Hazard Detection With VHDL in Combinational Logic Circuits with Fixed Delays

by

Ming-Cheung Chu

Thesis Submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirement for the degree of

Master of Science

in

Electrical Engineering

APPROVED:

James R. Armstrong

Dr. James R. Armstrong, Chairman

F L Gray

Dr. F. Gail Gray

Dong S Ha

Dr. Dong S. Ha

July 1992

Blacksburg, Virginia

Hazard Detection With VHDL in Combinational Logic Circuits With Fixed Delays

by

Ming-Cheung Chu

Dr. James R. Armstrong, Chairman

Electrical Engineering

(ABSTRACT)

Timing hazards are common problems found in logic circuits. A new integrated hazard detection system (HDS), which is implemented in VHDL, is proposed to detect the static, the dynamic, and the function hazards in any logic circuit that is described structurally in VHDL. This system adopts the IEEE VHDL Model Standard Group 1076-1164 Nine-Valued Multiple-Valued Logic package. Without any designer-supplied arbitrary input test patterns, the system predicts which input combinations will cause hazards, reports what type of hazards, and provides detailed timing information on the hazards in the combinational logic circuit with fixed gate delays.

Acknowledgments

I would like to thanks Dr. Armstrong for his support and guidance throughout my thesis research and for being my thesis advisor. Under his guidance, I have learned many different disciplines in technical areas and in engineering ethics. I also appreciate Dr. Gray and Dr. Ha for serving as my committee members.

I would like to thank Rob Greenwood, Chris Ryan, Noreen Sullivan, Dr. Caisy Ho, and Dr. Alex Chu for serving as my proof readers throughout my research. Also, I would like to thank Chang Cho, and Yi-Feng Jang for their encouragement. I would like to give my special thanks to my parents (Yuk-Kwong Chu and Sin-Jun Koo), my sisters (Shuk-Man Chu and Shuk-Kwan Chu), and also my best friend (Miss Tsun Yuen) for their invaluable support and patience. Finally, I praise GOD for His miraculus support.

Tables of Contents

List of Tables

List of Illustrations

Chapter 1. Introduction

1.1 Motivation

Timing hazards are common problems existing in logic circuits. From transistor level to chip level, timing hazards are produced by unfavorable input combinations and by device propagation delays. In a small scale digital circuit, a timing hazard can be detected easily. However, as VLSI technology is pushing forward development on compact size and high performance integrated circuits, timing hazard detection becomes a difficult problem during circuit design and timing analysis. In practice, when a harmful hazard exists in the circuit, a logic designer has to spend a great amount of time to find out what type of hazard it is and which input patterns are causing the hazard. In the past, some researchers had proposed the use of multiple-valued logic (MVL) to detect hazards. For example, Eichelberger, E.B.**[1]** has proposed the use of ternary (three-valued: 0, 1, and 1/2) logic to detect both static and function hazards. Daniel W. Lewis **[2]** has proposed the use of quinary (five-valued 0, 1, 0/1, 1/0, and 1/2) logic to detect static hazards. Breuer and Harrision **[3]** have proposed the use of eight-valued logic to detect both static and dynamic hazards. However, the theories behind MVL and their detection algorithms are complex. In most cases, the designer has to provide arbitrary input patterns to detect the possibility of hazards. Therefore, the goal of automation is difficult to achieve.

In today's technology, there are few existing simulators that are able to report glitches or hazards in logic circuits. One such product is called LADD 5.00 series E **[4].** This design tool

automatically finds spikes and propagation delays that will cause gate malfunction. However, the tool is unable to predict what kind of spike it is and what input combinations cause the spike.

Hardware description languages (HDLs) have traditionally been used to describe digital circuits of various sizes and complexity. From transistor level to system level and from behavioral models to structural models, HDLs have proven to be a necessary tool for design verification, logic synthesis, and design representation. One of the most widely accepted HDLs is the VHSIC Hardware Description Language (VHDL)**[5, 6, 7, 8, 9]**. Work with VHDL has successfully been done in the areas of behavioral modeling, structural modeling, abstract level modeling and logic synthesis. However, to date, little work exists in the open literatures as far as VHDL diagnostic tools are concerned. Hence, there is a growing interest in using VHDL as a diagnostic tool in today's Electronics Design Automation (EDA) industry.

The primary objectives of this thesis are to:

1. Assess the capability of VHDL as a diagnostic tool to describe and verify the timing and the logic behavior of a circuit which is described in VHDL.
2. Develop an *integrated system* that is capable of reporting common types of hazards, and capable of predicting all input combinations that cause hazards for any test circuits that are described in VHDL. The proposed integrated system is called the VHDL Hazards Detection System (VHDS).

This thesis concentrates on the theory, design and applications of the integrated system, in attempting to develop a proper modeling methodology for system design and verification using VHDL.

1.2 VHDL Features for the Hazards Detection System

VHDL is a rich and powerful hardware description language used to model analog and digital systems. It has a few important features that make it suitable for attempting to use it as a diagnostic tool for hazards detection. In particular, there are five major features of VHDL that distinguish it from other HDLs as a powerful tool for CAD design and they are as follows:

1. Design Library:

A VHDL design library holds a collection of error-free VHDL models, packages, and subprograms. As a diagnostic tool, the design library stores the VHDS and makes it visible to the designer by specifying the proper library clauses. The Synopsys VHDL user-option file (vhdl.uof) **[16]** for the VHDS is specified as follows:

```
TIMEBASE = ns
WORK > NIBBLE
NIBBLE: //vtvhdl/local_user/chu/nibble
VHDS_LIB > VHDS_DIR
VHDS_DIR: //vtvhdl/local_user/chu/vhds_dir
```

The integrated hazards detection system can be visible to the source program by adding the following library and use clauses in the source program.

```
library VHDS_LIB;
use VHDS_LIB.PACKAGE_IDENTIFIER.all;
```

2. Packages and Subprograms

Packages provide a method of holding useful resources so that different design units are allowed to share the same information. Subprograms define algorithms for computing values or

exhibiting behavior. Subprograms can be overloaded. That is, VHDL supports polymorphism and thus, it adds great flexibility and extension to the language. The use of packages and subprograms is especially useful for tool designing and building. It not only enhances the coding *modularity,* but also makes it easier for the designer to debug any problem and to improve the design. The VHDS takes the advantage of the packages and subprograms and creates several packages with unique purposes. For example, one package is designed for hazard detection, another package is designed for timing analysis, and yet another package is designed for the hazard patterns generator, etc.

3. Record Type

VHDL has pre-defined types such as TIME, BIT, and BOOLEAN, etc. In addition, it allows the designer to declare his or her own abstract data types and even abstract record types. A record type is a composite type consisting of a number of homogeneous or heterogeneous fields. Each element of the record can be accessed by specifying the correct field name. In the VHDS, the record type is particularly useful in defining complex abstract data types. For example, the following record type declaration holds the time values of minimum propagation delay and maximum propagation delay of a logic circuit.

```
type PROP_DELAY_SPEC is
record
MIN_PROP_DELAY: TIME;
MAX_PROP_DELAY: TIME;
end record;
```

Also, the VHDL record type declaration can expand the capability of the subprogram's use. In the VHDS, various data and control variables in the subprograms are required to memorize values

that are frequently used. With the advantage of record type declaration, all the values can be grouped into a record type so that only one composite variable is needed to hold all the values.

4. File Declaration and Text I/O

VHDL File Declaration and Text I/O provide a means to transfer information in and out from the host environment. With this capability, information on any hazard found in a logic circuit can be written to an external file through the file declaration and the text I/O. This external file becomes the hazard report summary of the test circuit. The summary describes which input combinations cause the existence of hazards, as well as the detailed description of their timing and logic behavior.

5. Assertion Statements

An assertion statement checks to see if a specific condition is true and reports an error if it is false. The assertion statement allows the designer to identify if various conditions have been violated in some parts of the design. In the VHDS, the assertion statement is a powerful mechanism to report hazards. It provides a detailed description on which type of hazard, the timing information of hazard, and which signal has found the hazard.

The features discussed above are important in designing the VHDS, as will be seen in later chapters.

1.3 Thesis Organization

The thesis is organized as follows: Chapter 2 reviews the fundamental concepts and definitions of timing hazards. It also discusses how to eliminate hazards. Chapter 3 introduces the integrated

system called the VHDL Hazard Detection System (VHDS). Detailed description of the multi-valued logic package, the timing package, the hazard patterns generator, the hazard detection package, and the components library will also be discussed. Chapter 4 discusses how to use the VHDS in testing a logic circuit. This includes creating the user-specification file, the VHDL source file design, and the test bench shell design. This chapter also explains how to interface the VHDS with the above files. Chapter 5 shows the descriptions and the simulation results of some sample test circuits. It also discusses the overall performance of the VHDS. Chapter 7 concludes the thesis work. The appendix is the source code listings of all the packages and subprograms in the VHDS.

Chapter 2. Overview of Hazards

2.1 Introduction to hazards

A *hazard* exists in a logic circuit when a certain input signal(s) produces an erroneous transient output signal. This erroneous transient output signal is called a *spike*. Hazards usually occur due to unfavorable input combinations, input transition times, and device propagation delays **[1, 8, 11, 12]**. For example, Figure 1 shows a combinational circuit connected to a sequential circuit. If the combinational circuit does not behave normally (i.e. has hazards), the hazards in the combinational circuit may cause an unexpected state transition in the sequential circuit and thus may cause circuit malfunction.

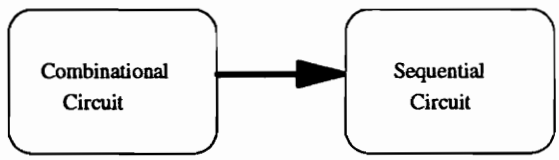


Figure 1. Block Diagam of a Switching Network

Hence, the most common effect of a hazard is circuit malfunction. Therefore, understanding hazards is absolutely essential for the design of a reliable circuit. Figure 2 on the next page shows a tree structure of common types of hazards that exist in a logic circuit **[1, 11, 12, 13, 14]**.

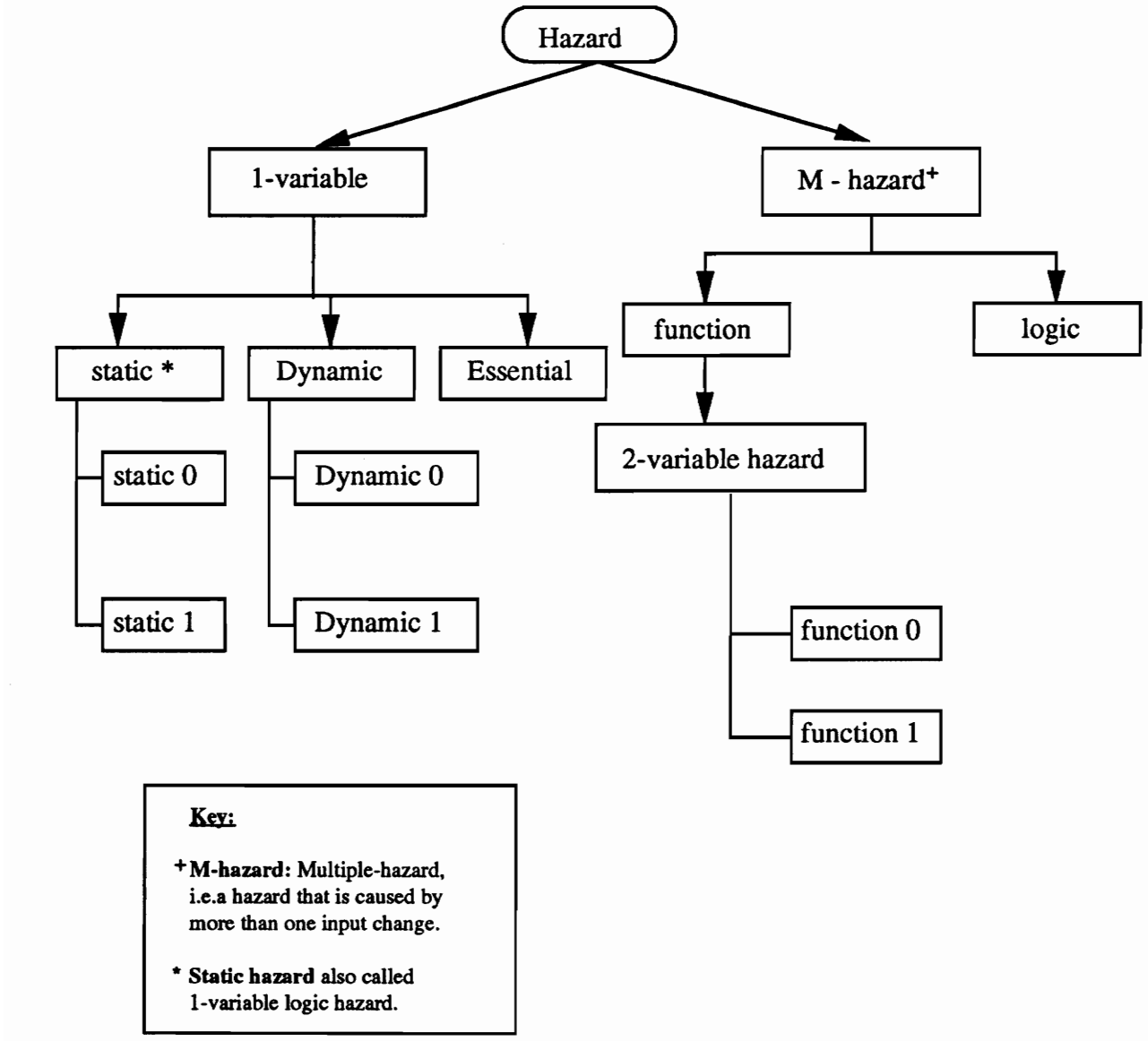


Figure 2. Summary of Hazard

From Figure 2, it can be seen that hazards can be divided into two major types. One type is called a *single-input change hazard (or one-variable hazard)* and the other type is called a *multiple-input change hazard (or M hazard)* **[1]**. Single-input change hazards have been defined in terms of single-input changes. The static hazard, the dynamic hazard, and the essential hazard are classified as *one-variable hazards.* Multiple-input change hazards have been defined in terms of multiple-input changes. That is, when two or more inputs are changed and a spurious hazard pulse exists on the output signal of a switching circuit, then a *M-hazard* is said to be present. Two major kinds of hazards are defined as M-hazards. They are the logic hazard and the function hazard. A logic hazard is similar to a static hazard and can always be eliminated by properly designing the logic circuit. A function hazard is inherent in the Boolean function and cannot be eliminated by modifying the logic circuit. A function hazard can be further classified into two types, the *function 0 hazard* and the *function 1 hazard* **[1, 10, 11, 13]**. Figures 3 and 4 illustrate the function 0 hazard and the function 1 hazard, respectively. In Figure 3, a function 0 hazard is present for the transition of WXYZ from "1100" (state A) to "1001" (state B). The output value of both input states A and B is '0'. Unless both inputs x and z change simultaneously, either state C or state D will be the intermediate state. The output value of both state C and state D is '1'. Hence, a spurious 1 output is present when the output is supposed to remain at '0'. In Figure 4, a function 1 hazard is present for the transition of WXYZ from "1110" to "0111" if the input z is the first to change. In this case, state F will be the intermediate state and has a '0' output. Therefore, a transient '0' output exists in the state transition from "1110" to "0111". In this thesis, VHDS concentrates on the theory and detection of the static hazard, the dynamic hazard, and the function hazard involving two-input changes, i.e. two-input change function hazard.

