Automatic Verification of VHDL Models

by

Raghu Ardeishar

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

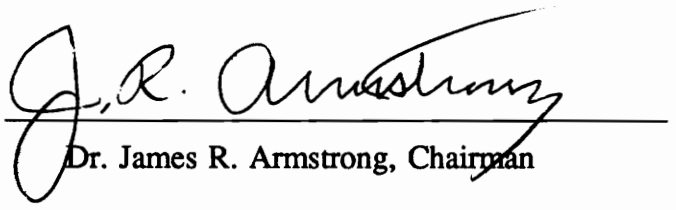
in partial fulfillment of the requirements for the degree of

Master of Science

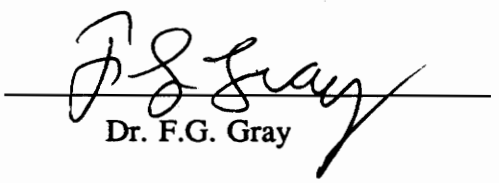
in

Electrical Engineering

APPROVED:



Dr. James R. Armstrong, Chairman



Dr. F.G. Gray

Dr. W.R. Cyre

July 1990
Blacksburg, Virginia

Automatic Verification of VHDL Models

by

Raghu Ardeishar

Dr.James R. Armstrong, Chairman

Electrical Engineering

(ABSTRACT)

Verification of a model describing a hardware system is very important for modeling and simulation purposes. It is necessary to ensure that the model accurately describes the hardware system. A scheme for the automatic verification of VHDL (VHSIC Hardware Description Language) models has been proposed. In the proposed scheme the specifications for the hardware system, i.e.,the timing constraints and relations between input and output signals are described by the designer in Modified Linear Time Temporal Logic, which is an extension to traditional boolean logic and can describe timing relation between signals. A semantic similarity between temporal operators and VHDL timings and delays has been drawn and an algorithm for comparing the VHDL model and temporal specifications has been developed. Comparisons are made between the simulation results on the VHDL model and the temporal logic specifications and discrepancies are reported.

Acknowledgements

I would like to thank my advisor Dr. James Armstrong for his guidance and support. I would also like to thank my parents and my brother for their encouragement. I would also like to thank Dr. Gray , Dr. Cyre for agreeing to serve on my committee. I would also like to thank Dr. Midkiff for providing help in correcting my thesis. I would like to thank Sriram from the Computer Science department for helping me. Finally thanks to my Bob Lineberry and my friends in the CRL lab.

Table of Contents

Chapter 1. Introduction 1

1.1 Model Verification 1

1.2 Contributions 3

1.3 Outline of Chapters 4

Chapter 2. Modified Linear Time Temporal Logic 6

2.1 Timings and Delays in VHDL 6

2.2 Operators in Modified Linear Time Temporal Logic 8

2.2.1 Boolean Operators 8

2.2.2 Arithmetic Operators 8

2.2.3 Temporal Operators 9

2.2.3.1 Description of Temporal Operators 9

2.3 Timing Diagram Specifications in Temporal Logic 14

2.4 Specification of Intervals in Temporal Logic 15

2.5 Specification of Variables in Temporal Logic 16

2.6 Parallelism in Temporal Logic 17

2.7 Example of a JK Flip-Flop 18

Chapter 3. Simplification of The Temporal Logic Formulae 20

Chapter 4. Verification of the VHDL model 24

4.1 Creation of Test Vectors 26

4.2 Verification of Simulation Results 27

4.2.1 Data Structures 30

4.2.1.1 Data Structures for Lexical Analysis and Parsing . 30

4.2.1.2 Data Structures for Verification 35

4.2.2 Modified Operator Precedence Parsing 37

4.2.2.1 Modified Operator Precedence Parsing Algorithm . 40

4.2.3 Reduce Algorithm 45

4.2.4 Verify Algorithm 47

Chapter 5. VHDL Model Verifier User's Manual 56

5.1 Creation of the Temporal Logic file 57

5.2 Using the Model Verifier 59

5.3 Verification of a Traffic Light Controller 63

5.3.1 Temporal Description of the Traffic Light Controller 65

5.3.2 Analyses of Verification Results 67

Chapter 6. Conclusions 72

References 74

Appendix A 76

A.1 Incorrect VHDL Model of Traffic Light Controller 76

A.2 Simulation Results of Incorrect VHDL Model 78

Appendix B. 85

B.1 Corrected VHDL Model of Traffic Light Controller 85

B.2 Simulation Results of Corrected VHDL Model 87

List of Illustrations

Figure 1. @(P > Q) 10

Figure 2. @(P iff Q) 10

Figure 3. @(P > #Q) 11

Figure 4. @(P > #~Q) 11

Figure 5. @(%P > Q) 12

Figure 6. @(%~P > Q) 12

Figure 7. @(P > next Q) 13

Figure 8. P while Q 13

Figure 9. P U Q 14

Figure 10. Falling Edge Representation 14

Figure 11. Rising Edge Representation 15

Figure 12. Intervals 16

Figure 13. Verification Process 25

Figure 14. Finite State Machine for Lexical Analyses 31

Figure 15. State Diagram for Traffic Light Controller 64

List of Tables

Table 1. Operator Precedence Table 29

Table 2. Lexical Analyses 34

Table 3. Sample Simulation Results 1 47

Table 4. Stack Operations 51

Table 5. Verification Output Code 52

Table 6. Sample Simulation Results 2 53

Table 7. Verify Algorithm Implementation 54

Table 8. Test Bench for Traffic Light Controller 69

Table 9. Verification Results of Incorrect VHDL Model 70

Table 10. Verification Results of Corrected VHDL Model 71

Chapter 1. Introduction

1.1 Model Verification

As circuits become more and more complex, good methods for specifying and validating a system and its submodules become more and more important. A problem related to the use of timing modeling is that of model validation. Chip-level models of large devices are complicated and their validation is a nontrivial task. Most methods existing today verify the validity on a DC basis, i.e. if they are logically correct, but no timing can be verified. A question can be raised as to the need to validate a model when the real chip can be used in simulation. The answer is that a simulation model allows for control of the response, e.g. time delays can be varied, to achieve the desired result, whereas with the use of a real chip, one has no control over the internal characteristics of the model. In this thesis we explore the possibility of automatic verification of VHDL models. VHDL is now an industry standard for hardware description languages. It incorporates structured programming and has many features that aid in the design of complex hardware systems. But as the system model becomes more and more complex it becomes exceedingly difficult to verify its validity manually. There thus exists a need to automatically and formally verify and validate the model describing a hardware system.

For the purpose of verification we formally specify the working of the hardware system i.e. the relation between the signals in MODIFIED LINEAR TIME TEMPORAL LOGIC, which is an extension to traditional boolean logic. Temporal logic was proposed for the specification of systems [1,4,5,6,7,8,9,10,11,12,13]. While traditional boolean logic is useful for specifying system states that are possible at some given time, temporal logic provides additional facilities for specifying possible state sequences, at different periods of time. Specifications in temporal logic can also describe in great detail the timing relations between the signals in a system. Timing in temporal logic has no real time concept but only considerations for eventuality, with some tolerance in time. Thus the approach of using temporal logic to specify systems gives great flexibility since the design of most digital modules does not require real-time considerations. Once we have specified the hardware system in temporal logic we are left with the task of verifying the model, i.e. checking to see if the VHDL model conforms to the specification of the hardware system in Modified Linear Time Temporal Logic. This research focuses on the development of a tool called the VHDL Model Verifier that accomplishes the above task. The tool checks the conformity between, the results obtained after simulating the VHDL model and the temporal logic description of the hardware system (Section 4.2).

The next section details the contributions made in the development of the VHDL Model Verifier.

1.2 Contributions

1. The reason for using temporal logic to describe VHDL models is because of the similarity of timing constructs between the two. A similarity in semantics between the timing constructs of VHDL and operators in temporal logic is developed. For example the temporal operator next is equivalent to the delta delay in VHDL and the # (eventually) operator in temporal logic is similar to the transport delay in VHDL. The similarities are detailed in Section 2.2.3.

2. An additional operator % (set-up and hold time) is introduced in Modified Linear Time Temporal Logic. This operator can be understood as being equivalent to the inertial delay in VHDL (see Section 2.2.3).

3. The semantics of temporal logic was expanded to include BIT_VECTORS (an array of bits) and numbers, both binary and hexadecimal. Also included were the arithmetic operators +,-,/ and *.

4. The operator precedence algorithm [14] is modified to incorporate the time factor associated with signals (see section 4.2.2). Also modifications are introduced to incorporate simplification of temporal logic formulae (see Chapter 3).

5. A reduction algorithm (Section 4.2.3) is developed to take actions once the parsing of the input temporal logic formula is complete.

6. A verification algorithm (Section 4.2.4) is developed that reads simulation results and verifies the conformity between the simulation results and the temporal logic formulae. The verification algorithm checks the validity of the model for every signal change.

7. The algorithms mentioned above were implemented and tested on a variety of models. The models verified include a JK flip-flop, a three stage synchronous counter, a traffic light controller (Chapter 5) and a memory unit.

In the next section an outline of the chapters in the thesis is given.

1.3 Outline of Chapters

Chapter 2 gives an overview of temporal logic and explains system specifications using it. Examples are given that explain using temporal logic to specify intervals, timing diagrams, variables, and numbers. Also given is an example of temporal logic specification of a JK flip-flop. Chapter 3 provides simplification procedures that are followed to ease the automation process. The chapters mentions the different identities

used in the simplification process. Chapter 4 then explains the algorithm for verifying the hardware system using the temporal logic description. The modified operator precedence algorithm, the reduce algorithm and the verify algorithm are explained in detail with examples. In Chapter 5 we give a detailed description of using the software, the VHDL Model Verifier, to validate models. An example of a traffic light controller is given to explain the use of the verifier.

Chapter 2. Modified Linear Time Temporal Logic

In this chapter we introduce Modified Linear Time Temporal Logic. We also show techniques of using it to specify hardware systems. Temporal logic is an extension of traditional boolean logic. In addition to the traditional boolean operators it has mathematical and temporal operators. The temporal operators are @ (always), # (eventually), % (set-up and hold time), next, while and U (Until). The list of operators with their meanings is given in Section 2.2. Before we proceed to introduce temporal logic we would like to give a brief description of VHDL timing constructs.

2.1 Timings and Delays in VHDL

Chip level modeling requires accurate representation of input/output timing [2]. The most important behavioral statement in VHDL to represent input/output timing is the signal assignment statement [3]. Signals are used to transmit information between processes. The simplest form of signal assignment statement is

signal_name <= value ;

The above signal assignment assigns *value* to the current value of the signal at the

beginning of the next simulation cycle. The left hand side of the signal assignment statement is referred to as the target of the assignment. When no explicit delay is given in the signal assignment statement a **delta delay** is implied. The delta delay is the smallest finite time in which a signal can change. While variables in VHDL can change immediately, i.e. without any delay, signals need at least one delta cycle before they can assume a new value.

It is also possible to assign a finite delay to a signal assignment statement. There are two types of delays that can be applied when assigning a time/value pair to the driver of a signal: **transport** and **inertial**. Transport delay is analogous to the delay incurred by passing a current through a wire. Inertial delay is used for devices that do not respond unless the value on its input persists or remains stable for a given amount of time. Inertial delay is useful in modeling devices that ignore spikes on their inputs. Given below are two examples that explain the two forms of delays:

P <= Q after 10ns ; **Inertial Delay**

P <= transport Q after 10ns ; **Transport Delay**

The first assignment statement implies inertial delay. P will assume the value Q if and only if the input persists at a given level for the specified period of time, in this case 10 ns. In the second assignment statement, where transport delay is specified, all changes in Q will propagate to P after the specified time delay, irrespective of how long the changes persist at a new level. In VHDL inertial delay is implied by the signal assignment statement unless the word transport is used.

In modified temporal logic there are operators that are equivalent to the delays in VHDL. The temporal operator # (eventually) is equivalent to the transport delay, **next** is equivalent to the delta delay and % (set-up and hold time) is equivalent to the inertial delay. The operators in temporal logic are explained in the next section.

2.2 Operators in Modified Linear Time Temporal Logic

2.2.1 Boolean Operators

Boolean operators in temporal logic are as follows.

and : Logical AND

or : Logical OR

~ : Logical not

Other boolean operators can be written as a combination of the above operators

2.2.2 Arithmetic Operators

Arithmetic operators in temporal logic are as follows.

+ : Binary addition

- : Binary Subtraction

/ : Binary division

* : Binary multiplication

= : Equals

The arithmetic operators are used to specify operations between BIT_VECTORS. During

verification the BIT_VECTORS are converted into the corresponding integer value. For example if the specification is Q = "110" + "#1" the verifier will check if Q is equal to 7 (111).

2.2.3 Temporal Operators

The following operators specify temporal relations.

> : implication

iff : if and only if (double implication)

@ : always

: eventually (within a certain time period)

% : set-up and hold time

next : next time instant

while

U : until

2.2.3.1 Description of Temporal Operators

P (with no operators) : P is true at present and nothing is specified about the future.

@P : P is true at present and at every time in the future.

For example as shown in figure 1, the assertion $@(P > Q)$ means that whenever P is true at some particular time , then Q is also true . Nothing is stated about the value of Q when P is false. @P can also be stated as P being invariant.

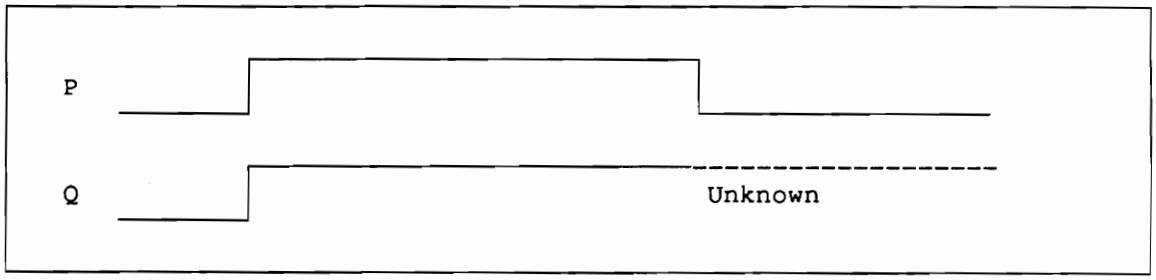


Figure 1. @ (P > Q)

iff : if and only iff (double implication)

For example as shown in figure 2, the assertion *@(P iff Q)* means that whenever P is true at any particular time, then Q is also true and vice versa. Also if P is false at any time, then Q is also false and vice versa.

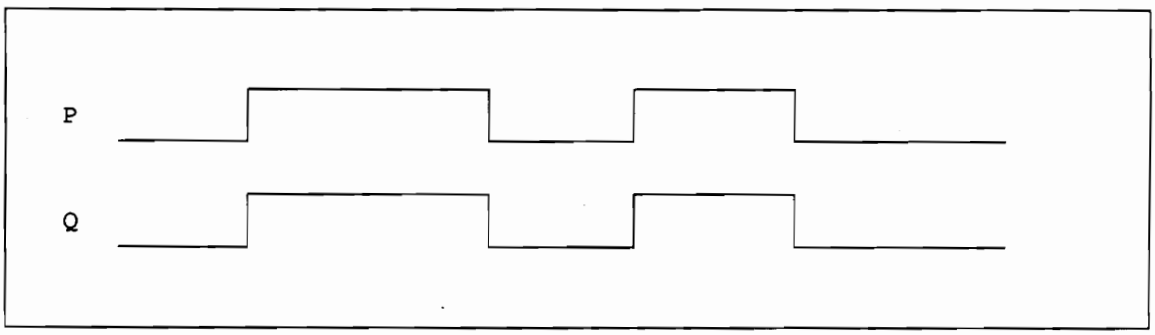


Figure 2. @ (P iff Q)

#P : P is true sometime in the present or the future.

For example as shown in figure 3, the assertion *@(P > #Q)* means that if P is true at some given time, e.g. the present time, then Q will eventually become true. This operator can be used to specify the eventual response of a module, with certain tolerance to some request. The operator # has an upper bound (in nanoseconds) associated with it, which has to be provided by the user during verification (see Chapter 5).

