1 changes first, instead of going directly to the stable state $y_1y_0 = 11$, the circuit will go to state $y_1y_0 = 01$ or $y_1y_0 = 10$. Such a condition, where two or more state variables are required to change at the same time, is called a race condition [3]. We notice in this particular example that regardless of the outcome of the race, the circuit will always end up in the same stable state $y_1y_0 = 11$. If the final state which the circuit has reached does not depend on the order in which the variables change, such a race condition is referred to as a **noncritical race** [1, 3, 4, 8]. Because the race outcome is not critical in determining the final stable state.

Now, consider the same table shown in Figure 4-2. Suppose the circuit is sitting in the stable state $(A, B, y_1, y_0) = (0, 0, 1, 1)$ and input B changes from "0" to "1". The required transition is to state $y_1y_0 = 00$, which involves a change in the values of two state variables. If y_1 changes value faster than y_0, the circuit will go to state $y_1y_0 = 01$, from which it will reach stable state $y_1y_0 = 00$. On the other hand, if y_0 changes value faster than y_1, the circuit will go to stable state $y_1y_0 = 10$ and remain there. Thus the circuit operation will be incorrect. In this case, the circuit may end up in one of two different stable states, depending upon the outcome of the race. Such a race condition is referred to as a **critical race** [1, 3].

Consider a fragment of another excitation table shown in Figure 4-3. Suppose that the circuit corresponding to this table is sitting in the stable state $(A, B, y_1, y_0) = (0, 1, 0, 1)$ and input B changes from "1" to "0". The required transition to state $y_1y_0 = 10$ involves a

change in the values of two state variables. An unstable state $y_1y_0 = 11$ is entered in row 01 column 00, thereby directing the circuit to row 11, from which it is directed to go to its final stable state $y_1y_0 = 10$. Such a condition, where a circuit goes through a unique sequence of unstable states, is called a **cycle** [3].

4.2 State-Assignment Model

Flow tables (see Figure 4-4(a)) are generally used to characterize the functional behavior of asynchronous sequential machines [12, 33, 34]. Each column of the flow table represents an input state and each row represents an internal state. Entries in the flow table specify the next internal state (state) of the machine. A graph, known as an adjacency diagram (see Figure 4-4(b)), is typically used to depict the set of allowed state transitions. Each **node** represents a state (row) in the flow table. The edge between two nodes is called a **link** and represents an allowed state transition. Two nodes are said to be **adjacent** if they are connected by a link. Traditionally, *ASLC* designers use adjacency diagrams and trial and error methods to make state assignments.

In order to develop algorithms to solve the state assignment problems, a different point of view is introduced to deal with the race conditions. Consider the flow table and adjacency table shown in Figure 4-4. There is no unique solution to encoding the states. For example, the following set of assignments could be made: 00 to *a*, 01 to *b*, 11 to *c*, and 10 to *d*. Since the Hamming distance for all possible state transitions is one (see Figure 4-4(c)), no race condition exists. Alternatively, the following state assignments could also be made: 00 to *a*, 11 to *b*, 01 to *c*, and 10 to *d*. Since the maximum Hamming distance for this set of state assignments is greater than one (see Figure 4-4(d)), a race condition exists. We refer to this type of race as a generated race.

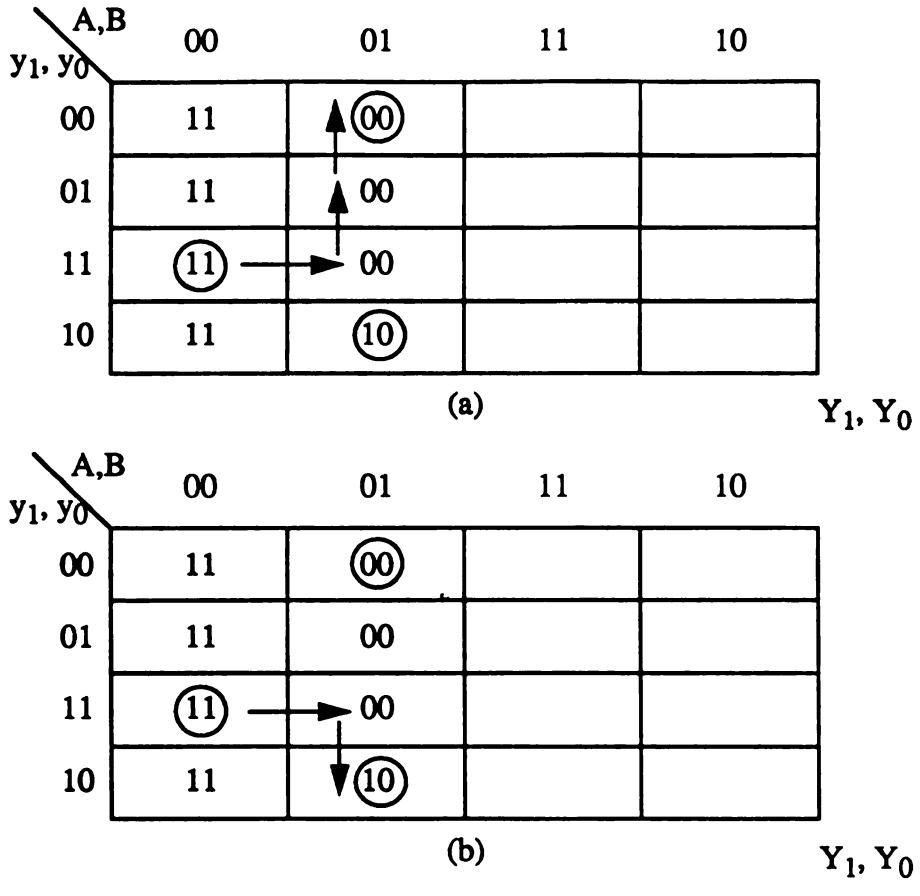


Figure 4-2. Illustration of a critical race (a) y_1 changes before y_0 (desired response); (b) y_0 changes before y_1 (incorrect response)

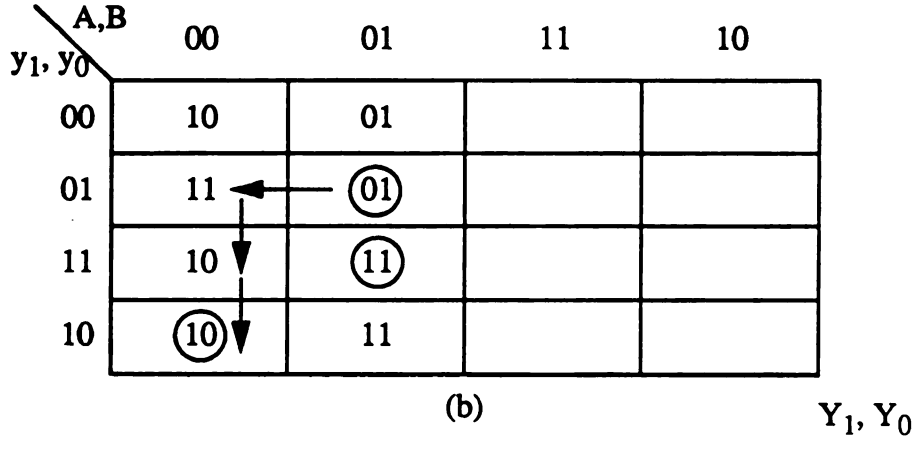


Figure 4-3. Illustration of a cycle

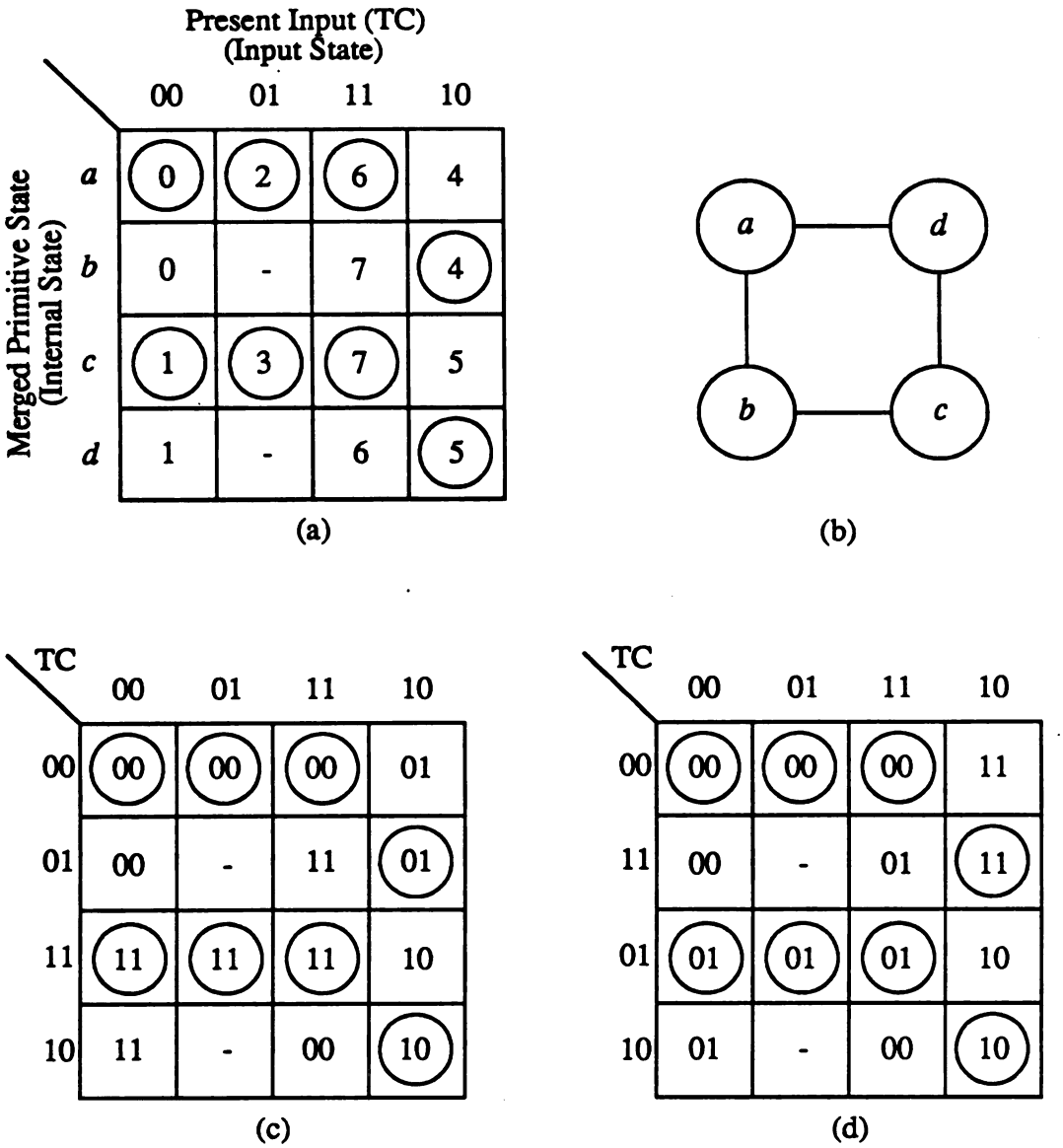


Figure 4-4. Illustration of how improper state assignments can introduce races: (a) flow table; (b) adjacency diagram; (c) excitation table with race-free state assignments; (d) excitation table with races present.

Definition 1 (Generated Race):

> ***A generated race (GR)*** *is a race caused by encoding the states in a manner that results in the existence of a race condition when alternative state assignments could have been made which do not result in a race condition.*

Generated races can be classified into two categories: one is *critical*; another is *noncritical*. This is because the critical races and noncritical races are identified from an excitation table after state assignments are made. For example, Figure 4-5(a) is a fragment of a merged state table corresponding to the excitation table shown in Figure 4-1 or Figure 4-2 which contains noncritical and critical races. However, if we give a different state assignment shown in Figure 4-5(b), there is no critical race or noncritical race appearing in the same part as those in the Figure 4-1 and Figure 4-2. Therefore, critical races or noncritical races are produced by improper state assignments. That is the reason we put critical and noncritical races into generated races. A proper state assignment will never produce any *GR*'s.

Consider the adjacency diagram shown in Figure 4-6(a). Two state variables are needed to represent four states. State *a* connects to three other states, as does *b*. In this example, the **link degree** (number of links) for state *a* is 3. No matter how the state assignments are made, a race condition will always exist, unless the corresponding flow table is modified. We refer to this type of race as an intrinsic race.

Definition 2 (Intrinsic Race):

> ***An intrinsic race (IR)*** *is a race that results when the minimum possible Hamming distance is greater than one.*

Because the *IR*'s in Figure 4-6(a) are easily identified by inspecting the adjacency diagram, we refer to them as visible intrinsic races.

Definition 3 (Visible Intrinsic Race):

A visible intrinsic race (VIR) is an IR that results when the maximum link degree in the adjacency diagram exceeds the number of digits in the state data word.

But another class of *IR*'s also exists. Consider the adjacency diagram shown in Figure 4-6(b). A minimum of three state variables are needed to encode the five states, and the maximum link degree is 2. There are no *VIR*'s since the maximum link degree is less than the number of digits in the state data word. However, a race will always exist no matter how

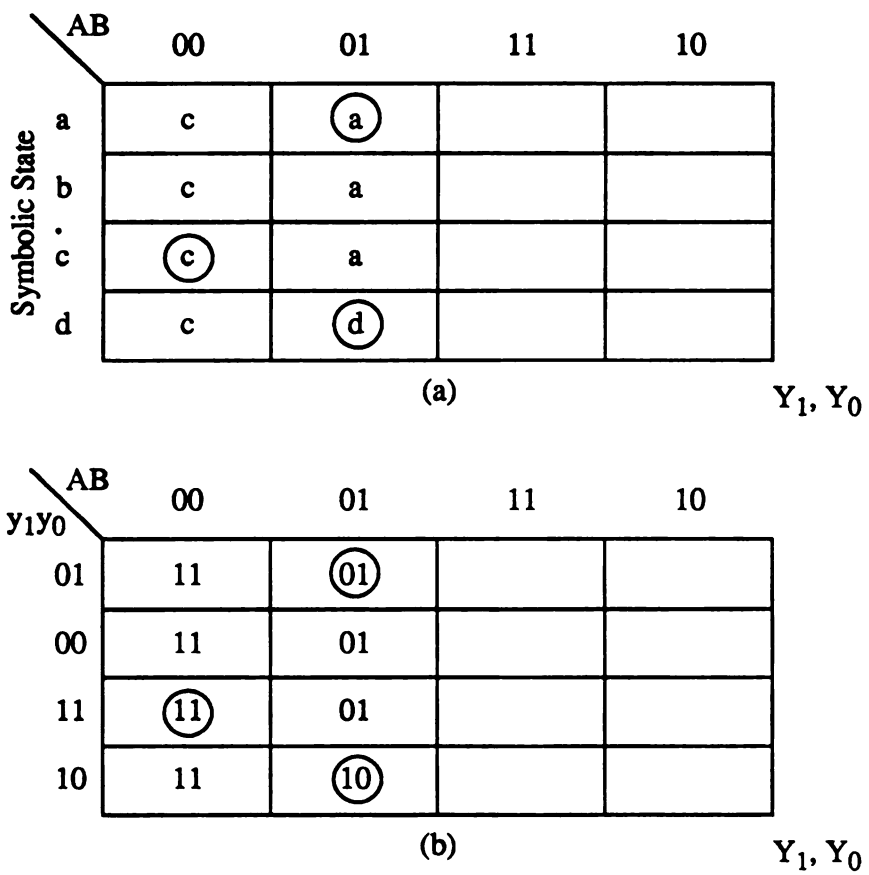


Figure 4-5. (a) A fragment of a merged state table corresponding to Figure 4-1 or Figure 4-2; (b) a different state assignment for (a).