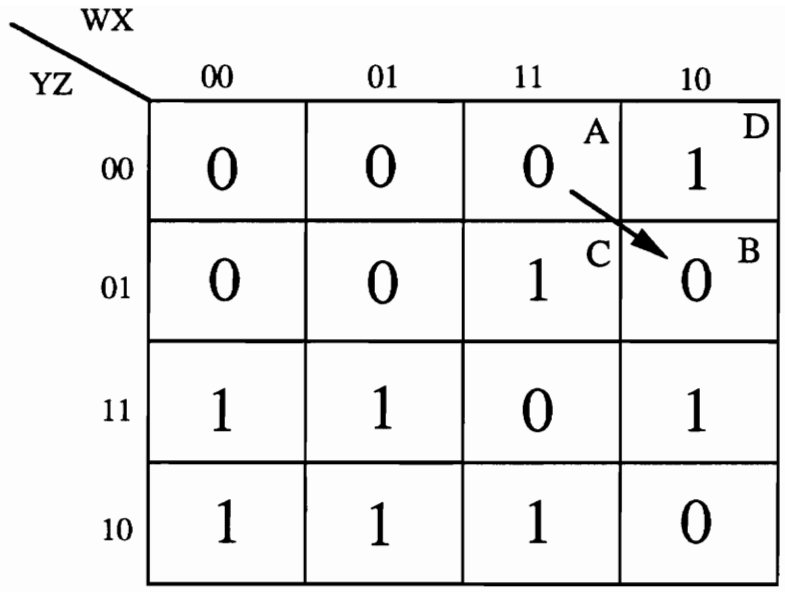


Figure 3. K-Map Illustration of the Function 0 Hazard

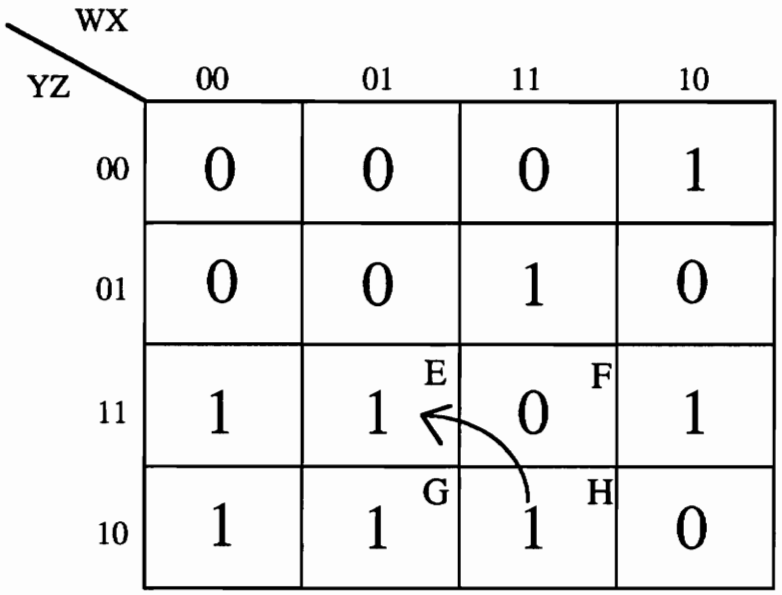


Figure 4. K-Map Illustration of the Function 1 Hazard

2.2 Definitions of Hazards

A *static 0 hazard* **[11, 12, 13, 14]** has the following characteristics:

(1) It ia caused by a sequence of two input combinations that only differ in one input variable,

(2) the output before the erroneous change is equal to the output after the erroneous change and has an output logic value '0', and

(3) during the transition in the input combinations pair, an erroneous momentary 1 pulse occurs at the output signal. The static 0 hazard waveform is illustrated in Figure 5a.

A *static 1 hazard* **[11, 12, 13, 14]** has the following characteristics:

(1) It is caused by a sequence of two input combinations that only differ in one input variable,

(2) the output before the erroneous change is equal to the output after the erroneous change and has an output logic value '1', and

(3) during the transition in the input combinations pair, an erroneous momentary 0 pulse occurs at the output signal. The static 1 hazard waveform is illustrated in Figure 5b.

A *dynamic 0 hazard* **[11, 12, 13, 14]** has the following characteristics:

(1) It is caused by a sequence of two input combinations that only differ in one input variable, and

(2) a triple change in the output (i.e. $1 \rightarrow 0 \rightarrow 1 \rightarrow 0$) is found. The dynamic 0 hazard waveform is illustrated in Figure 6a.

A *dynamic 1 hazard* **[11, 12, 13, 14]** has the following characteristics:

(1) It is caused by a sequence of two input combinations that only differ in one input variable, and

(2) a triple change in the output (i.e. $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$) is found. The dynamic 1 hazard waveform is illustrated in Figure 6b.

Static hazards and dynamic hazards are independent of time delay between the input combinations.

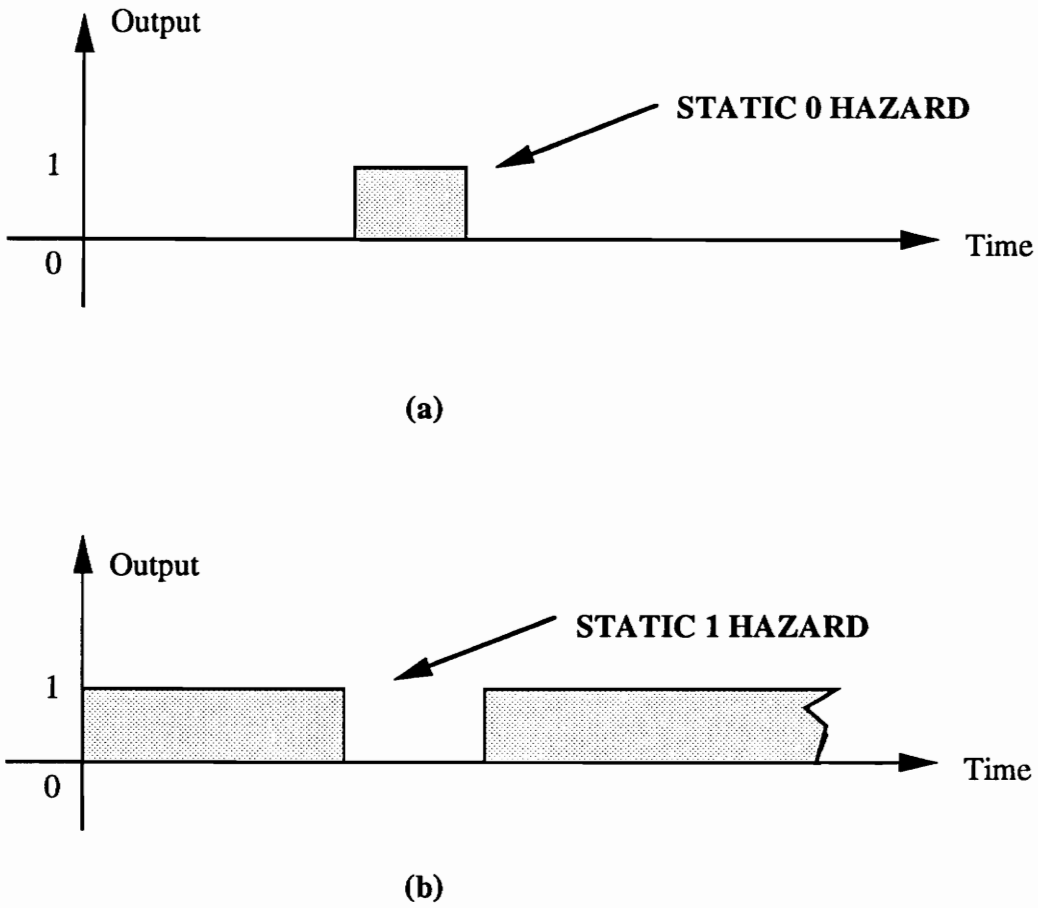


Figure 5. Illustration of the Static Hazard Waveform

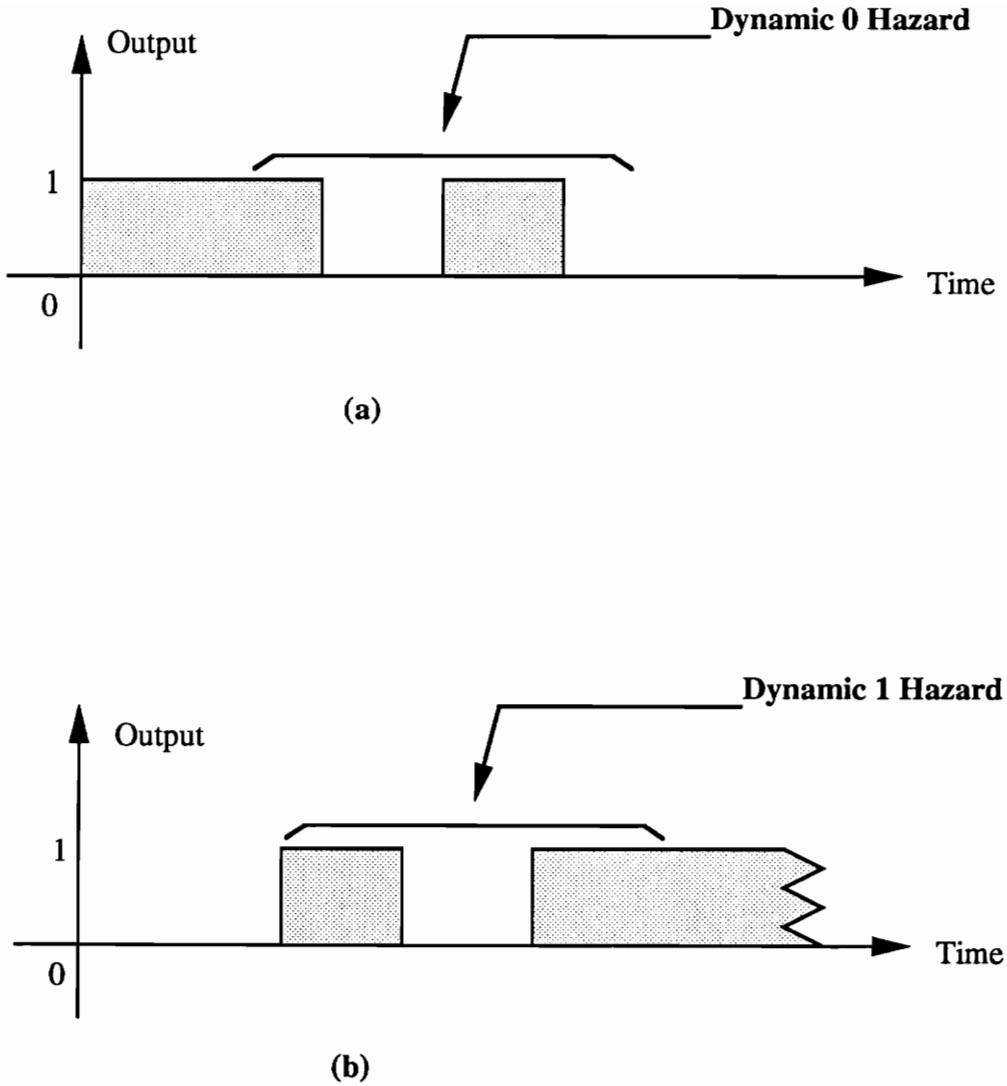


Figure 6. Illustration of the Dynamic Hazard Waveform

A *two-variable function 0 hazard* or *function 0 hazard* in short **[10, 11, 13]** has the following characteristics:

(1) It is caused by a pair of input combinations that only differ in two input variables and both input variables change simultaneously during the state transition,

(2) the output before the erroneous change is equal to the output after the erroneous change and has an output logic value '0', and

(3) during the transition in the input combinations pair, an erroneous spurious 1 pulse occurs on the output. The function 0 hazard waveform is illustrated in Figure 7a.

A *two-variable function 1 hazard* or *function 1 hazard* in short **[10, 11, 13]** has the following characteristics:

(1) It is caused by a pair of input combinations that only differ in two input variables and both input variables change simultaneously during the state transition,

(2) the output before the erroneous change is equal to the output after the erroneous change and has an output logic value '1', and

(3) during the transition in the input combinations pair, an erroneous spurious 0 pulse occurs on the output. The function 1 hazard waveform is illustrated in Figure 7b.

A function hazard is a delay-sensitive hazard. That is, it depends on the circuit delays.

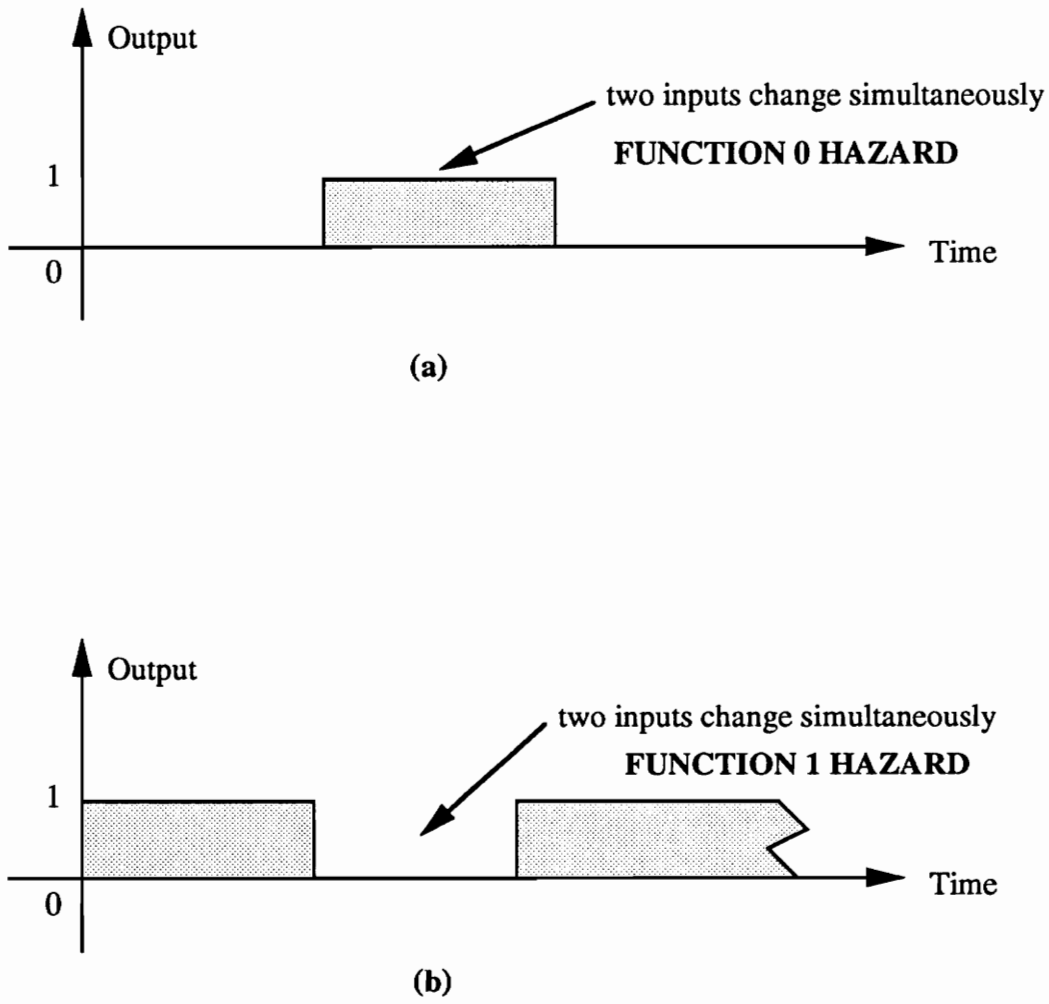


Figure 7. Illustration of the Function Hazard Waveform

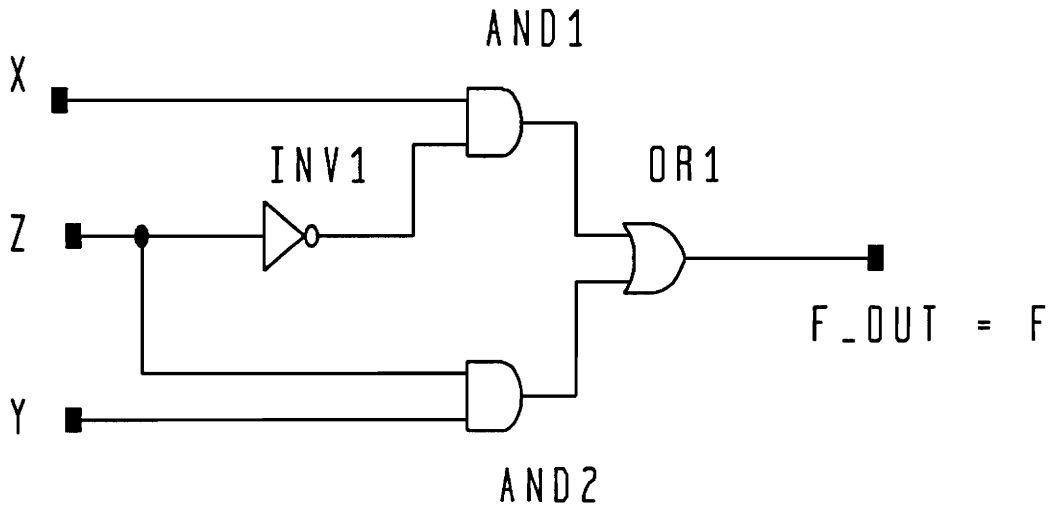
2.3 Elimination of Hazards

Static hazards can be eliminated logically by adding an extra prime implicant or *consensus term* to the two covered prime implicants. Figures 8 and 9 illustrate the phenomena of the static 1 hazard and how to eliminate the hazard, respectively. Figure 8a shows the circuit realized from the Karnaugh map which is shown in Figure 8b. The boolean equation for the circuit shown in Figure 8a is XZ' + YZ[1]. A static 1 hazard exists between the state transition of XYZ from 111 to 110. To eliminate the static 1 hazard, a consensus term (XY) is added to cover the two prime implicants, XZ' and YZ. This is illustrated in Figure 9a. Hence, the hazard-free logic network for the above example is shown in Figure 9b. A similar method can be applied to eliminate the static 0 hazard. Another way to eliminate the static hazard is to use output inertial delay to filter out the erroneous spike.