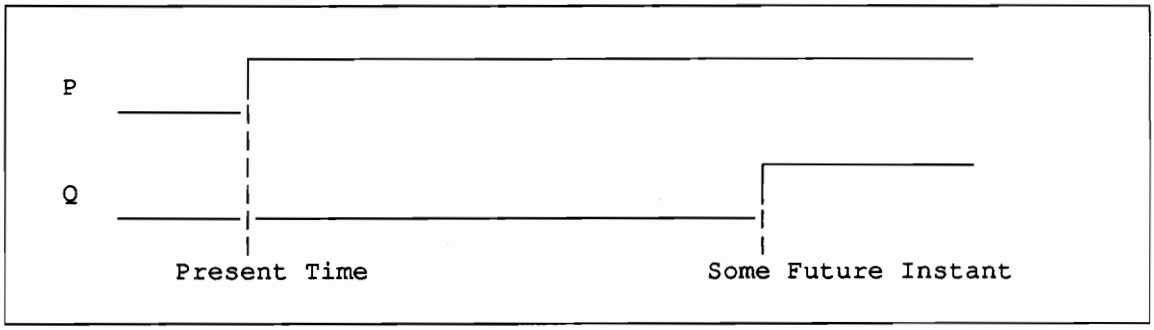


Figure 3. @(P > #Q)

 # ~P : P is false sometime in the present or the future.

For example as shown in figure 4, the assertion *@(P > #~Q)* means that if P is true at any given time, e.g. the present time, then Q will eventually become false. Again this operator can be used to specify the eventual response of a module, with certain tolerance to some request.

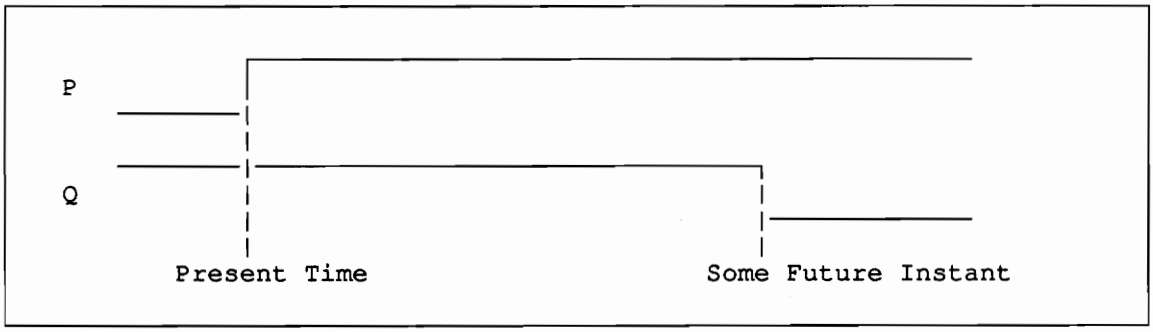


Figure 4. @(P > #~Q)

% P : P is true for a specified period of time.

For example as shown in figure 5, the assertion *@(%P > Q)* means that if P has remained true for some given time, analogous to the inertial delay in VHDL, then Q becomes true. This operator can be used to specify the response of a module to spikes in the input. The operator % also has an upper bound (in nanoseconds) associated with it,

which has to be provided by the user during verification (see Chapter 5).

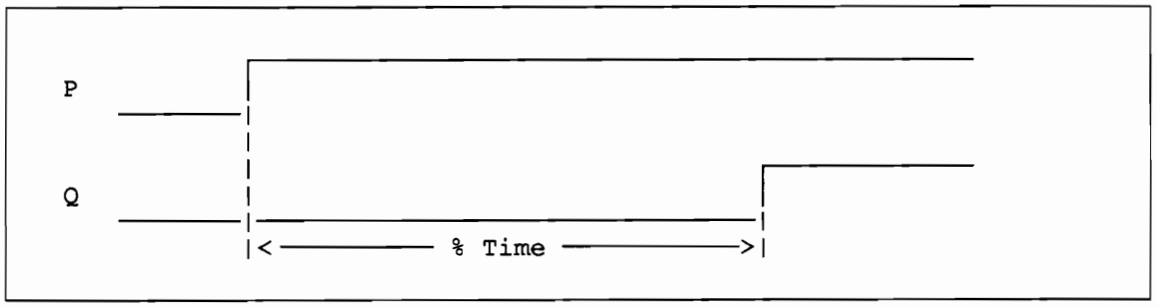


Figure 5. @(%P > Q)

% ~P : P is false for a specified period of time.

For example as shown in figure 6, the assertion *@(%~P > Q)* means that if P has remained false for some given time, then Q becomes true.

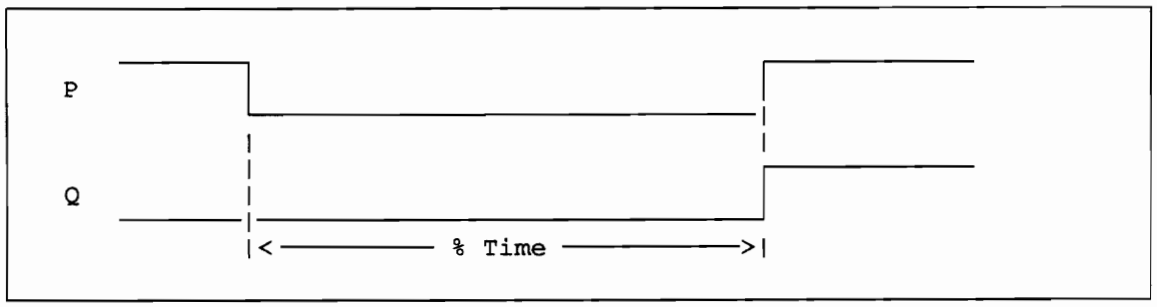


Figure 6. @(%~P > Q)

next P : P is true in the next time instant.

For example as shown in figure 7, the assertion *@(P > next Q)* means that if P is true at present then Q is true in the next time instant. This introduces the concept of discrete time and the concept of transition that occurs between subsequent time instants. The next time instant can be understood as being equivalent to the delta delay in the VHDL

language, where the delta delay is the smallest possible finite time in which a signal can change its value.

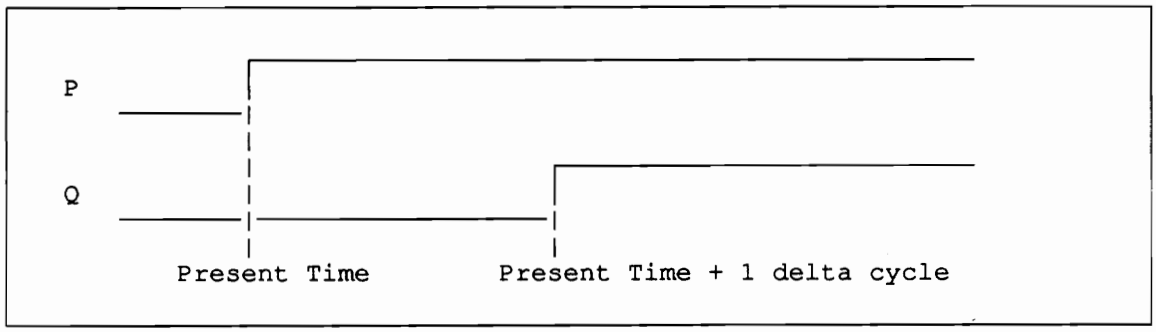


Figure 7. @(P > next Q)

P while Q : P is true only when Q is true.

As shown in figure 8 this assertion implies nothing for any time after Q is false. More formally we may define the while operator recursively using the next operator as follows

P while Q is equivalent to *Q > (P and next (P while Q))*

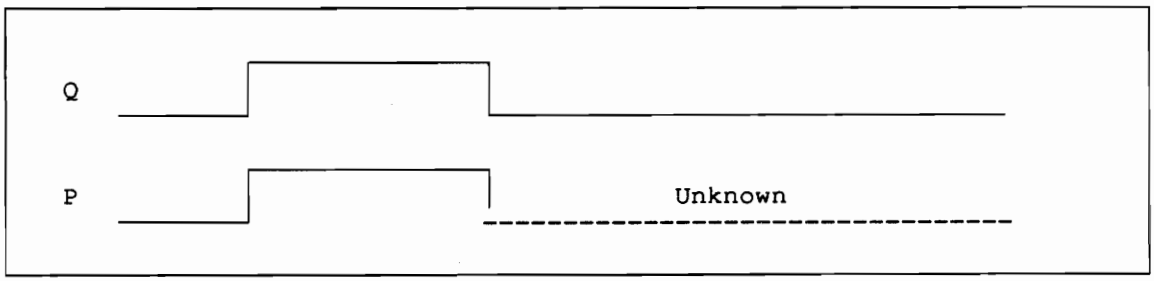


Figure 8. P while Q

P U Q : P is true at all times until the first time when Q is true after which it is false as shown in figure 9.

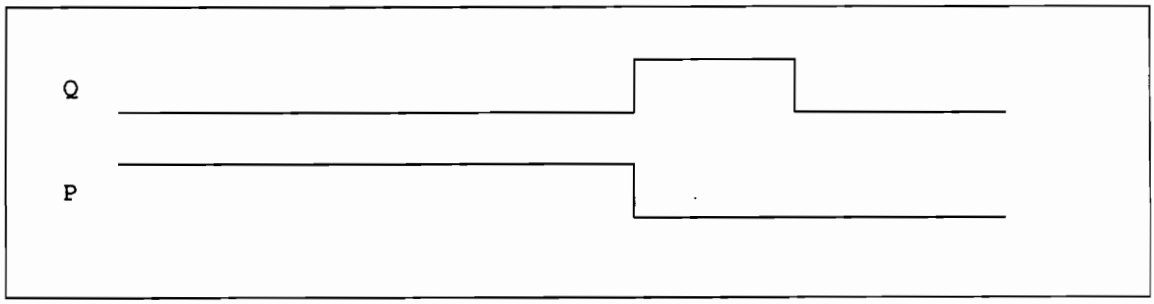


Figure 9. P U Q

2.3 Timing Diagram Specifications in Temporal Logic

Temporal logic can describe timing relations among signals as illustrated by the examples shown below.

@(P > next Q) : This states that whenever the signal P is true then in the next time instant signal Q is true as shown in figure 7.

@((P and next (~ P)) > next Q) : If P is true now and false in the next time instant then Q becomes true, i.e. Q is true on the falling edge of signal P as shown in figure 10.

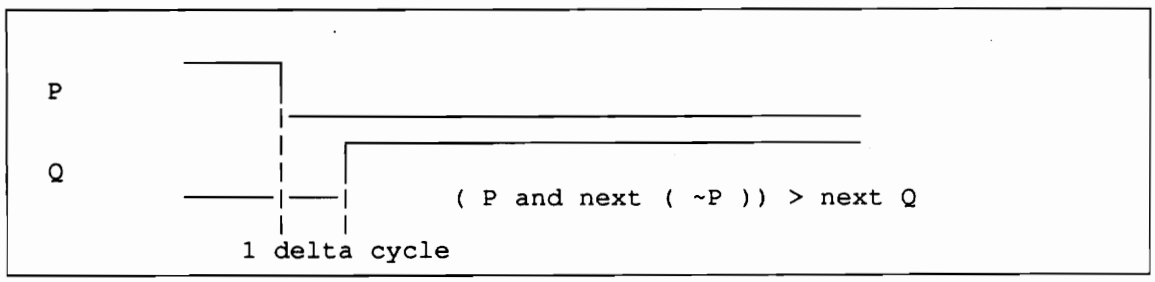


Figure 10. Falling Edge Representation

@((~P and next P) > next Q) : If P is false now and true in the next time instant then the signal Q becomes true, i.e. Q is true on the rising edge of signal P as shown in figure 11.

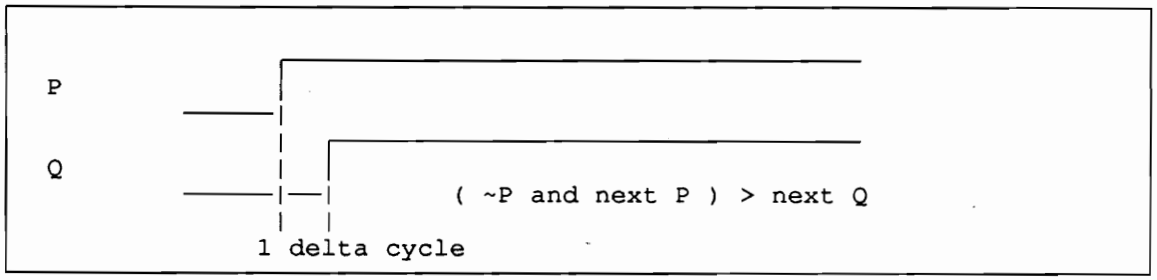


Figure 11. Rising Edge Representation

2.4 Specification of Intervals in Temporal Logic

Temporal logic can also be used to specify intervals [6] as shown below:

(S1 > ((S2 > ((P > next Q) U E2)) U E1))

In the above example, with timing diagram shown in figure 12, the outer interval is specified by signals S1 and E1 and the inner interval is specified by signals S2 and E2. The above specification states that between the time instants when S2 is high until the time instant E2 is high during the time when signal S1 is high until the time E1 is high, if signal P is high then the signal Q becomes high in the next time instant.

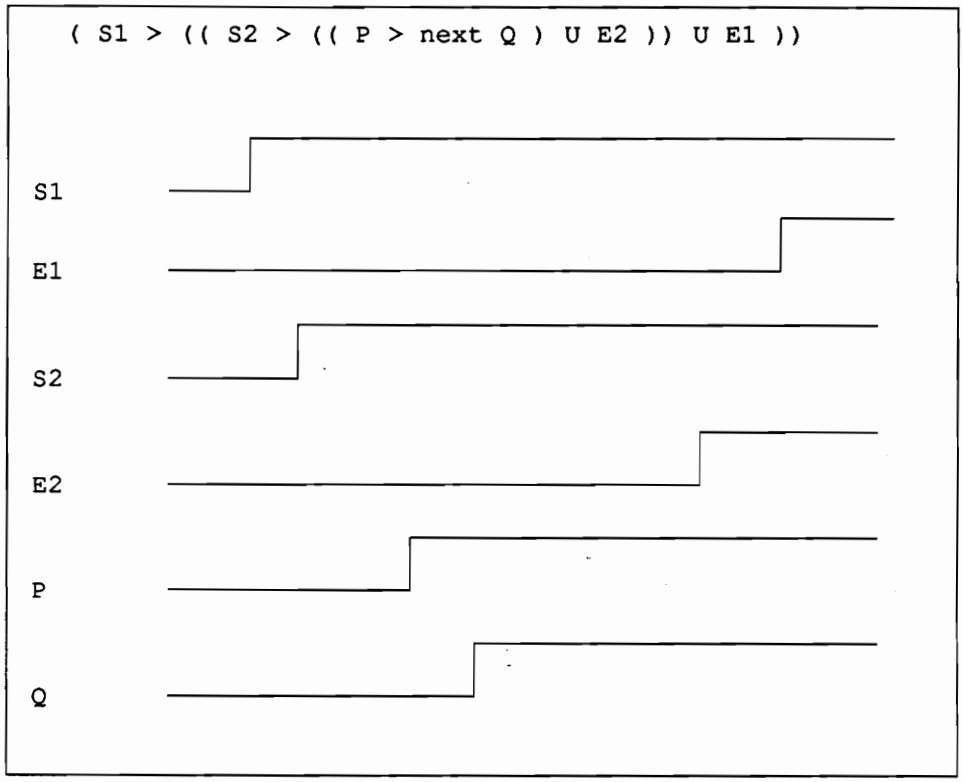


Figure 12. Intervals

2.5 Specification of Variables in Temporal Logic

There is also a provision for specifying variables and numbers (other than binary 0 and 1) in modified temporal logic. The unknown variables should not be confused with the unknown state 'X' in the four states possible for a signal ('0', '1', 'X' and 'Z').

(IN = "x") > (OUT = "x") while (~RESET and SET)

The above specification states that as long as signal RESET is low and signal SET is high, the signal OUT (which can be a BIT VECTOR) follows signal IN.

(IN = "x") > ((OUT = "x" + "#1") while (~RESET and SET))

The above specification states that while RESET is low and SET is high the signal OUT equals the value of signal IN incremented by 1.

2.6 Parallelism in Temporal Logic

@(P > Q while T) <*1*>

@(N > next R) <*2*>

The above statements specify that conditions <*1*> and <*2*> are to be satisfied by the system at the same time. The user also has the option of specifying sequentiality i.e, condition <*2*> has to be satisfied after condition <*1*> is satisfied. The default for the Model Verifier software is parallel behavior. The details of specifying sequential behavior are given in Chapter 5. In the next chapter we discuss the procedures for simplifying and expanding the temporal logic formulae for ease in automation, but first the following section presents an example.

2.7 Example of a JK Flip-Flop

The following example illustrates the temporal logic specifications of a positive edge triggered JK flip-flop. The temporal logic formulae describing the working of the JK flip-flop are given below for the non-inverting output Q.