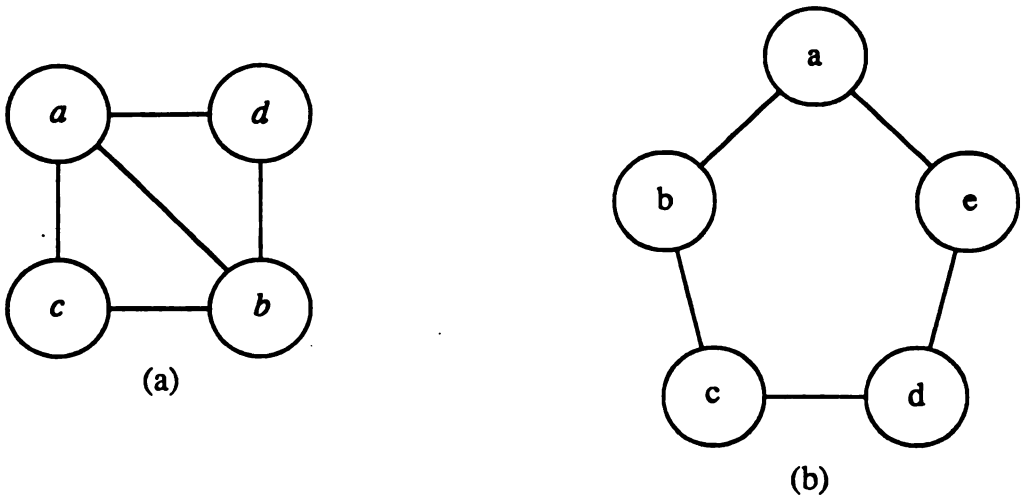


Figure 4-6. An example of intrinsic races (a) *VIR*; (b) *HIR*

the state assignments are made; hence, *IR*'s exist which are not *VIR*'s. We refer to this type of race conditions as a hidden intrinsic race.

Definition 4 (Hidden Intrinsic Race):

A hidden intrinsic race (HIR) is an IR that results when the maximum link degree in the adjacency diagram is not greater than the number of digits in the state data word.

4.3 Methods to Avoid Races

Races in *ASLC*'s can always be eliminated, although there may be a cost in terms of increased hardware complexity and reduced speed of operation. But, to achieve this, intrinsic races (*IR*'s) must be eliminated and generated races (*GR*'s) must not be produced. One method to eliminate *IR*'s is to direct the circuit through intermediate unstable states before it reaches its final desired stable state. For this method, cycles are created without adding any additional states [1, 3, 4]. Therefore, there will be no additional cost in

hardware, although the circuit's speed will be reduced in direct proportion to the number of cycles added. Another method involves adding states to create cycles [1, 3, 4]. However, this approach is less desirable since hardware complexity generally increases. In summary, the general approach to avoid races is to first identify and eliminate all *IR*'s and then make race-free state assignments to guarantee that no *GR*'s are produced.

Our design automation system (*MSUASLC*) always attempts to use the minimum number of states and minimum number of state variables. If *IR*'s exist, it always initially tries to find a cycle through intermediate unstable states without adding any additional states. If this approach does not succeed in eliminating the *IR*'s, then states are added to create a cycle(s) for eliminating the *IR*'s. This strategy guarantees achieving a minimum or near minimum number of states and state variables.

4.3.1 Identification of Races

Before making state assignments, *IR*'s must be detected, identified and eliminated. A rule for quickly identifying a *VIR* is given as follows:

Rule 1 (VIR Identification Rule):

If the link degree of any state is greater than the number of digits in the state data word (i.e., the number of state variables), a VIR exists.

Assume m is the number of states in a given flow table, n is the number of state variables needed to represent the m states, so $n = \lceil \log_2 m \rceil$. Consider the merged state table showing in Figure 4-7(a). The corresponding adjacency diagram is shown in Figure 4-7(b). The number of states is m = 4, and the number of state variables is $n = \lceil \log_2 m \rceil = 2$. The link degrees for states *a*, *b*, *c*, and *d* are 3, 3, 2, and 2, respectively. From the *VIR*

identification rule, we can immediately determine that *IR*'s exist. This merged flow table can be modified by creating cycles without adding additional states. We can make the unstable state 3 in row *a* go, first, to the unstable state 3 in row *c* and then to the stable state 3 in row *b*. Similarly, for the unstable state 4 in row *b*, we can make it first go to the unstable state 4 in row *d* and then to the stable state 4 in row *a*. Figure 4-7(c) shows the flow table in terms of rows and indicates the necessary cycles explicitly. This kind of table is referred to as a **modified flow (state) table**. Figure 4-7(d) is the modified adjacency diagram corresponding to Figure 4-7(c).

Consider another example shown in Figure 4-8. This example tells us adding states to create cycles and eliminate intrinsic races. There are four merged primitive states in Figure 4-8(a); so, two state variables are needed to represent four states. The link degree of any of these states is 3 (see Figure 4-8(b)), which is greater than the number of state variables, so *VIR*'s exist. We cannot eliminate intrinsic races in this example by using only two state variables to make any row simultaneously adjacent to three other rows. However, if we use three state variables instead of two to encode the rows of the merged state table, we may be able to accommodate all of the required adjacencies by creating cycles. Figure 4-8(c) shows the modified state table which does not contain any intrinsic races, and Figure 4-8(d) shows the modified adjacency diagram corresponding to Figure 4-8(c).

Based on the relation of the link degrees and the number of digits in the state data word, we have the following theorem:

Theorem 1:

> ***The necessary, but not sufficient,*** *condition for making race-free state assignments is that all of the link degrees in a given flow table (or adjacency diagram) must be less than or equal to the number of state variables.*

Proof: This theorem can be proven by using *Definition 3* (or *Rule 1*) and *Definition 4*.

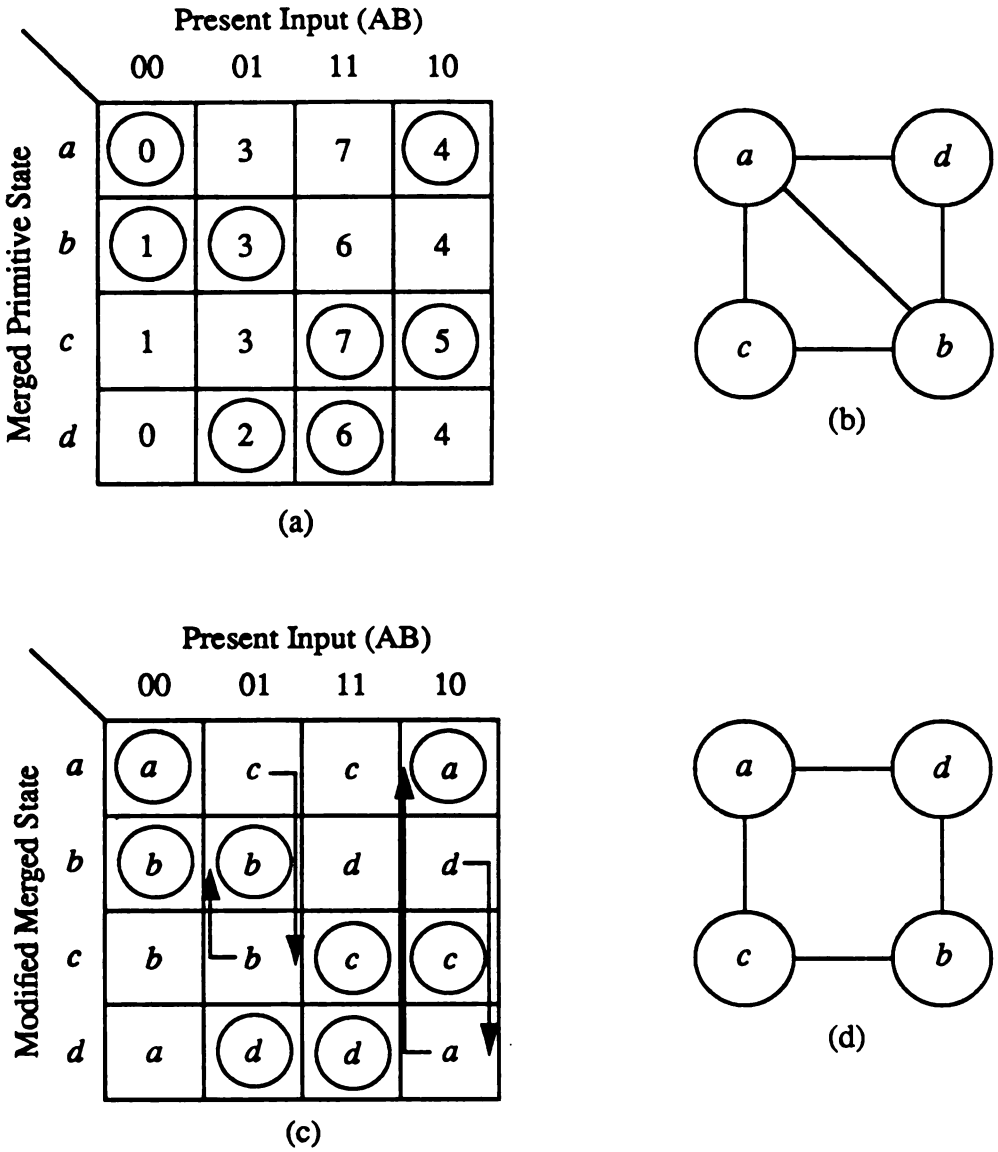


Figure 4-7. An example of avoiding races by creating cycles (a) merged state table; (b) adjacency diagram; (c) modified state table; (d) modified adjacency diagram.

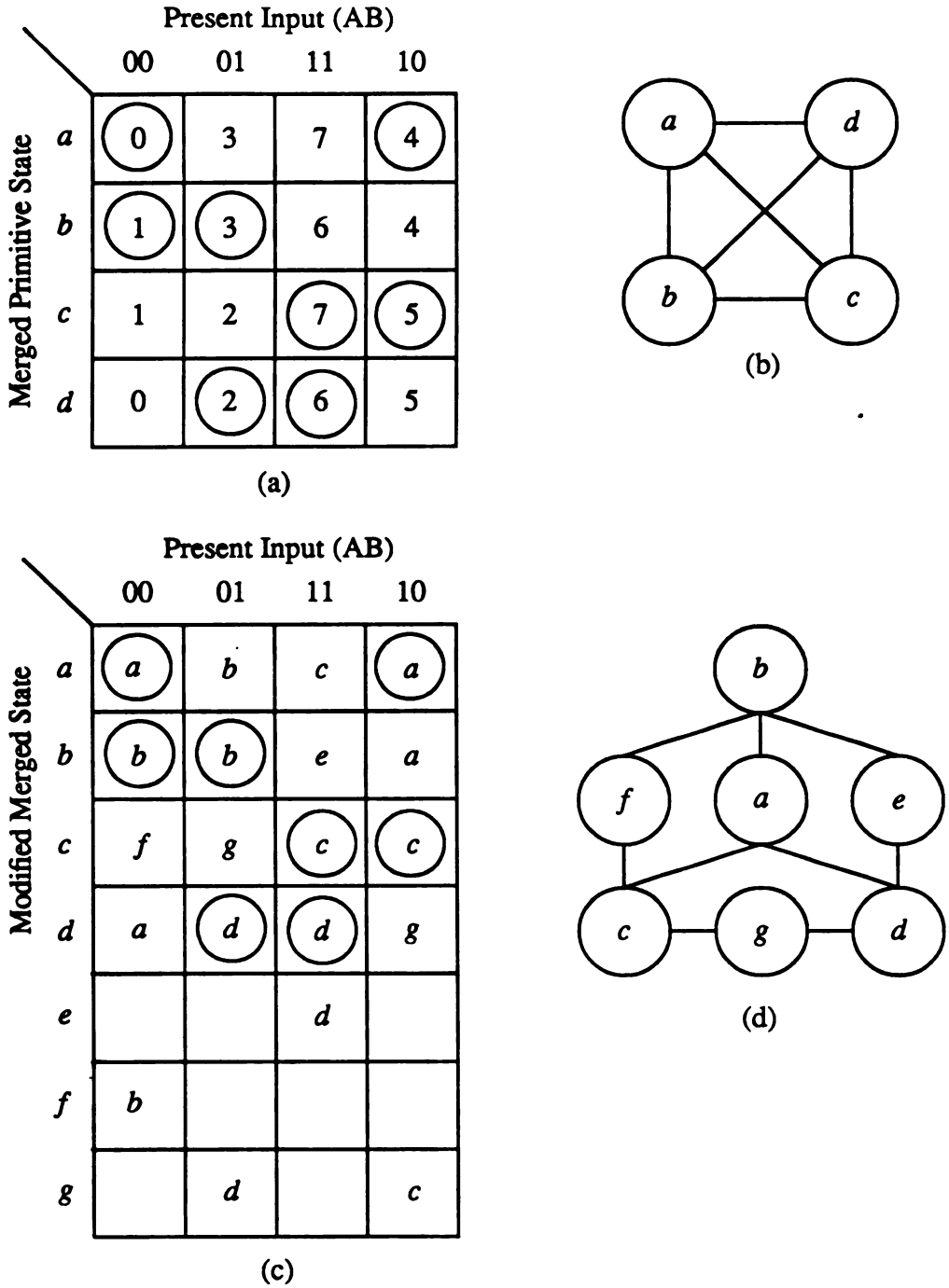


Figure 4-8. An example of adding states to create cycles and eliminate races (a) merged state table; (b) adjacency diagram; (c) modified state table; (d) modified adjacency diagram.

To guarantee that no *GR's* are generated in the state assignment procedure, a state assignment assumption is required.

State Assignment Assumption:

Encoded adjacent states must have a Hamming distance equal to one, ($H_d = 1$).

4.3.2 Node-Weight Diagram (NWD)

To facilitate the process of making state assignments, a graph, referred to as a node-weight diagram (or *n-NWD*) is introduced. This diagram is a variation of a binary **n-cube** (see Figure 4-9) connection diagram but provides a more convenient geometric representation of binary numbers for the purpose of making race-free state assignments. A four-bit *n-NWD* (a 4-*NWD*) is illustrated in Figure 4-10. The weight of node N_i, denoted by $|N_i|$, is defined as the total number of 1's in the binary representation of N_i. The nodes are

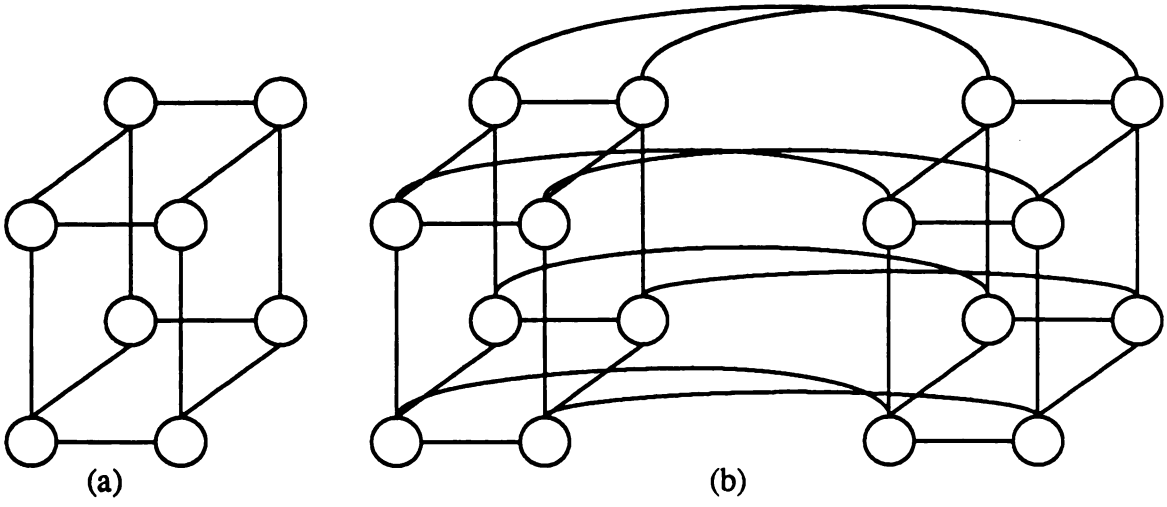


Figure 4-9. Examples of binary n-cube, (a) 3-cube, (b) 4-cube

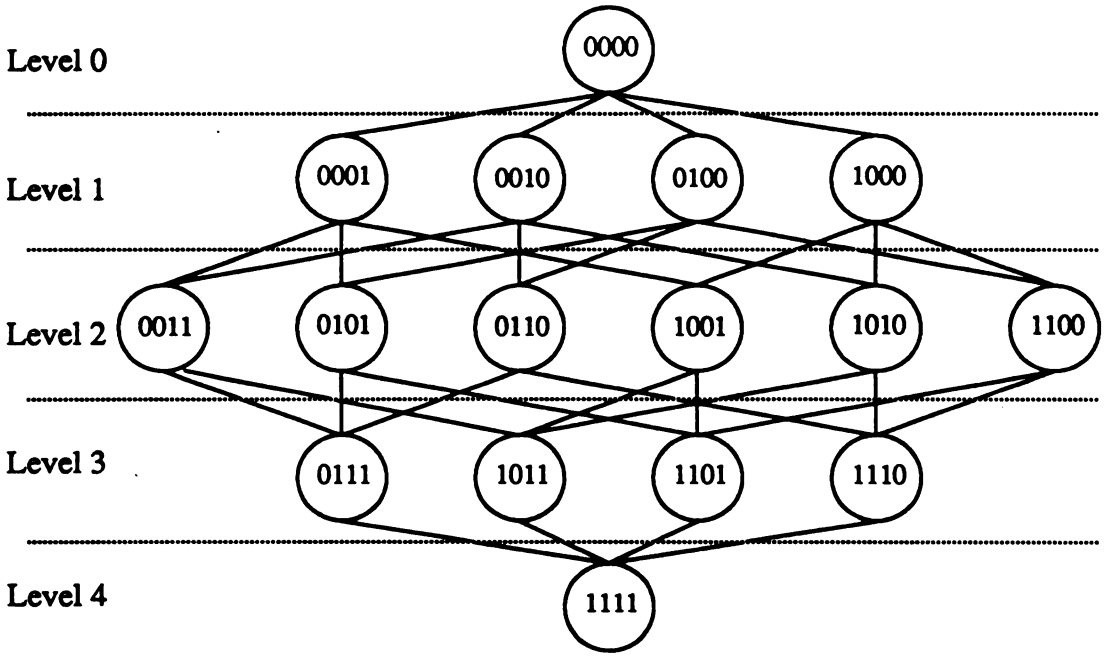


Figure 4-10. Geometric representation of a 4-*NWD*

arranged by weights in levels with level p containing all nodes with weight p.