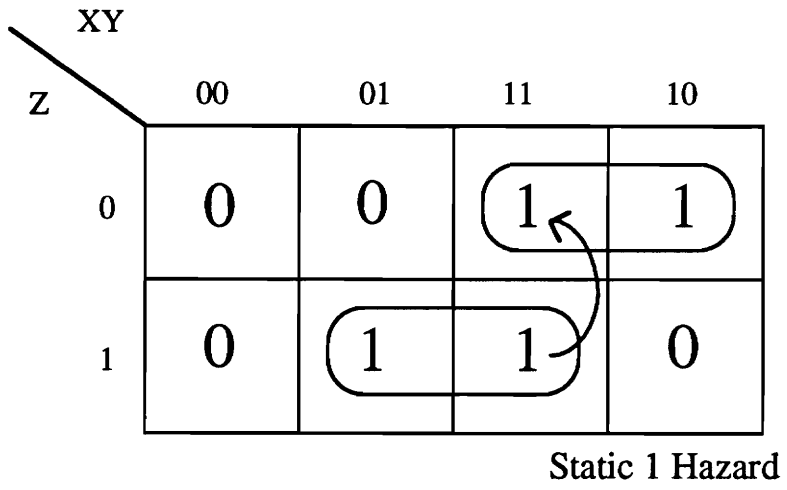
A dynamic hazard is caused by the propagation delay of the components inside the circuit. Dynamic hazards may exist when the event of an input signal has three different propagation paths with different propagation delays. One way to eliminate the dynamic hazard is to add delay elements in each delay path so that the triple output transitions are eliminated. Another way to eliminate the dynamic hazard is to use output inertial delay to filter out the erroneous spike.

A function hazard is inherent to the boolean function of the circuit. Hence, re-designing the logic network does not eliminate the function hazard. There are two ways to eliminate the function hazard. The first method is to add delay elements in the logic circuit so that the output spike is eliminated. The second method is to use output inertial delay to filter out the spike that

[1] XZ' + YZ implies (X and (complement of Z)) or (Y and Z)

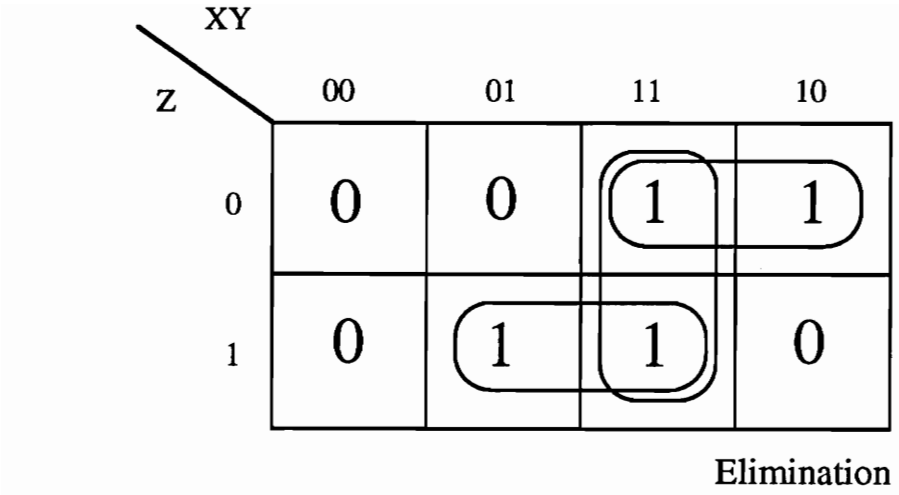


(a)

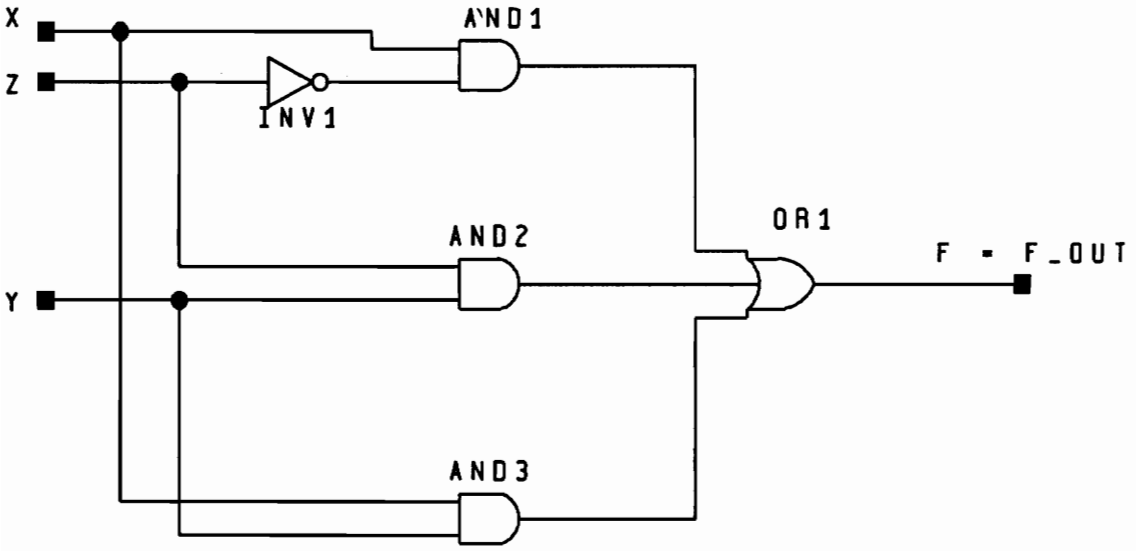


(b)

Figure 8. Illustration of the Static Hazard



(a)



(b)

Figure 9. Elimination of the Static Hazard and its Realization

contributes the function hazard. Table 1 compares the characteristics and the elimination of the static hazard, the dynamic hazard, and the function hazard.

Table 1. Summary of Hazards Characteristics and Elimination

Hazards	Characteristics	Elimination
Static Hazard	**Single-input**	**Re-design Logic Network; Output Inertial Delay.**
Dynamic Hazard	**Single input**	**Add Delay Element in Delay Path; Output Inertial Delay.**
Function Hazard	**Multiple-input**	**Add Delay Elements in Logic Circuit; Output Inertial Delay.**

2.4 Hazard Detection Techniques in VHDL

There are two main ways to detect hazard in a logic circuit which is described in VHDL. One technique is called *normal mode detection*. The other technique is called *test mode detection*. The normal mode detects hazards in a logic circuit during normal circuit simulation. The designer supplies test vectors to the test circuit, and the hazards based on that set of test vectors are detected. Usually, the normal mode detection does not able to find all the hazards exist in the test circuit.

The test mode detection is different from the normal mode detection in two ways. First, the test mode detection detects all the possible hazards that exist in a given logic circuit. Second, the designer is not required to supply the test patterns. The only requirement is to interface the test circuit with the detection system properly. Once the system is connected, the test mode detection

system should able to detect all the possible hazards that exist in the test circuit. The VHDS discussed in this thesis uses the test mode detection technique.

Chapter 3. The VHDL Hazard Detection System (VHDS)

3.1 Introduction to VHDS

The VHDL Hazard Detection System (VHDS) is an integrated system which uses the test mode detection technique. The system determines all the possible input combinations that cause hazards and reports what type of hazards exist. The VHDS works for circuits which are structurally described in VHDL. The VHDS is primarily used for accurate gate level modeling and testing. The comprehensive timing and the logic packages in the VHDS are used for describing the logic and the timing behavior of a given test circuit. With the dynamic nature of the VHDS, a logic designer is able to discover all the possible input combinations which cause the different types of hazards without supplying test patterns. Therefore, the system saves the designer a lot of time in finding out all the "potential" hazards in the circuit. In addition, the VHDS has the ability to compute the minimum (min) and the maximum (max) propagation delays (shortest and longest delay paths problem) for any logic circuit that is structurally described in VHDL.

The VHDS is coded in IEEE Standard 1076-1987 VHDL and is compiled and simulated under the Synopsys V2.1c VHDL environment on an Apollo Domain Series 3500 machine. The VHDS system consists of three major parts, the User-Specification, the VHDS Design Library, and the VHDS Output Files. A general block diagram for the VHDS is shown in Figure 10. The detailed overview of the VHDS is shown in Figure 11.

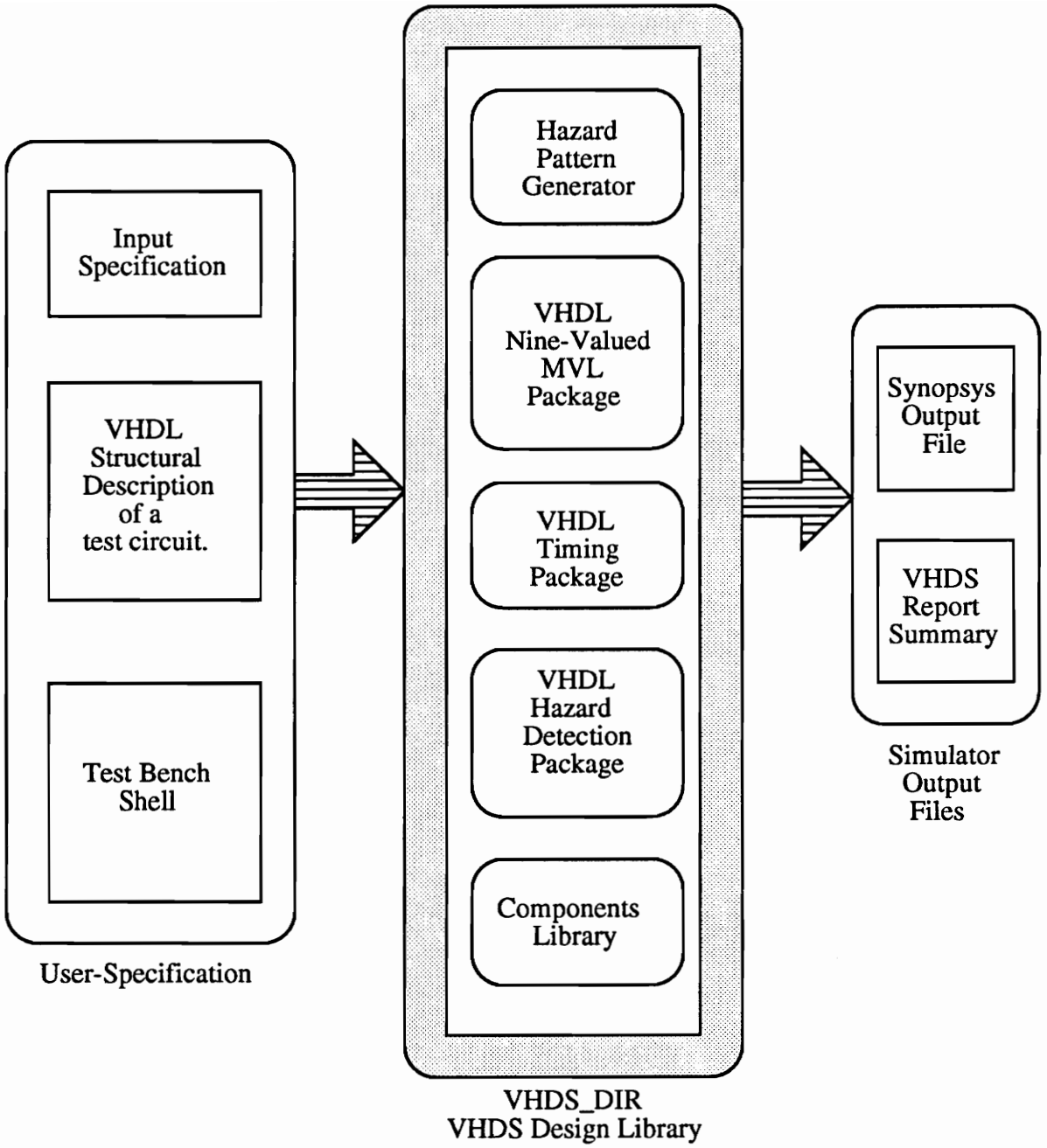


Figure 10. Block Diagram of VHDS (VHDL Hazard Detection System)