@(Q iff ~Qbar) ; <1>

@(~Reset and next Reset) > next # ~Q ; <2>

@((~Set and next Set) and ~Reset)> next # Q ; <3>

@(~Set and ~Reset)and(~clk and next clk) and (J and ~Q) > next #Q ; <4>

@(~Set and ~Reset)and(~clk and next clk) and (K and Q) > next #~Q ; <5>

<1> dictates that the non-inverting output Q is always the opposite of the inverting output Qbar.

<2> dictates that on the rising edge of Reset, Q eventually goes low.

<3> dictates that when Reset is low then on the rising edge of Set, Q eventually goes high.

<4> dictates that when Set and Reset are low then on the rising edge of the clock, if the J input of the flip-flop is high and output Q is low then the output Q eventually goes high.

<5> dictates that when Set and Reset are low then on the rising edge of the clock, if the K input of the flip-flop) is high and output Q is high then the output Q eventually goes low.

In a real flip-flop condition <1> is not always met and sometimes the two outputs Q and Qbar are the same. This is because of the uneven delays in producing the two outputs, i.e. Q and Qbar do not change simultaneously. The other conditions, however, are satisfied. So if we are modeling a practical system with uneven delays the condition <1> should not be specified.

In the next chapter the simplification of temporal logic formulae is explained. The chapter details the identities which are used in expanding the temporal logic formulae in the time frame. These are necessary to associate a TIME INSTANT with signals, e.g the "present time" or "eventually" (see section 4.2.2.1 for a discussion on TIME INSTANTS).

Chapter 3. Simplification of The Temporal Logic Formulae

During the verification of the VHDL model (of the system described in temporal logic) we manipulate the temporal logic formulae for ease in automation. This chapter details the procedures followed in manipulating the temporal logic formulae. The changes made depend on the equivalence of certain operators shown below.

P > Q is equivalent to **~P or Q**

P iff Q is equivalent to **(~P and ~Q) or (P and Q)**

The logical not operator (~) is moved inward using the following identities :

~(P and Q) is equivalent to **(~P or ~Q)**

~(P or Q) is equivalent to **(~P and ~Q)**

~ @P is equivalent to **# ~P** (~ @ P dictates that P is not always true which is equivalent to saying that P eventually becomes false as dictateed by # ~ P)

~ #P is equivalent to **@ ~P** (~ #P dictates that P never becomes true which is equivalent to saying that P is always false as dictated by @ ~P).

The above two forms of specification i.e., (~ @ P) and (~ # P), are not supported by the Model Verifier software. But any specification of the form (~ @P) can be written as # ~P (which is supported by the software) and also any specification of the form (~ #P) can be written as (@ ~P) which is also supported.

After we perform the above operations we expand the temporal logic formulae in the time frame using the identities [4] given below. These expansion rules help us to separate the expected results into events in the present and next time instants.

Identity 1

@ P = P and next @P

The above assertions states that if @P (always P) is true then P is true at all times i.e., P is true at present and all times from the next time instant.

Identity 2

#P = P or (~P and next #P)

The above assertion states that if #P (P is eventually true) is true then either P is true now or, P is false now and eventually true from the next time instant. Since the operator # has an upper bound associated with it P must become true during the time provided by the

user.

Identity 3

P U Q = Q or (P and ~Q and next (P U Q))

The above assertion states that if P is true until Q is true (P U Q) then either Q is true now, or P is true now and Q is false, and in the next time instant P U Q is satisfied.

Identity 4

~ @P = # ~P = ~P or (P and next (# ~P))

The above assertion states that if P is not always true then either P is false now (in the present time instant) or P is true now and from the next time instant #~P is true. Since the operator # has an upper bound associated with it P must become false during the time provided by the user.

Identity 5

~ #P = @~ P = ~P and next (@ ~P)

The above assertion states that if P is never eventually true i.e., P is always false then P is false at the present time instant and from the next time instant ~ # P is true.

Identity 6

~(P U Q) = (~P and ~Q) or (~Q and next (~ (P U Q)))

The above assertion states that if the statement P is true until Q is true, is false then it means that either, at the present time instant P and Q are false, or Q is false and from the next time instant ~(P U Q) is true.

As can be seen from above, the temporal operators are eliminated from the present time instant. They do however exist in the next time instant. The formulae and identities are used recursively to expand the specifications for the present and next time instants. During this simplification process signals are assigned a TIME INSTANT (see Section 4.2.2.1). This results in the elimination of temporal operators from the temporal logic formulae, and we are left with the task of verifying the boolean validity. During the verification process as explained in the next chapter, we check the simulation results using the modified temporal formulae thus simplifying our verification process.

Chapter 4. Verification of the VHDL model

This chapter details the various steps in the verification of the VHDL model which has been described by one or many temporal logic formulae. There are essentially three steps in the verification process as shown in figure 13.

1. The creation of test vectors for simulating the VHDL file,
2. The simulation of the VHDL file using the test vectors generated (using thr VHDL simulator), and
3. The verification of the output of the simulation results with the expected results as obtained from the temporal logic description of the system. The steps are explained below.

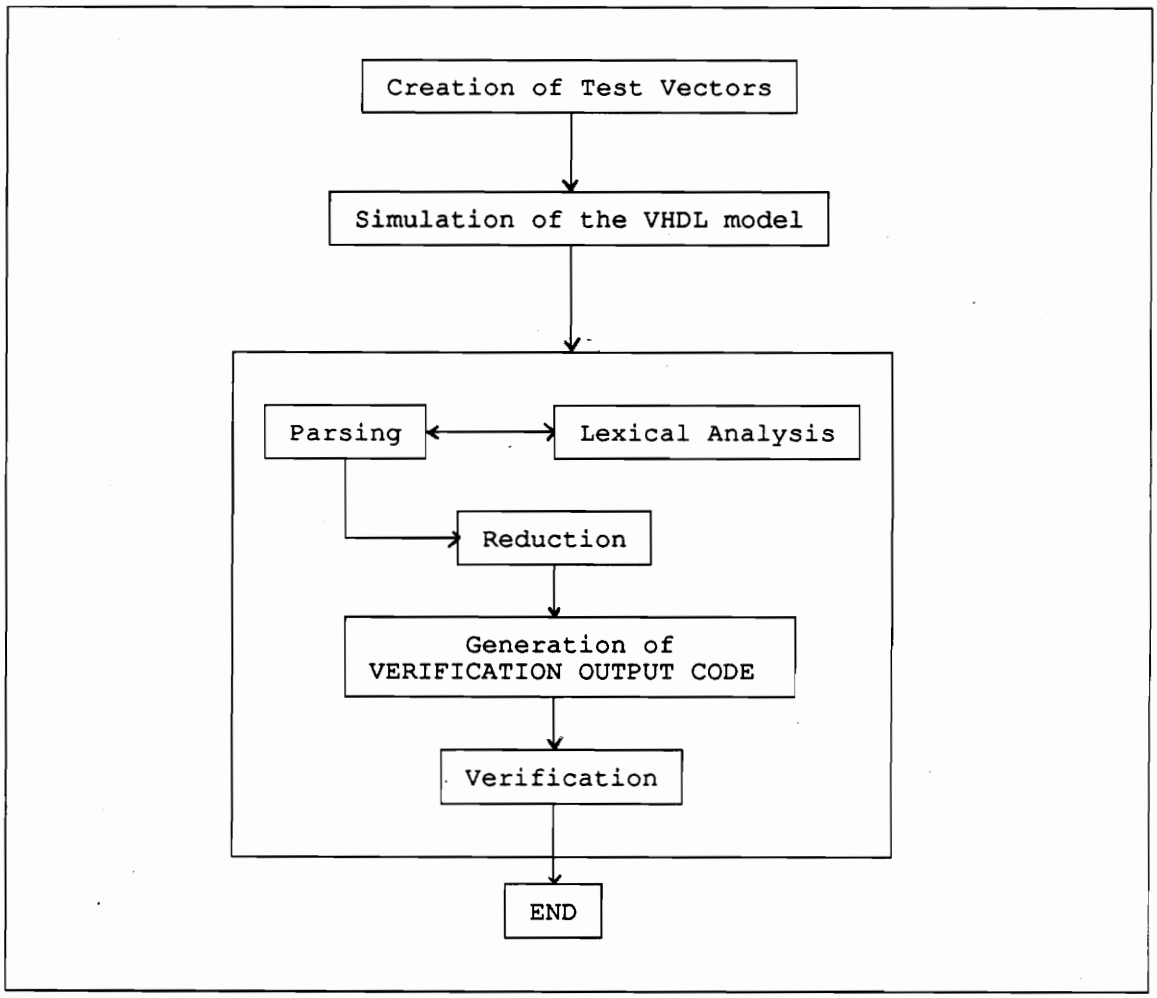


Figure 13. Verification Process

4.1 Creation of Test Vectors

The creation of test vectors is not the main focus of this research, but an option is provided if the user wants to create exhaustive test patterns. The patterns generated are useful if the user is verifying the boolean validity of the model, i.e. on a DC basis. If the user chooses to generate exhaustive test patterns as discussed in chapter 5, the VHDL model is read and exhaustive patterns are generated based on the input ports of the entity to be simulated. The user can specify an approximate settling time for the system, i.e. how many time instants, in nanoseconds, should elapse before the inputs can change. The user can also specify if he needs a certain input to be held at a certain value. Exhaustive binary patterns are generated based on the above data. The user also has the option of requesting the generation of random patterns of test vectors in case the generation of exhaustive patterns will involve lengthy simulation time. The user can also supply his own set of test vectors if he is trying to simulate the system for a particular set of faults.

The next step after the generation of test vectors is the simulation of the VHDL model. This step involves the analysis and simulation of the VHDL model of interest. The user specifies the entity and corresponding architecture to be simulated. The simulation is performed using the test vectors generated in step 1. The results of the simulation are stored in a file that is used in the verification process described in the next section.

4.2 Verification of Simulation Results

The verification of simulation results is the main part of the VHDL Model Verifier software and the central focus of this research. The verification is done using two files, the output of the simulation results and the temporal logic description of the system. The verification process is described in detail in a series of algorithms given below. The process as explained applies to a single temporal logic formulae. It can be repeated on other formulae after the verification of the first one.

The first steps are the lexical analysis and parsing of the input formula. This is done to recognize the input string and generate what we call the VERIFICATION OUTPUT CODE. This code is a series of steps that when executed will result in the verification of our simulation results. For efficient parsing and recognition of the input formulae, the different operators, boolean, arithmetic and temporal, have been assigned levels of precedence. The operators have a predefined precedence which will be in accordance with Table 1. The user can override the precedence level by using parentheses. Before we explain the algorithm used and the parsing techniques used we define the following terms.

Tokens: Tokens are strings that have a particular and unique meaning to the parser. In our verifier the lexical analyzer converts the stream of input characters into a stream of tokens.

Terminals: Terminals are the basic symbols from which strings are formed.

The word "token" is a synonym for "terminal" when talking about grammars for programming languages. In our context, the symbols @, #, next are terminals.

Nonterminals: Nonterminals are syntactic variables that denote sets of strings. The nonterminals define sets of strings that help define the language generated by the grammar.

Identifier: Languages use identifiers as names of variables, arrays and functions. A grammar for a language often treats an identifier as a token. A parser based on such a grammar wants to see the same token, say id, each time an identifier appears in the input. For example, the input

$ Input and Reset iff Output $

would be converted by the lexical analyzer into the stream

$ Id and Id iff Id $

This token stream would then be used for parsing. Note that at this time no mention has been made of a mechanism to distinguish between the different identifiers.

The next section gives an account of the DATA STRUCTURES used. These DATA STRUCTURES are used in lexical analysis, parsing and verification of the VHDL model and temporal logic formulae.

Table 1. Operator Precedence Table

	and	or	~	>	iff	+	-	/	*	@	#	%	U	while	next	(	)	id	=	U	N	$
and	>	>	<	>	>	>	>	>	>	>	>	>	>	>	<	<	>	<	>	<	<	>
or	>	>	<	>	>	>	>	>	>	>	>	>	>	>	<	<	>	<	>	<	<	>
~	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>	<	<	>
>	<	<	<	>	>	<	<	<	<	<	<	<	<	<	<	<	>	<	<	<	<	>
iff	<	<	<	>	>	<	<	<	<	<	<	<	<	<	<	<	>	<	<	<	<	>
+	<	<	<	>	>	>	>	<	<	>	>	>	>	>	>	<	>	<	>	<	<	>
-	<	<	<	>	>	>	>	<	<	>	>	>	>	>	>	<	>	<	>	<	<	>
/	<	<	<	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>	<	<	>
*	<	<	<	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>	<	<	>
@	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	<	<	<	<	>
#	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	<	<	<	<	>
%	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	<	<	<	<	>
U	<	<	<	<	<	<	<	<	<	>	>	>	>	>	<	<	>	<	<	<	<	>
while	<	<	<	<	<	<	<	<	>	>	>	<	>	<	<	<	>	<	<	<	<	>
next	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>	<	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	<	<	<	E
)	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	E	>	E	>	<	<	>
id	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	E	>	E	>	<	<	>
=	<	<	<	>	>	<	<	<	<	>	>	>	>	>	<	<	>	<	>	<	<	>
U	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	E	>	E	>	<	<	>
N	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	E	>	E	>	<	<	>
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	E	<	<	<	<	E

The rows and columns are the different Temporal Operators
id : Identifier, signal or port name
U : variables
N : Numbers

Relation	Meaning
a < b	a "yields precedence to" b
a = b	a "has the same precedence as" b
a > b	a "takes precedence over" b
a E b	a "has no relation to" b (an error)

4.2.1 Data Structures

4.2.1.1 Data Structures for Lexical Analysis and Parsing

The process of lexical analysis is essential for recognizing the input temporal logic formula. As explained earlier the lexical analyzer reads the input temporal logic input and returns TOKENS to the parser which takes actions based on the TOKENS. For the efficient recognition of the input string a finite state machine as shown in figure 14, is constructed. The starting state is 1. The final states are shown in concentric circles. The finite state machine is traversed based on the input string and if an input is valid, a final state is reached. At any final state we return a unique TOKEN to the parser. This token can be any one of the following:

An Identifier

A Variable

A number

An operator (arithmetic, boolean or temporal).

For example if the input string is

Reset > next Set ;

The sequence traversed in the finite state machine is shown in Table 2.