An *n-NWD* is constructed as follows: For m states, there are 2^n nodes in the *n-NWD*, where $n = \lceil log_2 m \rceil$. These nodes are arranged in $(n + 1)$ levels, where the pth level contains $C\binom{n}{p}$ nodes of weight p, where,

$$C\binom{n}{p} = \frac{n!}{(n-p)!p!}.$$

Next, all possible binary vectors with n digits are assigned to the nodes in an *n-NWD* so that the link relation between nodes can be established. The rule for assigning a binary vector to a node is as follows:

Rule 2 (n-NWD Code-Assignment Rule):

There are n digits in each binary vector for an n-NWD. The number of 1's in a binary vector assigned to a node is equal to the level number at which the node resides. The binary vectors in each level are arranged from left-to-right in ascending magnitude of the binary vector.

After the binary vectors have been assigned to the nodes in an *n-NWD*, a link between two nodes is generated if these two nodes have only one digit difference in their binary vectors. A link may only exist between pairs of nodes in adjacent levels since only these pairs of nodes may have a Hamming distance equal to one. For any pair of adjacent nodes, the node with lower weight is called the **parent node**, while the adjacent node with higher weight is called the **child node**. Nodes with the same weight are called **sibling nodes**. No links exist between sibling nodes since the Hamming distance between sibling nodes is always greater than one.

Lemma 1: In an n-level state weight connection diagram, the Hamming distance between any two nodes at the same level i (with the same weight i) is greater than 1; i.e., $H_d > 1$.

Proof: Assume any two nodes, a and b, are at the same level i. Node a and node b have at most $(i - 1)$'s 1 at the same bit positions. Then, node a has a bit with value 1 at bit position j, node b has a bit with value 1 at bit position k, and $j \neq k$ (note that all the nodes in the state weight connection diagram are different). Hence, $H_d > 1$.

Q.E.D.

Property: Each node at level p of an n-*NWD* has at most p parent nodes

The data structure for each node is shown in Figure 4-11. For an *n-NWD*, two states can be connected through a **path** composed of a single link or series of links. Cycles and

states can be added by referring to the *n-NWD* so that the flow table can be easily modified to ensure that no race conditions exist. The **length** of a path is defined as the total number of links connecting the endpoints. Since these endpoints represent the initial state and next state in the original flow table, the objectives in assigning paths to required state transitions may be expressed as follows: No path should contain a race, i.e., the Hamming distance for all state transitions must be equal to one. Second, the length of each path should be as short as possible to ensure that timing delays between state transitions is minimized. And, third, only the minimum number of new states should be added to help ensure that hardware complexity is minimized.

Theorem 2: An *HIR* exists if a circuit exists in an adjacency diagram and the circuit size $D_c = 2 \times q + 1$, $q \in Z_n$, where Z_n is a set of positive integer numbers greater than 0; i.e., there are odd number of links in the closed path.

Proof: Given a circuit with circuit size D_c, we can choose any one of the links in the circuit, and cut it (assume the nodes connected to this cut link are labeled *a* and *b*). Assume

flag	level_no	std_code	node_no	parent	child	next

flag: indicates whether or not node has been previously checked
level_no: level in *the NWD* in which the node is located
std_code: binary code for this node
node_no: symbolic name (i.e., number) for this node
parent: pointer to parent list
child: pointer to child list
next: pointer to next sibling node

Figure 4-11. The data structure for a node in an *n-NWD*

a is the starting node (state) and *b* is the end node after the circuit is cut. We define a path length, *P*, is equal to the number of links between node *a* and node *b*, so $P = D_c - 1$. Then we map this circuit (after cut) into the state weight diagram (suppose node *a* is mapped into level *i* with weight *i*). Since $H_d = 1$ between two adjacent nodes, the two adjacent nodes must be at adjacent levels. According to this, node *b* should be at level *i - 1* or level *i + 1* (because node *a* is adjacent to node *b*). Although there may exist many different mapping patterns, the difference between the number of up links (L_u) (the link from node *p* to node *q* is called an up link, if node *p* (with higher weight) is at one level lower than node *q* (with lower weight)) and the number of down links (L_d) should be equal to 1 such that node *b* is located at level *i - 1* or level *i + 1*. Note that we will never choose the mapping patterns which make node *b* be 2 more (include 2) levels above or below level *i*, since they will always make *IR*'s exist. However, *P* is an even number which makes $L_u = L_d$, so state *b* is at the same level (level *i*) as state *a*. By *Lemma 1*, the H_d between state *a* and state *b* is greater than 1. Therefore, an *IR* (*HIR*) exists.

Q.E.D.

Consider a very simple merged state table shown in figure 4-12(a). The corresponding adjacency diagram is shown in Figure 4-12(b). Two state variables are needed to encode three states. The link degree of each state is 2, so no *VIR*'s exist. By Theorem 2, *HIR*'s exist. Since we cannot create a cycle through any of three states (a, b, c), a state must be added to create a cycle. Figure 4-12(c) shows the modified state table by adding a state to the merged state table. Figure 4-12(d) is the modified adjacency diagram corresponding to Figure 4-12(c). This example shows that we can add states to create cycles and eliminate intrinsic races without increasing the number of state variables.

Consider another adjacency diagram shown in Figure 4-13. Three state variables can

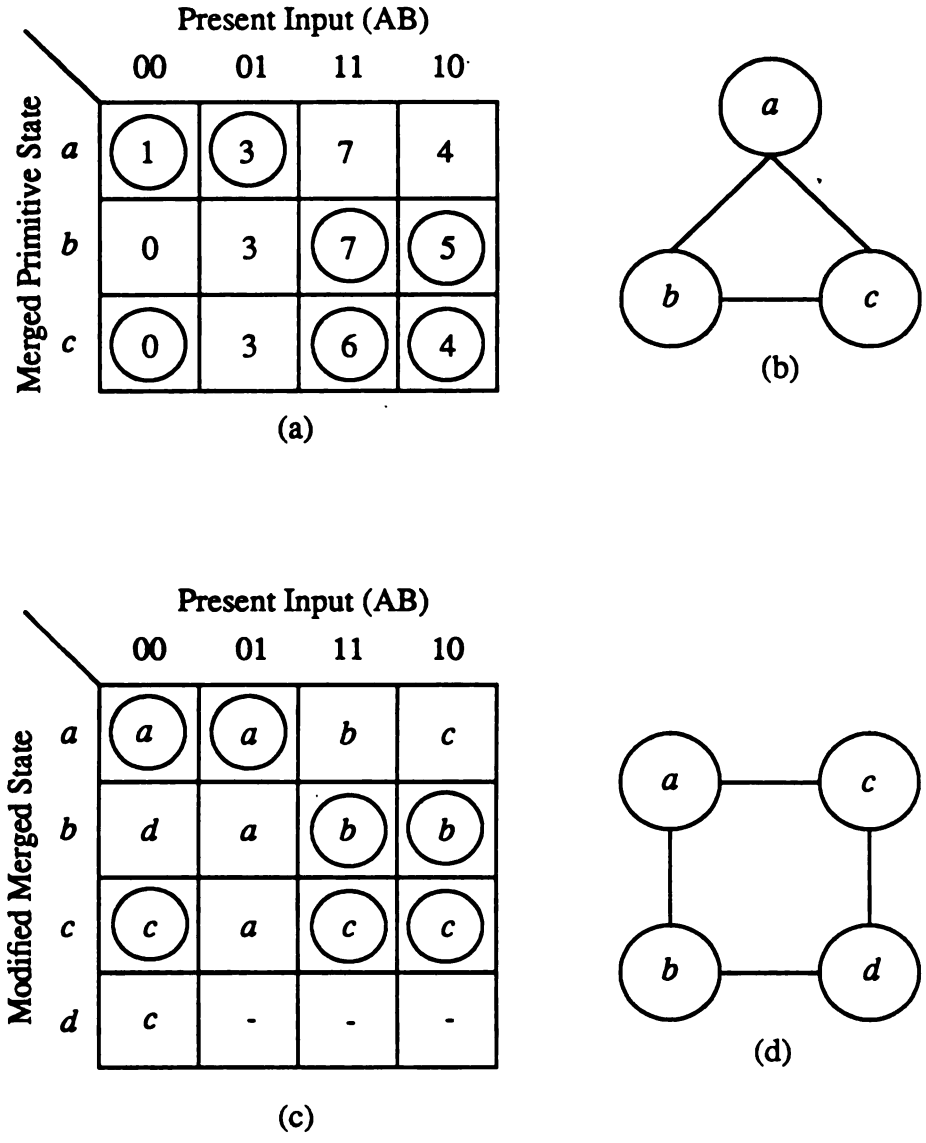


Figure 4-12. An example of adding states to create cycles and eliminate races (a) merged state table; (b) adjacency diagram; (c) modified state table; (d) modified adjacency diagram

be used to represent five states. No link degrees are greater than the number of state variables, so no *VIR*'s exist. But, no matter how the state assignments are made, races always exist. For example, if we assign 000 to *a*, 001 to *b*, 010 to *c*, and 100 to *d*, then any of the remaining four binary vectors: 011, 101, 110, and 111 assigned to *e* will cause race conditions happen.

Theorem 3:An adjacency diagram is mapped into a state weight diagram such that there are n nodes at level *i*. For these n nodes, if they have a common parent node at level $i-1$, a common child node at level $i+1$, and $n > 2$, then an *HIR* exists.

Proof: Because of the symmetrical property of the state weight diagram, for any kind of this subgraph, we can view these three levels as the first three levels (level 0, 1, 2). According to the property of the state weight diagram, the node at level 2 has at most two parent nodes (because each node at level 2 has weight value equal to 2). Therefore, an *HIR* exists if $n > 2$.

Q.E.D.

Based on the concepts introduced above, four race-free state assignment algorithms have been developed and integrated into the *MSUASLC Design Automation System.* Following Section presents these algorithms and gives some examples to show how these algorithms work.

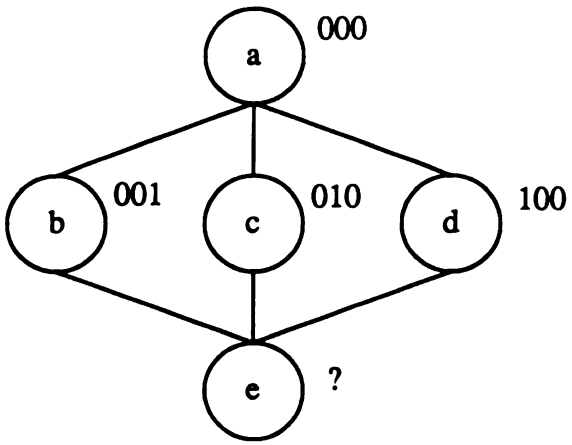


Figure 4-13. An example of adjacency diagram which cannot avoid race conditions.

4.4 Algorithms and Examples for Race-Free State Assignment

Algorithm 1 (identifying and removing intrinsic races):

Step 1. Relabel the merged state table (*MST*):

Change the stable state number in each row corresponding to its row number and change the remaining unstable states. (The purpose is to reduce the redundant representations for the same state.)

Step 2. Compute the number of state variables according to the number of rows in a given *MST*. (The number of state variables is equal to $\lceil \log_2 m \rceil$, m is the number of given states.)

Step 3. Generate the adjacency table:

3.1 Initialize an adjacency table. (Table size depends on the number of rows in a given merged state table.)

3.2 Fill the entries in the adjacency table according to the relabeled *MST*.

3.3 Compute link degree for each merged state.

Step 4. Identify visible intrinsic races (*VIR*):

Check all the link degrees:

4.1 Check link degrees one by one. If one of the link degrees is greater than the number of state variables, there exists a *VIR*. Stop checking the remaining link degrees. Go to Step 6.

4.2 If all the link degrees are less than or equal to the number of state variables go to Step 5.

Step 5. Identify hidden intrinsic races (*HIR*) (see Algorithm 2).

5.1 If there are *HIR*'s in the given *MST*, go to Step 6.

5.2 If there are no *HIR*'s in the given *MST*, go to Step 7. (No any *IR*'s in the given merged flow table)

Step 6. Remove intrinsic races (see Algorithm 3).

Step 7. End. Next is to do state assignment (see Algorithm 4).

Algorithm 2 (identifying hidden intrinsic races):

Step 1. Select a merged primitive state with maximum link degree as the root node of an assignment tree (see Figure 4-14).

Step 2. Initialize the first two levels (level 0 and level 1) of assignment tree.

2.1 Assign 0 (n-bits, n is the number of state variables) to the root node at level 0. Fill out the parent nodes and child nodes information for the root node.

2.2 Generate the tree nodes at level 1 according to the child nodes information at level 0. Assign binary vectors, which are one bit difference with root node, to each tree node at level 1. Fill out the parent

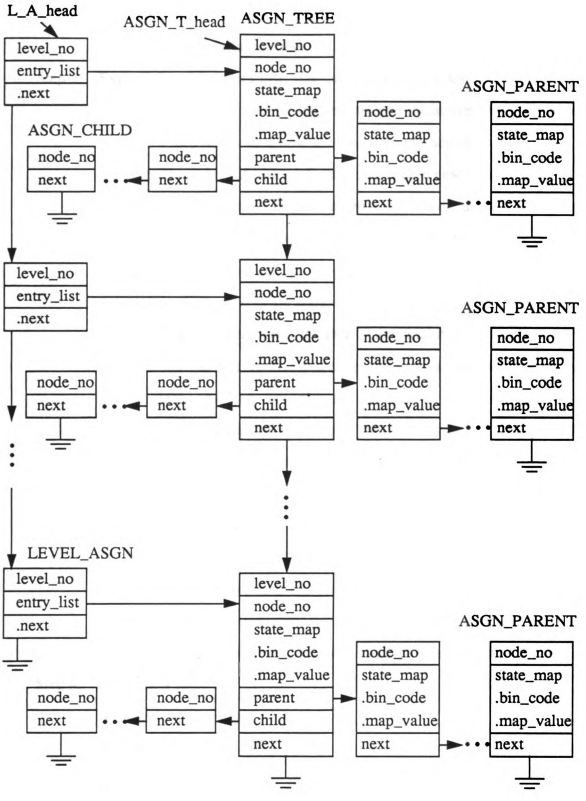


Figure 4-14. The data structure of an assignment tree

nodes and child nodes information for each tree node at level 1. Set level 1 to be the active level.

Step 3. Expand the assignment tree: (Generate new tree nodes based on the child nodes information of each active tree node at the active level, and append these new tree nodes to a new level next to the active level. Fill out the parent nodes and child nodes information for each new tree node)

3.1 For each child node in the current level, check if the child node is already a tree node in the assignment tree. (Each time, check one child node.)

3.2 **IF** it is a tree node in the assignment tree

THEN check if the level number of the child node is the same as the level number of the active tree node. (The active tree node is a parent of the child node. They are adjacent)

(1) If they are at the same level, then there is an *HIR*. Exit from this algorithm and go to algorithm 3 for removing *IR*'s.

(2) If they are not in the same level, then check next child node. Go to 3.1.

ELSE (which means it is not a tree node in the assignment tree) generate a new tree node and append it to the assignment tree. Go to 3.1.

Note: The above two steps 3.1 and 3.2 are repeated until all the child nodes at the active level have been checked or exit if an *IR* is found.

Step 4. Check new level

IF there is no new level generated

THEN there are no *HIR*'s. Based on the assignment tree, an n-*NWD* is generated. Exit from this algorithm and go to algorithm 4 for state assignment.

ELSE compute the number of parent nodes for each new tree node in this new level and do the following things.