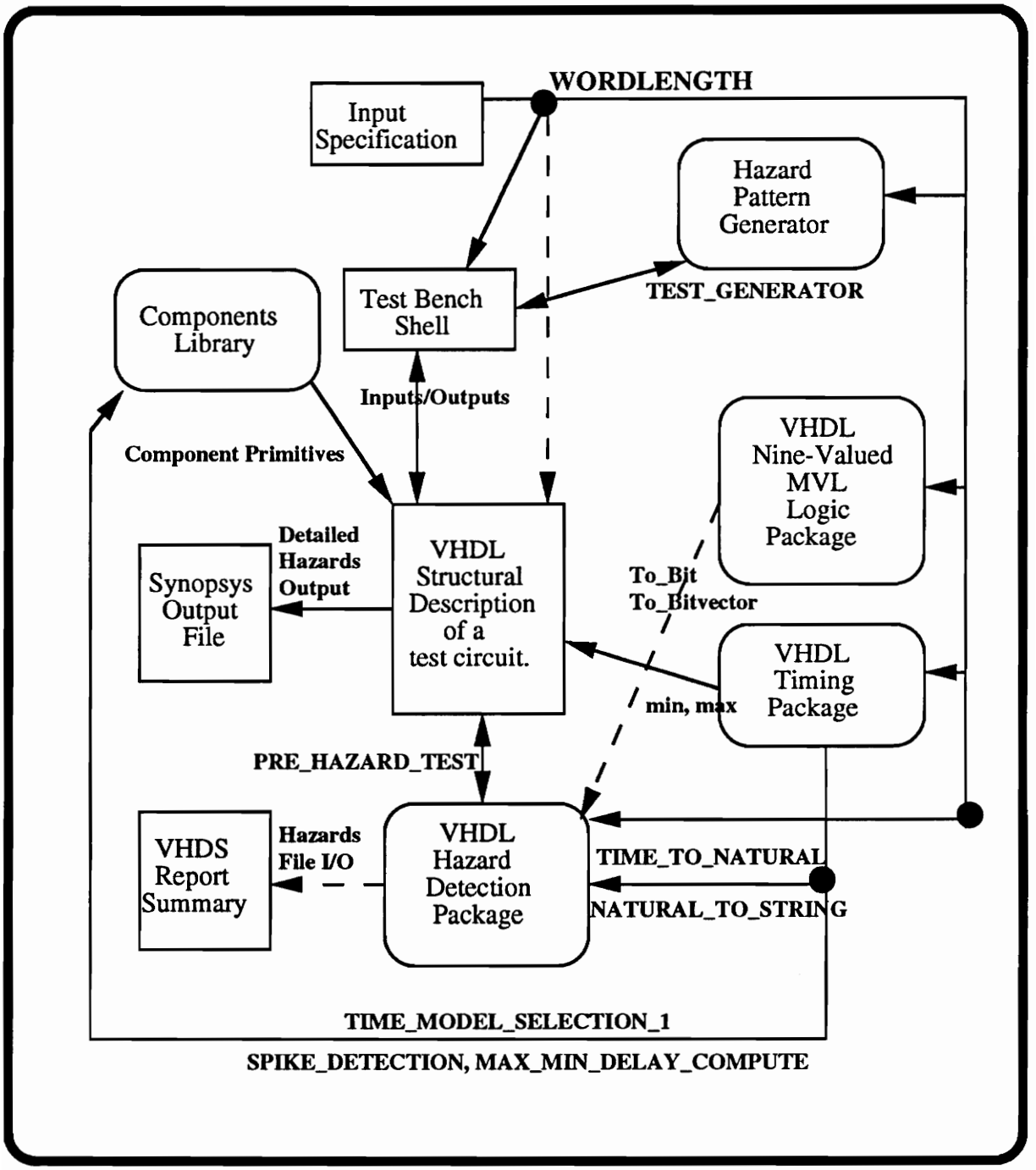


Figure 11. Detailed Overview of the VHDS

The User-Specification specifies the physical information on the test circuit and the interface descriptions between the test circuit and the VHDS Design Library. The information includes the number of inputs of the circuit, the VHDL structural description of the circuit, and the test bench shell which specifies the interface descriptions between the test circuit and the VHDS Design Library.

The VHDS Design Library is the most important portion of the system. The Design Library contains five components. They are the Hazard Patterns Generator (HPG), the VHDL nine-valued Multiple-Valued Logic (MVL) package, the nine-valued Timing Package, the nine-valued VHDL Hazards Detection package, and the nine-valued Components Library. The Hazard Patterns Generator (HPG) is a package which produces all the possible input combinations that are required for hazards detection. The generator produces all the possible input combinations that are suitable for the single-input change hazards and the two-input change hazards. The VHDL nine-valued MVL package provides the modeling capability for accurately describing and modeling the logic behavior of the test circuit. The VHDL nine-valued timing package provides a mechanism for modeling the timing behavior of the test circuit. The VHDL Hazard Detection package contains detection mechanisms which are used to distinguish all the different hazard types. The Components Library collects some common primitives for gate level modeling.

The VHDS Output Files are the simulation results. They consist of two kinds of files, the Synopsys output file and the VHDS report summary. The Synopsys output file contains the detailed timing values for the input combinations, the output values, and the hazard status for each input combination. The VHDS report summary summarizes the hazard status of the test circuit. That is, the report only describes the timing and the logic values for all the input combinations in which the hazards are found.

This chapter begins with the VHDL nine-valued logic package. Then, the VHDS timing package will be explained in detail. The design algorithm for the HPG will then be presented. Next, the methodology of hazard detection in the VHDS Hazard Detection package will be discussed. Finally, the chapter concludes with a brief description of the VHDS components library.

3.2 Logic Design in VHDS

VHDL is a unique HDL with great flexibility in specifying logic systems. This capability offers the designer a wide variety of choices for modeling different kinds of systems effectively. It also ensures accurate simulation of a variety of logic families. Hence, a powerful logic package can provide a great aid for the designer to build a more realistic and reliable system. The VHDS logic package adopted the package that was proposed by the IEEE VHDL Model Standard Group. The package is called STD_LOGIC_1164 **[15, 18]** and was designed by William Billowitch. The logic values in the package are {'U', 'X', '0', '1', 'W', 'H', 'L', 'Z', '-'}.

3.2.1 Introduction to Multiple-Valued Logic (MVL)

MVL can represent more information than the binary logic {'0', '1'}. So, MVL has an advantage over the binary logic in circuit description and modeling. Today, modeling and testing of real logic circuits and systems frequently employ the use of more than two states {'0', '1'}. The most common example is the addition of the high impedance state 'Z' which appears mostly in tri-state logic and bus modeling. More logic values are required when the circuits or systems are becoming more complex. Hence, there is a growing trend in developing a more powerful logic package that is more suitable in modeling today's rapid changing technology. For example, the Synopsys company has developed a seven-valued logic package **[16]** {'X', '0', '1', 'Z', 'W, 'H', 'L'}

which is used for VHDL modeling. The Vantage Analysis, Inc. has also developed a forty-six valued logic package **[9, 17]** based on the idea of interval logic. The use of MVL enhances the representation of a logic system in the areas of accuracy and unknown handling. However, the simulation efficiency decreases as the number of MVL values increases. Since there are no industrial standard packages available at this moment, incompability and type conversion problems always occur when one system is trying to map to another system in which both systems employ different MVL logic systems. Thus, the nine-valued logic system described above is a de-facto standard logic system in describing VHDL models from gate level to system level.

3.2.2 Standard 1076-1164 Nine-Valued Logic Package

The VHDS MVL package employs three logic states {'X', '0', '1'}, three logic strengths {f(forcing/strong), w (weak), Z (tri-state/high-impedance)}, and two special states {'U', '-'}. Hence, the total nine-valued logic values are {'U', fX => 'X', f0 => '0', f1 => '1', wX => 'W', w1 => 'H', w0 => 'L', ZX = Z0 = Z1 = 'Z', '-'}. The logic 'U' is called the *Uninitialized Logic Value* and is used for circuit initialization purposes. The logic value 'X' is called the *forcing (strong) unknown value*. It is used for modeling bus conflict problems. Logic values '0' and '1' are called the strong '0' and strong '1', respectively. They are the normal binary logic values. Logic value 'W' is called the weak unknown value or weak 'X'. Logic value 'L' and 'H' are called the weak '0' and weak '1', respectively. Logic value 'Z' is called the high-impedance state and is useful for modeling tri-state logic as well as bus contention. Finally, the logic value '-' is called the *don't care* value. This logic value is in the special state category and is not used in simulations but in logic synthesis. The nine-valued logic package describes the circuit effects and the logic behavior more accurately than the conventional binary logic {'0', '1'} and the four-valued logic {'X', '0', '1',

'Z'} during circuit simulation. It is also suitable for modeling different technologies such as TTL, CMOS, GaAs, NMOS, PMOS, and ECL digital devices. The logic package may be insufficient for transistor level modeling. For the VHDS requirements and needs, the nine-valued logic package is adequate.

The filename for the nine-valued logic package in the VHDS is called *mvl9.vhd*. The package identifier is called MVL9_SYSTEM. The nine-valued logic type is declared as type MVL9. Vectorized arrays are defined as type MVL9_VECTOR. This package supports logic operations such as "*and*", "*nand*", "*or*", "*nor*", "*xor*", and "*not*". These operations also work for multiple inputs. The function *resolved* is a wired-X bus resolution function which handles bus contention problems. The function *To_bit* is a type conversion function which converts any nine-valued logic (MVL9) into binary logic (BIT). The function *To_bitvector* is also a type conversion function which converts any nine-valued logic vector (MVL9_VECTOR) into binary logic vector (BIT_VECTOR). Finally, the table *convert_to_X01* is a look-up table which converts any nine-valued logic values (MVL9) into its logic state {'X', '0', or '1'}.

3.3 Timing Design in VHDS

Timing design is one of the most important considerations in modeling analog and digital circuits. In accurate timing design, timing relationships among signals within a system and timing requirements of external signals are precisely described. Timing design also provides detailed timing information when timing errors occur. One such example is the detection of timing hazards. Since timing hazards are caused by unfavorable input transition timing and device propagation delays, accurate timing representation is essential to predict the existence of hazards.

The following section begins with the fundamental concept in timing design. Then, the basic VHDL timing delay models will be introduced. Next, the nine-valued VHDS timing package is presented. All important VHDS timing features will then be discussed in detail.

3.3.1 Fundamental Circuit Timing Concept

All logic circuits have delays associated with their wiring and logic devices. That is, when an input has an event, the output may change after a certain propagation delay. Two common delay parameters are usually specified for a given circuit. They are the *minimum propagation delay* (min) and the *maximum propagation delay* (max). Minimum propagation delay specifies the minimum amount of time that must pass before an output will respond to an input event. Maximum propagation delay specifies the maximum amount of time that must pass before an output will respond to an input event. This is shown in Figure 12.

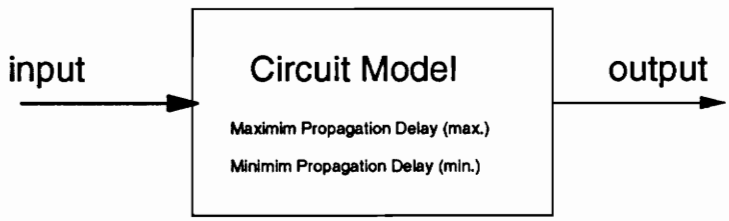


Figure 12. Basic Circuit Model with Min and Max Propagation Delays

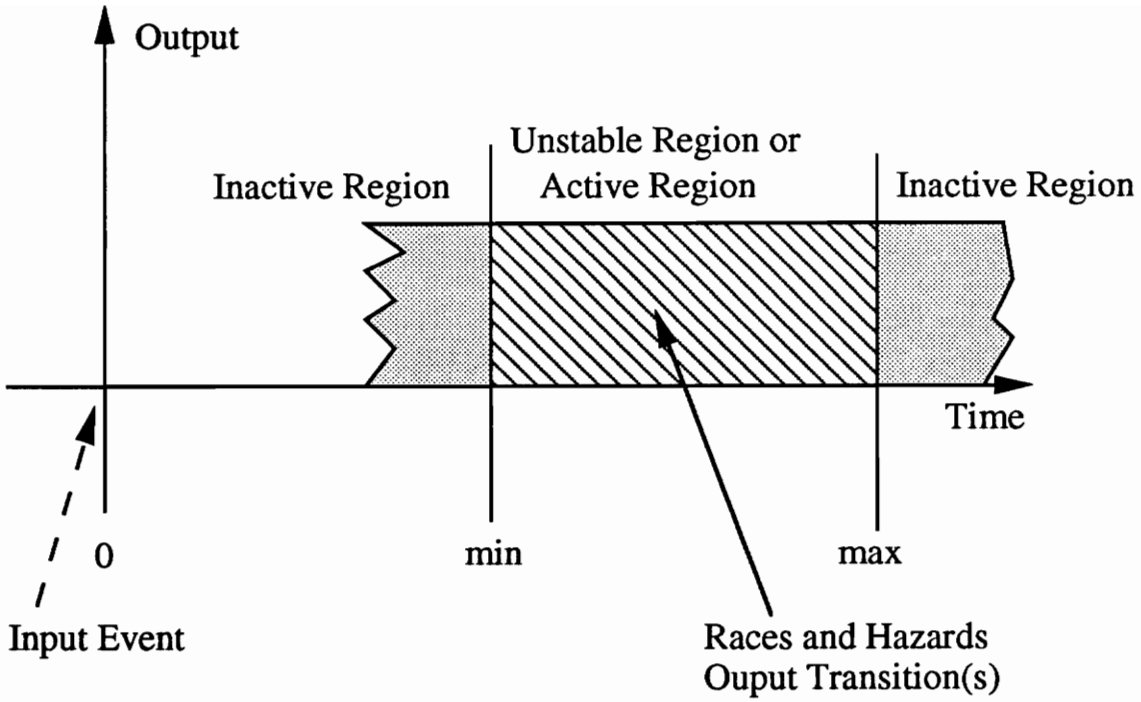


Figure 13. Timing Relationship Between Input and Output Events

Figure 13 shows the general timing waveform for the input and output timing relationship. Assume the input has an event at time unit 0. There are two regions known as the *inactive region*. One region between the time unit 0 and the min. The other region is from the max and beyond. In these regions, there is no output event due to the input event at time unit 0. The region between the min and max is called the *active region*. That is, the output may have an event(s) inside the region due to the input event at time unit 0. Inside this region, either a normal output transition or an erroneous hazard spike will occur.

3.3.2 VHDL Timing Delay Models

VHDL has three different timing delay models. They are the *delta (unit) delay model, the inertial delay model, and the transport (pure) delay model* **[5, 6, 7]**.

The delta delay model corresponds to a physical system that executes in an infinitely small, but non-zero time. In VHDL, each unit delta corresponds to the time from one simulation cycle to another simulation cycle in which time does not advance. That is, delta delay is usually used to model a sequence of events without considering any actual timing behavior. Delta inertial delay and transport inertial delay are equivalent.

The inertial delay model is the default timing model in VHDL because it is commonly used in modeling digital circuits and switching circuits. Sometimes, a logic system does not always respond instantaneously to the input stimulus, but rather requires that an input signal contains a minimum amount of energy before the output signal begins to react. For electrical signals, this minimum energy requirement refers to a minimum pulse width requirement. This energy represents the inertial delay requirement. Hence, if the minimum energy is not met, the output

signal will be suppressed. This behavior is quite similar to the real logic design world. Hence, in the inertial delay model, if an input does not persist at a given amount of energy for a specified amount of time, the output will not have any effect. In other words, the inertial delay model is a "*spike-suppression*" timing delay model.

The transport delay model is not a spike-suppression delay model. Instead, transport delay is a "*pulse transmitter*" delay model. Whenever an input changes, the output changes after a pre-defined propagation delay, regardless of the minimum energy and delay requirement. A simple transmission line behaves like a transport delay. Therefore, transport delay is usually used for transmission line modeling and for accurate path delay analysis in the logic devices. Because it passes all pulses regardless of duration, transport delay is the most appropriate delay model for analyzing circuits with hazards and in illustrating the hazard spike.

3.3.3 VHDS Nine-Valued Timing Package

VHDS has a built-in timing package. The filename of this package is called *time1_test.vhd.* The package identifier is called TIME_PACKAGE. This timing package is used to describe the accurate timing information of a logic circuit and to detect timing hazards in a logic circuit. This package was developed based on the nine-valued MVL as mentioned above. Hence, it is fully compatible and consistent with the VHDS MVL package. The primary unit of time in the VHDS is "ns". The timing package is divided into three parts. Part one is the nine-valued propagation delays model scheme. For example, the package supports three different delay parameters which are the binary delays, the tri-state delays and the wire (or interconnection) delay. Part two contains a function how to compute the minimum and maximum propagation delays of a VHDL model from the subprograms in the timing package. Part three contains the miscellaneous type

conversion functions and the spike detection routine to support hazard detection. Each of them will be described in detail later.