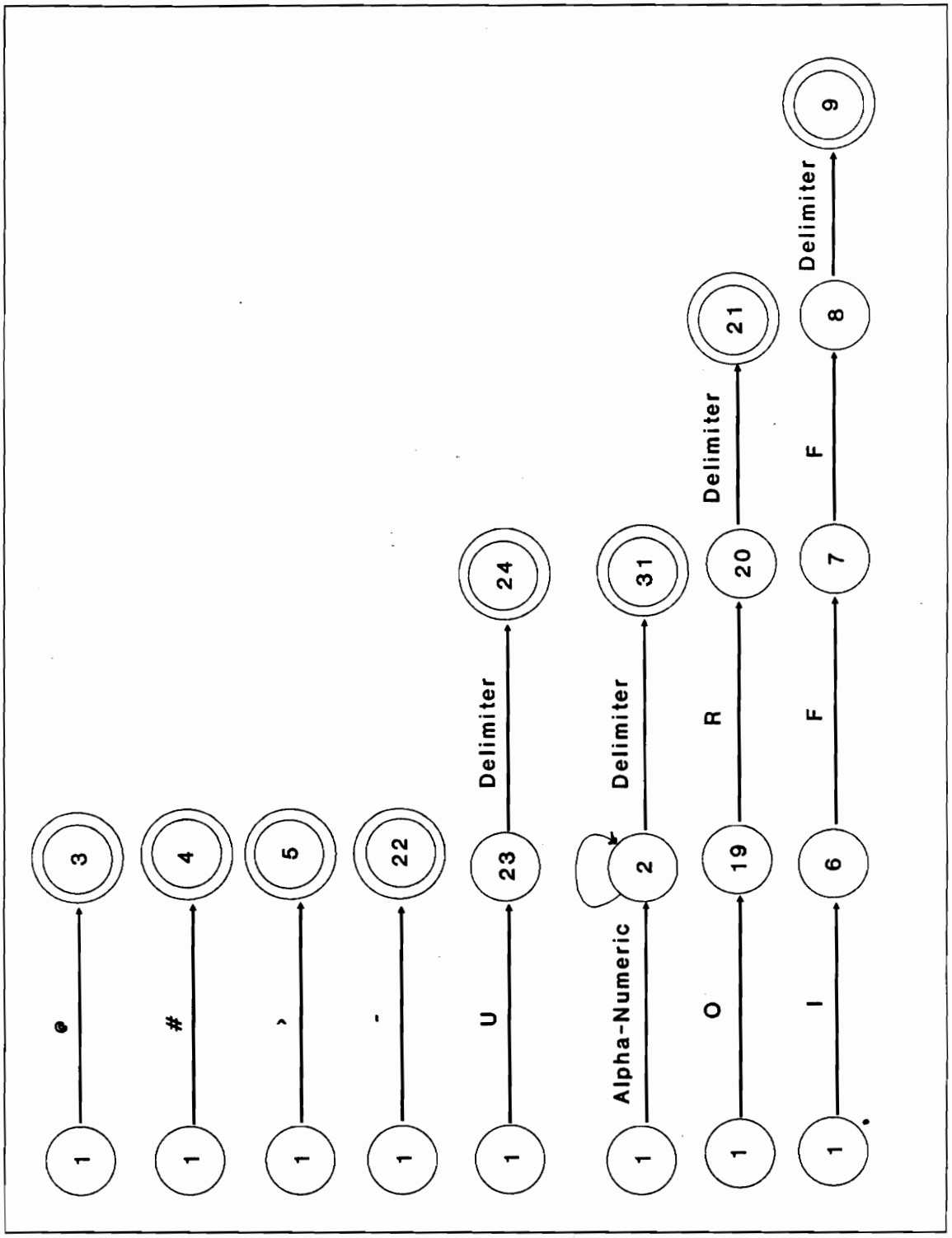


Figure 14. Finite State Machine for Lexical Analyses

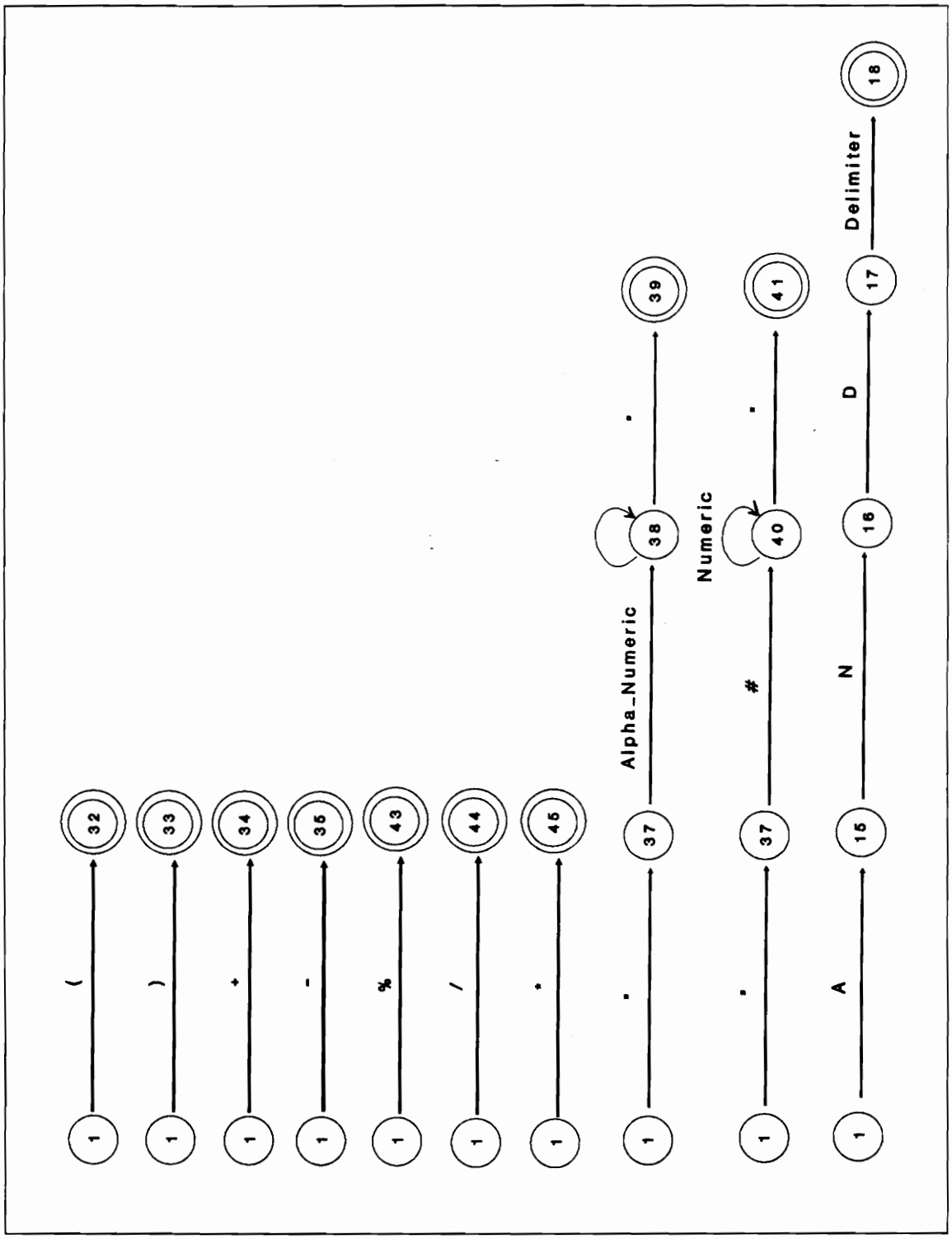


Figure 14 (Contd). Finite State Machine for Lexical Analyses

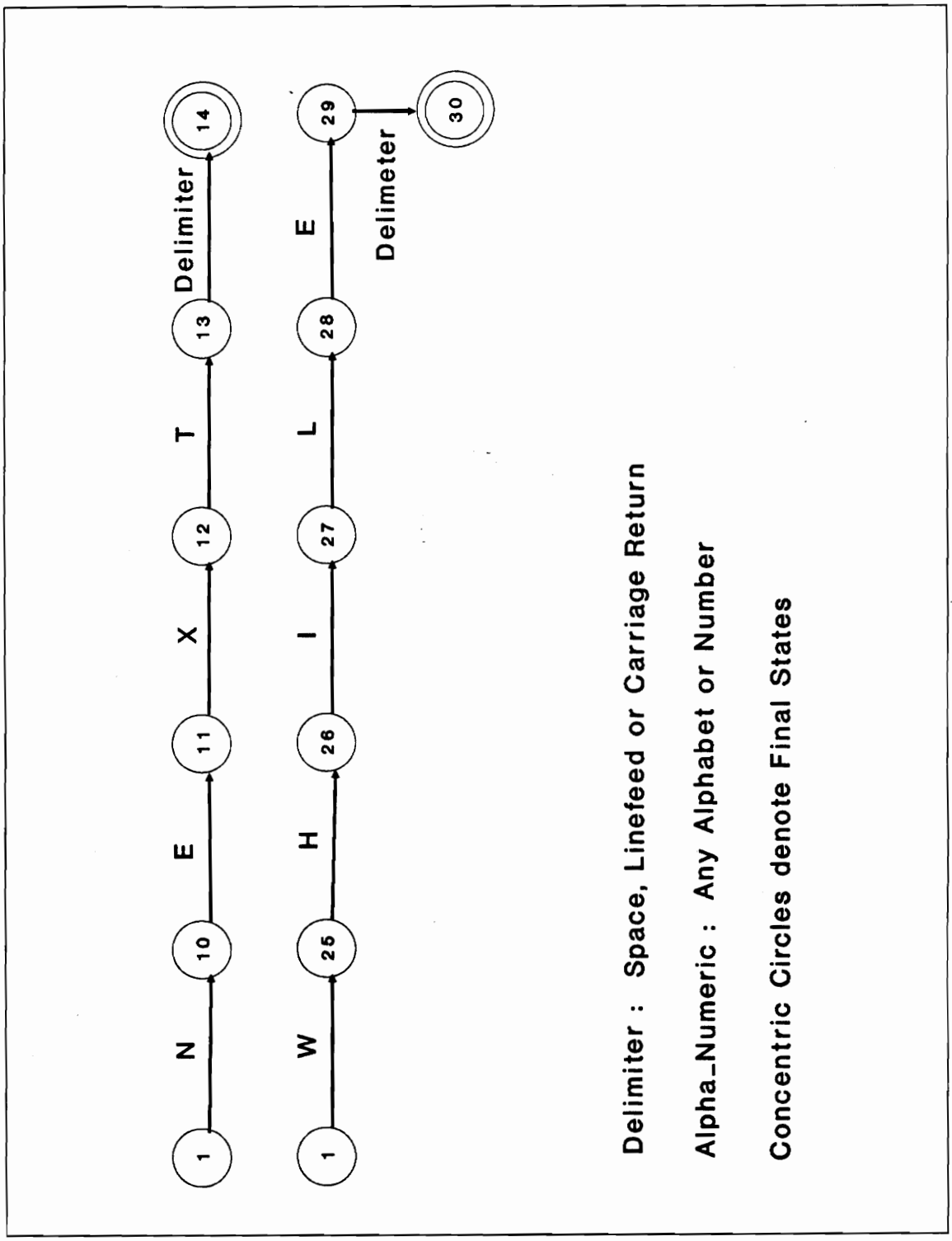


Figure 14 (Contd). Finite State Machine for Lexical Analyses

Table 2. Lexical Analyses

Present State	Input	Next State	TOKEN
1	R	2	
2	e	2	
2	s	2	
2	e	2	
2	t	2	
2	space	31	Identifier **Reset**
1	>	5	Operator **>**
1	n	10	
10	e	11	
11	x	12	
12	t	13	
13	space	14	Operator **next**
1	S	2	
2	e	2	
2	t	2	
2	space	31	Identifier **Set**

For parsing, the main data structure used is the operator precedence array as shown in table 1. The parsing algorithm shown later is carried out using this array. The entire parsing algorithm is implemented in a STACK. The stack holds the input string while being parsed. Based on the input string the stack is incremented and decremented and actions are taken based on the operator precedence table. The use of the stack will become clear in the example given in Section 4.3.

4.2.1.2 Data Structures for Verification

The result of parsing is the generation of the VERIFICATION OUTPUT CODE. The VERIFICATION OUTPUT CODE consists of a linked list of structures of the form

```
structure EXECUTE
        integer OPERATION ;
        character string IDENTIFIER ;
        integer TIME_INSTANT ;
        integer EVENTUALLY ;
end structure ;
```

There is a unique linked list of structures, such as above, for each temporal formula. Elements of the above data structure are defined below.

OPERATION is one of the temporal operations to be performed. For example one of the operations can be GET IDENTIFIER, i.e. fetch the value of the required identifier from the simulation results.

IDENTIFIER is the character string that represents a signal name or an unknown variable. For example, in the above example Set, Reset, Q are character strings

representing signal names.

TIME_INSTANT is the time for which the signal value is needed i.e., if the verification is performed for T = 200 ns we may either need the value of a signal for T=200 ns (present time) or for the next TIME_INSTANT (T = 200ns + delta). TIME_INSTANT can assume different values as discussed in Section 4.2.2.1.

EVENTUALLY is another variation of the TIME_INSTANT in which the user specifies the duration of tolerance. For example, EVENTUALLY can be equal to EVENTUALLY '0', or EVENTUALLY '1'. (See Section 4.2.2.1.)

Another data structure used is the SYMBOL TABLE. A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data allows us to find the record for each identifier quickly and store or retrieve data from it. When an identifier in the source temporal logic formula is detected by the lexical analyzer, an identifier is entered in the SYMBOL TABLE. However the attributes of the identifier cannot be determined during the lexical analysis. In our software we maintain separate symbol tables for signals and variables. This is done because the attributes of signals are far more complicated than those of variables. The symbol table is used in situations such as in

(Set = "y") > (Reset = "y" + "#1") $

In this case an entry is made for variable **y**. When the variable **y** is first encountered the value of **y** is unknown so it is given the current value of signal Set. When the variable is next encountered the value which equals the value of Set, is fetched from the symbol table. It is then incremented by 1, since "#1" is equal to 1, and compared with the value of Reset.

For each operation to be performed there is a corresponding entry in the VERIFICATION OUTPUT CODE. The STACK operation is performed according to the algorithm given below and is illustrated in Table 3. Table 4 gives the VERIFICATION OUTPUT CODE which is later used in verification. The explanation for both is given below.

4.2.2 Modified Operator Precedence Parsing

"In operator-precedence parsing [14], we define three disjoint precedence relations <, >, and = between certain pair of terminals (The precedence relations operators should not be confused with the temporal logic operators). These precedence relations have the following meanings.

Relation	Meaning
a < b	a "yields precedence to" b
a = b	a "has the same precedence as" b
a > b	a "takes precedence over" b
a E b	a "has no relation to" b (an error)

These relations may appear similar to arithmetic relations "less than" , "greater than", and "equal to", but have quite different properties. For example we could have a < b and a > b for the same language, or we might have none of a < b, a = b, and a > b holding for some terminals a and b. The method we use in determining precedence between operators is intuitive and is based on the traditional notions of associativity and precedence of operators." For example if * (multiplication) is to have a higher precedence than + (addition), we make + < * and * > +. The purpose of using the precedence relation will become clear when we explain the parsing algorithm used in the next section. Given an input string we consult the operator-precedence relation table and insert the precedence relations between them. For example the string

$ Input and Reset iff Output $ would be modified as

$ Id and Id iff Id $

$ < Id > and < Id > iff < Id > $

In the above example the precedence relation < is inserted between the leftmost $ and Id since < is the entry in the row $ and column Id. Now having done that we execute the following steps:

1. "Scan the string from left end until the first > is encountered. In the example the first > occurs between the first **Id** and **and.**

2. From that point onwards scan backward (to the left) over any ='s until a < is encountered. In the example we scan backwards to **$.**

3. The handle contains everything to the left of the first > and to the right of the first < encountered in step 2, including any intervening or surrounding non-terminals (the inclusion of surrounding nonterminals is essential so that two adjacent nonterminals do not appear). In the above example the handle is the first Id". Once we have isolated the handle we are in a position to take some action we call now as reduction. At this juncture we are in a position to recognize the handle. For example we recognize the present handle as Id (an identifier or a signal name) and proceed to fetch its value. At this point we also are in a position to enter an executable code in the VERIFICATION OUTPUT CODE file. The code in this case would be fetch Signal (which would be the value of the signal during the simulation). The other elements of the VERIFICATION OUTPUT CODE file will become clearer in the example shown below. If no precedence relation holds between

a pair of entries (as indicated by a blank in the operator precedence relation table) an error in the input temporal logic string is detected.

The operator precedence algorithm [14] as explained above is effective if the operators have no time factor associated with them. Since in our temporal specification, signals have a time factor associated with them the parsing algorithm has to modified. Moreover the simplification procedures mentioned in Chapter 3 make the modifications necessary. The Modified Operator Precedence algorithm is explained in the next section with a detailed example following it.

4.2.2.1 Modified Operator Precedence Parsing Algorithm

The process of parsing involves determining the actions to be taken based on the TOKENS returned by the lexical analyzer. The parsing technique used is a modification of the operator parsing algorithm [14]. The modification is done to take into account the value of signals at different time instants.

Before we explain the modified parsing technique we detail the timing aspect related to signals. By using the modified operator precedence algorithm we can keep an account of the TIME_INSTANT during which the signal value is needed. The TIME_INSTANT can be any of the following:

Present time : The value of the signal at the present simulation time is needed.

(K)delta cycles : The value of the signal is needed **K** delta cycles after the present time.

Eventually (#) : There can be three different situations if an EVENTUALLY operator is associated with a signal. They are as follows:

1. As in the case of *P > #Q* : We have to see if the signal becomes true after a certain time instant.

2. As in the case of *P > #~Q* : We have to see if the signal becomes false after a certain time instant.

3. As in the case of *P > # (Q = "x")* : In this case we have to check if Q attains the value "x" (x is any variable). In this case the operation has to be delayed because the value of "x" is unknown during the verification process. The operation is delayed until the value of the expression on the right of the "=" (EQUAL operator) is known. The expression on the right can be any expression involving variables e.g., Q = "x" + "y" * "#1010".

Set-Up and Hold Time(%) : In this case we have to check if the signal has remained stationary at a particular value for a certain period of time. Again there can be two cases in this situation :

% P : Check to see if P has been stable at '1' for a certain period of time.

% ~P : Check to see if P has been stable at '0' for a certain period of time.

% (Q = "x") : In this case we have to check if Q retains the value "x" (x is any variable). In this case the operation has to be delayed because the value of "x" is unknown during the verification process. The operation is delayed until the value of the expression on the right of the "=" (EQUAL operator) is known.