4.1 If the number of parent nodes for anyone of these new tree nodes is greater than the level number of this new level, then there is an *HIR*. Exit from this algorithm and go to algorithm 3 for removing *IR*'s.

4.2 If the number of parent nodes is less than or equal to the new level number, then check next new tree node. If all the new tree nodes have been checked, then go to Step 5.

Step 5 Try to assign a binary vector to each new tree node at this new level. The method is as follows:

5.1 **FOR** each new tree node which has more than 1 parent node

DO the "OR" operation on the binary vectors of its parent nodes, and the resulting binary vector is assigned to the new tree node. If the resulting binary vector is already assigned to another tree node, then exit from this algorithm and go to algorithm 3 for removing *IR*'s because an *HIR* is detected. Otherwise, go to step 5.2.

5.2 **FOR** the remaining new tree nodes which have only 1 parent node

DO trying to find an available binary vector which is one bit difference with the binary vector of its parent node.

IF all the possible binary vectors (one bit difference with its parent node) have been assigned,

THEN there is an *HIR*. Exit from this algorithm and go to algorithm 3 for removing *IR*'s.

5.3 Generate the child nodes information for each new tree nodes at the new level.

5.4 Set the new level to be the active level. Go to Step 3.

Algorithm 3 (removing intrinsic races):

Step 1. Copy the relabeled merged flow table (MFT) to a new MFT for modification. If the number of rows (m) is less than 2^n, then add additional rows (2^n - m) to the new MFT, where $n = \lceil \log_2 m \rceil$. The additional rows are marked "checked". In the following, the modification is done to the new MFT.

Step 2. Generate an n-*NWD*.

2.1 Initialize the n-*NWD*: generate nodes for first two levels, level 0 and level 1. The number of nodes in level 1 is equal to the number of state variables currently used.

2.2 Complete the n-*NWD*: generate all the link information.

Step 3. Find a state with a minimum link degree (greater than 0) (or alternatively, a link degree which is equal to or close to the number of state variables) as the root node in the n-*NWD*. Fill the parent nodes and child nodes link information.

Step 4. Fill the nodes in the first level of the n-*NWD* with the states adjacent to the root node. Fill the parent nodes and child nodes link information. Set level 1 as the active level.

Step 5. Pick up a effective node in the active level as active node. Here, we mean effective node is that the node number of this effective node must be > 0 and <= number of rows in the MFT. The active node is marked "checked".

5.1 Find a row in a MFT as the active row. The row number of this active row must be equal to the node number of the active node. The active row is marked "checked".

5.2 Check all the entries from first column to the last column in the active row:

IF the entry is in an UNSTABLE state,

THEN check if this entry is in the n-*NWD*:

IF it is in the n-*NWD*

THEN check if it is directly connected (which means one bit difference) to the active node:

IF yes, check next entry.

IF no, try to find an available path (create cycle) (shortest path first) from the active node to the node corresponding to the active entry:

IF an available path is found

THEN modify the MFT, and then check next entry.

ELSE go to Step 10 (expand MFT and n-*NWD*, n is increased by 1).

ELSE try to find such an empty node that can provide an available path (shortest path first) to the active node.

IF such an empty node is found

THEN fill the empty node with a node number, which is the same as the state number of the active entry. Fill the parent nodes and child nodes link information. If path length is greater than 1, then modify the MFT according to the path.

ELSE go to Step 10 (expand MFT and n-*NWD*).

ELSE check next entry

Step 6. Repeat Step 5 until all the effective nodes in the active level have been checked.

Step 7. Take the next level as the active level and go to Step 5, until all the remaining levels have been checked.

Step 8. Recheck the n-*NWD* from first level to the last level to see if there are new effective nodes generated.

IF a new effective node is found in some level

THEN Set that level as an active level. Go to Step 5

ELSE Go to Step 9.

Step 9. Check if all the rows in MFT have been checked.

IF all the rows have been checked

THEN exit from this algorithm and go to algorithm 4 for state assignment.

ELSE take an empty node in n-*NWD* as the node corresponding to an unchecked row in MFT. The empty node becomes an effective node and the corresponding level becomes an active level. Go to Step 5.

Step 10. Reset new MFT according to the relabeled MFT.

Step 11. Increase the number of state variables by 1. Expand the new MFT and generate an new n-*NWD* according to the current number of state Variables.

Step 12. Go to Step 3.

Algorithm 4 (state assignment):

Step 1. Select one state assignment from the following:

Choice 1. alternative state assignment: user assign any legal binary vector to one of the states.

Choice 2. system assignment

Step 2. If Choice 1 is selected, then user will be asked to give a binary vector to one of the existing states. Go to Step 4.

Step 3. If Choice 2 is selected, then excitation tables and output tables can be

generated according to the standard codes on the n-*NWD*. End.

Step 4. Perform an "XOR" operation on the assigned binary vector and the standard code (in an n-*NWD*) of the designated state. Take the result of the "XOR" operation as the "mask vector".

Step 5. **IF** the value of "mask vector" is equal to zero

THEN generate the excitation tables and output tables based on the n-*NWD*. End.

ELSE go to Step 6.

Step 6. Use the "mask vector" to do "XOR" operation with each standard code in the n-*NWD*. Each result is a binary vector assigned to each state.

Step 7. Generate the excitation tables and output tables. End.

Algorithm 1 gives an easy way to identify *VIR*'s. Algorithm 2 presents an efficient method to identify *HIR*'s. Algorithm 3 is the most important and most difficult part in all of the four algorithms. Algorithm 4 provides us a very quick and very efficient method to generate **alternative state assignments.**

The following examples illustrate the principal features of the four algorithms and demonstrate their utility. In each example, a merged flow (state) table is given before applying the four algorithms. Other tables shown for each example were generated using the *MSUASLC* Design Automation System. These include a re-labeled merged state table (*MST*), an adjacency table, a modified *MST*, and an excitation table.

Example 4-1: (*VIR's* Present)

Figure 4-15(a) depicts a reduced *MST* for an *ASLC* with two inputs and four internal states. The re-labeled *MST* given in Figure 4-15(b) is obtained by applying *Algorithm 1*. The adjacency table for the re-labeled *MST* is shown in Figure 4-15(c). The table entries *S*, *Y*, and *N* indicate whether the next state is the same state as the present state, is adjacent to

the present state, or is neither, respectively. The number in parentheses next to each row number indicates the link degree for the present state. The link degree of each present state equals the number of *Y's* in its row. For this example, since the link degree of row 1 is greater than the number of state variables (=2), a *VIR* exists.

In A*lgorithm 3*, the row (state) 3, which has minimum link degree in the given *MST*, is selected as the root node of the 2-*NWD* (see Figure 4-15(d)). States 1 and 2 become the child nodes of the root node, since they are adjacent to the state 3. *Algorithm 3* sets node 1 of the 2-*NWD* as the active node and row 1 of the re-labeled *MST* as the active row. The first entry in row 1 is a stable state, so it is skipped and next entry checked. This is unstable state 2 and is node 2 in the 2-*NWD*. Node 2 is not directly connected to the active node 1 (there is no path 1-2); hence, an available race-free path between the present state and next state must be found. In this example, path 1-3-2 from active node 1 to node 2 located in the 2-*NWD* and is also available in that column of the *MST*. Thus, the active entry 2 is changed to 3 as shown in the modified *MST* (see Figure 4-15(e)). Next entry 3 in the row 1 is checked. It is an unstable state. Because there is a direct link 1-3, this entry is not modified. The next row entry is a stable state; so, it is not altered. This completes the path-assignment process for the first row in the *MST*. The remaining rows are checked and modified in like manner. This path identification and modification process yields the modified *MST* (Figure 4-15(e)) and corresponding adjacency table (Figure 4-15(f)). These two tables identify all allowed transitions.

Algorithm 4 is next applied to encode the states. The excitation table (see Figure 4-15(g)) is generated according to the state assignments. In this excitation table, the binary vectors 00, 01, 10, and 11 are represented by the decimal number 0, 1, 2, and 3. Figure 4-15(h) is an alternative state assignment by assigning binary vector 00 to row 2 first. Then the mask vector is obtained by the following: $00 \oplus 10 = 10$. The binary vectors for the remaining states can be obtained by XORing the mask vector with each vector shown on

the 2-*NWD*. The resulting set of state assignment is provided in Figure 4-15(h), and the excitation table based on this state assignment is depicted in Figure 4-15(i).

Simple examples, like the one illustrated in Figure 4-15, can easily be done by inspection of the *MST*; however, more complex *ASLC's requ*ire formal procedures to identify and eliminate races. For example, it is not an easy process to modify the *MST* shown in Figure 4-16(a) by inspection even though it is a relatively simple *MST* (see Example 4-2).

Example 4-2: (*VIR's* Present)

An *MST* is shown in Figure 4-16(a). Applying Algorithm 1, a relabeled *MST* is obtained and shown in Figure 4-16(b) which is the same as Figure 4-16(a) for this example. The adjacency table for the relabeled *MST* is shown in Figure 4-16(c). The link degrees for row 1, 2, 3, 4, 5, 6, 7, and 8 are 3, 5, 3, 3, 3, 3, 3, and 3, respectively. Since the link degree of row 2 is 5, which is greater than the number of state variables (=3), a *VIR* exists.

Using Algorithm 3, a 3-*NWD* (shown in Figure 4-16(d)) is constructed to modify the *MST* and eliminate *IR*'s. The modified *MST* is shown in Figure 4-16(e) and the corresponding adjacency table is shown in Figure 4-16(f)) in which the link degrees of row 2, 4, and 6 have been changed from 5, 3, 3 to 3, 2, 2.

Algorithm 4 is next applied to encode the states. The excitation table (see Figure 4-16(g)) is generated according to the state assignments. Figure 4-16(h) is an alternative state assignment by assigning binary vector 101 to row 3 first. Then the mask vector is obtained by the following: $001 \oplus 101 = 100$. The binary vectors for the remaining states can be obtained by XORing the mask vector with each vector shown on the 3-*NWD*. The resulting set of state assignment is provided in Figure 4-16(h), and the excitation table based on this state assignment is depicted in Figure 4-16(j).

Merged Primitive State \ Present Input (AB)	00	01	11	10
	(0)	3	7	(4)
	(1)	(3)	6	4
	1	3	(7)	(5)
	0	(2)	(6)	4

(a)

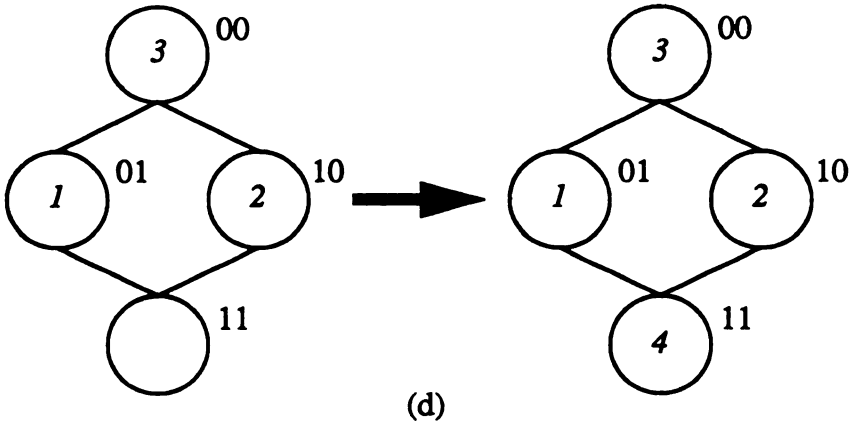
Merged State (relabeled) \ INPUT (AB)	0	1	3	2
	1 Y	2 N	3 N	1 Y
	2 Y	2 Y	4 N	1 N
	2 N	2 N	3 Y	3 Y
	1 N	4 Y	4 Y	1 N

(b)

	1(3)	2(3)	3(2)	4(2)	Merged State (relabeled)
1(3)	S	Y	Y	Y	
2(3)	Y	S	Y	Y	
3(2)	Y	Y	S	N	
4(2)	Y	Y	N	S	

(c)

Figure 4-15. Example 4-1 (a)a merged state table; (b) relabeled *MST*; (c) adjacency table corresponding to the relabeled *MST*; (d) 2-*NWD*; (e) modified state table; (f) adjacency table corresponding to the modified state table; (g) an excitation table due to system assignment; (h) an alternative state assignment; (i) an excitation table due to an alternative state assignment.



(d)

	INPUT (AB) 0	1	3	2	
1	1 Y	3 N	3 N	1 Y	Merged State (relabeled)
2	2 Y	2 Y	4 N	4 N	
3	2 N	2 N	3 Y	3 Y	
4	1 N	4 Y	4 Y	1 N	

(e)

	1(2)	2(2)	3(2)	4(2)	
1(2)	S	N	Y	Y	Merged State (relabeled)
2(2)	N	S	Y	Y	
3(2)	Y	Y	S	N	
4(2)	Y	Y	N	S	

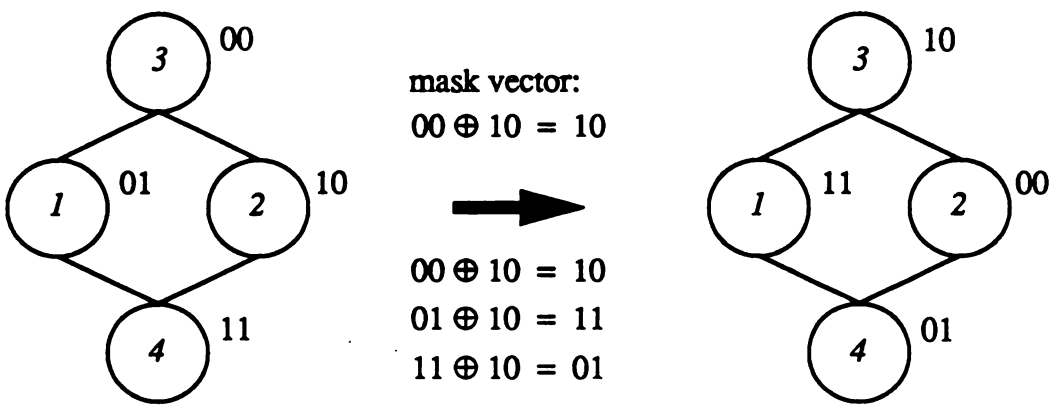
(f)

Figure 4-15 (cont'd).

		INPUT (AB) 0	1	3	2
1	Present State	1	0	0	1
2		2	2	3	3
0		2	2	0	0
3		1	3	3	1

Binary vectors 01, 10, 00, and 11 are assigned to row 1, 2, 3, and 4, respectively.

(g)



(h)

		INPUT (AB) 0	1	3	2
3	Present State	3	2	2	3
0		0	0	1	1
2		0	0	2	2
1		3	1	1	3

(i)

Figure 4-15 (cont'd).

Merged Primitive State \ Present Input (AB)	00	01	11	10
1	(1)	5	-	3
2	(2)	8	(2)	3
7	-	2	(3)	
1	(4)	2	(4)	
6	(5)	2	-	
6	(6)	8	2	(6)
7	(7)	(7)	8	4
7	(8)	(8)	8	6

(a)

Merged State (relabeled) \ INPUT (AB)	0	1	3	2
	1 Y	5 N	-	3 N
	2 Y	8 N	2 Y	3 N
	7 N	-	2 N	3 Y
	1 N	4 Y	2 N	4 Y
	6 N	5 Y	2 N	-
	6 Y	8 N	2 N	6 Y
	7 Y	7 Y	8 N	4 N
	7 N	8 Y	8 Y	6 N

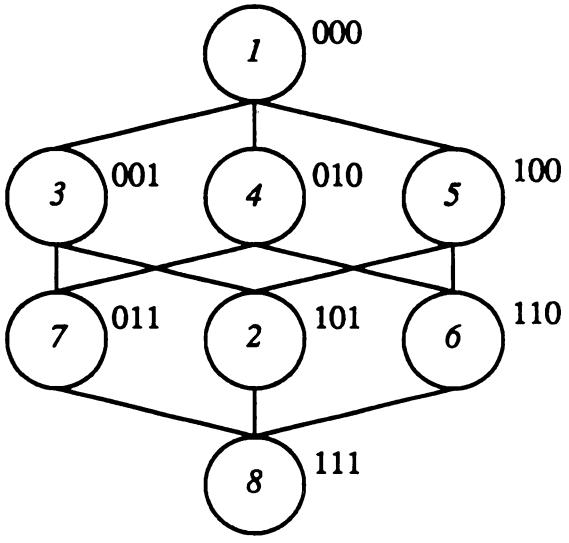
(b)

Figure 4-16. Example 4-2 (a)a merged state table; (b) relabeled *MST*; (c) adjacency table corresponding to the relabeled *MST*; (d) 3-*NWD*; (e) modified state table; (f) adjacency table corresponding to the modified state table; (g) an excitation table due to system assignment; (h) an alternative state assignment; (i) an excitation table due to an alternative state assignment.