Before the detailed discussion of the nine-valued timing package, several important type declarations are first introduced. Record type PROP_DELAY_SPEC is defined as follows:

```
type PROP_DELAY_SPEC is
record
MIN_PROP_DELAY: TIME;          -- minimum propagation delay
MAX_PROP_DELAY: TIME;          -- maximum propagation delay
end record;
```

This record type consists of two fields. Both of them are of type TIME. The first field, MIN_PROP_DELAY, represents the minimum propagation delay of a VHDL model. The second field, MAX_PROP_DELAY, represents the maximum propagation delay of a VHDL model. The record type PROP_DELAY_SPEC provides a more convenient way to compute the min and max of a VHDL model.

The vectorized array of the above record type is defined below:

```
type PROP_DELAY_SPEC_VECTOR is array (POSITIVE RANGE <>) of PROP_DELAY_SPEC;
```

The enumeration type of delay models selection is defined as follows:

```
type TIME_MODEL_CHOICE is (INERTIAL_DELAY, TRANSPORT_DELAY, DELTA_DELAY).
```

The vectorized array of type TIME is defined as follows:

```
type TIME_VECTOR is array (NATURAL RANGE <>) of TIME;
```

The vectorized array of type NATURAL is defined as follows:

```
type NATURAL_VECTOR is array (NATURAL RANGE <>) of NATURAL;
```

3.3.3.1 Basic VHDL Delay Modeling

All logic circuits have propagation delays. In fact, propagation delay is closely related to real logic circuit modeling. In VHDL, there are two basic propagation delay schemes. They are the *fixed delay* scheme and the *variable delay* scheme. The fixed delay scheme propagates an input event to the output by a fixed pre-defined delay. This is shown below:

```
OUTPUT <= [transport] INPUT after  X ns;
```

The fixed delay is the most basic one. It provides the basic signal propagation delay through the generic value X. It is useful for timing verification, but it does not accurately represent the real behavior of a circuit.

In the variable delay scheme, gate delays are specified in a more sophisticated way. When refering to any manufacturer's data book, a logic device usually has the rise propagation delay (TP_01) and the fall propagation delay (TP_10). This is shown below:

```
if (OLD_OUTPUT = '0' and NEW_OUTPUT = '1') then
DELAYED_OUTPUT <= [transport] NEW_OUTPUT after TP_01;
elsif (OLD_OUTPUT = '1' and NEW_OUTPUT = '0') then
DELAYED_OUTPUT <= [transport] NEW_OUTPUT after TP_10;
end if;
```

The variable delay scheme more closely represents the actual behavior of a circuit. Therefore, it is especially useful for accurate gate level modeling.

3.3.3.2 VHDS Timing Delays Model

The VHDS nine-valued timing package has a subprogram called TIME_MODEL_SELECTION_1. This subprogram employs a delay scheme which is similar to the variable delay scheme. The derivation of the rise and fall delays in the VHDS are based on the diagram shown in Figure 14.

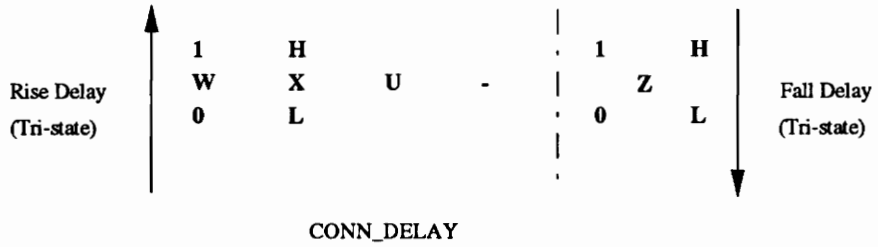


Figure 14. VHDS Nine-Valued Rise/Fall Delays Model

The delay scheme above was adopted from the Synopsys seven-valued timing design **[16].** From the diagram above, logic values '1' and 'H' are defined as logical high. Logic values '0' and 'L' are defined as logical low. Logic values 'X', 'U', '-', and 'Z' are defined as logical unknown and are in between logical high and logical low. Logic value 'Z' is the high-impedance state and has its own propagation delay relationship with logical high and logical low. A rise propagation delay (TP_01) occurs when there is an output transition from '0' or 'L' to '1' or 'H'. Similarly, a TP_01 also occurs when there is an output transition from 'X', 'U', and '-' to '1' or 'H'. A fall propagation

delay occurs when the direction of the above transition is reversed.

Depending on circuit technology, such as MOS device modeling, the propagation delay from logic value '1' to 'Z' (TP_1Z) may be longer than the propagation delay from '1' to '0'. Hence, there is another delay called the tri-state fall delay for this output transition. This is also true for TP_Z0. Similarly, the tri-state rise delays (TP_Z1 and TP_0Z) describe the output transition from logic values 'Z' to '1' and '0' to 'Z', respectively.

Finally, there is the wire delay (CONN_DELAY). In real circuit behavior, propagation delay sometimes exists due to interconnection of logic devices using metal wire. Usually the wire delay is present in large circuits. The default values of TP_1Z, TP_Z1, TP_0Z, TP_Z0, and CONN_DELAY are 0 ns.

The delay parameters (basic time set) in the VHDS timing package are summarized as follows:

- TP_01: Propagation delay for an output to have a low to high transition.
- TP_10: Propagation delay for an output to have a high to low transition.
- TP_1Z: Propagation delay for an output to have a high to high-impedance transition.
- TP_Z1: Propagation delay for an output to have a high-impedance to high transition.
- TP_0Z: Propagation delay for an output to have a low to high-impedance transition.
- TP_Z0: Propagation delay for an output to have a high-impedance to low transition.
- CONN_DELAY: Wire delay or interconnection delay between logic devices.

All the parameters above form the basic delay unit in the VHDS timing package. Depending on the kind of technology employed, TP_10 may not be equal to TP_01. For example, in TTL technology, TP_01 may be equal to TP_10. However, in CMOS technology, TP_10 is

approximately equal to three times the TP_01. Also, in tri-state logic modeling, TP_01 is not equal to TP_0Z or TP_Z1, and TP_10 is not equal to TP_Z0 or TP_1Z.

The procedure specification for the TIME_MODEL_SELECTION_1 is defined as follows:

```
procedure TIME_MODEL_SELECTION_1
(OUTPUT_SIG: in MVL9; signal OUTPUT: out MVL9; CHOICE: TIME_MODEL_CHOICE;
LAST_DELAY_TIME: inout TIME; OUTPUT_SIG_LAST_VALUE: inout MVL9;
signal OUTPUT_LAST_VALUE: inout MVL9;
TP_01, TP_10, TP_1Z, TP_Z1, TP_0Z, TP_Z0, CONN_DELAY: TIME);
```

The input variable OUTPUT_SIG is the undelayed output logic value from the behavioral part of the model primitive. More details can be found in section 3.6.1. The signal OUTPUT is the output signal as controlled by the VHDS nine-valued rise and fall delay model. The input variable CHOICE selects between inertial delay, transport delay, and delta delay. The variable LAST_DELAY_TIME memorizes the most recent gate delay value of the nine-valued rise and fall delays. The signal OUTPUT_LAST_VALUE memorizes the most recent output value (OUTPUT) in the procedure. The variable OUTPUT_SIG_LAST_VALUE memorizes the most recent input variable (OUTPUT_SIG) in the procedure. The timing values TP_01, TP_10, TP_1Z, TP_Z1, TP_0Z, TP_Z0, and CONN_DELAY are the nine-valued generic rise and fall delays.

TIME_MODEL_SELECTION_1 provides an option for the designer to choose the timing delay model generically. Three options are allowed in the TIME_MODEL_SELECTION_1. They are (1) the inertial delay, (2) the transport delay, and (3) the delta delay. All of them are defined in the enumeration type TIME_MODEL_CHOICE. With this selection, the designer can easily switch from one delay model to the other delay model without changing the descriptions of the original primitive models. One precaution of using such delay model selection is to avoid

inconsistent delay model selection. That is, if a VHDL description involves multiple copies of the same components, then all copies of the component description should follow the same timing delay model selection. This restriction avoids inaccurate timing behavior and improper simulation results for a logic circuit.

3.3.4 Minimum and Maximum Propagation Delays Computation

Minimum propagation delay (min) and maximum propagation delay (max) are common parameters in describing circuit delays. As mentioned before, the existence of hazard in a logic device are closely related to circuit timing. Hence, min and max become the important parameters to find hazards in a logic circuit. However, how to compute the min and max of any model that is described structurally in VHDL is a problem.

There are six functions (MIN_TIME, MAX_TIME, MIN_TIME_ALL, MAX_TIME_ALL, MAX_MIN_DELAY_COMPUTE, and TIME_BUS_FUN) in the TIME_PACKAGE which are designated to compute the min and max of a VHDL structural model. The function specifications are listed below. By specifying the delay parameters (the normal rise and fall delay, and optionally, the interconnection delay, and the tri-state rise and fall delay) of each component in the VHDL model, the functions will then automatically compute the min and max for this VHDL model. This approach works with any size VHDL structural model.

```
function MIN_TIME(TP_01, TP_10, TP_1Z, TP_Z1, TP_0Z, TP_Z0: TIME;
                  TRI_STATE_OPTION: STRING := "OFF") return TIME;

function MAX_TIME(TP_01, TP_10, TP_1Z, TP_Z1, TP_0Z, TP_Z0: TIME;
                  TRI_STATE_OPTION: STRING := "OFF") return TIME;
```

```
function MIN_TIME_ALL(INPUT_TIME_VECTOR: TIME_VECTOR) return TIME;

function MAX_TIME_ALL(INPUT_TIME_VECTOR: TIME_VECTOR) return TIME;

function MAX_MIN_DELAY_COMPUTE(TP_01, TP_10, TP_1Z, TP_Z1,
                               TP_0Z, TP_Z0, CONN_DELAY: TIME;
                               OPTION: STRING := "OFF";
                               TIME_SPEC_IN: PROP_DELAY_SPEC :=
                               (0 ns, 0 ns)) return PROP_DELAY_SPEC;

function TIME_BUS_FUN(MAX_MIN_TIME_SET: PROP_DELAY_SPEC_VECTOR)
                      return PROP_DELAY_SPEC;
```

The function MIN_TIME computes the minimum time value of the delay parameters (TP_01, TP_10, TP_1Z, TP_Z1, TP_0Z, and TP_Z0). If the TRI_STATE_OPTION specified in the function is "OFF", then the tri-state rise and fall delay parameters (TP_1Z, TP_Z1, TP_0Z, and TP_Z0) will not be considered. Hence, the function will only compute the minimum value of TP_01 and TP_10. If the TRI_STATE_OPTION is "ON", then all the delay parameters will be considered. The default value of the TRI_STATE_OPTION is "OFF".

The function MAX_TIME computes the maximum time value of the delay parameters (TP_01, TP_10, TP_1Z, TP_Z1, TP_0Z, and TP_Z0). If the TRI_STATE_OPTION specified in the function is "OFF", then the tri-state rise and fall delay parameters (TP_1Z, TP_Z1, TP_0Z, and TP_Z0) will not be considered. Hence, the function will only compute the maximum value of TP_01 and TP_10. If the TRI_STATE_OPTION is "ON", then all the delay parameters will be considered. The default value of the TRI_STATE_OPTION is "OFF".

The function MIN_TIME_ALL computes the most minimum time value (TIME) from a vectorized timing value (TIME_VECTOR). Similarly, the function MAX_TIME_ALL computes the most maximum time value (TIME) from a vectorized timing value (TIME_VECTOR).

The function MAX_MIN_DELAY_COMPUTE updates the min and max for each component when an input timing event occurs at the beginning of the simulation (0 ns). The function TIME_BUS_FUN finds the most min and max timing values when multiple mins and maxs exist in a component.

3.3.4.1 Minimum and Maximum Propagation Delays Calculation in a VHDL Structural Model

The minimum and maximum propagation delays of a VHDL model are determined by using the "*individual component self-updating technique.*" That is, each component in the VHDL structural model is self-updated whenever it has an input timing event at the beginning of the simulation (0 ns). Hence, the determination of the min and max values is analogous to the determination of the output value in a VHDL model. Figure 15 illustrates how the technique works.

In Figure 15, a VHDL model called TEST_DEVICE is shown. Inside the TEST_DEVICE, component A and component B are connected to component C. Each component is given its own TP_01 and TP_10. In order to find the min and the max from the input port (input_1 or input_2) to the output port (output_logic) in the TEST_DEVICE, a special procedure, described below, is required.

First, two kinds of signal declarations are required in each component. One kind of signal declaration is the usual input and output logic signal declaration. The second kind of signal declaration is the new timing input and output signal declaration. For example, the component A declaration is shown below. The logic signals are of type MVL9 and the timing signals are of type PROP_DELAY_SPEC.

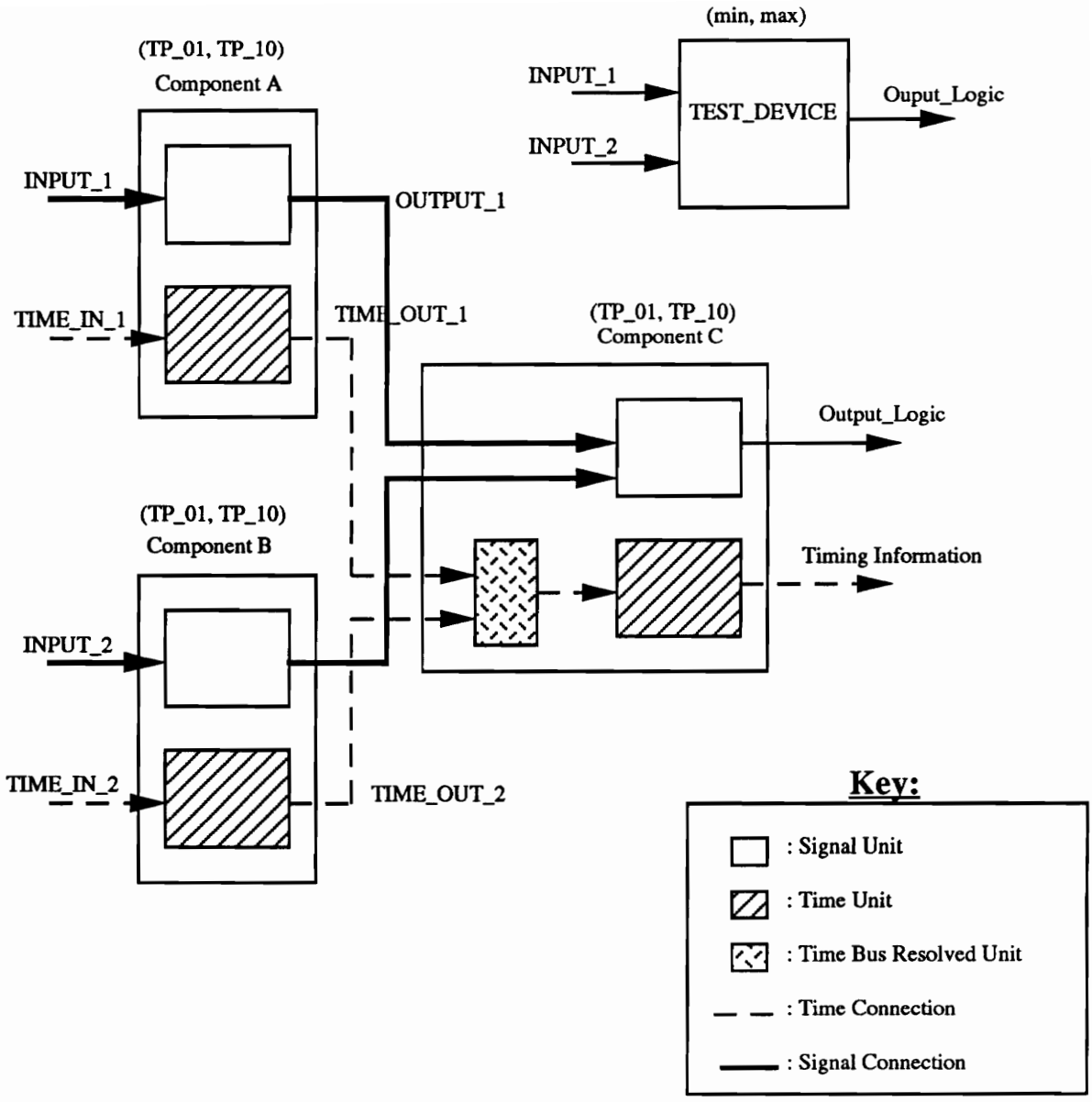


Figure 15. Min and Max Propagation Delays Computation in VHDL Structural Model

```
component MODEL              -- component A
generic(TP_01, TP_10, DELAY_MODEL);
port(INPUT: in MVL9; OUTPUT: out MVL9;  -- PROP_DELAY_SPEC: (MIN, MAX)
     TIME_IN: in PROP_DELAY_SPEC; TIME_OUT: out PROP_DELAY_SPEC);
```