The modified operator precedence algorithm is given below. The steps 10-22, and 26-34 are a modification of the regular operator precedence algorithm.

Start Algorithm

Input : An input temporal logic string w and an Operator Precedence Table

Output : If w is a valid string an VERIFICATION OUTPUT CODE file.

Method : Initially, the stack contains $ and the input buffer the string w$. To parse we execute the following steps:

```
1 set ip to point to the first symbol of w$;
2 repeat forever
3       if $ is on top of the stack and ip points to $ then
4             return  /* END OF PARSING */
5       else begin
6             let a be the topmost terminal symbol on the stack
7                   and let b be the symbol pointed to by ip;
```

```
8       if a < b or a = b then begin
9           push b onto the stack
10          if b = NEXT
11              Increment TIME_INSTANT ;
12          If b = EVENTUALLY ;
13              TIME_INSTANT = EVENTUALLY ;
14          if b = SET-UP and HOLD time
15              TIME_INSTANT = SET-UP
16          if b = NOT operator
17              COUNTER_NOT = not(COUNTER_NOT)
18          advance ip to the next input signal;
19          if ip is equal to EQUAL operator
20              if TIME_INSTANT is EVENTUALLY
21                  delay the OPERATION
22      end;
23
24      else if a > b then
25          repeat
26          pop the stack
27          if b = NEXT
28              decrement TIME_INSTANT ;
```

```
29                  If b = EVENTUALLY ;
30                      TIME_INSTANT = not(EVENTUALLY) ;
31                  if b = SET-UP and HOLD time
32                      TIME_INSTANT = not(SET-UP)
33                  if b = NOT operator
34                      COUNTER_NOT = not(COUNTER_NOT)
35
36                  until the top of the stack is related by <
37                      to the terminal most recently popped
38                          Reduce() ; /* Explained Below */
39          else error()
40      End Algorithm
```

The information from above, as shown later, is used while making an entry in the VERIFICATION OUTPUT CODE. The Reduce algorithm discussed in the next section when executed results in the generation of the VERIFICATION OUTPUT CODE. For purposes of understanding only a broad outline of the algorithm is given in the next section.

4.2.3 Reduce Algorithm

At this stage we are in a position to understand the TOKEN returned to our parser by the lexical analyzer. After we recognize the TOKEN we make an entry in the VERIFICATION OUTPUT CODE which details the actions to be performed once the TOKENS have been recognized.

Start Algorithm

If TOKEN = IDENTIFIER /* a signal or port name */

check if IDENTIFIER is a primary port signal ;

OPERATION = Fetch Signal Value ;

TIME_INSTANT = As evaluated earlier ;

IDENTIFIER = Name of Signal as returned by Lexical Analyzer

If TOKEN = Variable

OPERATION = Fetch Value of Variable from SYMBOL_TABLE ;

IDENTIFIER = Name of Variable as returned by Lexical Analyzer ;

If TOKEN = Iff

OPERATION = Iff ;

Repeat the last step if TOKEN is any other TEMPORAL OPERATOR ;

VERIFY() ; /* The Verification Algorithm

Next Section */

end Reduce() ;

End Algorithm

The above algorithm generates the VERIFICATION OUTPUT CODE. In section 4.3 an example which details the process is given.

4.2.4 Verify Algorithm

VERIFY is the algorithm that is executed once the VERIFICATION OUTPUT CODE is generated. The VERIFY routine reads the simulation results and creates a map between the signal names in the temporal logic file and the signal values in the simulation output results file. We would like to mention at this stage that the verification of simulation results is carried for simulation time instants explicitly or implicitly stated in the simulation results file i.e., for all input signal changes. For example in Table 3, verification is performed at T = 100ns, T = 100ns + delta, T = 200ns (explicitly stated in the table) and also at T = (200ns - delta) (implicitly stated) during which the signal J changes from 0 to 1 (i.e.,a rising edge), K changes from 1 to 0 (i.e., a falling edge) etc.

Table 3. Sample Simulation Results 1

Time(ns)	J	K	Set	Reset	Q	Qbar
100(0)	1	0	1	1	0	1
100(1)	0	1	0	1	1	0
200	1	0	0	0	0	1

J, K, Set, Reset, Q and Qbar are Signal names
100(0) specifies T = 100ns, Present Time
100(1) specifies T = 100ns, Delta Time later

Start Algorithm

```
1  if for each signal name in the temporal logic file corresponding signal exists in the
2  Simulation Results File.
3          Map Into the VERIFICATION OUTPUT CODE, the position of
4          signal in the Simulation Results File ;
5  else
6          Report "Signal Not Found"

7  if OPERATION = Fetch Signal Value
8          Check TIME_INSTANT ;
9          if TIME_INSTANT = PRESENT_TIME
10                 Fetch Signal Value from Simulation Results ;
11         else if TIME_INSTANT = (K) delta cycles later
12                 Fetch Signal K delta cycles later ;
13         else if TIME_INSTANT = EVENTUALLY
14                 Check if Signal has attained desired value
15                 within desired Tolerance time.
16         else if TIME_INSTANT = SET-UP
17                 Check if Signal has retained desired value
18                 for the desired time.
19         else if TIME_INSTANT has to be delayed
```

20 delay operation till OPERATION is equal to "="

21 **Perform** the above task for each delta cycle during which a change has occurred.

22 **if** OPERATION = Fetch Variable Value

23 check value of Variable from SYMBOL_TABLE ;

24 **if** value exists

25 fetch value

26 **else**

27 assign value of terminal to the right of = sign

28 Update SYMBOL_TABLE ;

29 **if** OPERATION = + (Binary addition)

30 obtain the top two elements from the stack

31 add them ;

32 **Repeat** steps 29-31 for any arithmetic or boolean operator.

The steps (22-28) deal with recognition with unknown variables. They exist in two forms: One as in (**Set = "x" ...**) where "x" has no previous value and has to be assigned the value of the signal to the left of the = sign, in this case Set.

The other form is (**Set = "x"**) > (**Reset = "x" + "#1"**) where the value of "x" is the value to be fetched from the SYMBOL_TABLE.

4.3 Example

Let us assume that the input temporal logic formulae is

@(((~Clock and next Clock) and (J and ~K)) iff (#~ Q)) $

The Stack operation is carried out as in table 4. The VERIFICATION OUTPUT CODE which is produced as a result of the **Reduce** algorithm for the above formula. The VERIFICATION OUTPUT CODE is given Table 5.

The above formula covers the temporal primitives such as next, eventually and @(always). The parsing is performed according to the algorithm in section 4.2.2.1. During the parsing a reduction is performed once we have recognized a TOKEN (returned from the lexical analyzer) and we are ready to take some action. At this point an entry is made in the VERIFICATION OUTPUT CODE. Table 5 gives the VERIFICATION OUTPUT CODE which gives the entry corresponding to each reduction step in Table 4. After the VERIFICATION OUTPUT CODE is completed the **Verify** algorithm is executed. This algorithm takes two inputs : the VERIFICATION OUTPUT CODE and the simulation results file. A sample simulation output file is given in Table 6. Table 7 shows the execution of the Verify algorithm.

Table 4. Stack Operations

STACK	INPUT	ACTION
$	@	Shift
@	(	Shift
(	(	Shift
(	(	Shift
(	~	Shift
~	Clock	Shift
Clock	and	Reduce(1)
~	and	Reduce(2)
and	next	Shift
next	Clock	Shift
Clock	)	Reduce(3)
next	)	Reduce(4)
and	)	Reduce(5)
(	)	Reduce(6)
(	and	Shift
and	(	Shift
(	J	Shift
J	and	Reduce(7)
and	~	Shift
~	K	Shift
K	)	Reduce(8)
~	)	Reduce(9)
and	)	Reduce(10)
(	)	Reduce(11)
and	)	Reduce(12)
(	)	Reduce(13)
(	iff	Shift
iff	#	Shift
#	~	Shift
~	Q	Shift
Q	)	Reduce(14)
~	)	Reduce(15)
#	)	Reduce(16)
(	)	Reduce(17)
iff	)	Reduce(18)
(	)	Reduce(19)
@	$	Reduce(20)
$	$	**End Parsing**

Stack: Denotes the content at the top of the stack
Input: Denotes the TOKEN returned by the Lexical Analyzer

Action:
Shift : Moving the input to the top of the stack
Reduce(k) : An action taken based on algorithm Reduce()
Also includes popping the top of the stack.
(k) specifies the k'th reduction step.

Table 5. VERIFICATION OUTPUT CODE

Reduction	Op_No	VERIFICATION OUTPUT CODE
Reduce(1)	1	OPERATION : fetch signal value IDENTIFIER : Clock ; TIME_INSTANT : present ; EVENTUALLY = 0 ;
Reduce(2)	2	OPERATION : not ; IDENTIFIER : Clock ;
Reduce(3)	3	OPERATION : fetch signal value ; IDENTIFIER : Clock ; TIME_INSTANT : 1 delta cycle later
Reduce(4)	4	OPERATION : next
Reduce(5)	5	OPERATION : and
Reduce(6)	6	OPERATION : parentheses ;
Reduce(7)	7	OPERATION : fetch signal value ; IDENTIFIER : J ; TIME_INSTANT : present ; EVENTUALLY = 0 ;
Reduce(8)	8	OPERATION : fetch signal value ; IDENTIFIER : K ; TIME_INSTANT : present ; EVENTUALLY = 0 ;
Reduce(9)	9	OPERATION : not ;
Reduce(10)	10	OPERATION : and ;
Reduce(11)	11	OPERATION : parentheses
Reduce(12)	12	OPERATION : and
Reduce(13)	13	OPERATION : parentheses
Reduce(14)	14	OPERATION : fetch signal value IDENTIFIER : Q ; TIME INSTANT : EVENTUALLY(0); EVENTUALLY = Yes ;
Reduce(15)	15	OPERATION = not ;

Table 5. Continued

Reduction	Op_No	VERIFICATION OUTPUT CODE
Reduce(16)	16	OPERATION = EVENTUALLY ;
Reduce(17)	17	OPERATION = parentheses ;
Reduce(18)	18	OPERATION = iff
Reduce(19)	19	OPERATION = parentheses ;
Reduce(20)	20	OPERATION = @ (always) ;

Reduce(k) : An action taken based on algorithm Reduce() Also includes popping the top of the stack. (k) specifies the k'th reduction step.

Op_No : VERIFICATION OUTPUT CODE operation number

VERIFICATION OUTPUT CODE : Output of Reduce algorithm. Used in Verification of VHDL model.

Table 6. Sample Simulation Results 2

Time(ns)	J	K	Clock	Reset	Q	QBar
100(0)	1	0	0	1	1	1
100(1)	1	0	1	1	1	0
125	1	0	0	0	0	1
150	1	0	1	0	1	1
175	1	0	1	0	0	1

J, K, Set, Reset, Q and QBar are Signal names
100(0) specifies T = 100ns, Present Time
100(1) specifies T = 100ns, Delta Time later

Table 7. Verify Algorithm Implementation

Op_No	TIME	STACK VALUE	Comments
1	100(0)	0	Fetch Value of Clock
2		1	not of value fetched
3		1	Fetch Value of Clock 1 delta cycle later i.e., T = 100(1)
4		1	No Operation
5		1	and Result of 2 & 4
6		1	No Operation
7		1	Fetch Value of J
8		0	Fetch Value of K
9		1	not (value of K)
10		1	and result of 7 & 9
11		1	No Operation
12		1	and result of 5 & 10
13		1	No Operation
14		0	See if Q becomes '0' Eventually i.e., Present Time + 25ns Q = '0' at T = 150ns
15		1	not of 14
16		1	No Operation
17		1	No Operation
18		1	See if 12 implies 15 if YES STACK = 1
19		1	No Operation
20		1	Start for next Simulation Time
1	100(1)	1	Fetch Value of Clock
2		0	not of value fetched
3		1	Fetch Value of Clock 1 delta cycle later i.e., T = 100(2)
4		1	No Operation
5		0	and Result of 2 & 4
6		1	No Operation
7		1	Fetch Value of J
8		0	Fetch Value of K
9		1	not (value of K)
10		1	and result of 7 & 9
11		1	No Operation
12		0	and result of 5 & 10
13		1	No Operation

Table 7. Continued

Op_No	TIME	STACK VALUE	Comments
14		0	See if Q becomes '0' Eventually i.e., Present Time + 25ns Q = '0' at T = 150ns
15		1	not of 14
16		1	No Operation
17		1	No Operation
18		0	See if 12 implies 15 if YES STACK = 1
19		0	No Operation
20		0	Start for next Simulation Time
1	150 - 1 delta cycle	0	Fetch Value of Clock
2		1	not of value fetched
3		1	Fetch Value of Clock 1 delta cycle later i.e., T = 150ns
4		1	No Operation
5		1	and Result of 2 & 4
6		1	No Operation
7		1	Fetch Value of J
8		0	Fetch Value of K
9		1	not (value of K)
10		1	and result of 7 & 9
11		1	No Operation
12		1	and result of 5 & 10
13		1	No Operation
14		0	See if Q becomes '0' Eventually i.e., Present Time + 25ns Q = '0' at T = 175ns
15		1	not of 14
16		1	No Operation
17		1	No Operation
18		1	See if 12 implies 15 if YES STACK = 1
19		1	No Operation
20		1	Start for next Simulation Time

The Verification shown above is only for 3 simulation times. The first two are explicit for which entries are present in Table 5. The last verification is for an implicit entry. The results are :
True at T = 100(0)ns and 150ns - 1 delta cycle
False at T = 100(1) ns.

Chapter 5. VHDL Model Verifier User's Manual

This chapter instructs the user on how to use the VHDL Model Verifier. The software runs on the output of the MCC simulator on the APOLLO DN3500 series.

First we would like to list the files a user will need to begin a session of the VHDL model verifier. The files are:

1. The VHDL model verifier file **verifier**.
2. The VHDL file describing a hardware system.
3. The temporal logic description of the system modeled by the VHDL file.
4. An input file containing TEST VECTORS (optional).
5. A pattern template file **pattern.fil**. This file is needed if exhaustive pattern generation is required.
6. The MCC VHDL software.

The next section describes the creation of the temporal logic files and how to use the VHDL Model Verifier.

5.1 Creation of the Temporal Logic file

For the process of verification we need the description of the system in modified linear time temporal logic (as described in Chapter 2). The description of the system in temporal logic is then compared with the results from the simulation of the VHDL model and discrepancies are pointed out. There are a few rules to be followed in creating the temporal logic files which are enumerated below.

1. You can include one or many temporal logic formulae in any given file. However each formula must be separated (or terminated if it is the last one) from the other, using the $ sign.

For example:

Input > next Output Reset > # Test **INVALID**

Input > next Output $ Reset > # Test $ **VALID**

2. Each terminal must be separated from the other using either spaces, linefeed, or carriage return. For example:

Input>nextOutput$ **INVALID**

Input > next Output $ **VALID**

3. Use parentheses to avoid ambiguities between operators. If not the automatic precedence between operators as defined in Table 1 takes effect. The precedence defined in Table 1 can be overridden by using parentheses.

4. Any operator defined in chapter 2 can be used to create a formula.

5. There is no limit on the degree of nesting in the temporal logic formulae i.e., any number of parentheses can be used.

6. Only signals have a time factor associated with them. Variables do not assume any time factor. For e.g., in the specification **# (P = "x")** the operation will be to check if P attains the current value of **"x"** within the desired period of time. Also you cannot give specifications such as **# (Q = R and S)**. If the intent is that Q eventually attains the logical and of the current value of R and S, the specification has to be restated as **@((R = "x") and (S = "y") > # (Q = "x" and "y"))**.

7. The two forms of specification i.e., **(~ @ P)** , **(~ # P)** are not supported by the software. Any specification of the form **(~ @P)** can be written as **# ~P** (which is supported by the software). Also any specification of the form **(~ #P)** can be written as **(@ ~P)** which is also supported.

8. Variables are supported by the software. You have to cast the variable in "" to distinguish it from signals. You can use any arithmetic or boolean operator to cast variables in expressions. Variables can appear within temporal operators but no time factor will be associated with it.

e.g., **@(Q = ("var_1" and "var_2") + "var_3") $**

9. Numbers are also supported by the software. You have to cast the numbers in "#" e.g., **Q = "#1011"**. Numbers can be used in any arithmetic or boolean expression.

Numbers have to be written with the LSB first and MSB last. For eg., if Q were a 4 bit BIT_VECTOR (integer value 8) you would write **Q = "#0001"** and not as **Q = "#1000"**. This is done to maintain compatibility with the MCC simulator which represents BIT_VECTOR with the LSB first and MSB last.