	1(3)	2(5)	3(3)	4(3)	5(3)	6(3)	7(3)	8(3)
1(3)	S	N	Y	Y	Y	N	N	N
2(5)	N	S	Y	Y	Y	Y	N	Y
3(3)	Y	Y	S	N	N	N	Y	N
4(3)	Y	Y	N	S	N	N	Y	N
5(3)	Y	Y	N	N	S	Y	N	N
6(3)	N	Y	N	N	Y	S	N	Y
7(3)	N	N	Y	Y	N	N	S	Y
8(3)	N	Y	N	N	N	Y	Y	S

Merged State (relabeled)

(c)



(d)

Figure 4-16 (cont'd).

	INPUT (AB) 0	1	3	2	
1	1 Y	5 N	3 N	3 N	
2	2 Y	8 N	2 Y	3 N	
3	7 N	-	2 N	3 Y	Merged State (relabeled)
4	1 N	4 Y	1 N	4 Y	
5	6 N	5 Y	2 N	-	
6	6 Y	8 N	5 N	6 Y	
7	7 Y	7 Y	8 N	4 N	
8	7 N	8 Y	8 Y	6 N	

(e)

	1(3)	2(3)	3(3)	4(2)	5(3)	6(2)	7(3)	8(3)	
1(3)	S	N	Y	Y	Y	N	N	N	
2(3)	N	S	Y	N	Y	N	N	Y	
3(3)	Y	Y	S	N	N	N	Y	N	
4(2)	Y	N	N	S	N	N	Y	N	Merged State (relabeled)
5(3)	Y	Y	N	N	S	Y	N	N	
6(2)	N	N	N	N	Y	S	N	Y	
7(3)	N	N	Y	Y	N	N	S	Y	
8(3)	N	Y	N	N	N	Y	Y	S	

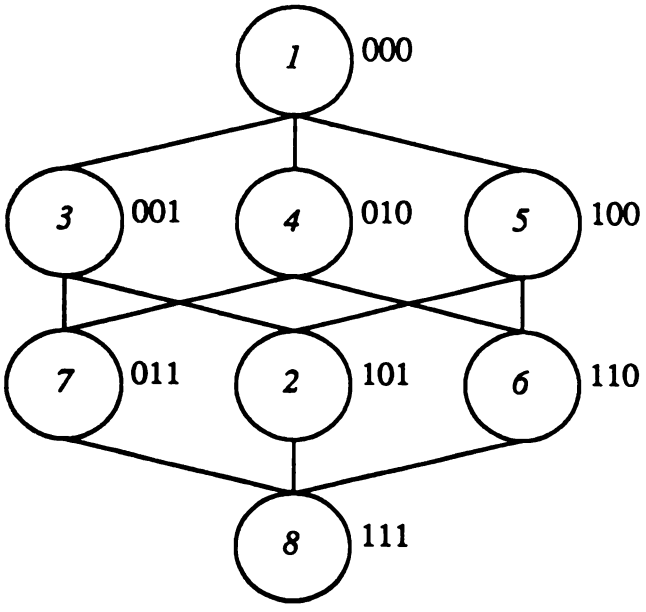
(f)

Figure 4-16 (cont'd).

		INPUT (AB)			
		0	1	3	2
0	Present State	0	4	1	1
5		5	7	5	1
1		3	-	5	1
2		0	2	0	2
4		6	4	5	-
6		6	7	4	6
3		3	3	7	2
7		3	7	7	6

(g)

Figure 4-16 (cont'd).



(1) A 3-*NWD* with standard code

Assign binary vector 101 to state 3

mark vector:

101 ⊕ 001 = 100

node 1: 100 ⊕ 000 = 100

node 3: 100 ⊕ 001 = 101

node 4: 100 ⊕ 010 = 110

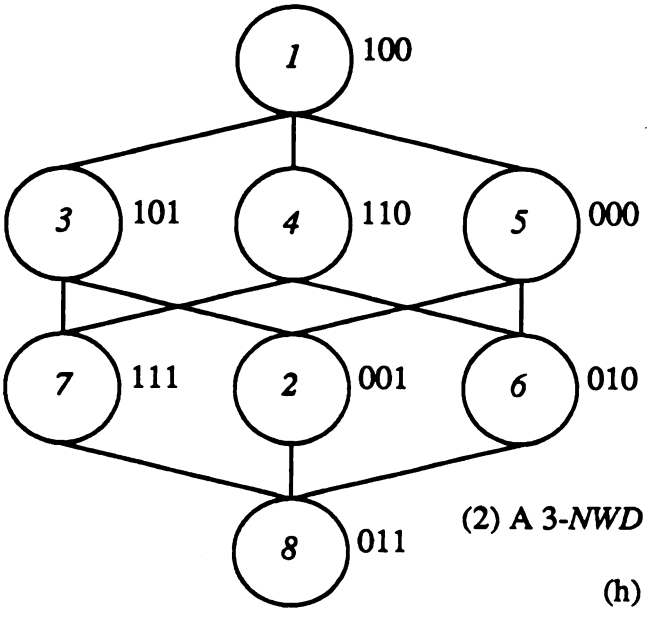
node 5: 100 ⊕ 100 = 000

node 7: 100 ⊕ 011 = 111

node 2: 100 ⊕ 101 = 001

node 6: 100 ⊕ 110 = 010

node 8: 100 ⊕ 111 = 011



(2) A 3-*NWD* with an alternative state assignment

(h)

Figure 4-16 (cont'd).

		INPUT (AB)			
		0	1	3	2
4		4	0	5	5
1		1	3	1	5
5		7	-	1	5
6	Present State	4	6	4	6
0		2	0	1	-
2		2	3	0	2
7		7	7	3	6
3		7	3	3	2

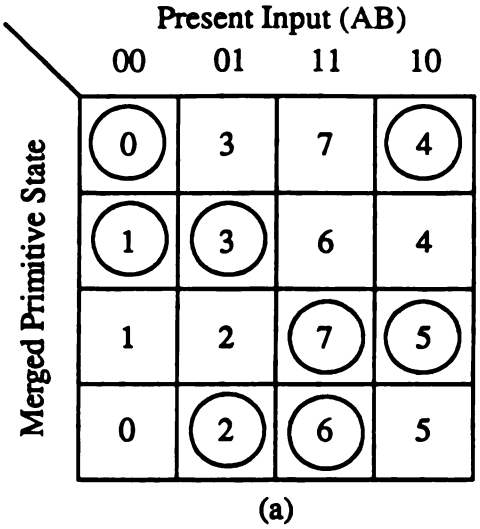
(i)

Figure 4-16 (cont'd).

Example 4-3: (*VIR's* Present)

In the above two examples, the size of the *MST* is not increased after the *MST* is modified. However, an *MST* (see Figure 4-17(a)) will be expanded in this example. Applying *Algorithm 1*, a relabeled *MST* is obtained and shown in Figure 4-17(b). The adjacency table for the relabeled *MST* is shown in Figure 4-17(c). The link degree of each row is 3 which is greater than the number of state variables (=2), so *VIR*'s exist.

Applying *Algorithm 3*, a 3-*NWD* (see Figure 4-17(d)) is generated to eliminate all the *IR*'s. The modified *MST is* shown in Figure 4-17(e). The *MST* is expanded from 4 rows to 7 rows, so the number of state variables is increased by 1 (=3). Look at the node 2 in Figure 4-17(d). Assume it is an active node on the active level (level 1) at this time. The nodes 5, 6, and 7 are not in the 3-*NWD* now. Examine the entries in the second row of the relabeled *MST*, an unstable state 4 is under column 3. In the 3-*NWD*, there is a path 2-1-4. However, the next state of state 1 under column 3 is state 3, so the path 2-1-4 is not available. Another path 2-*x*-4 is available, where *x* represents an empty node in the 3-*NWD*. Since no other rows in the column 3 of the relabeled *MST a*re connected to the row 4, row 5 is added as the intermediate node of the path 2-*x*-4. Then the path 2-*x*-4 becomes 2-5-4. Similarly for the entry at the row 3 (the third row) and column 0 (the first column) of the relabeled *MST*, the path 3-1-2 is not available. However, there is a path 3-*x*-2 can be used to connect node 3 to node 2. The *x* in the path 3-*x*-2 is replaced by the number 6. Hence node 6 is in the 3-*NWD* and a new row (row 6) is added into the new *MST*. Notice that we cannot use row 5 as the intermediate node for the path 3-*x*-2, even though it is unused under column 0 (a don't care "-" in that entry). Because there is no such path 3-5-2 existing in the 3-*NWD*. Therefore, modifying an *MST* is not so straightforward, it refers to both 3-*NWD* and *MST*. The adjacency table corresponding to the modified *MST* is shown in Figure 4-17(f). The excitation table for an alternative state assignment (assign a binary vector 000 to state 2 first) is shown in Figure 4-17(g).



(a)

INPUT (AB)

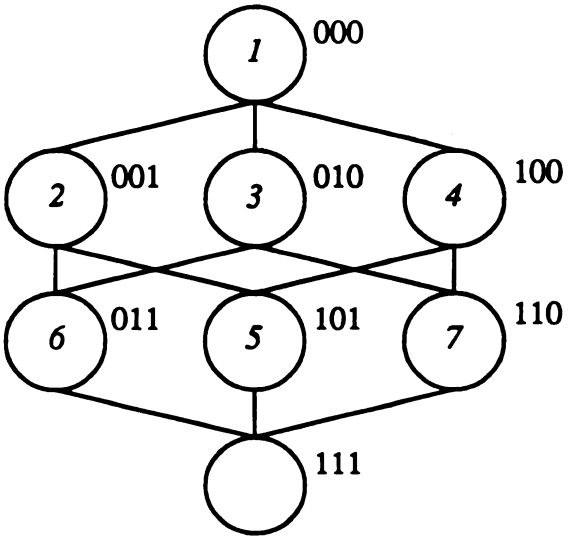
Merged State (relabeled)	0	1	3	2
	1 Y	2 N	3 N	1 Y
	2 Y	2 Y	4 N	1 N
	2 N	4 N	3 Y	3 Y
	1 N	4 Y	4 Y	3 N

(b)

	1(3)	2(3)	3(3)	4(3)	Merged State (relabeled)
1(3)	S	Y	Y	Y	
2(3)	Y	S	Y	Y	
3(3)	Y	Y	S	Y	
4(3)	Y	Y	Y	S	

(c)

Figure 4-17. Example 4-3 (a)a merged state table; (b) relabeled *MST*; (c) adjacency table corresponding to the relabeled *MST*; (d) 3-*NWD*; (e) modified state table; (f) adjacency table corresponding to the modified state table; (g) an excitation table due to an alternative state assignment (assign 000 to state 2).



(d)

	INPUT (AB) 0	1	3	2	Merged State (relabeled)
1	1 Y	2 N	3 N	1 Y	
2	2 Y	2 Y	5 N	1 N	
3	6 N	7 N	3 Y	3 Y	
4	1 N	4 Y	4 Y	7 N	
5	-	-	4 N	-	
6	2 N	-	-	-	
7	-	4 N	-	3 N	

(e)

Figure 4-17 (cont'd).

	1(3)	2(3)	3(3)	4(3)	5(2)	6(2)	7(2)	
1(3)	S	Y	Y	Y	N	N	N	
2(3)	Y	S	N	N	Y	Y	N	
3(3)	Y	N	S	N	N	Y	Y	Merged State (relabeled)
4(3)	Y	N	N	S	Y	N	Y	
5(2)	N	Y	N	Y	S	N	N	
6(2)	N	Y	Y	N	N	S	N	
7(2)	N	N	Y	Y	N	N	S	

(f)

		INPUT (AB)			
		0	1	3	2
1		1	0	3	1
0		0	0	4	1
3	Present State	2	7	3	3
5		1	5	5	7
4		-	-	5	-
2		0	-	-	-
7		-	5	-	3

(g)

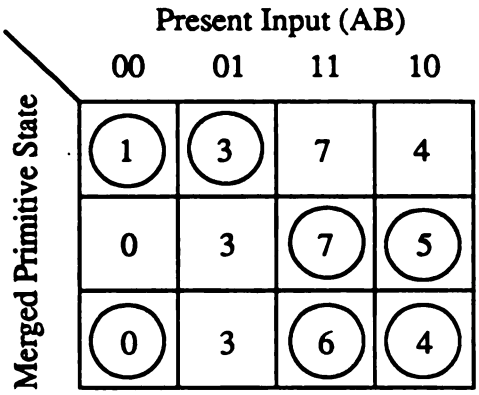
Figure 4-17 (cont'd).

Example 4-4: (*HIR's* Present)

This example illustrates the process of identifying and eliminating hidden intrinsic races. Consider the *MST* illustrated in Figure 4-18(a). Results of applying *Algorithm 1* are illustrated in Figure 4-18(b). The adjacency table for the re-labeled *MST* is shown in Figure 4-18(c). The link degree of each row is 2 which is equal to the number of state variables; so, *no VIR's* exist. But, do any *HIR's* exist? To answer this question, *Algorithm 2* is used to construct an assignment tree, which is shown in Figure 4-18(d). Node 2 is both a parent node and a child node of node 3, and they are in the same level. But nodes at the same level (sibling nodes) are always separated by a Hamming distance which is greater than one; hence, an *HIR* exists.

Algorithm 3 is next applied to eliminate all *IR's*. A 2-*NWD* is generated and shown in Figure 4-18(e). The modified *MST* is shown in Figure 4-18(f). Notice that the entry in row 2 and column 0 has been changed from 3 to 4, and one row (row 4) is added into the *MST*. Although there is a path 2-1-3 existing in the 2-*NWD*, this path is not available under the input 00 (i.e., column 0). Therefore, a new path 2-4-3 is created so that it can start from state 2, via state 4, and reach state 3. The adjacency table corresponding to the modified *MST* is shown in Figure 4-18(g). *Algorithm 4* is next applied to encode the states. The excitation table (see Figure 4-18(h)) is generated according to the state assignments.

So far, the examples we have seen are small size of *MST*. In the following, we are going to see an example with a very large size of *MST*. Smith [13] pointed out a flow table with more than 150 cells is a very large flow table. The number of cells (entries) is equal to the number of rows times the number of columns. It can be used as a rough measure of flow table complexity. It will be shown that the algorithms presented can be used to rapidly determine state assignments for extremely large tables.



(a)

INPUT (AB)

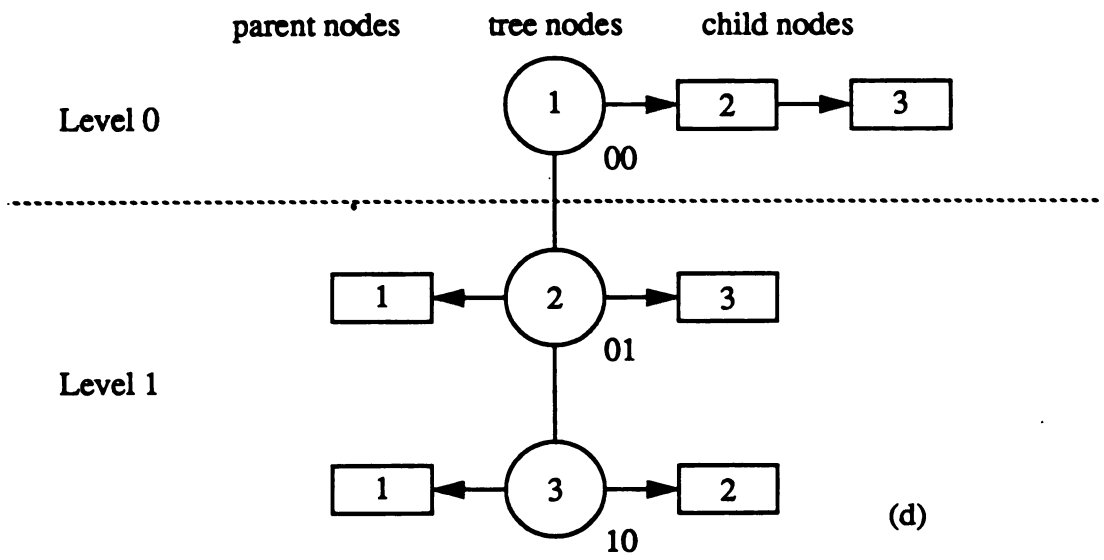
Merged State (relabeled)	0	1	3	2
	1 Y	1 Y	2 N	3 N
	3 N	1 N	2 Y	2 Y
	3 Y	1 N	3 Y	3 Y

(b

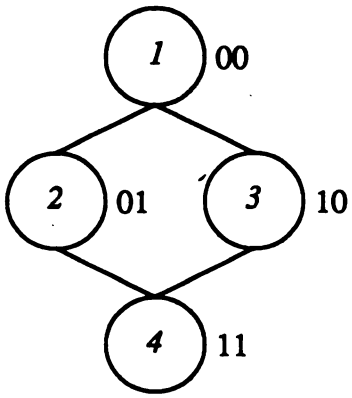
	1(2)	2(2)	3(2)	Merged State (relabeled)
1(2)	S	Y	Y	
2(2)	Y	S	Y	
3(2)	Y	Y	S	

(c)

Figure 4-18. Example 4-4 (a)a merged state table; (b) relabeled *MST*; (c) adjacency table corresponding to the relabeled *MST*; (d) assignment tree; (e) 2-*NWD*; (f) modified state table; (g) adjacency table corresponding to the modified state table; (h) an excitation table due to system assignment.