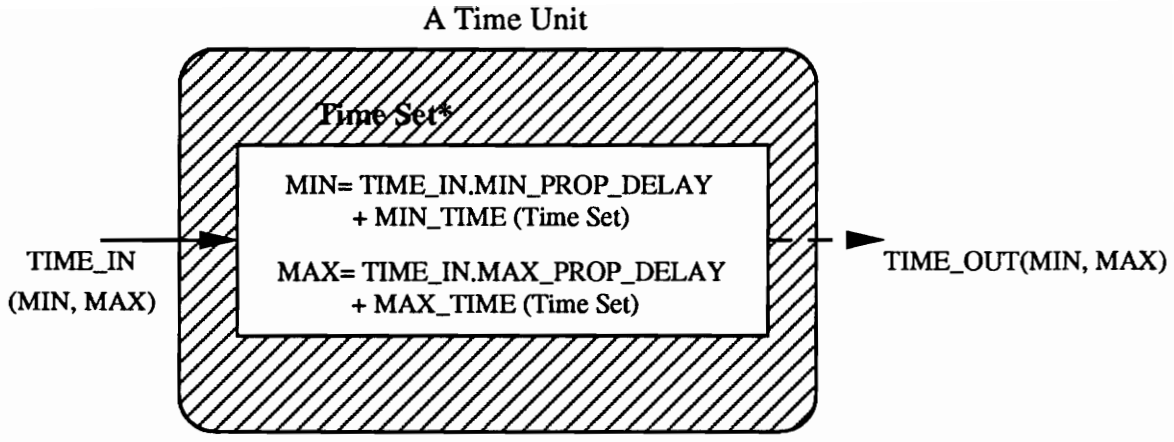
The input timing signal TIME_IN transfers the timing information into the component. The output timing signal TIME_OUT transfers the timing information out of the component.

The component instantiation statement of component A is shown as follows:

```
A: MODEL
generic map(5 ns, 5 ns, TRANSPORT_DELAY)
port map(INPUT => INPUT_1, OUTPUT => OUTPUT_1,
         TIME_IN => TIME_IN_1, TIME_OUT => TIME_OUT_1);
```

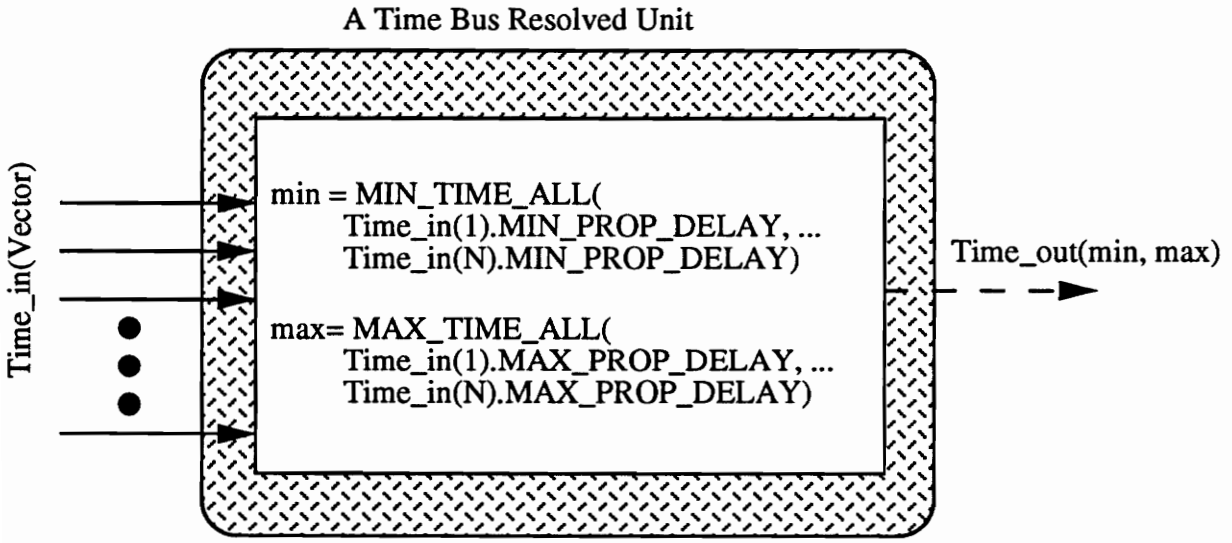
Logic signals are interconnected from one component to another component to form the VHDL structural model. In the same manner, timing signals are also interconnected between the components. This is shown in Figure 15 on page 40. Hence, two kinds of connection netlists exist in the TEST_DEVICE. They are the logic signals netlist and the time signals netlist. Both netlists are connected in the same configuration except one is the logic connection and the other is the time connection. So, each component has two units: the *logic unit* and the *time unit*. The logic unit updates the logic value of the component with the signals INPUT and OUTPUT while the time unit updates the timing values of the component with the signals TIME_IN and TIME_OUT.

The internal structure of the time unit is shown in Figure 16a. The time unit is basically an operator which updates the new timing output signals. This is analogous to the boolean operator in the logic signals. The function MAX_MIN_DELAY_COMPUTE represents the time unit. The two subroutines called by the time unit are the min calculation subroutine and the max



Time Set:= {(TP_01, TP_10, TP_0Z, TP_Z0, TP_1Z, TP_Z1), and CONN_DELAY}
Default: Time Set = {TP_01, TP_10}
MIN_TIME (Time Set) = MIN_TIME (TP_01, TP_10) and
MAX_TIME (Time Set) = MAX_TIME (TP_01, TP_10).

(a) Internal Structure of a Time Unit



(b) Internal Structure of a Time Bus Resolved Unit

Figure 16. Overview of a Time Unit and a Time Bus Resolved Unit

calculation subroutine. The min calculation subroutine will determine the minimum delay of the component. The max calculation subroutine will determine the maximum delay of the component. Whenever there is an input timing event, such as TIME_IN, the time unit will first split the TIME_IN into two values, the TIME_IN.MIN_PROP_DELAY and the TIME_IN.MAX_PROP_DELAY. The minimum value can be updated by adding the new TIME_IN.MIN_PROP_DELAY to the minimum value of the basic time set. Similarly, the maximum value can be updated by adding the new TIME_IN.MAX_PROP_DELAY to the maximum value of the basic time set. The minimum value of the basic time set is computed by the function MIN_TIME. The maximum value of the basic time set is computed by the function MAX_TIME. The new min and max values are then combined and become the new TIME_OUT. The new value is then either propagated to the next component or reported to the output results. Since each component in the VHDL structural model is updated by the above method, the whole VHDL model is always updated until the final min and max values are determined. This updating process takes place at the beginning of the simulation.

3.3.4.2 Multiple Timing Resolution in Minimum and Maximum Delays Computation

The time unit described above is a single-input, single-output min and max computation unit. Most frequently, there may be more than one input timing signal on the time unit or a component. The situation is similar to component C in Figure 15. In this case, a function which is similar to the bus resolution function is needed to resolve the multiple timing drivers. This bus resolution function is called the "*time bus resolved unit*" and is used to resolve multiple drivers in the *time domain*. The time bus resolved unit resolves multiple timing drivers into a single timing signal and propagates it to the corresponding connected time unit. A time bus resolved unit is

shown in Figure 16b. The function TIME_BUS_FUN in the TIME_PACKAGE represents the time bus resolved unit.

When multiple timing drivers (TIME_IN(VECTOR)) occur, the time bus resolved unit will split all the multiple drivers into two sets. One set consists of all the minimum values of TIME_IN(VECTOR), i.e. the set of (TIME_IN(1).MIN_PROP_DELAY, TIME_IN(2).MIN_PROP_DELAY, ... , and TIME_IN(N).MIN_PROP_DELAY). The other set consists of all the maximum values of TIME_IN(VECTOR), i.e. the set of (TIME_IN(1).MAX_PROP_DELAY, TIME_IN(2).MAX_PROP_DELAY, ..., and TIME_IN(N).MAX_PROP_DELAY)). The resolved unit will then find the *most* minimum value in the set of TIME_IN(VECTOR).MIN_PROP_DELAY by the function MIN_TIME_ALL. The resolved unit also finds the *most* maximum value in the set of TIME_IN(VECTOR).MAX_PROP_DELAY by the function MAX_TIME_ALL. These two values will combine and become the resolved TIME_OUT in which it contains the most min and max values. The timing signal TIME_OUT is either propagated to the attached time unit or reported to the output results. With the time unit design and the time bus resolved unit, the minimum and the maximum propagation delays of any circuit can be computed.

3.3.5 Type Conversion Functions in the VHDS Timing Package

Inside the TIME_PACKAGE, there are several subprograms that perform type conversion. The type conversion functions apply between the data types TIME, NATURAL, STRING (CHARACTER), and REAL. Those conversion functions are useful in hazard analysis. By first converting the timing information of a hazard into type NATURAL; and then converting the

natural value into STRING, an assertion message can be displayed to alert the designer. An example of a hazard assertion statement is shown below.

```
assert condition
report HT & "Function 0 hazard is detected on " & SIGNAL_NAME & " at "
            NATURAL_TO_STRING(TIME_TO_NATURAL(HAZARD_TIME)) & " ns"
-- Assume the value of HAZARD_TIME = 1000 ns
severity SEVERITY_LEVEL;
Example:    Function 0 hazard is detected on OUTPUT at    1000 ns
```

The functions TIME_TO_NATURAL convert a time value (TIME) in "ns", or vectorized time values (TIME_VECTOR) in "ns" into a natural value (NATURAL), or vectorized natural values (NATURAL_VECTOR), respectively. The functions NATURAL_TO_TIME convert a natural value (NATURAL), or vectorized natural values (NATURAL_VECTOR) into a time value (TIME) in "ns", or vectorized time values (TIME_VECTOR) in "ns", respectively. The functions TIME_TO_NATURAL perform the opposite operation from the functions NATURAL_TO_TIME. The function NATURAL_TO_STRING converts a non-negative integer range from 0 to 1,000,000,000 to type STRING (or CHARACTER). The function NATURAL_TO_REAL converts any natural value (NATURAL) to a floating value (REAL).

3.3.6 Spike Detection in a VHDL Component and its Implicit Use

The last procedure in the TIME_PACKAGE is called SPIKE_DETECTION. The procedure SPIKE_DETECTION is used to find spikes in a component. Therefore, the procedure call is placed in the primitive model description of the component. If a spike is found in the output, the procedure will flag an assertion message to the designer. This is especially useful when the test circuit is multi-level (i.e. level > 3). This is illustrated in Figure 17. When a spike is found in a multi-level component (e.g. component F), the spike message will print and the propagation of

the spike will produce a series of spike assertion messages from one component to the other components (F to J to L). From the flow of the assertion messages in the model, the starting location of the hazard and the flow of the hazard may be deduced.

The procedure specification of the SPIKE_DETECTION is shown below:

```
procedure SPIKE_DETECTION(signal OUTPUT_SIGNAL: in MVL9;
                          OUTPUT_LAST_EVENT: inout TIME;
                          GLITCH_0_TIME, GLITCH_1_TIME: TIME);
```

The input MVL9 signal OUPUT_SIGNAL represents the output signal which is under spike detection. The timing value OUTPUT_LAST_EVENT memorizes the time of the last output event. The generic values GLITCH_0_TIME and GLITCH_1_TIME specifies the output pulse width margin for the 0-spike and the 1-spike, respectively. If the time duration of the output pulse is less than or equal to the GLITCH_0_TIME or GLITCH_1_TIME, the spike detection assertion mesage will be printed.

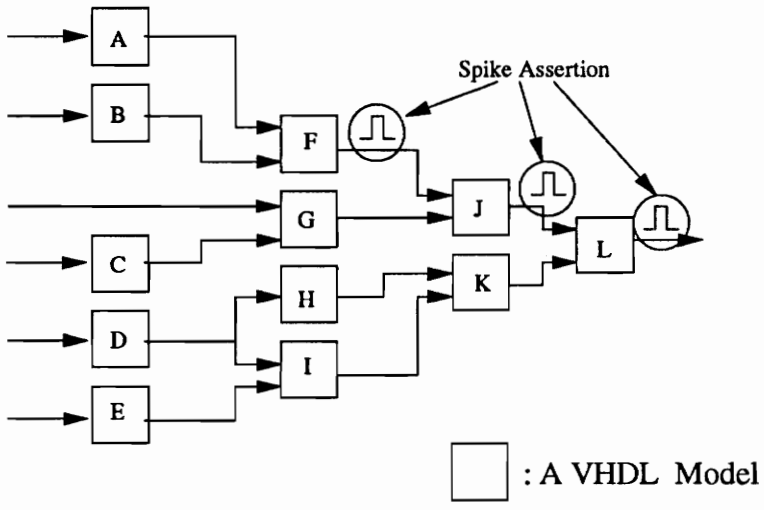


Figure 17. Spike Detection in a VHDL Structural Model

3.4 VHDS Hazard Pattern Generator (VHDS HPG)

The VHDS Hazard Pattern Generator is a package which produces all the possible bit vector patterns for common hazard detection. It also provides bit pattern injectors that inject all the generated patterns to the test circuit. The HPG was developed using the nine-valued logic system. The HPG has three major functions. They are the single-input change bit pattern generator, and the two-input change bit pattern generator. The single-input change bit pattern generator is used for detection of single-input change hazards in a logic circuit, that is the static hazard and the dynamic hazard. The two-input change bit pattern generator is used for detection of the two-input change hazard in a logic circuit, that is the function hazard. With the capability of HPG, all the static, dynamic, and function hazards are automatically detected in any logic circuit by the HPG without any user-supplied input test vector combinations. Therefore, the designer knows all the hazard information immediately by interfacing the test circuit with the VHDS. The filename for the HPG is called *tpgred.vhd*. The package identifier of the HPG is called TEST_GENERATOR.

This section will discuss the theory and the design of the generators in detail. It starts with a discussion on the single-input pattern generator. Next, the two-input pattern generator will be presented. Finally, other support features in the HPG will also be introduced.

The number of inputs specified by the designer is a constant called WORDLENGTH where **WORDLENGTH ≥ 2**. WORDLENGTH is a keyword in VHDS and is only allowed to define the number of inputs of the test circuit. The constant value WORDLENGTH will be used throughout the VHDS.

Before the detailed discussion of the HPG, several important type declarations are first introduced. Record type MVL9_VECTOR_TUPLE is defined as follows:

```
type MVL9_VECTOR_TUPLE is
record
        ORG_CELL: MVL9_VECTOR(WORDLENGTH - 1 downto 0); -- original pattern
        NEG_CELL: MVL9_VECTOR(WORDLENGTH - 1 downto 0); -- neighbor pattern
end record;
```

This record type describes the single-input change patterns. It consists of two fields. The first field ORG_CELL describes the starting bit pattern. The second field NEG_CELL describes the final bit pattern.