Once you have the files described above you are ready to start a session using the VHDL model verifier which is explained in the next section.

5.2 Using the Model Verifier

This section explains how to use the VHDL Model Verifier to verify VHDL models. In the text below the letters in bold are those which appear on the screen as the verification proceeds, and the inputs the user has to provide is shown in italics. Also a note of explanation is given for each section. To start the session type:

> *verifier.exe*

verifier.exe is the name of the VHDL model validation software. The following questions would then be asked by the verifier.

> **Enter Temporal Logic filename** : *temporal.des*

temporal.des is the name of the file which contains the temporal logic descriptions of the system which is being modeled. The temporal.des file contains one or more temporal

logic formulae separated by the $ sign.

> Enter Output filename : *output.prt*

The output file (output.prt) will contain the result of the simulation carried on the VHDL model. This file along with the temporal logic file is used to verify the model.

> Enter input VHDL filename : *input.vhdl*

The file input.vhdl contains the VHDL model of the system.

> Enter Entity for Simulation : *jkflip*

jkflip is the name of the entity of interest to be verified. You can specify any entity in the file input.vhdl.

> Enter Architecture for entity : *behavior*

The architecture behavior is the architecture corresponding to the entity jkflip. You can specify any architecture corresponding to the entity JKFLIP.

> Do you wish verbose information : *0*

Enter 1 if you wish verbose information else enter 0. If you enter 1 you get a detailed information about the value of the signals during the different time instants as the verification proceeds. It will give you the value of the stack so you can estimate at what time instants the model fails.

> Do you wish to Reanalyze and Simulate : *0*

If you wish to reanalyze and simulate enter 1 else enter 0. If you have already analyzed and simulated the model with your test vectors or during a previous run of the verifier you would want to bypass this step.

> Enter Tolerance Time for Eventualities : *50*

This would be the tolerance time (an upper bound for eventualities) before which an event scheduled "eventually" should happen. For e.g., in **@((P and Q) > # R)** if P and Q are true for the present time instant R should become true within 50 ns from the next time instant. This query is issued for each temporal formula.

> Enter Set-Up and Hold Time : *20*

This would be the tolerance time (an upper bound for set-up and hold time) for which a signal has to maintain a desired value for an event scheduled to happen. For e.g., in **@(%P > next R)** if P is true for 20 ns, R will become true in the next time instant.

> Do You wish Generation of Exhaustive Test Vectors : *1*

The verifier software allows you to generate exhaustive test vectors for any primary port of type IN or INOUT. You can also supply a stationary value (0 or 1) for any primary port. In case you wish any of the above enter 1, else enter 0. If you entered 0 and you also desired reanalyses of the model, the model is reanalysed and simulated with 100 random test vectors. If you enter 1 to the question above you will be asked the following questions for each primary port (of type IN or INOUT).

> Enter Settling Time if Exhaustive Test Vectors Used : *200*

Settling Time constitutes the smallest interval before which the inputs should not change. This can be the time the circuit needs to settle i.e., for the result of any input to propagate to the output. The exhaustive patterns are generated such that no input changes before the settling time.

> Do you want exhaustive patterns for port portname (TYPE IN/INOUT) :

Enter 0 (No)

Enter 1 (Yes)

Enter 2 (for stationary binary value) : *1*

In a VHDL model you may have declared a primary output port as INOUT to enable readability inside the model, but would not like to include it for exhaustive input patterns. If you wish to exclude any port from exhaustive patterns enter 0. If you wish that the port remains at a stationary binary value enter 2. If you wish that a port be a part of an exhaustive set of test vectors enter 1. If your response to any port is 2 the following question is asked:

> Enter Stationary Value for port portname : *1*

You would enter either 0 or 1.

The default for verification is parallel behavior i.e., all formulae have to be satisfied at the same time. If you want sequential verification you just have to ignore the results for the time instants for which you do not want a particular formula to be satisfied.

The verification then proceeds and gives the results for each time instant during the

simulation. An example of the verification of a **Traffic Light Controller** is given in the next section.

5.3 Verification of a Traffic Light Controller

The following is a model of a traffic light control [15]. The model is as follows :
A HIGHWAY meets a FARMROAD at crossroads. We start at a time when the HIGHWAY light is green. The machine remains in this state (figure 15) as long as no cars are detected or the long timeout has not occurred i.e., as long as (CAR and TL)= 0. After a long time out occurs and any cars are detected the machine restarts the timer and switches to HY where the HIGHWAY lights are yellow. It remains in this state only till the short timeout occurs, then restarts the timer and goes to state FG where the FARMROADS lights are green. It remains in this state until no cars are detected or the long timeout occurs. It then restarts the timer and goes to state FY where the FARMROAD lights are yellow. After a short timeout occurs it restarts the timer and goes to state HG where the HIGHWAY lights are green.

The temporal logic descriptions of the Traffic light controller are given in the next section. Also given are the incorrect VHDL model for the controller and the results of simulation (Appendix A). A fault in the AND array in the VHDL model causes the traffic light to function incorrectly. This is corrected and the corrected VHDL model and simulation results are given in Appendix B.

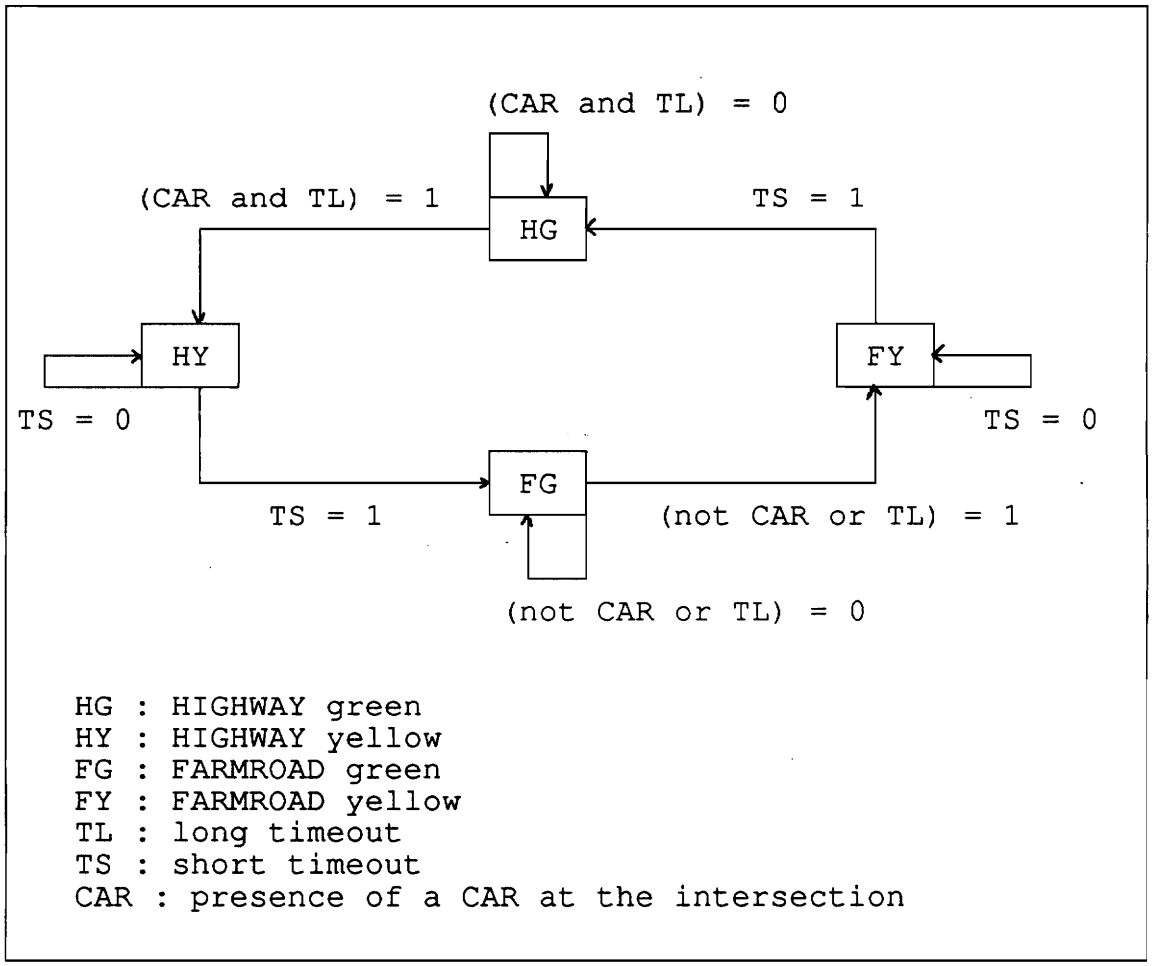


Figure 15. State Diagram of a Traffic Light Controller

5.3.1 Temporal Description of the Traffic Light Controller

@((HL = "#00") > (FL = "#10")) $ **<1>**

@((HL = "#01") > (FL = "#10")) $ **<2>**

@((FL = "#01") > (HL = "#10")) $ **<3>**

@((FL = "#00") > (HL = "#10")) $ **<4>**

@((FL = "#10") > ((HL = "#00") or (HL = "#01"))) $ **<5>**

@((HL = "#10") > ((FL = "#00") or (FL = "#01"))) $ **<6>**

@(((HL = "#00") and ~(CAR and TL) and (~ clktwo and next clktwo)) > (next (HL = "#00"))) $ **<7>**

@(((HL = "#00") and (CAR and TL) and (~ clktwo and next clktwo)) > (# (HL = "#01"))) $ **<8>**

@(((HL = "#01") and ~TS and (~ clktwo and next clktwo)) > (next (HL = "#01"))) $ **<9>**

@(((HL = "#01") and TS and (~ clktwo and next clktwo)) > (# (FL = "#00"))) $ **<10>**

@(((FL = "#00") and ~(~CAR or TL) and (~ clktwo and next clktwo)) > (next (FL = "#00"))) $ **<11>**

@(((FL = "#00") and (~CAR or TL) and (~ clktwo and next clktwo)) > (# (FL = "#01"))) $ **<12>**

@(((FL = "#01") and ~TS and (~ clktwo and next clktwo)) > (next (FL = "#01"))) $ **<13>**

@(((FL = "#01") and TS and (~ clktwo and next clktwo)) > (# (HL = "#00")))

$ **<14>**

<1> dictates that when HL is equal to green then FL should be red.

<2> dictates that when HL is equal to yellow then FL should be red.

<3> dictates that when FL is equal to yellow then HL should be red.

<4> dictates that when FL is equal to green then HL should be red.

<5> dictates that when FL is equal to red then HL can be green or yellow.

<6> dictates that when HL is equal to red then FL can be green or yellow.

<7> - **<14>** are a representation of the state diagram (figure 14) in temporal logic.

The VHDL model as given in Appendix A and B is a lower level model of the traffic light controller. The PLA has two planes, the AND and the OR plane. The inputs and their complemented values are fed into the AND planes. These are called the BIT lines. Lines which run across the BIT lines are called product lines. These are entered as '1' in the personality matrix of the PLA (see constant AND_PLANE and constant OR_PLANE in the entity PLA_IMPL(TRAF_LIGHT)).

A two phase clock is used to latch inputs and outputs. When CLKone rises the inputs are latched. The outputs are latched by the second clock CLKtwo. The outputs change 5ns after the clock CLKtwo goes high.

5.3.2 Analyses of Verification Results

The incorrect VHDL model along with the simulation results are given in Appendix A. The result of the verification run on the incorrect VHDL model is given in Table 9. The VHDL file is then corrected and the correct VHDL file along with the new simulation results are given in Appendix B. The result of verification on the corrected VHDL model is given in Table 10. The results of verification are shown only until T = 1620 ns. Please note that the same test vectors are used for simulating the incorrect and corrected VHDL models. The test bench is given in table 8. The results of a few cases are explained below (please see simulation result tables in Appendix A and Appendix B):

Verification of Temporal Formula 1

During the start of the operation during the first 20 ns neither Clkone nor Clktwo change from 0 to 1 hence both the lights are green. However due two an error in the AND_PLANE in the incorrect file there are other time instants during which both lights are green. However this situation is corrected in the correct VHDL model and there are no other time instants during which both lights are green. The same argument applies to formula 4.

Verification of Temporal Formula 8

There are errors for this formula in the correct and incorrect VHDL models. In the correct VHDL model the error arises because of the following reasons. At T = (20 - delta)ns, Clkone has not yet changed from 0 to 1 hence an error because the changes have not latched to the OR plane of the PLA. At T = (820 - delta)ns, clktwo goes from 0 to 1 and (CAR and TL) are 1. Thus HL should eventually become yellow. But this does not happen because when at T = (800 -delta)ns, when clkone goes from 0 to 1 TL is 0 and the change is not latched to the OR array. The same holds good for T = (1620 - delta)ns. However in the incorrect VHDL model there are other instants during which the error occurs. For example at T = (140 - delta)ns, Clktwo rises from 0 to 1 and (CAR and TL) are equal to 1. Also at T = (120 - delta)ns, Clkone rises and (CAR and TL) is equal to 1. Hence HL should eventually become yellow. But due to an error in the AND plane this does not occur and the error is flagged.

Verification of Temporal Formula 12

There are errors for this formula in the correct and incorrect VHDL models. In the correct VHDL model the error arises because of the following reasons. At T = (20 - delta)ns, Clkone has not yet changed from 0 to 1 hence an error because the changes have not latched to the OR array of the PLA. At T = (140 - delta)ns, clktwo goes from 0 to 1 and (~CAR or TL) is equal to 1 because TL = '1'. Also at T = (120 - delta)ns, FL is green, (~CAR or TL) = '1' because TL = '1', and Clkone rises thus latching the changes to the OR plane of the PLA. Thus FL should eventually become yellow. But this does not happen because of the error in the AND plane. However in the correct VHDL model the AND plane is correct and there is no error at this time instant.

Verification of Temporal Formula 14

This formula we get an error in the correct VHDL file but not in the incorrect file. The reason is because at T = (220 -delta)ns, FL = 01(yellow) and Clktwo rises and TS = 1, so HL should become green. But at T = (200 - delta)ns, when Clkone rises TS = 0 and the change is not latched. In the incorrect model we do not get this error because FL never becomes yellow. The reason FL does not become yellow is because of the fault in the AND plane. This error is flagged in the verification of formula 12.