(d)



(e)

	INPUT (AB) 0	1	3	2	
1	1 Y	1 Y	2 N	3 N	Merged State (relabeled)
2	4 N	1 N	2 Y	2 Y	
3	3 Y	1 N	3 Y	3 Y	
4	3 N	-	-	-	

(f)

Figure 4-18 (cont'd).

	1(2)	2(2)	3(2)	4(2)	
1(2)	S	Y	Y	N	Merged State (relabeled)
2(2)	Y	S	N	Y	
3(2)	Y	N	S	Y	
4(2)	N	Y	Y	S	

(g)

		INPUT (AB)			
		0	1	3	2
0	Present State	0	0	1	2
1		3	0	1	1
2		2	0	2	2
3		2	-	-	-

(h)

Figure 4-18 (cont'd).

Example 4-5: (*VIR's* Present)

This example demonstrates a very large flow table with 256 entries (8 rows * 32 columns). The merged state table is given in Figure 4-19(a). Applying *Algorithm 1*, a relabeled *MST* is obtained and shown in Figure 4-19(b). The adjacency table for the relabeled *MST* is shown in Figure 4-19(c). The link degrees are 7, 3, 3, 7, 5, 3, 3, and 3 for row 1, 2, 3, 4, 5, 6, 7, and 8, respectively. Therefore *VIR*'s exist.

Now, *Algorithm 3* is applied to eliminate the *IR*'s. A 4-*NWD* (see Figure 4-19(d) is generated for this purpose. The modified *MST* is shown in Figure 4-19(e), and the corresponding adjacency table is shown in Figure 4-19(f). The excitation table based on a system assignment is shown in Figure 4-19(g). This example fully shows that it is almost impossible to do this by pen-and-paper methods, because it takes too much effort and too much time. Also, human designers may make mistakes because of the complexity of the merged flow table.

INPUT (JKCSR)							
0	1	3	2	6	7	5	4
1 Y	5 Y	13 N	10 N	26 N	29 N	21 Y	17 Y
1 N	5 N	13 Y	9 Y	26 N	29 N	-	-
-	-	13 N	10 N	25 Y	29 Y	21 N	17 N
2 N	6 N	14 Y	10 Y	26 Y	30 N	21 N	18 Y
2 N	6 Y	14 N	-	-	30 N	21 N	-
2 Y	5 N	14 N	10 N	-	-	21 N	18 N
-	5 N	14 N	-	26 N	30 Y	22 Y	18 N
-	5 N	-	-	-	-	-	-

12	13	15	14	10	11	9	8
49 Y	53 Y	61 N	58 N	42 N	45 N	37 Y	33 Y
49 N	53 N	61 Y	57 Y	42 N	45 N	-	33 N
-	-	61 N	58 N	41 Y	45 Y	37 N	33 N
50 N	54 Y	62 Y	58 Y	42 Y	46 N	37 N	34 N
49 N	-	62 N	-	42 N	-	37 N	34 Y
50 Y	53 N	62 N	58 N	42 N	46 Y	38 Y	34 N
-	53 N	62 N	-	-	-	37 N	-
-	53 N	-	-	-	-	-	-

Figure 4-19. Example 4-5 (a)an example of large size *MST*; (b) relabeled *MST*; (c) adjacency table corresponding to the relabeled *MST*; (d) 4-*NWD*; (e) modified state table; (f) adjacency table corresponding to the modified state table; (g) an excitation table due to system assignment.

24	25	27	26	30	31	29	28
97 N	101 Y	109 Y	105 Y	122 N	125 N	117 Y	113 Y
97 Y	101 N	-	106 N	122 N	125 N	-	114 N
-	-	109 N	106 N	121 Y	125 Y	117 N	113 N
98 N	102 N	110 Y	106 Y	122 Y	126 N	117 N	114 Y
98 Y	101 N	110 N	106 N	122 N	126 Y	118 N	113 N
-	101 N	110 N	-	-	-	117 N	114 N
98 N	102 Y	110 N	-	-	-	117 N	-
-	101 N	-	-	-	126 N	118 Y	114 N

20	21	23	22	18	19	17	16
81 N	85 Y	93 Y	89 Y	74 N	77 N	69 Y	65 N
82 N	-	-	-	74 N	77 N	69 N	65 Y
81 Y	85 N	93 N	90 N	73 Y	77 Y	69 N	65 N
82 Y	85 N	94 N	90 Y	74 Y	78 Y	70 N	66 N
82 N	86 N	94 Y	90 N	74 N	78 N	69 N	66 Y
82 N	86 Y	94 N	-	-	-	69 N	66 N
-	85 N	94 N	-	-	-	69 N	-
-	85 N	-	-	-	78 N	70 Y	66 N

(a)

INPUT (JKCSR)

0	1	3	2	6	7	5	4
1 Y	1 Y	2 N	4 N	4 N	3 N	1 Y	1 Y
1 N	1 N	2 Y	2 Y	4 N	3 N	-	-
-	-	2 N	4 N	3 Y	3 Y	1 N	1 N
6 N	5 N	4 Y	4 Y	4 Y	7 N	1 N	4 Y
6 N	5 Y	4 N	-	-	7 N	1 N	-
6 Y	1 N	4 N	4 N	-	-	1 N	4 N
-	1 N	4 N	-	4 N	7 Y	7 Y	4 N
-	1 N	-	-	-	-	-	-

Figure 4-19 (cont'd).

12	13	15	14	10	11	9	8
1 Y	1 Y	2 N	4 N	4 N	3 N	1 Y	1 Y
1 N	1 N	2 Y	2 Y	4 N	3 N	-	1 N
-	-	2 N	4 N	3 Y	3 Y	1 N	1 N
6 N	4 Y	4 Y	4 Y	4 Y	6 N	1 N	5 N
1 N	-	4 N	-	4 N	-	1 N	5 Y
6 Y	1 N	4 N	4 N	4 N	6 Y	6 Y	5 N
-	1 N	4 N	-	-	-	1 N	-
-	1 N	-	-	-	-	-	-

24	25	27	26	30	31	29	28
2 N	1 Y	1 Y	1 Y	4 N	3 N	1 Y	1 Y
2 Y	1 N	-	4 N	4 N	3 N	-	4 N
-	-	1 N	4 N	3 Y	3 Y	1 N	1 N
5 N	7 N	4 Y	4 Y	4 Y	5 N	1 N	4 Y
5 Y	1 N	4 N	4 N	4 N	5 Y	8 N	1 N
-	1 N	4 N	-	-	-	1 N	4 N
5 N	7 Y	4 N	-	-	-	1 N	-
-	1 N	-	-	-	5 N	8 Y	4 N

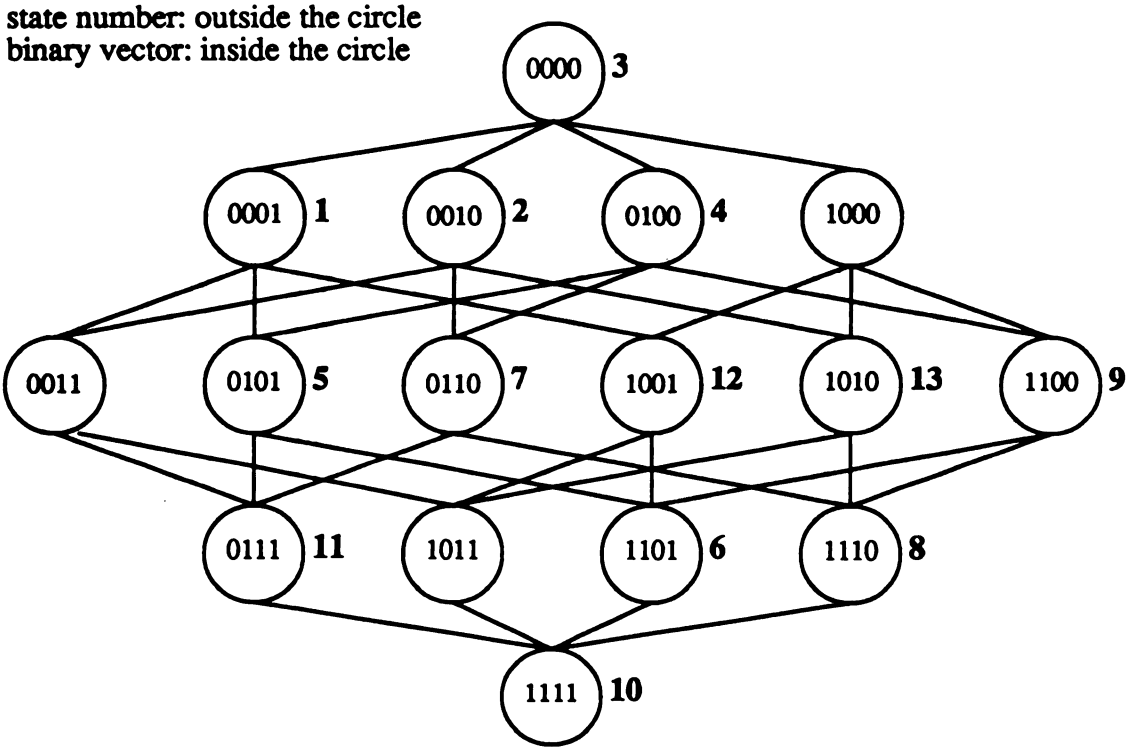
20	21	23	22	18	19	17	16
3 N	1 Y	1 Y	1 Y	4 N	3 N	1 Y	2 N
4 N	-	-	-	4 N	3 N	1 N	2 Y
3 Y	1 N	1 N	4 N	3 Y	3 Y	1 N	2 N
4 Y	1 N	5 N	4 Y	4 Y	4 Y	8 N	5 N
4 N	6 N	5 Y	4 N	4 N	4 N	1 N	5 Y
4 N	6 Y	5 N	-	-	-	1 N	5 N
-	1 N	5 N	-	-	-	1 N	-
-	1 N	-	-	-	4 N	8 Y	5 N

(b)

Figure 4-19 (cont'd).

	1(7)	2(3)	3(3)	4(7)	5(5)	6(3)	7(3)	8(3)
1(7)	S	Y	Y	Y	Y	Y	Y	Y
2(3)	Y	S	Y	Y	N	N	N	N
3(3)	Y	Y	S	Y	N	N	N	N
4(7)	Y	Y	Y	S	Y	Y	Y	Y
5(5)	Y	N	N	Y	S	Y	Y	Y
6(3)	Y	N	N	Y	Y	S	N	N
7(3)	Y	N	N	Y	Y	N	S	N
8(3)	Y	N	N	Y	Y	N	N	S

(c)



(d)

Figure 4-19 (cont'd).

	INPUT (JKCSR)							
	0	1	3	2	6	7	5	4
1	1 Y	1 Y	3 N	3 N	5 N	3 N	1 Y	1 Y
2	3 N	3 N	2 Y	2 Y	7 N	3 N	-	-
3	1 N	1 N	2 N	4 N	3 Y	3 Y	1 N	1 N
4	9 N	5 N	4 Y	4 Y	4 Y	7 N	3 N	4 Y
5	6 N	5 Y	4 N	4 N	4 N	4 N	1 N	4 N
6	6 Y	12 N	5 N	5 N	-	-	5 N	5 N
7	-	2 N	4 N	-	4 N	7 Y	7 Y	4 N
8	-	7 N	-	-	-	-	-	-
9	6 N	-	-	-	-	-	-	-
10	-	-	-	-	-	-	-	-
11	-	-	-	-	-	-	-	-
12	-	1 N	-	-	-	-	-	-
13	-	-	-	-	-	-	-	-

12	13	15	14	10	11	9	8
1 Y	1 Y	3 N	3 N	5 N	3 N	1 Y	1 Y
3 N	3 N	2 Y	2 Y	7 N	3 N	3 N	3 N
1 N	1 N	2 N	4 N	3 Y	3 Y	1 N	1 N
9 N	4 Y	4 Y	4 Y	4 Y	5 N	3 N	5 N
1 N	1 N	4 N	4 N	4 N	6 N	1 N	5 Y
6 Y	5 N	5 N	5 N	5 N	6 Y	6 Y	5 N
-	2 N	4 N	-	4 N	-	2 N	-
-	7 N	-	-	-	-	-	-
6 N	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-

Figure 4-19 (cont'd).

24	25	27	26	30	31	29	28
3 N	1 Y	1 Y	1 Y	5 N	3 N	1 Y	1 Y
2 Y	3 N	-	3 N	7 N	3 N	3 N	7 N
2 N	1 N	1 N	4 N	3 Y	3 Y	1 N	1 N
5 N	7 N	4 Y	4 Y	4 Y	5 N	3 N	4 Y
5 Y	1 N	4 N	4 N	4 N	5 Y	11 N	1 N
-	5 N	5 N	-	-	-	12 N	9 N
4 N	7 Y	4 N	-	4 N	4 N	2 N	4 N
-	13 N	-	-	-	7 N	8 Y	7 N
-	-	-	-	-	-	-	4 N
-	-	-	-	-	-	8 N	-
-	-	-	-	-	-	10 N	-
-	-	-	-	-	-	1 N	-
-	2 N	-	-	-	-	-	-

20	21	23	22	18	19	17	16
3 N	1 Y	1 Y	1 Y	5 N	3 N	1 Y	3 N
7 N	3 N	-	-	7 N	3 N	3 N	2 Y
3 Y	1 N	1 N	4 N	3 Y	3 Y	1 N	2 N
4 Y	3 N	5 N	4 Y	4 Y	4 Y	9 N	5 N
4 N	6 N	5 Y	4 N	4 N	4 N	1 N	5 Y
5 N	6 Y	5 N	-	-	-	5 N	5 N
4 N	2 N	4 N	-	4 N	4 N	2 N	4 N
-	7 N	-	-	-	7 N	8 Y	7 N
-	-	-	-	-	-	8 N	-
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-

(e)

Figure 4-19 (cont'd).

	1(3)	2(3)	3(3)	4(4)	5(4)	6(3)	7(3)	8(4)
1(3)	S	N	Y	N	Y	N	N	N
2(3)	N	S	Y	N	N	N	Y	N
3(3)	Y	Y	S	Y	N	N	N	N
4(4)	N	N	Y	S	Y	N	Y	N
5(4)	Y	N	N	Y	S	Y	N	N
6(3)	N	N	N	N	Y	S	N	N
7(3)	N	Y	N	Y	N	N	S	Y
8(4)	N	N	N	N	N	N	Y	S
9(3)	N	N	N	Y	N	Y	N	Y
10(2)	N	N	N	N	N	N	N	Y
11(2)	N	N	N	N	Y	N	N	N
12(2)	Y	N	N	N	N	Y	N	N
13(2)	N	Y	N	N	N	N	N	Y

9(3)	10(2)	11(2)	12(2)	13(2)
N	N	N	Y	N
N	N	N	N	Y
N	N	N	N	N
Y	N	N	N	N
N	N	Y	N	N
Y	N	N	Y	N
N	N	N	N	N
Y	Y	N	N	Y
S	N	N	N	N
N	S	Y	N	N
N	Y	S	N	N
N	N	N	S	N
N	N	N	N	S

(f)

Figure 4-19 (cont'd).

	INPUT (JKCSR)							
	0	1	3	2	6	7	5	4
1	1	1	0	0	5	0	1	1
2	0	0	2	2	6	0	-	-
0	1	1	2	4	0	0	1	1
4	12	5	4	4	4	6	0	4
5	13	5	4	4	4	4	1	4
13	13	9	5	5	-	-	5	5
6	-	2	4	-	4	6	6	4
14	-	6	-	-	-	-	-	-
12	13	-	-	-	-	-	-	-
15	-	-	-	-	-	-	-	-
7	-	-	-	-	-	-	-	-
9	-	1	-	-	-	-	-	-
10	-	-	-	-	-	-	-	-

12	13	15	14	10	11	9	8
1	1	0	0	5	0	1	1
0	0	2	2	6	0	0	0
1	1	2	4	0	0	1	1
12	4	4	4	4	5	0	5
1	1	4	4	4	13	1	5
13	5	5	5	5	13	13	5
-	2	4	-	4	-	2	-
-	6	-	-	-	-	-	-
13	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-

Figure 4-19 (cont'd).