Record type MVL9_VECTOR_TRIPLE is defined as follows:

```
type MVL9_VECTOR_TRIPLE is
record
ORG_CELL: MVL9_VECTOR(WORDLENGTH - 1 downto 0); -- original pattern
NEG_CELL: MVL9_VECTOR(WORDLENGTH - 1 downto 0); -- 1st neighbor pattern
NEG_NEG_CELL: MVL9_VECTOR(WORDLENGTH - 1 downto 0); -- 2nd neighbor pat.
end record;
```

This record type describes the two-input change patterns. It consists of three fields. The first field ORG_CELL describes the starting bit pattern. The second field NEG_CELL describes the intermediate bit pattern. The third field NEG_NEG_CELL describes the final bit pattern. The intermediate bit pattern differs from the starting bit pattern by one bit. The final bit pattern differs from the intermediate bit pattern by one bit also. Therefore, the final bit pattern differs from the starting bit pattern by two bits. The starting bit patterns and the final bit patterns are used to describe a pair of input combinations with two inputs that change simultaneously.

The vectorized array of the MVL9_VECTOR_TUPLE is defined as follows:

```
type MVL9_VECTOR_TUPLE_VECTOR is array (NATURAL range <>) of
MVL9_VECTOR_TUPLE;
```

The vectorized array of the MVL9_VECTOR_TRIPLE is defined as follows:

```
type MVL9_VECTOR_TRIPLE_VECTOR is array (NATURAL range <>) of
MVL9_VECTOR_TRIPLE;
```

Record type MVL9_STATE_LOGIC is defined as follows:

```
type MVL9_STATE_LOGIC is
record
        STATE:              MVL9_VECTOR(WORDLENGTH -1 downto 0);
        LOGIC_VALUE:        MVL9;
end record;
```

The above record type describes the input logic value of the circuit and its corresponding output logic value. It consists of two fields. The first field STATE describes the present input logic value of the circuit. The second field LOGIC_VALUE describes the corresponding stable output logic values associated with the input logic value in the first field.

The vectorized array of the MVL9_STATE_LOGIC is defined as follows:

```
type MVL9_STATE_LOGIC_VECTOR is array (NATURAL range <>) of MVL9_STATE_LOGIC;
```

Record type STATIC_INFO is defined as follows:

```
type STATIC_INFO is
record
        STATE_PATT1:            MVL9_VECTOR(WORDLENGTH - 1 downto 0);
        STATE_PATT2:            MVL9_VECTOR(WORDLENGTH - 1 downto 0);
        STATIC_EVENT_TIME:      TIME;
        PULSE_DURATION:         TIME;
end record;
```

The above record type describes the timing and logic information of the static hazard. It consists of four fields. The first field STATE_PATT1 describes the starting input combination that causes the static hazard. The second field STATE_PATT2 describes the corresponding final input combination that causes the static hazard. The third field STATIC_EVENT_TIME describes the amount of time between the event of the final input combination and the first output event that is caused by the final input combination. The fourth field PULSE_DURATION describes the pulse width of the static hazard.

The vectorized array of the STATIC_INFO is defines as follows:

```
type STATIC_INFO_VECTOR is array (NATURAL range <>) of STATIC_INFO;
```

The enumeration type of the simulation test mode is defined as follows:

```
type TEST_ORDER is (STATIC, FUNCT);
```

Two test modes are allowed in the VHDS simulation. They are the single-input change hazard detection (STATIC), and the two-input change hazard detection (FUNCT).

3.4.1 Generation of Single-Input Change Patterns

The single-input pattern generator produces all the possible bit patterns for single-input change hazards (i.e., the static hazard and the dynamic hazard). The generator only needs the number of inputs of the test circuit. Given this value, the generator will generate all the possible bit patterns for this test circuit. Figure 18 shows the flow chart of the single-input patterns generator. The generator first accepts an input-specification file. The input-specification file is a VHDL file which is specified by the designer. The file is used to specify the number of inputs of the test circuit. The filename is called "*datax.vhd*" where ***x*** is usually the number of inputs specified for the test circuit. The content of the input-specified file is described below.

```
use WORK.all;
package INPUT_DATA is

-- Please enter the number of inputs for the test circuit as below:

constant WORDLENGTH: NATURAL := user_specified_number_of_inputs;
end INPUT_DATA;
```

From the constant value WORDLENGTH, the generator produces a range of naturals from {0, 1, 2, 3, ..., $2^{WORDLENGTH} - 1$} using the function called NATURAL_GENERATION in the package TEST_GENERATOR. Each natural in the range is then converted into its nine-valued bit vector pattern. The function NATURAL_TO_BIT_PATTERN implements the whole process of converting the constant value WORDLENGTH into the range of the nine-valued bit vector patterns. Each bit vector pattern in the range is called the "*original bit vector pattern.*" The function STATIC_NEIGHBOR uses each original bit vector pattern and looks for all the possible bit vector patterns that differ by one bit from the original bit vector pattern. Each of the

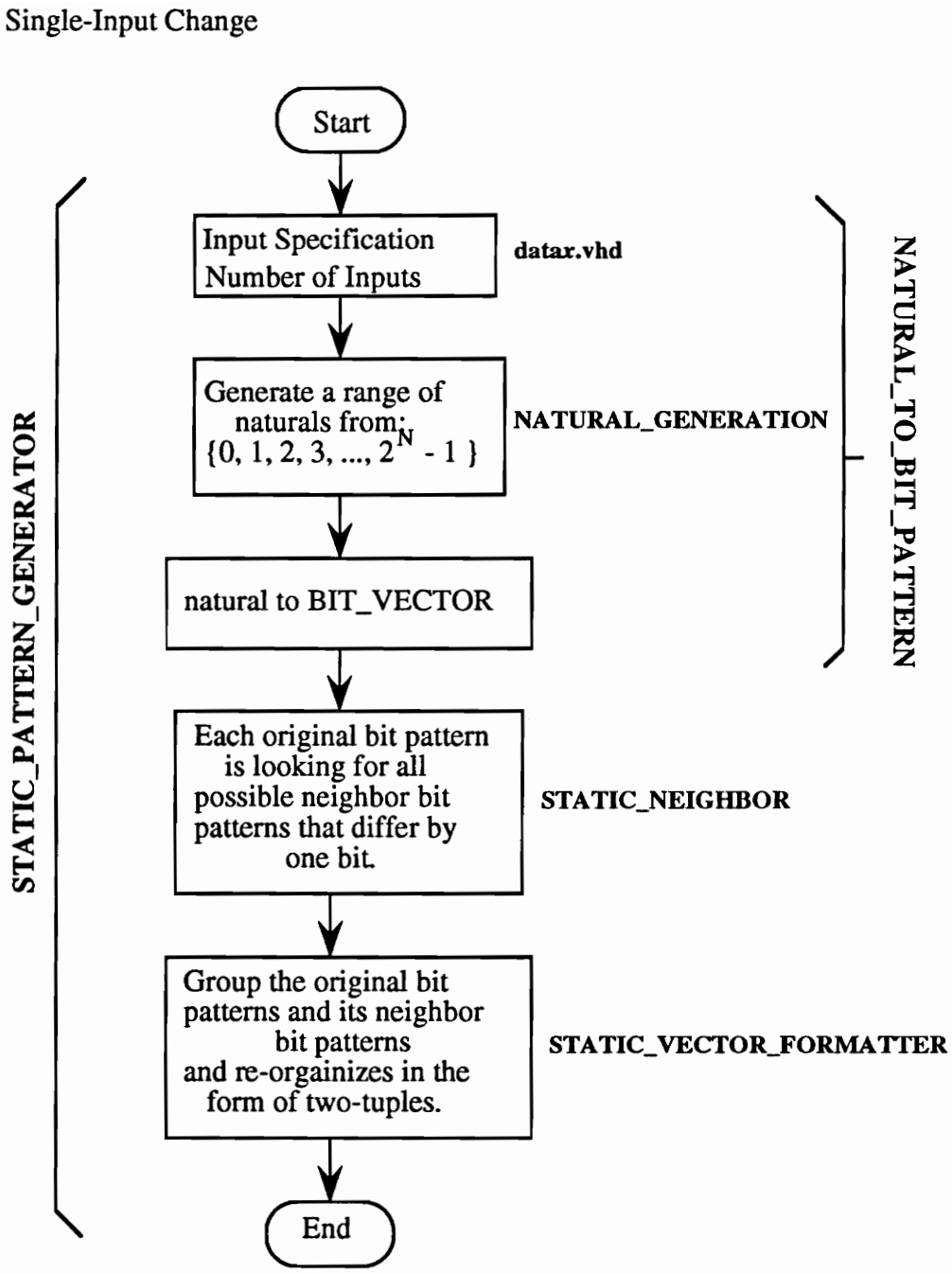


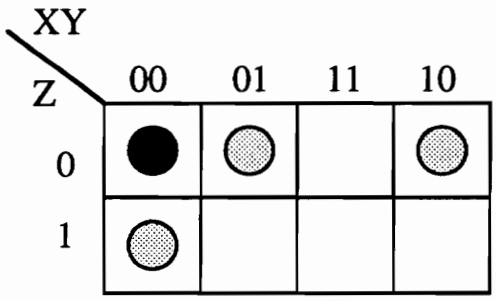
Figure 18. Flow Chart of Hazard Pattern Generator

possible bit vector patterns found is called the "*neighbor bit vector pattern.*" Figure 19 illustrates how the function STATIC_NEIGHBOR works.

Figure 19a shows a three-variable Karnaugh Map. The dark dot in the cell location "000" represents an original bit vector pattern. The shaded dots represent all the possible neighbor bit vector patterns that differ by one bit with respect to the bit vector "000". In this case, three neighbor bit vector patterns are found. They are "001", "100", and "010". Since each original bit vector pattern has three neighbor bit vector patterns found, the total number of bit vector test patterns for this case is equal to 1 * 3 = 3. For the three-variable case, there are eight original bit vector patterns. Thus, the total number of bit vector test patterns for the three-variable case = 8 * 3 = 24. Similarly, Figure 19b shows a four-variable Karnaugh Map. The dark dot in the cell location "0000" represents an original bit vector pattern. The shaded dots represent all the possible neighbor bit vector patterns that differ by one bit with respect to the bit vector "0000". In this case, four neighbor bit vector patterns are found. They are "0001", "0100", "1000", and "0010". Since each original bit vector pattern has found four neighbor bit vector patterns, the total number of bit vector test patterns for this case becomes 1 * 4 = 4. For the four-variable case, there are sixteen original bit vector patterns. Thus, the total number of bit vector test patterns for the four-variable case = 16 * 4 = 64. **In general, the total number of bit vector test patterns required for N inputs is equal to (Number of Original Bit Vector Patterns) * (Number of Neighbor Bit Vector Patterns) = $2^{WORDLENGTH}$ * WORDLENGTH.**

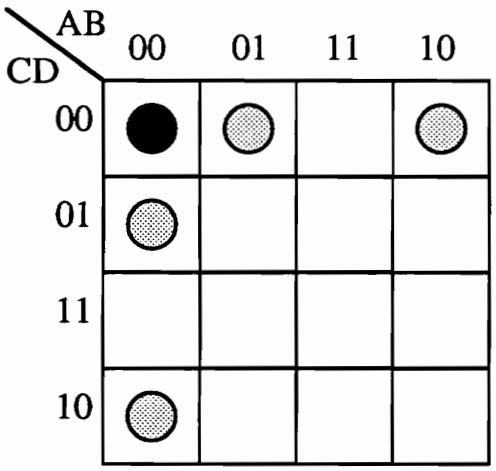
After all the original patterns and the neighbor patterns are found by the function STATIC_NEIGHBOR, the function STATIC_VECTOR_FORMATTER reformats the original bit vector patterns and the neighbor bit vector patterns in the form of *two-tuples*. Each of

Single-Input Change
(a) 3-variable case:



Each cell has three neighbor cells which differs by one bit. So, the total number of test pattern = 8 x 3 = 24

(b) 4-variable case:



Each cell has four neighbor cells which differs by one bit. So, the total number of test pattern = 16 x 4 = 64

In general, the total number of patterns required for N inputs = $2^N * N$

Figure 19. Illustration of the Function STATIC_NEIGHBOR

the two-tuples (ORIGINAL_PATTERN, NEIGHBOR_PATTERN) becomes a single-input change pattern for the test circuit.

Figure 20 describes an example of how the single-input pattern generator works. The value of WORDLENGTH is equal to 3. The range of naturals becomes {0, 1, 2, 3, 4, 5, 6, 7}. From the function NATURAL_TO_BIT_PATTERN, the original bit vector patterns set is {"000", "001", "010", "011", "100", "101", "110", "111"}. From the function STATIC_NEIGHBOR, the set of original bit vector patterns and the neighbor bit vector patterns are shown in figure 20a. After reformating by the function STATIC_VECTOR_FORMATTER, the final two-tuples bit vector patterns are shown in Figure 20b.

The function STATIC_PATTERN_GENERATOR implements the whole process of transforming the given constant input value WORDLENGTH to the desired set of two-tuples.

The specification of the function STATIC_PATTERN_GENERATOR is defined as follows:

```
function STATIC_PATTERN_GENERATOR(INPUT: NATURAL)  -- INPUT = WORDLENGTH
return MVL9_VECTOR_TUPLE_VECTOR;                   -- return the desired set of two-tuples.
```

The next step is to provide a mechanism to inject the two-tuples into the test circuit. Three subprograms in the TEST_GENERATOR are used for single-input change patterns injection. They are the function FIRST_STATIC_PATTERN, the function SECOND_STATIC_PATTERN, and the procedure STATIC_PATTERN_TESTER. The function FIRST_STATIC_PATTERN collects the first set (ORIGINAL_PATTERN) of the two-tuples. The function SECOND_STATIC_PATTERN collects the second set

Example:

Number of Inputs, N = 3;

Range of Naturals = 0, 1, 2, 3, 4, 5, 6, 7.

Natural to BIT_VECTOR = ("000", "001", "010", "011", "100", "101", "110", "111");

Single-Input Change:

Each original bit patterns is looking for all possible neighbor bit patterns that differ by one bit.

```
(((("000", "100"), ("000", "010"), ("000", "001")),
 (("001", "101"), ("001", "011"), ("001", "000")),
                          ⋮

 (("111", "011"), ("111", "101"), ("111", "110")));
```

(a)

Reformat the two-tuples as follows:

```
(("000", "100"),
 ("000", "010"),
 ("000", "001"),
 ("001", "101"),
 ("001", "011"),
 ("001", "000"),
       ⋮

 ("111", "011"),
 ("111", "101"),
 ("111", "110"));
```

(b)

Figure 20. Illustration of the Operation of the Single-Input Pattern Generator

(NEIGHBOR_PATTERN) of the two-tuples. The procedure STATIC_PATTERN_TESTER injects the two-tuples by controlling the functions FIRST_STATIC_PATTERN and SECOND_STATIC_PATTERN. Each pattern is injected into the test circuit in a particular order with a time gap of maximum propagation delay to ensure stable state transitions.