Table 8. Test Bench for Traffic Light Controller

```
make base time units ns
cycle clkone   high 20ns low 20ns
starting at 0ns value 1 stopping at 5000ns

cycle clktwo   high 20ns low 20ns
starting at 0ns value 0 stopping at 5000ns

cycle ts   high 100ns low 100ns
starting at 0ns value 1 stopping at 5000ns

cycle tl   high 200ns low 200ns
starting at 0ns value 1 stopping at 5000ns

cycle car   high 400ns low 400ns
starting at 0ns value 1 stopping at 5000ns
```

Table 9. Verification Results of Incorrect VHDL Model

```
Starting VERIFICATION of VHDL model

 Starting Verification Of Temporal Logic Formula 1
     FALSE AT 0 ( 0 ) ns
     FALSE AT 20 ( 0 ) ns
     FALSE AT 105 ( 0 ) ns
     FALSE AT 120 ( 0 ) ns
     FALSE AT 140 ( 0 ) ns
     FALSE AT 905 ( 0 ) ns
     FALSE AT 920 ( 0 ) ns
     FALSE AT 940 ( 0 ) ns
 Starting Verification Of Temporal Logic Formula 2
 Starting Verification Of Temporal Logic Formula 3
 Starting Verification Of Temporal Logic Formula 4
     FALSE AT 0 ( 0 ) ns
     FALSE AT 20 ( 0 ) ns
     FALSE AT 105 ( 0 ) ns
     FALSE AT 120 ( 0 ) ns
     FALSE AT 140 ( 0 ) ns
     FALSE AT 905 ( 0 ) ns
     FALSE AT 920 ( 0 ) ns
     FALSE AT 940 ( 0 ) ns
 Starting Verification Of Temporal Logic Formula 5
 Starting Verification Of Temporal Logic Formula 6
 Starting Verification Of Temporal Logic Formula 7
 Starting Verification Of Temporal Logic Formula 8
     FALSE AT 20 ( 0 )- delta ns
     FALSE AT 140 ( 0 )- delta ns
     FALSE AT 820 ( 0 )- delta ns
     FALSE AT 940 ( 0 )- delta ns
     FALSE AT 1620 ( 0 )- delta ns
 Starting Verification Of Temporal Logic Formula 9
 Starting Verification Of Temporal Logic Formula 10
     FALSE AT 220 ( 0 )- delta ns
     FALSE AT 1020 ( 0 )- delta ns
 Starting Verification Of Temporal Logic Formula 11
 Starting Verification Of Temporal Logic Formula 12
     FALSE AT 20 ( 0 )- delta ns
     FALSE AT 140 ( 0 )- delta ns
     FALSE AT 420 ( 0 )- delta ns
     FALSE AT 940 ( 0 )- delta ns
     FALSE AT 1220 ( 0 )- delta ns
 Starting Verification Of Temporal Logic Formula 13
 Starting Verification Of Temporal Logic Formula 14
```

Table 10. Verification Results of Corrected VHDL Model

```
Starting VERIFICATION of VHDL model
 Starting Verification Of Temporal Logic Formula 1
     FALSE AT 0 ( 0 ) ns
     FALSE AT 20 ( 0 ) ns
 Starting Verification Of Temporal Logic Formula 2
 Starting Verification Of Temporal Logic Formula 3
 Starting Verification Of Temporal Logic Formula 4
     FALSE AT 0 ( 0 ) ns
     FALSE AT 20 ( 0 ) ns
 Starting Verification Of Temporal Logic Formula 5
 Starting Verification Of Temporal Logic Formula 6
 Starting Verification Of Temporal Logic Formula 7
 Starting Verification Of Temporal Logic Formula 8
     FALSE AT 20 ( 0 )- delta ns
     FALSE AT 820 ( 0 )- delta ns
     FALSE AT 1620 ( 0 )- delta ns
 Starting Verification Of Temporal Logic Formula 9
 Starting Verification Of Temporal Logic Formula 10
 Starting Verification Of Temporal Logic Formula 11
 Starting Verification Of Temporal Logic Formula 12
     FALSE AT 20 ( 0 )- delta ns
 Starting Verification Of Temporal Logic Formula 13
 Starting Verification Of Temporal Logic Formula 14
     FALSE AT 220 ( 0 )- delta ns
     FALSE AT 1025 ( 0 )- delta ns
```

Chapter 6. Conclusions

We found that temporal logic is a versatile tool for specifying various properties of hardware systems. Examples illustrating the use of temporal logic to specify hardware systems were given. We also showed the equivalence between the different timing constructs of VHDL and the operators in temporal logic. An additional operator % (Set-Up and Hold time) was introduced into modified linear time temporal logic to model the inertial delay of VHDL. The semantics of temporal logic was expanded to incorporate arithmetic operators, numbers and BIT_VECTORS. A VHDL Model Verifier was developed for comparing a VHDL model with a description given in temporal logic. For this purpose the regular operator precedence algorithm was modified for taking into account the time factor associated with signals. Also reduction and verification algorithms were developed to verify the VHDL model. The verifier was tested on models such as a JK flip-flop, a three stage synchronous counter, a memory unit and a traffic controller. The tool however assumes the existence of an temporal logic description of the hardware system. This temporal logic description is used as the "gold" description of the hardware system while comparing it with the simulation results.

Further Research

The VHDL model verifier supports all the operators in modified temporal logic. It also supports the additional operators introduced. This tool can be improved to take timing

diagrams as inputs instead of temporal logic formulae. The timing diagrams can then be converted to temporal logic formulae. The simulation results can also be shown graphically and the errors can be highlighted on the graphics screen.

This tool does not generate test vectors though it does give an option to generate exhaustive tests (which are effective if logical validity is being verified). A set of primitives can be enumerated and a scheme for generating tests for those can be developed.

The above tool combined with an automatic test pattern generater can be a very effective tool which can help reduce model debugging time considerably.

References

[1] Z.Manna and A.Pnueli, "The Modal Logic of Programs," Dep. Comput. Sci.,Stanford Univ., Stanford, CA, Rep. STAN CS 79751, 1979

[2] James R. Armstrong, "Chip Level Modeling with VHDL", Prentice Hall, New Jersey, 1989.

[3] IEEE Standard VHDL Language Reference Manual - Std 1076-1987. (New York: IEEE: 1988).

[4] M.Fujita, H.Tanaka, and T.Moto-oka, " Specifying Hardware in Temporal Logic and Efficient Synthesis of State-Diagrams Using Prolog", FGCS'84, Tokyo, November, 1984.

[5] G.V. Bochmann, "Hardware Specification with Temporal Logic: An Example", IEEE Trans. Computer, C-31, No. 3, March, 1982.

[6] M.Fujita and S.Kono, H.Tanaka and T.Moto-aka," Assistance in Hierarchial and Structured Logic Design Using Temporal Logic and Prolog", IEE Proceedings. Pt.E, Vol 133, No.5, September, 1986.

[7] D.L. Dill and E.M. Clarke," Automatic Verification of Synchronous Circuits using Temporal Logic", IEEE Proceedings. Pt.E, Vol 133, No.5, September, 1986.

[8] M.Browne, E.Clarke, D.L. Dill and B.Mishra," Automatic Verification of Sequential Circuits using Temporal Logic", IFIP 7th Computer Hardware Description Languages and their Applications, August 1985.

[9] Masahiro Fujita, Hidehiko Tanaka and Tohru Moto-Oka," Logic Design Assistance with Temporal Logic", IFIP 7th Computer Hardware Description Languages and their Applications, August 1985.

[10] B. Moszkowski," A Temporal Logic for Multi Level Reasoning about Hardware", IFIP 6th Computer Hardware Description Languages and their Applications, May 1983.

[11] Masahiro Fujita, Hidehiko Tanaka and Tohru Moto-Oka," Verification with Prolog and Temporal Logic", IFIP 6th Computer Hardware Description Languages and their Applications, May 1983

[12] W.M VanCleemput, "A Hierarchial Language for the Structural Description of Digital Systems", ACM IEEE DA Conference, June 1977.

[13] J.R. Juley and D.L. Dietmeyer, "A Digital System Design Language (DDL)", IEEE Trans. on Computer, Vol.C-17, No. 9, pp850-861, Sept 1968.

[14]. A.V. Aho, Ravi Sethi and J.D. Ullman, "Compilers, Principles, Techniques and Tools", Addison Wesley, 1986.

[15]. C.A. Mead and L.A. Conway, Introduction to VLSI Systems, MA: Addison Wesley, 1980.

Appendix A

A.1 Incorrect VHDL Model of Traffic Light Controller

```
package BV is
type BV_AND is array (0 to 9,0 to 9) of BIT ;
type BV_OR is array (0 to 9,0 to 6) of BIT ;
subtype BV10 is BIT_VECTOR (0 to 9) ;
subtype BV7 is BIT_VECTOR(0 to 7) ;
subtype BV2 is BIT_VECTOR(0 to 1) ;
end BV ;
use work.all, work.BV.all ;

entity PLA_IMPL is
port( CLKone, CLKtwo, TL, TS, CAR : in BIT ;
        HL, FL : inout BV2 ;
      ST : inout BIT );
end PLA_IMPL ;

architecture TRAF_LIGHT of PLA_IMPL is

signal input : BV10 ;
signal Y0, Y1 : BIT ;
signal OUTPUT : BV7 ;
constant TOTAL_DEL : TIME := 5 ns ;

constant AND_PLANE : BV_AND := (
        ('1', '0', '0', '0', '0', '0', '1', '0', '1', '0'),
        ('0', '0', '1', '0', '0', '0', '1', '0', '1', '0'),
        ('0', '1', '0', '1', '0', '0', '1', '0', '1', '0'),
        ('0', '0', '0', '0', '1', '0', '0', '1', '1', '0'),
        ('0', '0', '0', '0', '0', '1', '0', '1', '1', '0'),
        ('0', '1', '1', '0', '0', '0', '0', '1', '0', '1'),
        ('1', '0', '0', '0', '0', '0', '0', '1', '0', '1'),
        ('0', '0', '0', '1', '0', '0', '1', '1', '0', '1'),  -- error in 7th position
        ('0', '0', '0', '0', '1', '0', '1', '0', '0', '1'),
        ('0', '0', '0', '0', '0', '1', '1', '0', '0', '1'));

constant OR_PLANE : BV_OR := (
        ('0', '0', '0', '0', '0', '1', '0'),
        ('0', '0', '0', '0', '0', '1', '0'),
        ('0', '1', '1', '0', '0', '1', '0'),
        ('0', '1', '0', '0', '1', '1', '0'),
        ('1', '1', '1', '0', '1', '1', '0'),
        ('1', '1', '0', '1', '0', '0', '0'),
        ('1', '0', '1', '1', '0', '0', '0'),
```

```
        ('1', '0', '1', '1', '0', '0', '0'),
        ('1', '0', '0', '1', '0', '0', '1'),
        ('0', '0', '1', '1', '0', '0', '1')) ;


begin
B0 : block (CLKone = '1' and (not CLKone'stable))
 begin
 P1 : process(GUARD)
        begin
        if(CLKone = '1' and ( not CLKone'stable )) then
                INPUT <= CAR & ( not CAR ) & (TL) & (not TL ) & (TS )
                        &(not TS) & ( Y1 ) & (not Y1) & (Y0 ) &  (not Y0 ) after 1ns ;
end if ;
end process P1 ;
end BLOCK B0 ;
P2 : process (INPUT)
variable ROW : BV10 ;
begin
for I in 0 to 9 loop
ROW(I) := '0' ;
        for J in 0 to 9 loop
                if (AND_PLANE(I, J) = '0' ) then null ;
            else ROW(I) := ROW(I) or INPUT(J) ;
                end if ;
        end loop ;
end loop ;

for K in 0 to 6 loop
OUTPUT(K) <=
        (not ROW(0) and OR_PLANE(0, K)) or (not ROW(1) and OR_PLANE(1, K)) or
        (not ROW(2) and OR_PLANE(2, K)) or (not ROW(3) and OR_PLANE(3, K)) or
        (not ROW(4) and OR_PLANE(4, K)) or (not ROW(5) and OR_PLANE(5, K)) or
        (not ROW(6) and OR_PLANE(6, K)) or (not ROW(7) and OR_PLANE(7, K)) or
        (not ROW(8) and OR_PLANE(8, K)) or (not ROW(9) and OR_PLANE(9, K));
end loop ;
end process P2 ;

B2 : block(CLKtwo = '1' and (not CLKtwo'stable))
 begin
 Y0 <= guarded OUTPUT(0)       after TOTAL_DEL ;
 Y1 <= guarded OUTPUT(1)       after TOTAL_DEL ;
 ST <= guarded OUTPUT(2)       after TOTAL_DEL ;
 HL(0) <= guarded OUTPUT(3)    after TOTAL_DEL ;
 HL(1) <= guarded OUTPUT(4)    after TOTAL_DEL ;
 FL(0) <= guarded OUTPUT(5)    after TOTAL_DEL ;
 FL(1) <= guarded OUTPUT(6)    after TOTAL_DEL ;
end block B2;
end TRAF_LIGHT ;
```

A.2 Simulation Results of Incorrect VHDL Model

pattern file: input.temp, created: Mon Jun 4 12:29:07 1990

	Work.Pla_Impl.Clkone	Work.Pla_Impl.Clktwo	Work.Pla_Impl.Tl
0(0):	1	0	1
20(0):	0	1	1
25(0):	0	1	1
40(0):	1	0	1
60(0):	0	1	1
65(0):	0	1	1
80(0):	1	0	1
100(0):	0	1	1
105(0):	0	1	1
120(0):	1	0	1
140(0):	0	1	1
145(0):	0	1	1
160(0):	1	0	1
180(0):	0	1	1
185(0):	0	1	1
200(0):	1	0	0
220(0):	0	1	0
225(0):	0	1	0
240(0):	1	0	0
260(0):	0	1	0
265(0):	0	1	0
280(0):	1	0	0
300(0):	0	1	0
320(0):	1	0	0
340(0):	0	1	0
360(0):	1	0	0
380(0):	0	1	0
400(0):	1	0	1
420(0):	0	1	1
425(0):	0	1	1
440(0):	1	0	1
460(0):	0	1	1
465(0):	0	1	1
480(0):	1	0	1
500(0):	0	1	1
505(0):	0	1	1
520(0):	1	0	1
540(0):	0	1	1
560(0):	1	0	1
580(0):	0	1	1
600(0):	1	0	0
620(0):	0	1	0
640(0):	1	0	0

660(0):	0	1	0
680(0):	1	0	0
700(0):	0	1	0
720(0):	1	0	0
740(0):	0	1	0
760(0):	1	0	0
780(0):	0	1	0
800(0):	1	0	1
820(0):	0	1	1
825(0):	0	1	1
840(0):	1	0	1
860(0):	0	1	1
865(0):	0	1	1
880(0):	1	0	1
900(0):	0	1	1
905(0):	0	1	1
920(0):	1	0	1
940(0):	0	1	1

pattern file: input.temp, created: Mon Jun 4 12:29:07 1990

	Work.Pla_Impl.Ts	Work.Pla_Impl.Car	Work.Pla_Impl.Hl
0(0):	1	1	00
20(0):	1	1	00
25(0):	1	1	00
40(0):	1	1	00
60(0):	1	1	00
65(0):	1	1	01
80(0):	1	1	01
100(0):	0	1	01
105(0):	0	1	00
120(0):	0	1	00
140(0):	0	1	00
145(0):	0	1	00
160(0):	0	1	00
180(0):	0	1	00
185(0):	0	1	01
200(0):	1	1	01
220(0):	1	1	01
225(0):	1	1	01
240(0):	1	1	01
260(0):	1	1	01
265(0):	1	1	10
280(0):	1	1	10
300(0):	0	1	10
320(0):	0	1	10
340(0):	0	1	10
360(0):	0	1	10
380(0):	0	1	10

400(0):	1	0	10
420(0):	1	0	10
425(0):	1	0	10
440(0):	1	0	10
460(0):	1	0	10
465(0):	1	0	10
480(0):	1	0	10
500(0):	0	0	10
505(0):	0	0	00
520(0):	0	0	00
540(0):	0	0	00
560(0):	0	0	00
580(0):	0	0	00
600(0):	1	0	00
620(0):	1	0	00
640(0):	1	0	00
660(0):	1	0	00
680(0):	1	0	00
700(0):	0	0	00
720(0):	0	0	00
740(0):	0	0	00
760(0):	0	0	00
780(0):	0	0	00
800(0):	1	1	00
820(0):	1	1	00
825(0):	1	1	00
840(0):	1	1	00
860(0):	1	1	00
865(0):	1	1	01
880(0):	1	1	01
900(0):	0	1	01
905(0):	0	1	00
920(0):	0	1	00
940(0):	0	1	00

pattern file: input.temp, created: Mon Jun 4 12:29:07 1990

	Work.Pla_Impl.Fl	Work.Pla_Impl.St
0(0):	00	0
20(0):	00	0
25(0):	10	1
40(0):	10	1
60(0):	10	1
65(0):	10	1
80(0):	10	1
100(0):	10	1
105(0):	00	0
120(0):	00	0
140(0):	00	0
145(0):	10	1