24	25	27	26	30	31	29	28
0	1	1	1	5	0	1	1
2	0	-	0	6	0	0	6
2	1	1	4	0	0	1	1
5	6	4	4	4	5	0	4
5	1	4	4	4	5	7	1
-	5	5	-	-	-	9	12
4	6	4	-	4	4	2	4
-	10	-	-	-	6	14	6
-	-	-	-	-	-	-	4
-	-	-	-	-	-	14	-
-	-	-	-	-	-	15	-
-	-	-	-	-	-	1	-
-	2	-	-	-	-	-	-

20	21	23	22	18	19	17	16
0	1	1	1	5	0	1	0
6	0	-	-	6	0	0	2
0	1	1	4	0	0	1	2
4	0	5	4	4	4	12	5
4	13	5	4	4	4	1	5
5	13	5	-	-	-	5	5
4	2	4	-	4	4	2	4
-	6	-	-	-	6	14	6
-	-	-	-	-	-	14	-
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-

(g)

Figure 4-19 (cont'd).

Example 4-6: (No *IR's* Present)

An *ASLC's* flow table is given in Figure 4-20(a). It has 6 rows and 4 columns. A re-labeled *MST* (see Figure 4-20(b)) and its corresponding adjacency table (see Figure 4-20(c)) are generated by applying *Algorithm 1*. Because no link degrees are greater than the number of state variables, no *VIR's* exist. *Algorithm 2* is next applied for the purpose of identifying *HIR's*. An assignment tree is generated, (see Figure 4-20(d)); it shows that no *HIR's* exist either. Since no *VIR's* and no *HIR's* exist, the flow table is free of intrinsic races (*IR's*); hence, the modified flow table (see Figure 4-20(e)) is the same as the re-labeled flow table. Also, the adjacency table (see Figure 4-20(f)) corresponding to the modified flow table is the same as the one shown in Figure 4-20(c). From the assignment tree, a 3-*NWD* is generated (see Figure 4-20(g)), and race-free state assignments are made. The excitation table is shown in Figure 4-20(h).

Merged State	INPUT (BA) 0	1	3	2
	0 Y	4 Y	12 N	8 Y
	-	5 N	12 Y	8 N
	1 Y	5 Y	13 N	9 N
	0 N	7 N	13 Y	9 Y
	3 Y	7 Y	15 Y	11 N
	1 N	-	15 N	11 Y

(a)

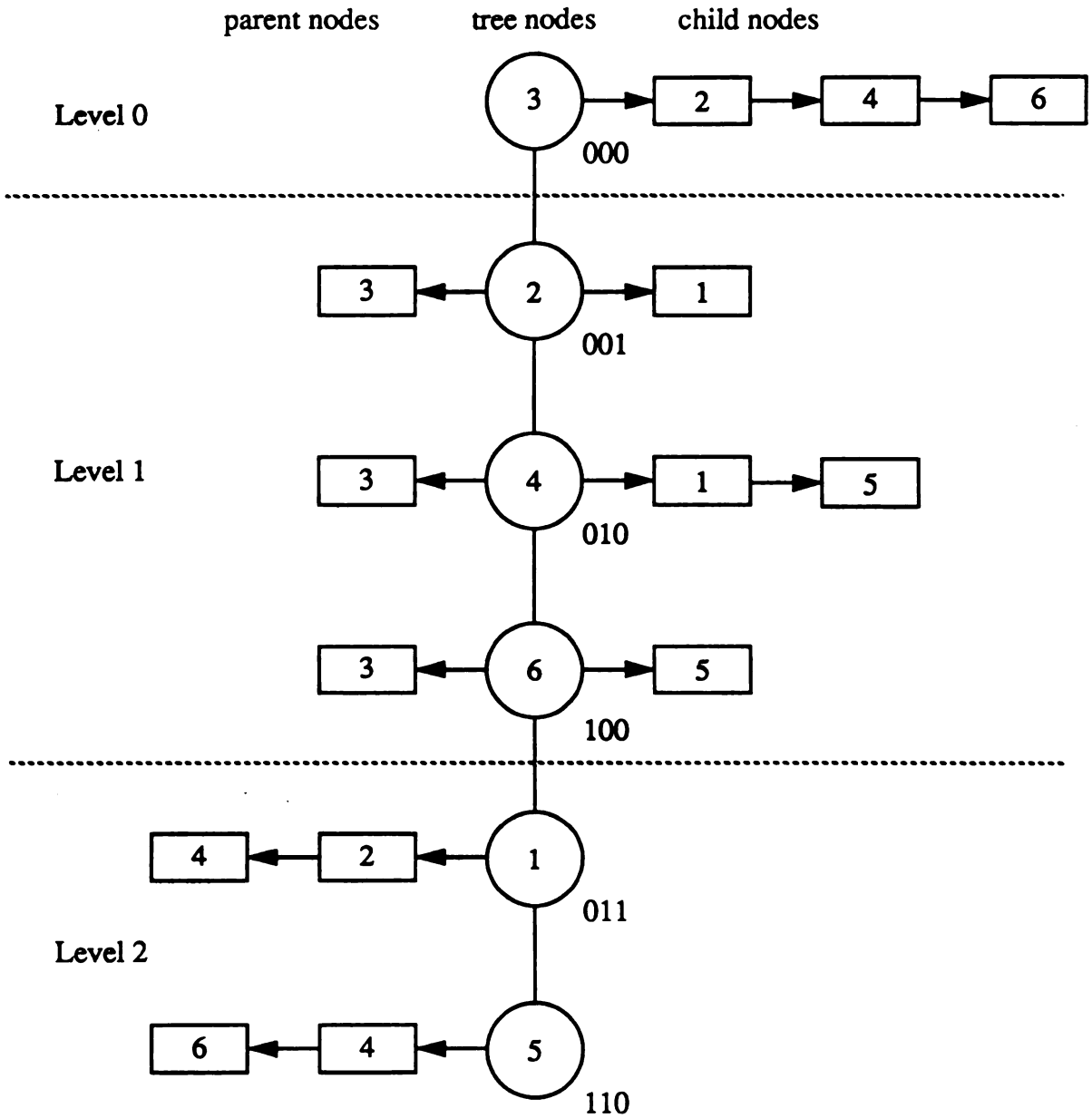
Merged State (relabeled)	INPUT (BA) 0	1	3	2
	1 Y	1 Y	2 N	1 Y
	-	3 N	2 Y	1 N
	3 Y	3 Y	4 N	4 N
	1 N	5 N	4 Y	4 Y
	5 Y	5 Y	5 Y	6 N
	3 N	-	5 N	6 Y

(b)

	1(2)	2(2)	3(3)	4(3)	5(2)	6(2)	Merged State (relabeled)
1(2)	S	Y	N	Y	N	N	
2(2)	Y	S	Y	N	N	N	
3(3)	N	Y	S	Y	N	Y	
4(3)	Y	N	Y	S	Y	N	
5(2)	N	N	N	Y	S	Y	
6(2)	N	N	Y	N	Y	S	

(c)

Figure 4-20. Example 4-6 (a)a merged state table; (b) relabeled *MST*; (c) adjacency table corresponding to the relabeled *MST*; (d) assignment tree; (e) modified state table; (f) adjacency table corresponding to the modified state table; (g) 3-*NWD*; (h) an excitation table due to system assignment.



(d)

Figure 4-20 (cont'd).

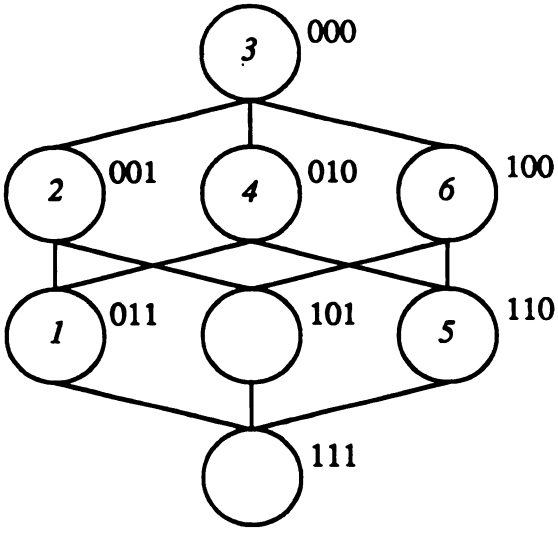
	INPUT (BA) 0	1	3	2	
1	1 Y	1 Y	2 N	1 Y	
2	-	3 N	2 Y	1 N	
3	3 Y	3 Y	4 N	4 N	Merged State (relabeled)
4	1 N	5 N	4 Y	4 Y	
5	5 Y	5 Y	5 Y	6 N	
6	3 N	-	5 N	6 Y	

(e)

	1(2)	2(2)	3(3)	4(3)	5(2)	6(2)	
1(2)	S	Y	N	Y	N	N	
2(2)	Y	S	Y	N	N	N	
3(3)	N	Y	S	Y	N	Y	Merged State (relabeled)
4(3)	Y	N	Y	S	Y	N	
5(2)	N	N	N	Y	S	Y	
6(2)	N	N	Y	N	Y	S	

(f)

Figure 4-20 (cont'd).



(g)

		INPUT (BA)			
		0	1	3	2
3	Present State	3	3	1	3
1		-	0	1	3
0		0	0	2	2
2		3	6	2	2
6		6	6	6	4
4		0	-	6	4

(h)

Figure 4-20 (cont'd).

4.5 Experimental Results and Comparisons

State assignment techniques for *ASLC*'s have been widely studied by many researchers [11-13, 33-41]. Most of the research relates to the single transition-time (*STT*) state assignment technique for fundamental-mode *ASLC*'s; i.e., only one digit in the input may change at a time, and no changes can take place in the input until the state machine stabilizes. *STT* state-assignment techniques were first presented by Liu [36] and later extended by Tracey [12]. Other improvements or variations on *STT* state assignments were studied by many researchers [13, 34-35, 37-39]. Tracey's methods [12] are important representative approaches of *STT* state assignments.

Tracey [12] described a method for finding minimum variable unicode *STT* assignments for normal, fundamental mode *ASLC's*. He noted that for "large" (8-12 rows) flow tables, the effort required to calculate minimum variable *USTT* codes became prohibitively complex. Therefore, he suggested two related algorithms which would require less effort, but would usually produce near minimum variable assignments.

Smith [13] demonstrated that for flow tables that contained more than 50 cells, even automated generation of minimum-variable assignments was not practical. Furthermore, he discovered that for very large flow tables (more than 150 cells), even Tracey's near-minimum variable methods required extremely large computation time. Smith noted that one 12- row by 4-column table (48 cells) from the literature requires 50 transition constraints and has more than 300 maximal intersectable classes. Even using very large computers, it has been economically impractical to calculate minimum variable assignments for tables with more than a few dozen constraints. Since the number of constraints is related to the number of next state entries rather than number of rows, Smith uses the number of cells (the product of the number of rows and columns) as a rough measure of flow table complexity. Smith also noted that none of Tracey's assignment

methods appear to be suitable for dense (mostly specified) flow tables of more than about 250 cells.

Smith presented an extension of Tracey's techniques which produces near-minimum variable assignments for very large tables, while requiring much less computational effort than previous methods. In the Fig. 9 in Reference [13], Smith gives a comparison of computational requirements for five assignment techniques. For the flow tables of *Machine 4* (table size 12 rows by 4 columns) and *Machine 6* (table size 6 rows by 4 columns), our approach yields fewer state variables and requires less CPU time than either Smith's or Tracey's state-assignment techniques. With the *Machine 4* example, Smith's and Tracey's assignments yield 6 state variables. In contrast, our method solves the race-free state assignment problem using only 5 state variables. For the *Machine 6* example, Smith's and Tracey's assignments use 4 state variables [13], Tan's assignment needs 12 state variables, Ullman's (Friedman's) assignment needs 9 state variables, Kuhl's assignment needs 6 variables [35], and our method requires only 3 state variables. All the experimental results that we have examined to date demonstrate that the *MSUASLC D*esign Automation System always yields the minimum number of state variables.

The *MSUASLC* currently runs on a Sun workstation. For *Machine 4*, Tracey's *Method D* requires 3.4 seconds (*CPU* time), Smith's method requires 3.3 seconds. In contrast, *MSUASLC* requires less than 10% of their shortest time. For *Machine 6*, both Smith's and Tracey's methods require approximately 1 second, while *MSUASLC* requires less than 0.1 second.

For a flow table with 250 cells, Smith's method needs 49.9 seconds, and Tracey's method needs 179.2 seconds [13]. However, *MSUASLC* takes less than 1 second to complete state assignment for a flow table with 256 entries (cells). Although these two tables (250 cells and 256 cells) differ slightly, the results demonstrate that *MSUASLC* reduces the computation time significantly.

The *One-Hot State-Assignment* method [5] is an interesting special class of unicode row assignment techniques. It is characterized by the fact that for each row in the flow table, exactly one of the state variables is assigned the value of 1. Therefore, a flow table with n rows will have n state variables. The advantage of this technique is that state equations can be obtained in a relatively straightforward manner. The disadvantage is that too many state variables are used; hence, the hardware cost can be quite high. Consider the Example Machine described by Unger and illustrated in Fig. 4-21 [5]. The number of state variables is 5 in using the *One-Hot State-Assignment* method and the state equations are as follows:

$$Y_1 = \bar{x}_1\bar{x}_2y_5 + x_1\bar{x}_2y_3 + y_1\bar{y}_2\bar{y}_3$$

$$Y_2 = \bar{x}_1x_2y_1 + y_2\bar{y}_3$$

$$Y_3 = x_1x_2y_1 + x_1x_2y_2 + y_3\bar{y}_1\bar{y}_4$$

$$Y_4 = \bar{x}_1\bar{x}_2y_3 + x_1x_2y_5 + y_4\bar{y}_5$$

$$Y_5 = x_1\bar{x}_2y_4 + y_5\bar{y}_1\bar{y}_4$$

By way of contrast, the number of state variables obtained using the *MSUASLC is* 3, and the state equations are as follows:

$$Y_0 = \bar{y}_2\bar{y}_1x_2 + \bar{y}_2\bar{y}_1\bar{y}_0x_1$$

$$Y_1 = y_1x_2 + y_1\bar{x}_1 + y_2x_1x_2$$

$$Y_2 = y_2x_2 + y_2x_1 + y_2y_1 + y_1\bar{x}_1\bar{x}_2$$

The race-free state-assignment method described here reduces the complexity of solving the race-free state-assignment problem by decomposing the problem into two principal stages. In the first stage, the flow table is scanned for intrinsic races (*IR's*). Visible intrinsic races (*VIR's*) and hidden intrinsic races (*HIR's*) are efficiently identified and eliminated. This stage yields a modified flow table that is free of intrinsic races. In the second stage, race-free state assignments are systematically made.

The node-weight diagram (*NWD*) was introduced and used in both stages of the race-free state-assignment method described here. This diagram is a variation of the binary **n-cube** connection diagram and provides a more convenient geometric representation of binary numbers for the purpose of making race-free state assignments than the n-cube. The *NWD* provides a framework for efficiently adding cycles and states to eliminate intrinsic races (*IR's*) and for guaranteeing that no generated races (*GR's*) are introduced when the symbolic states are assigned binary codes.

Many asynchronous state-machines have been synthesized using the state-assignment method described here and comparisons made with other state-assignment methods reported in the literature. Experimental results show that this method provides significantly better results than other approaches in terms of the computation time required to make the assignments and the number of state variables required to achieve race-free *ASLC's*.

	X_1X_2 00	01	11	10
1	(1)	2	3	(1)
2	-	(2)	3	-
3	4	(3)	(3)	1
4	(4)	-	(4)	5
5	1	(5)	4	(5)

Figure 4-21. Flow table for Unger's example machine

Chapter 5

Summary and Conclusions

5.1 Summary

The research reported here has focused on investigating the process of designing asynchronous sequential logical circuits (*ASLC*'s). For a given sequential logic function, the process of designing *ASLC*'s is significantly more complex than that of their clocked sequential logical circuit (*CSLC*) counterparts. This is due in part to critical-race and hazard problems that are associated with *ASLC* architectures. But, through appropriate state assignments, the race conditions can be avoided. Although it can be a very tedious process, the *ASLC* designer can verify that a particular design is functionally correct, as well as race and hazard free. Some computer-aided design (*CAD*) tools can be used to simulate the designed circuits to assist in the design verification process. But, because of the complexity of the overall *ASLC* design process, only the simplest sequential logic functions have been implemented using *ASLC* architectures. An automated set of design tools would enable state machine designers to evaluate alternative sequential logic function architectures more thoroughly before committing the design to a specific architectural implementation.

An *ASLC* design automation system, which is called the **MSUASLC Design Automation System**, has been developed and tested. Software was written in the *C* programming language with approximately 20,000 lines of source code. While the current version of this software runs on Sun workstations, it is readily transportable to other platforms. It is both modular and interactive and automatically validates the correctness of each step in the design process. Each module provides intermediate output information necessary to document the design. This design system significantly reduces the *ASLC* design cycle time.

The *MSUASLC Design Automation System* consists of five modules as illustrated in Figure 1-1. Each module can accept data files from either an up-stream module or interactively from the circuit designer. This modular *CAD* system architecture has clearly defined entry and exit points and permits each module (sub-system) to be accessed independently. Therefore, concurrent execution of these modules can be achieved for different design tasks.

Constructing a primitive flow table (*PFT*) is the first important step in designing an *ASLC*. Generating a large *PFT* by pen-and-paper methods may require several hours or days, and the results may be incorrect. In order to ensure a *PFT* match a required functional design specification, the generated *PFT* must be verified to guarantee a correct *ASLC* behavior. Therefore, verifying a *PFT* may take longer time than the generating a *PFT*. Many inputs/outputs can make flow tables unmanageably large. The *Behavioral Descriptor* (*BD*) overcomes these complexity problems. The *BD* uses an artificial intelligence approach to map the functional design specification into a primitive flow table (*PFT*), which completely captures the sequential logic function's behavior. For a table with 2^{12} entries, the *BD* can generate the *PFT* in about one second. Therefore, the *BD* can quickly generate a very large *PFT*'s. This significantly reduces the time for generating and verifying the *PFT*.

Merging a large flow table is more time consuming and more difficult than generating a large flow table. It becomes almost impossible to draw a merger diagram by hand from a large *PFT* and to identify the strongly connected subgraphs by visual process from a complex merger diagram. The *Merger* uses graph techniques to reduce the complexity of the merging process. It reduces the *PFT* into a merged flow table and merged output table, thereby minimizing the number of states by eliminating redundant primitive state assignments. Specific steps in this state merging process are as follows: The merger diagram (*MD*) and the merged flow table (*MFT*) are initialized. The *MD* is completed by identifying identical rows in the *PFT*. Each set of identical rows in the *PFT* becomes a new row in the *MFT* and is assigned a new symbolic state name. A given primitive state may be capable of being merged in more than one way, but primitive states must only be assigned to one of these merged states. Therefore, additional constraints are necessary in order to decide which grouping is best. One such constraint might be to require that only primitive states with identical output states be merged. Four different merging methods are provided in the *Merger*. The major purpose is to give designers or researchers more opportunities to investigate the merged results. The traditional merging method (identifying the largest strongly connected subsets) is not included in the *Merger* module because it takes too much computation time (effort), especially when the *PFT* is large. As indicated in Chapter 3, the traditional merging method does not guarantee a minimum number of merged states, i.e., rows in the merged flow table.

Verifying all the allowed state transitions and modifying the *MFT* to avoid races are very difficult in the design of *ASLC*'s, especially for a large *MFT*'s. The *Connector* and *Assigner* overcome the difficulty of making race-free state assignments. The race-free state-assignment method described here reduces the complexity of solving the race-free state-assignment problem. The node-weight diagram (*NWD*) is introduced. It is a variation of the binary n-cube connection diagram and provides a more convenient geometric

representation of binary numbers for the purpose of making race-free state assignments than the n-cube. The *NWD* provides a framework for efficiently adding cycles and states to eliminate intrinsic races (*IR's*) and for guaranteeing that no generated races (*GR's*) are introduced when the symbolic states are assigned binary codes. Experimental results show that this method provides significantly better results than other approaches in terms of the computation time required to make the assignments and the number of state variables required to achieve race-free *ASLC's*.

The process of mapping symbolic states into race-free binary-coded state vectors is not unique; i.e., more than one set of race-free state assignments can be generated for a given flow table. The approach described here is currently being used to investigate alternative *ASLC* implementations for a given sequential-logic element specification. The objective of this investigation is to develop rules for determining the best set of state assignments for a specific set of *ASLC* design constraints.

Finding consensus terms to eliminate hazards is also a time-consuming and difficult process, especially when the excitation tables and output tables become large. The purpose of most Boolean minimization tools [42-45] is to minimize a given switching function, so they do not provide the function for eliminating hazards. The *Equation Generator* quickly generates hazard-free *ASLC* state equations and output equations. The *Equation Generator* module reads the state excitation table and modified output table and applies the Quine-McCluskey algorithm [46] to generate the state equations and output equations. These equations are expressed in two-level, sum-of-products form. Static hazards are identified by searching each of these equations for adjacent pairs of prime implicants. If a pair of adjacent prime implicants do not possess a consensus term, it is added to eliminate the static hazard. The *Equation Generator* consists of two major parts: one is *Boolean Generator*; the other is *Hazard Eliminator*. The *Boolean Generator* generates a minimized Boolean function. The *Hazard Eliminator* realizes the Boolean function (generated by the *Boolean Generator*) in its hazard-free, two-level sum-of-products form.

One primary objective of the work reported here is to map human (expert) knowledge into the implementation of an *ASLC* design system to speed up the overall *ASLC* design process. This system lessens the burden on sequential logic function circuit designers by greatly reducing the chances that errors would creep into the design and by greatly reducing the overall design cycle time. This increase in designer productivity could be used in part to explore alternative implementations for purposes of optimizing the overall circuit's performance.

The *MSUASLC Design Automation System* not only overcomes the design complexity and difficulty but also provides for design flexibility. Different merging methods may generate different *MFT*'s. Different state assignments may generate different excitation tables and output tables. Any of these may result in a different *ASLC* implementation. Designers can choose the one which satisfies their particular requirements.

Results of the research reported here will impact the *ASLC* design process as follows:

1. It provides a tool that will enable researchers to investigate the general implications of alternative *ASLC* implementations. The intent would be to develop a general set of rules for optimizing designs.
2. It provides a tool that will enable *ASLC* designers to explore alternative *ASLC* implementations for purposes of optimizing a given design.
3. It provides a means to design complex *ASLC*'s (large-scale *ASLC*'s) and thereby provide an alternative to the *CSLC* implementation of sequential circuits.
4. It provides a tool to help the researchers or designers investigate specific designs such as fault-tolerant or testable *ASLC* designs. Researchers or designers can modify the tables generated or can make tables themselves and then submit these tables to the system to generate the *ASLC* equations which satisfy their design constraints.

5.2 Future Research and Development

One important research issue that has not yet been fully addressed is the design of fault-tolerant or testable *ASLC*'s. Since using the *MSUASLC Design Automation System* can quickly generate the *ASLC* equations, one can readily investigate the internal characteristics of the *ASLC*'s. In order to explore fault-tolerant or testable *ASLC* design, some redundant states may be added to make the circuit have some specific features, or some special arrangements can be made to the flow table, state assignments, excitation tables, or output tables. Based on these, researchers may develop rules for fault-tolerant or testable *ASLC* design. Because of its modular design, the *MSUASLC Design Automation System* can be easily expanded. We can expand the capability of *MSUASLC*, e.g., add some features to allow the design automation for fault-tolerant or testable *ASLC* design.

Another research issue could be the asynchronous control portion of an *ASIC* sequential logic function. The design time of an *ASIC* is only as fast as the design time of its slowest portion. Keutzer [47] pointed out the lack of synthesis and verification procedures for asynchronous and analog portions of circuits is a severe problem. The *MSUASLC Design Automation System* can be used to assist the research in *ASIC* design. Some specific asynchronous control applications can also be explored or implemented. One interesting research issue would be in the prediction and avoidance of hazard conditions.

LIST OF REFERENCES

LIST OF REFERENCES

[1] E. J. McCluskey, *Logical Design Principles*, Prentice Hall, Englewood Cliffs, New Jersey, 1986.

[2] L. A. Hollaar, "Direct Implementation of Asynchronous Control Units," *IEEE Trans. Computers*, vol. C-31, pp. 1133-1141, Dec. 1982.

[3] K. J. Breeding, *Digital Design Fundamentals*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[4] A. D. Friedman, *Fundamentals of Logic Design and Switching Theory*, Computer Science Press, Inc., Rockville, Md.,1986.

[5] S. H. Unger, *Asynchronous Sequential Circuits*, Wiley Interscience, New York, 1969.

[6] Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill, New York, 1978.

[7] T. H.-Y. Meng, R. W. Brodersen, and D. G. Messerschmitt, "Asynchronous Logic Synthesis for Signal Processing from High-Level Specifications," *IEEE ICCAD'87 Dig. Technical Papers*, pp. 514-517, Nov. 1987.

[8] A. E. A. Almaini, *Electronic Logic Systems*, 2nd ed., Prentice-Hall International (UK) Ltd., 1989.

[9] A. L. Fisher and H. T. Kung, "Synchronizing Large VLSI Arrays," *IEEE Trans. Computers*, vol. C-34, pp. 734-740, Aug. 1985.

[10] C. E. Molnar, T. P. Fang, and F. U. Rosenberger, "Synthesis of delay-insensitive modules," in *Proc. 1985 Chapel Hill Conf. VLSI*, Chapel Hill, NC, May 15-17, 1985, pp. 67-86.

[11] T. Nanya and Y. Tohma, "On Universal Single Transition Time Asynchronous State Assignments," *IEEE Trans. Computers*, vol. C-27, pp. 781-782, Aug. 1978.

[12] J. H. Tracey, "Internal State Assignments for Asynchronous Sequential Machines," *IEEE Trans. Elect. Computers*, vol. EC-15, pp. 551-560, August 1966.

[13] R. J. Smith, II, "Generation of Internal State Assignments for Large Asynchronous Sequential Machines," *IEEE Trans. Computers*, vol. C-23, pp. 924-932, Sep, 1974.

[14] S. H. Unger, "Hazards and Delays Detection in Asynchronous Sequential Switching Circuits," *IRE Trans. Circuit Theory*, vol. CT-6, pp. 12-25, March 1959.

[15] S. B. Lerner, "Hazard Correction in Asynchronous Sequential Circuits," *IEEE Trans. Electron. Computers*, vol. EC-14, pp. 265-267, Apr. 1965.

[16] W. S. Meisel and R. S. Kashef, "Hazards in Asynchronous Sequential Circuits," *IEEE Trans. Computers*, vol. C-18, pp. 752-759, Aug. 1969.

[17] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra, "Automatic Verification of Sequential Circuits Using Temporal Logic," *IEEE Trans. Comput*ers, vol. C-35, pp. 1035-1044, Dec. 1986.

[18] C. Berthet and E. Cerny, "An Algebraic Model for Asynchronous Circuits Verification," *IEEE Trans. Comput*ers, vol. C-37, pp. 835-847, July 1988.

[19] T. S. Chen, and P. D. Fisher, A *Complete Logical Design System for Application Specific CMOS Digital Integrated Circuits*, Technical Report, Department of Electrical Engineering, Michigan State University, E. Lansing, May 1988.

[20] Wu, Sheng-Fu and P. David Fisher, "Automating the Design of Asynchronous Sequential Logic Circuits", *IEEE Journal of Solid-State Circuits*, Vol. 26, pp. 364-370 March 1991.

[21] S. F. Wu and P. D. Fisher, "Automating the Design of Asynchronous Sequential Logic Circuits," *IEEE 1990 Custom Integrated Circuits Conference*, pp. 29.5.1 - 29.5.4, May 1990.

[22] S. F. Wu and P. D. Fisher, "An Artificial Intelligence Approach to the Behavioral Modeling of Asynchronous Sequential Logic Circuits," *33rd Midwest Symposium on Circuits and Systems*, August 1990 (in press).

[23] I. E. Sutherland, "Micropipelines," *Comm. of the ACM*, vol. 32, pp. 720-738, June 1989.

[24] S. F. Wu and P. D. Fisher, *MSUASLC Design Automation System -- Technical Report*, Department of Electrical Engineering, Michigan State University, 1991.

[25] V. E. Kelly, "The Critter Systems: Automated Critiquing of Digital Circuit Designs," *Proceedings of the 21st ACM/IEEE Design Automation Conference,* pp. 419-425, IEEE and ACM-SIGDA, IEEE Computer Society, June 1984.

[26] R. Joobbani, *An Artificial Intelligence Approach to VLSI Routing,* Kluwer Academic Publishers, Boston/Dordrecht/Lancaster, 1986.

[27] R. Joobbani and D. P. Siewiorek, "WEAVER: A Knowledge-Based Routing Expert," *Proceedings of the 22nd ACM/IEEE Design Automation Conference,* pp. 266-272, IEEE and ACM-SIGDA, IEEE Computer Society, June 23-26, 1985.

[28] W. P. Bermingham, and J. H. Kim, "DAS/Logic: A Rule-based Logic Design Assistant," *Proceedings of the Second Conference, Miami Beach, Florida, Artificial Intelligence Applications The Engineering of Knowledge-Based Systems,* pp. 264-268, IEEE Computer Society Press, Dec. 1985.

[29] W. P. Birmingham, D. P. Siewiorek, "MICON: A Knowledge Based Single Board Computer Designer," *Proceedings of the 21st ACM/IEEE Design Automation Conference,* pp. 565-571, IEEE and ACM-SIGDA, IEEE Computer Society, June 25-27, 1985.

[30] L. I. Steinberg, T. M. Mitchell, "A Knowledge-based Approach to VLSI CAD," *Proceedings of the 21st ACM/IEEE Design Automation Conference,* pp. 412-418, IEEE and ACM-SIGDA, IEEE Computer Society, June 25-27, 1985.

[31] W. B. Rauch_Hindin, *Artificial Intelligence in Business, Science, and Industry,* Prentice Hall, Englewood Cliffs, New Jersey, 1985.

[32] E. Rich, *Artificial Intelligence,* McGraw-Hill, Inc., New York, 1983.

[33] D. A. Huffman, "The Synthesis of Sequential Switching Circuits," *J. Franklin Inst.,* vol. 257, pp. 161-190, March 1954 and pp. 275-303, April 1954.

[34] C. J. Tan, "State Assignments for Asynchronous Sequential Machines," *IEEE Trans. Computers,* vol. C-20, pp. 382-391, Apr. 1971.

[35] J. G. Kuhl, and S. M. Reddy, "A Multicode Single Transition-time State Assignment for Asynchronous Sequential Machines," *IEEE Trans. Comput.,* vol. C-27, pp. 927-934, Oct. 1978.

[36] C. N. Liu, "A State Variable Assignment Method for Asynchronous Sequential Switching Circuits," *J. ACM,* vol. 10, pp. 209-216, April 1963.

[37] G. K. Maki, J. H. Tracy, and R. J. Smith, II, "Generation of Design Equations in Asynchronous Sequential Circuits," *IEEE Trans. Computers,* vol. C-18, pp. 467-472, May 1969.

[38] A. D. Friedman, R. L. Graham, and J. D. Ullman, "Universal Single Transition Time Asynchronous State Assignments," *IEEE Trans. Computers,* vol. C-18, pp. 541-547, June 1969.

[39] T. Nanya and Y. Tohma, "Universal Multicode STT State Assignments for Asynchronous Sequential Machines," *IEEE Trans. Comput.,* vol. C-28, pp. 811-818, Nov. 1979.

[40] G. K. Maki, and J. H. Tracy, "A State Assignment Procedure for Asynchronous Sequential Circuits," *IEEE Trans. Computers,* vol. C-20, pp. 666-668, June 1971.

[41] G. Saucier, "Encoding of Asynchronous Sequential Networks," *IEEE Trans. on Elec.Computers,* pp. 365-369, June 1967.

[42] M. R. Dagenais, V. K. Agarwal and N. C. Rumin, "McBoole: A New Procedure for Exact Logic Minimization," *IEEE Trans. on CAD*, pp. 229-238, Jan. 1986.

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

[44] Richard L. Rudell, *Multiple-Valued Logic Minimization for PLA Synthesis*, Thesis, University of California, Berkeley, June 1978.

[45] S. J. Hong, R. G. Cain and D. L. Ostapko, "MINI: A Heuristic Approach for Logic Minimization," *IBM J. of R&D*, pp. 443-458, Sep. 1974.

[46] E. J. McCluskey, Jr., "Minimization of Boolean Functions," *Bell Syst. Tech. J.*, vol. 35, pp. 1417-1444, Apr. 1957.

[47] Kurt Keutzer, "Three Competing Design Methodologies for ASIC's: Architectural Synthesis, Logic Synthesis and Module Generation," *26th ACM/IEEE Design Automation Conference*, pp. 308-313, June 1989.