160(0):	10	1
180(0):	10	1
185(0):	10	0
200(0):	10	0
220(0):	10	0
225(0):	10	1
240(0):	10	1
260(0):	10	1
265(0):	00	0
280(0):	00	0
300(0):	00	0
320(0):	00	0
340(0):	00	0
360(0):	00	0
380(0):	00	0
400(0):	00	0
420(0):	00	0
425(0):	00	1
440(0):	00	1
460(0):	00	1
465(0):	01	1
480(0):	01	1
500(0):	01	1
505(0):	10	0
520(0):	10	0
540(0):	10	0
560(0):	10	0
580(0):	10	0
600(0):	10	0
620(0):	10	0
640(0):	10	0
660(0):	10	0
680(0):	10	0
700(0):	10	0
720(0):	10	0
740(0):	10	0
760(0):	10	0
780(0):	10	0
800(0):	10	0
820(0):	10	0
825(0):	10	1
840(0):	10	1
860(0):	10	1
865(0):	10	1
880(0):	10	1
900(0):	10	1
905(0):	00	0
920(0):	00	0
940(0):	00	0

pattern file: input.temp, created: Mon Jun 4 12:29:07 1990

	Work.Pla_Impl.Clkone	Work.Pla_Impl.Clktwo	Work.Pla_Impl.Tl
960(0):	1	0	1
980(0):	0	1	1
985(0):	0	1	1
1000(0):	1	0	0
1020(0):	0	1	0
1025(0):	0	1	0
1040(0):	1	0	0
1060(0):	0	1	0
1065(0):	0	1	0
1080(0):	1	0	0
1100(0):	0	1	0
1120(0):	1	0	0
1140(0):	0	1	0
1160(0):	1	0	0
1180(0):	0	1	0
1200(0):	1	0	1
1220(0):	0	1	1
1225(0):	0	1	1
1240(0):	1	0	1
1260(0):	0	1	1
1265(0):	0	1	1
1280(0):	1	0	1
1300(0):	0	1	1
1305(0):	0	1	1
1320(0):	1	0	1
1340(0):	0	1	1
1360(0):	1	0	1
1380(0):	0	1	1
1400(0):	1	0	0
1420(0):	0	1	0
1440(0):	1	0	0
1460(0):	0	1	0
1480(0):	1	0	0
1500(0):	0	1	0
1520(0):	1	0	0
1540(0):	0	1	0
1560(0):	1	0	0
1580(0):	0	1	0
1600(0):	1	0	1
1620(0):	0	1	1

pattern file: input.temp, created: Mon Jun 4 12:29:07 1990

	Work.Pla_Impl.Ts	Work.Pla_Impl.Car	Work.Pla_Impl.Hl
960(0):	0	1	00
980(0):	0	1	00
985(0):	0	1	01
1000(0):	1	1	01
1020(0):	1	1	01
1025(0):	1	1	01
1040(0):	1	1	01
1060(0):	1	1	01
1065(0):	1	1	10
1080(0):	1	1	10
1100(0):	0	1	10
1120(0):	0	1	10
1140(0):	0	1	10
1160(0):	0	1	10
1180(0):	0	1	10
1200(0):	1	0	10
1220(0):	1	0	10
1225(0):	1	0	10
1240(0):	1	0	10
1260(0):	1	0	10
1265(0):	1	0	10
1280(0):	1	0	10
1300(0):	0	0	10
1305(0):	0	0	00
1320(0):	0	0	00
1340(0):	0	0	00
1360(0):	0	0	00
1380(0):	0	0	00
1400(0):	1	0	00
1420(0):	1	0	00
1440(0):	1	0	00
1460(0):	1	0	00
1480(0):	1	0	00
1500(0):	0	0	00
1520(0):	0	0	00
1540(0):	0	0	00
1560(0):	0	0	00
1580(0):	0	0	00
1600(0):	1	1	00
1620(0):	1	1	00
1625(0):	1	1	00

pattern file: input.temp, created: Mon Jun 4 12:29:07 1990

	Work.Pla_Impl.Fl	Work.Pla_Impl.St
960(0):	10	1
980(0):	10	1
985(0):	10	0
1000(0):	10	0
1020(0):	10	0
1025(0):	10	1
1040(0):	10	1
1060(0):	10	1
1065(0):	00	0
1080(0):	00	0
1100(0):	00	0
1120(0):	00	0
1140(0):	00	0
1160(0):	00	0
1180(0):	00	0
1200(0):	00	0
1220(0):	00	0
1225(0):	00	1
1240(0):	00	1
1260(0):	00	1
1265(0):	01	1
1280(0):	01	1
1300(0):	01	1
1305(0):	10	0
1320(0):	10	0
1340(0):	10	0
1360(0):	10	0
1380(0):	10	0
1400(0):	10	0
1420(0):	10	0
1440(0):	10	0
1460(0):	10	0
1480(0):	10	0
1500(0):	10	0
1520(0):	10	0
1540(0):	10	0
1560(0):	10	0
1580(0):	10	0
1600(0):	10	0
1620(0):	10	0

Appendix B.

B.1 Corrected VHDL Model of Traffic Light Controller

```
package BV is
type BV_AND is array (0 to 9,0 to 9) of BIT ;
type BV_OR is array (0 to 9,0 to 6) of BIT ;
subtype BV10 is BIT_VECTOR (0 to 9) ;
subtype BV7 is BIT_VECTOR(0 to 7) ;
subtype BV2 is BIT_VECTOR(0 to 1) ;
end BV ;

use work.all, work.BV.all ;
entity PLA_IMPL is
port( CLKone, CLKtwo, TL, TS, CAR : in BIT ;
        HL, FL : inout BV2 ;
      ST : inout BIT );
end PLA_IMPL ;

architecture TRAF_LIGHT of PLA_IMPL is

signal input : BV10 ;
signal Y0, Y1 : BIT ;
signal OUTPUT : BV7 ;
constant TOTAL_DEL : TIME := 5 ns ;

constant AND_PLANE : BV_AND := (
        ('1', '0', '0', '0', '0', '0', '1', '0', '1', '0'),
        ('0', '0', '1', '0', '0', '0', '1', '0', '1', '0'),
        ('0', '1', '0', '1', '0', '0', '1', '0', '1', '0'),
        ('0', '0', '0', '0', '1', '0', '0', '1', '1', '0'),
        ('0', '0', '0', '0', '0', '1', '0', '1', '1', '0'),
        ('0', '1', '1', '0', '0', '0', '0', '1', '0', '1'),
        ('1', '0', '0', '0', '0', '0', '0', '1', '0', '1'),
        ('0', '0', '0', '1', '0', '0', '0', '1', '0', '1'),  --error corrected
        ('0', '0', '0', '0', '1', '0', '1', '0', '0', '1'),
        ('0', '0', '0', '0', '0', '1', '1', '0', '0', '1'));

constant OR_PLANE : BV_OR := (
        ('0', '0', '0', '0', '0', '1', '0'),
        ('0', '0', '0', '0', '0', '1', '0'),
        ('0', '1', '1', '0', '0', '1', '0'),
        ('0', '1', '0', '0', '1', '1', '0'),
        ('1', '1', '1', '0', '1', '1', '0'),
        ('1', '1', '0', '1', '0', '0', '0'),
        ('1', '0', '1', '1', '0', '0', '0'),
```

```
        ('1', '0', '1', '1', '0', '0', '0'),
        ('1', '0', '0', '1', '0', '0', '1'),
        ('0', '0', '1', '1', '0', '0', '1')) ;


begin
B0 : block (CLKone = '1' and (not CLKone'stable))
 begin
 P1 : process(GUARD)
        begin
        if(CLKone = '1' and ( not CLKone'stable )) then
                INPUT <= CAR & ( not CAR ) & (TL) & (not TL ) & (TS )
                        &(not TS) & ( Y1 ) & (not Y1) & (Y0 ) &  (not Y0 ) after 1ns ;
end if ;
end process P1 ;
end BLOCK B0 ;
P2 : process (INPUT)
variable ROW : BV10 ;
begin
for I in 0 to 9 loop
ROW(I) := '0' ;
        for J in 0 to 9 loop
                if (AND_PLANE(I, J) = '0' ) then null ;
            else ROW(I) := ROW(I) or INPUT(J) ;
                end if ;
        end loop ;
end loop ;

for K in 0 to 6 loop
OUTPUT(K) <=
        (not ROW(0) and OR_PLANE(0, K)) or (not ROW(1) and OR_PLANE(1, K)) or
        (not ROW(2) and OR_PLANE(2, K)) or (not ROW(3) and OR_PLANE(3, K)) or
        (not ROW(4) and OR_PLANE(4, K)) or (not ROW(5) and OR_PLANE(5, K)) or
        (not ROW(6) and OR_PLANE(6, K)) or (not ROW(7) and OR_PLANE(7, K)) or
        (not ROW(8) and OR_PLANE(8, K)) or (not ROW(9) and OR_PLANE(9, K));
end loop ;
end process P2 ;

B2 : block(CLKtwo = '1' and (not CLKtwo'stable))
 begin
 Y0 <= guarded OUTPUT(0)      after TOTAL_DEL ;
 Y1 <= guarded OUTPUT(1)      after TOTAL_DEL ;
 ST <= guarded OUTPUT(2)      after TOTAL_DEL ;
 HL(0) <= guarded OUTPUT(3)   after TOTAL_DEL ;
 HL(1) <= guarded OUTPUT(4)   after TOTAL_DEL ;
 FL(0) <= guarded OUTPUT(5)   after TOTAL_DEL ;
 FL(1) <= guarded OUTPUT(6)   after TOTAL_DEL ;
end block B2;
end TRAF_LIGHT ;
```

B.2 Simulation Results of Corrected VHDL Model

pattern file: input.temp, created: Fri Jun 1 11:07:46 1990

	Work.Pla_Impl.Clkone	Work.Pla_Impl.Clktwo	Work.Pla_Impl.Tl
0(0):	1	0	1
20(0):	0	1	1
25(0):	0	1	1
40(0):	1	0	1
60(0):	0	1	1
65(0):	0	1	1
80(0):	1	0	1
100(0):	0	1	1
105(0):	0	1	1
120(0):	1	0	1
140(0):	0	1	1
145(0):	0	1	1
160(0):	1	0	1
180(0):	0	1	1
200(0):	1	0	0
220(0):	0	1	0
225(0):	0	1	0
240(0):	1	0	0
260(0):	0	1	0
265(0):	0	1	0
280(0):	1	0	0
300(0):	0	1	0
320(0):	1	0	0
340(0):	0	1	0
360(0):	1	0	0
380(0):	0	1	0
400(0):	1	0	1
420(0):	0	1	1
440(0):	1	0	1
460(0):	0	1	1
480(0):	1	0	1
500(0):	0	1	1
520(0):	1	0	1
540(0):	0	1	1
560(0):	1	0	1
580(0):	0	1	1
600(0):	1	0	0
620(0):	0	1	0
640(0):	1	0	0
660(0):	0	1	0
680(0):	1	0	0
700(0):	0	1	0
720(0):	1	0	0

740(0):	0	1	0
760(0):	1	0	0
780(0):	0	1	0
800(0):	1	0	1
820(0):	0	1	1
825(0):	0	1	1
840(0):	1	0	1
860(0):	0	1	1
865(0):	0	1	1
880(0):	1	0	1
900(0):	0	1	1
905(0):	0	1	1
920(0):	1	0	1
940(0):	0	1	1
945(0):	0	1	1
960(0):	1	0	1
980(0):	0	1	1
1000(0):	1	0	0

pattern file: input.temp, created: Fri Jun 1 11:07:46 1990

	Work.Pla_Impl.Ts	Work.Pla_Impl.Car	Work.Pla_Impl.Hl
0(0):	1	1	00
20(0):	1	1	00
25(0):	1	1	00
40(0):	1	1	00
60(0):	1	1	00
65(0):	1	1	01
80(0):	1	1	01
100(0):	0	1	01
105(0):	0	1	10
120(0):	0	1	10
140(0):	0	1	10
145(0):	0	1	10
160(0):	0	1	10
180(0):	0	1	10
200(0):	1	1	10
220(0):	1	1	10
225(0):	1	1	10
240(0):	1	1	10
260(0):	1	1	10
265(0):	1	1	00
280(0):	1	1	00
300(0):	0	1	00
320(0):	0	1	00
340(0):	0	1	00
360(0):	0	1	00
380(0):	0	1	00
400(0):	1	0	00

420(0):	1	0	00
440(0):	1	0	00
460(0):	1	0	00
480(0):	1	0	00
500(0):	0	0	00
520(0):	0	0	00
540(0):	0	0	00
560(0):	0	0	00
580(0):	0	0	00
600(0):	1	0	00
620(0):	1	0	00
640(0):	1	0	00
660(0):	1	0	00
680(0):	1	0	00
700(0):	0	0	00
720(0):	0	0	00
740(0):	0	0	00
760(0):	0	0	00
780(0):	0	0	00
800(0):	1	1	00
820(0):	1	1	00
825(0):	1	1	00
840(0):	1	1	00
860(0):	1	1	00
865(0):	1	1	01
880(0):	1	1	01
900(0):	0	1	01
905(0):	0	1	10
920(0):	0	1	10
940(0):	0	1	10
945(0):	0	1	10
960(0):	0	1	10
980(0):	0	1	10
1000(0):	1	1	10

pattern file: input.temp, created: Fri Jun 1 11:07:46 1990

	Work.Pla_Impl.Fl	Work.Pla_Impl.St
0(0):	00	0
20(0):	00	0
25(0):	10	1
40(0):	10	1
60(0):	10	1
65(0):	10	1
80(0):	10	1
100(0):	10	1
105(0):	00	1
120(0):	00	1
140(0):	00	1
145(0):	01	0

160(0):	01	0
180(0):	01	0
200(0):	01	0
220(0):	01	0
225(0):	01	1
240(0):	01	1
260(0):	01	1
265(0):	10	0
280(0):	10	0
300(0):	10	0
320(0):	10	0
340(0):	10	0
360(0):	10	0
380(0):	10	0
400(0):	10	0
420(0):	10	0
440(0):	10	0
460(0):	10	0
480(0):	10	0
500(0):	10	0
520(0):	10	0
540(0):	10	0
560(0):	10	0
580(0):	10	0
600(0):	10	0
620(0):	10	0
640(0):	10	0
660(0):	10	0
680(0):	10	0
700(0):	10	0
720(0):	10	0
740(0):	10	0
760(0):	10	0
780(0):	10	0
800(0):	10	0
820(0):	10	0
825(0):	10	1
840(0):	10	1
860(0):	10	1
865(0):	10	1
880(0):	10	1
900(0):	10	1
905(0):	00	1
920(0):	00	1
940(0):	00	1
945(0):	01	0
960(0):	01	0
980(0):	01	0
1000(0):	01	0

pattern file: input.temp, created: Fri Jun 1 11:07:46 1990

	Work.Pla_Impl.Clkone	Work.Pla_Impl.Clktwo	Work.Pla_Impl.Tl
1025(0):	0	1	0
1040(0):	1	0	0
1060(0):	0	1	0
1065(0):	0	1	0
1080(0):	1	0	0
1100(0):	0	1	0
1120(0):	1	0	0
1140(0):	0	1	0
1160(0):	1	0	0
1180(0):	0	1	0
1200(0):	1	0	1
1220(0):	0	1	1
1240(0):	1	0	1
1260(0):	0	1	1
1280(0):	1	0	1
1300(0):	0	1	1
1320(0):	1	0	1
1340(0):	0	1	1
1360(0):	1	0	1
1380(0):	0	1	1
1400(0):	1	0	0
1420(0):	0	1	0
1440(0):	1	0	0
1460(0):	0	1	0
1480(0):	1	0	0
1500(0):	0	1	0
1520(0):	1	0	0
1540(0):	0	1	0
1560(0):	1	0	0
1580(0):	0	1	0
1600(0):	1	0	1
1620(0):	0	1	1

pattern file: input.temp, created: Fri Jun 1 11:07:46 1990

	Work.Pla_Impl.Ts	Work.Pla_Impl.Car	Work.Pla_Impl.Hl
1025(0):	1	1	10
1040(0):	1	1	10
1060(0):	1	1	10
1065(0):	1	1	00
1080(0):	1	1	00
1100(0):	0	1	00
1120(0):	0	1	00
1140(0):	0	1	00
1160(0):	0	1	00
1180(0):	0	1	00
1200(0):	1	0	00
1220(0):	1	0	00
1240(0):	1	0	00
1260(0):	1	0	00
1280(0):	1	0	00
1300(0):	0	0	00
1320(0):	0	0	00
1340(0):	0	0	00
1360(0):	0	0	00
1380(0):	0	0	00
1400(0):	1	0	00
1420(0):	1	0	00
1440(0):	1	0	00
1460(0):	1	0	00
1480(0):	1	0	00
1500(0):	0	0	00
1520(0):	0	0	00
1540(0):	0	0	00
1560(0):	0	0	00
1580(0):	0	0	00
1600(0):	1	1	00
1620(0):	1	1	00

pattern file: input.temp, created: Fri Jun 1 11:07:46 1990

	Work.Pla_Impl.Fl	Work.Pla_Impl.St
1025(0):	01	1
1040(0):	01	1
1060(0):	01	1
1065(0):	10	0
1080(0):	10	0
1100(0):	10	0
1120(0):	10	0
1140(0):	10	0
1160(0):	10	0
1180(0):	10	0
1200(0):	10	0
1220(0):	10	0
1240(0):	10	0
1260(0):	10	0
1280(0):	10	0
1300(0):	10	0
1320(0):	10	0
1340(0):	10	0
1360(0):	10	0
1380(0):	10	0
1400(0):	10	0
1420(0):	10	0
1440(0):	10	0
1460(0):	10	0
1480(0):	10	0
1500(0):	10	0
1520(0):	10	0
1540(0):	10	0
1560(0):	10	0
1580(0):	10	0
1600(0):	10	0
1620(0):	10	0

Vita

Raghu Ardeishar was born on May 30, 1967 in New Delhi, India. He graduated with a Bachelor of Engineering degree in Electrical and Electronics Engineering (with honors) in June 1988 from Birla Institute of Technology and Science, Pilani. He then went to pursue a Master's degree in Electrical Engineering at Virginia Polytechnic Institute and State University in August 1988. He will be working as an IC design engineer at Delco, Indiana after his graduation.