The specification of the procedure STATIC_PATTERN_TESTER is defined as follows:

```
procedure STATIC_PATTERN_TESTER(
INPUT_NUM: in MVL9_VECTOR_TUPLE_VECTOR(0 to (2**WORDLENGTH)*WORDLENGTH
-1);
signal TIME_DELAY: in PROP_DELAY_SPEC;
signal PATTERN1, PATTERN2, OUT_PATTERN: out MVL9_VECTOR;
MODE: TEST_ORDER);
```

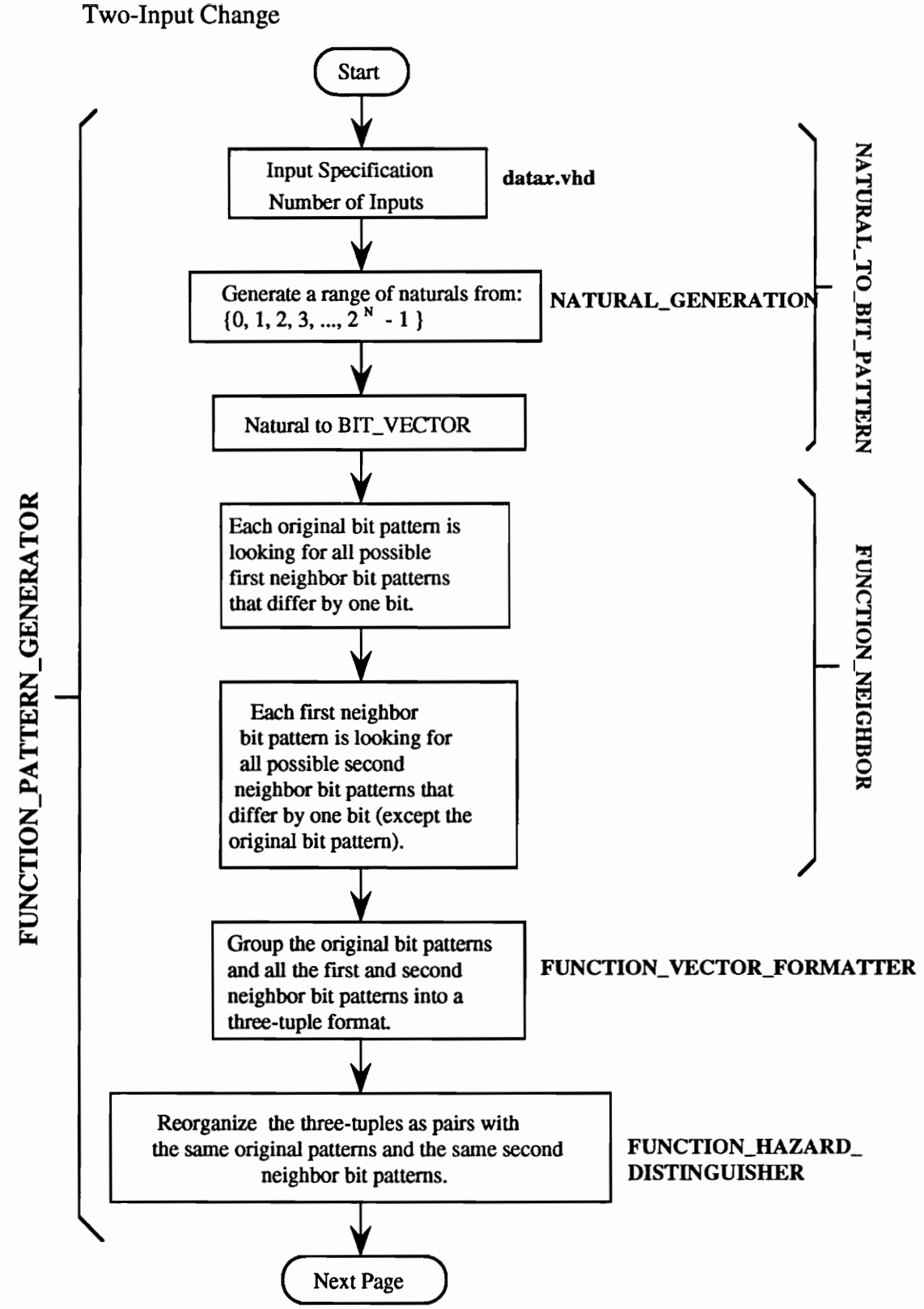
The procedure STATIC_PATTERN_TESTER schedules all the input pattern injection at once. The input variable INPUT_NUM is the desired two-tuples set which is generated from the function STATIC_PATTERN_GENERATOR. The input timing signal TIME_DELAY is a record type that holds the min and max of the whole VHDL model. The output signals PATTERN1 and PATTERN2 represent the current original bit vector pattern and the neighbor bit vector pattern, respectively. The output signal OUT_PATTERN describes the current injected pattern in the test circuit. The variable MODE describes which simulation test mode (STATIC, FUNCT) is currently in use. In this case, the single-input change hazard detection mode (STATIC) is used.

After each two-tuple is injected into the test circuit and its corresponding output transition(s) recorded, then all the previous input and output information is reset to 'X' by injecting a reset pattern to clear the circuit. Then, the test circuit will wait for the next two-tuple injection. More details about the reset pattern will be discussed in section 3.4.3.

3.4.2 Generation of Two-Input Change Patterns

The second bit pattern generator in the HPG is called the "*two-input pattern generator.*" This generator produces all the possible bit vector patterns for detection of the two-input change hazard (i.e., the function hazard). Like the single-input pattern generator, the two-input pattern generator only needs the number of inputs of the test circuit, i.e. the constant value WORDLENGTH. In the same manner, the generator accepts the input-specification file as in the single-input pattern generator.

The design of the two-input pattern generator is similar to the single-input pattern generator. The flow chart for the two-input pattern generator is shown in Figure 21. Like the previous generator, the value WORDLENGTH is used to generate a range of naturals from {0, 1, 2, 3, ..., $2^{WORDLENGTH} - 1$} using the function NATURAL_GENERATION. The original bit vector patterns are created by converting each natural inside the range to its corresponding bit vector pattern in the nine-valued logic. The function NATURAL_TO_BIT_PATTERN performs the task of converting the constant value WORDLENGTH into the range of the nine-valued bit vector patterns. Next, each original bit vector pattern looks for all the possible "*first neighbor bit vector patterns*" in which each differs by one bit from the original bit vector pattern. After that, each first neighbor bit vector pattern then looks for all the possible "*second neighbor bit vector patterns*" in which each differs by one bit from the first neighbor bit vector pattern. That is, the second neighbor bit vector pattern is different from the original bit vector pattern by two bits. Hence, the original bit vector pattern and the second neighbor bit vector pattern are used to describe an input transition where two input variables change simultaneously. These two steps in finding all the neighbor cells are performed by a specific function in the HPG package called


Two-Input Change
Start
Input Specification
Number of Inputs
datax.vhd
Generate a range of naturals from:
{0, 1, 2, 3, ..., 2 N - 1 }
NATURAL_GENERATION
Natural to BIT_VECTOR
NATURAL_TO_BIT_PATTERN
Each original bit pattern is looking for all possible first neighbor bit patterns that differ by one bit.
Each first neighbor bit pattern is looking for all possible second neighbor bit patterns that differ by one bit (except the original bit pattern).
FUNCTION_NEIGHBOR
Group the original bit patterns and all the first and second neighbor bit patterns into a three-tuple format.
FUNCTION_VECTOR_FORMATTER
Reorganize the three-tuples as pairs with the same original patterns and the same second neighbor bit patterns.
FUNCTION_HAZARD_
DISTINGUISHER
Next Page
FUNCTION_PATTERN_GENERATOR

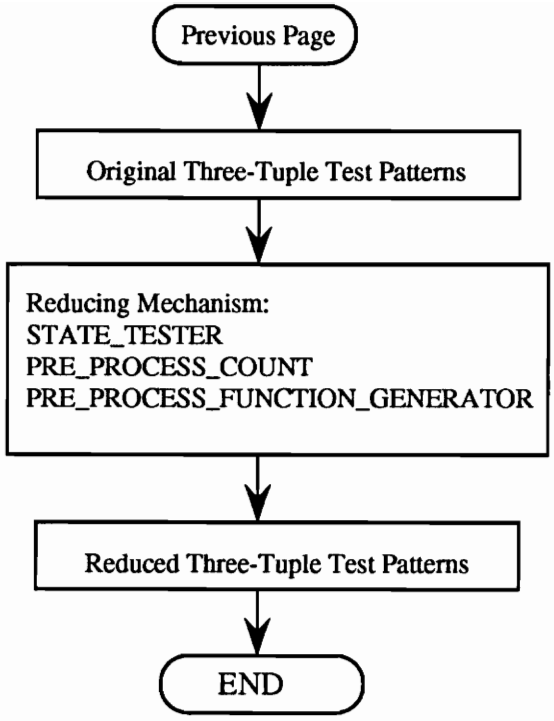


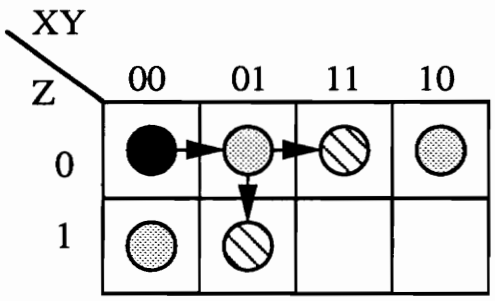
Figure 21. Flow Chart of Hazard Pattern Generator

FUNCTION_NEIGHBOR. Figure 22 shows the detailed design of FUNCTION_NEIGHBOR.

Figure 22a shows a three-variable Karnaugh Map. The dark dot in the cell location "000" represents one of the original bit vector patterns that are generated by the function NATURAL_TO_BIT_PATTERN. The shaded dot represents all the possible first neighbor bit vector patterns with respect to the bit vector "000". In this case, three first neighbor bit vector patterns are found. They are "001", "100", and "010". From each of the first neighbor bit vector patterns, "010" for example, all the second neighbor bit vector patterns will then be found. In this case, only two second neighbor bit vector patterns are found. They are "110" and "011". Bit vector "000" is not counted as the second neighbor bit vector pattern since it was the starting original bit vector pattern. Thus, the total number of patterns becomes (1 * 3 * 2) = 6 patterns. So, for the three-variable case, there are eight original bit vector patterns. The total number of patterns is equal to 8 * (1 * 3 * 2) = 48 patterns.

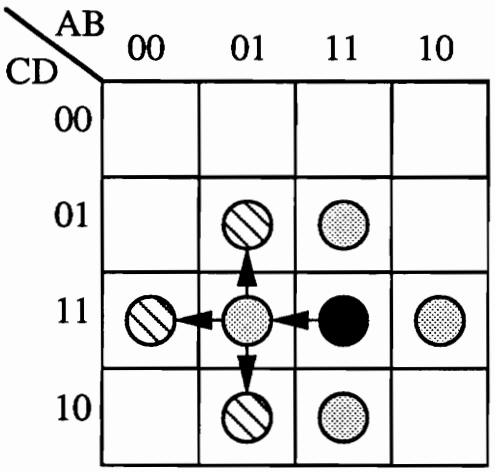
Figure 22b shows a four-variable Karnaugh Map. The dark dots in the cell location "0000" and "1111" represent the original bit vector patterns. The shaded dots with respect to "1111" are the first neighbor bit vector patterns. In this case, four bit patterns are found. They are "1101", "0111", "1110", and "1011". For the first neighbor bit vector pattern "0111", three second neighbor bit vector patterns are found. They are "0101", "0011", and "0110". So, in this case, the total number of patterns generated is equal to (1 * 4 * 3) = 12 patterns. For the four-variable case, there are sixteen original bit vector patterns. The total number of patterns = 16 * (1 * 4 * 3) = 192 patterns. **In general, the total number of patterns generated for N inputs is equal to (Number of Original Bit Vector Patterns) * (Number of First Neighbor Bit Vector Patterns) * (Number of Second Neighbor Bit Vector Patterns) = $2^{WORDLENGTH}$ * (1 *

Two-Input Change:
(a) 3-variable case:



For the first move, the starting cell has three neighbors. Then, at the second move, the seond cells has two neighbors. So, the total number of pattens:
8 x (1 x 3 x 2) = 48 patterns

(b) 4-variable case:



For the first move, the starting cell has four neighbors. Then, at the second move, the seond cells has three neighbors. So, the total number of pattens:
16 x (1 x 4 x 3) = 192 patterns

In general, the total number of the original three-tuple test patterns for N inputs: $2^N * N * (N - 1)$

Figure 22. Illustration of the Function FUNCTION_NEIGHBOR

WORDLENGTH * (WORDLENGTH - 1)). It can be seen that the number of patterns grows exponentially as the number of primary input increases.

From the function FUNCTION_NEIGHBOR, all the original bit vector patterns, all the first and the second neighbor bit vector patterns for the given test circuit are found. The function FUNCTION_VECTOR_FORMATTER then re-formats the patterns into a *three-tuples* format. Each of the three-tuples (ORIGINAL_PATTERN, FIRST_NEIGHBOR_PATTERN, SECOND_NEIGHBOR_PATTERN) is re-organized by using a function in HPG called FUNCTION_HAZARD_DISTINGUISHER. This function re-organizes all the three-tuples by grouping on all three-tuples which have the same original bit patterns and the same second neighbor bit patterns adjacently.

The function FUNCTION_PATTERN_GENERATOR implements the whole process of transforming the given constant value WORDLENGTH to the final desired set of three-tuples.

The specification of the function FUNCTION_PATTERN_GENERATOR is listed below:

```
function FUNCTION_PATTERN_GENERATOR(INPUT: NATURAL)  -- INPUT = WORDLENGTH
return MVL9_VECTOR_TRIPLE_VECTOR;                    -- return the original three-tuples set.
```

Figure 23 describes an example showing how the function FUNCTION_PATTERN_GENERATOR works. The value WORDLENGTH is equal to 3. From the function NATURAL_GENERATION, the range of naturals becomes {0, 1, 2, 3, 4, 5, 6, 7}. From the function NATURAL_TO_BIT_PATTERN, the original bit vector patterns set is {"000", "001", "010", "011", "100", "101", "110", "111"}. The function FUNCTION_NEIGHBOR then finds the set of original bit patterns and all the corresponding first and second neighbor bit vector patterns. These are shown in Figure 23a. The three-tuple

Example:

Number of Inputs, N = 3;

Range of Naturals = 0, 1, 2, 3, 4, 5, 6, 7.

Natural to BIT_VECTOR = ("000", "001", "010", "011","100", "101", "110", "111");

Two-Input Change:

Each original bit patterns is looking for all possible first and second neighbor bit patterns.

```
(((("000", "100", "110"), ("000", "100", "101"),
 ("000", "010", "110"), ("000", "010", "011"),
 ("000", "001", "101"), ("000", "001", "011")),
                        •
                        •
                        •
 (("111", "011", "001"), ("111", "011", "010"),
 ("111", "101", "001"), ("111", "101", "100"),
 ("111", "110", "010"), ("111", "110", "100")));
```

(a)

Reformat the three-tuples as follows:

```
(("000", "100", "110"), ("000", "100", "101"),
 ("000", "010", "110"), ("000", "010", "011"),
 ("000", "001", "101"), ("000", "001", "011"),
                        •
                        •
                        •
 ("111", "011", "001"), ("111", "011", "010"),
 ("111", "101", "001"), ("111", "101", "100"),
 ("111", "110", "010"), ("111", "110", "100"));
```

(b)

Reformat the three-tuples by grouping on all three-tuples which have the same original bit patterns and the same second neighbor bit patterns adjacently.

```
(("000", "100", "110"), ("000", "010", "110"),
 ("000", "100", "101"), ("000", "001", "101"),
 ("000", "010", "011"), ("000", "001", "011"),
                        •
                        •
                        •
 ("111", "011", "001"), ("111", "101", "001"),
 ("111", "011", "010"), ("111", "110", "010"),
 ("111", "101", "100"), ("111", "110", "100"));
```

(c)

Figure 23. Illustration of the Operation of the Two-Input Pattern Generator

patterns are shown in Figure 23b after being re-formated by the function FUNCTION_VECTOR_FORMATTER. The final three-tuple patterns re-organized by the function FUNCTION_HAZARD_DISTINGUISHER are shown in Figure 23c.

3.4.2.1 Reducing Mechanism in the Two-Input Pattern Generator

The next step is to provide a mechanism to reduce the number of patterns in the three-tuples set. In VHDS, the real time required for simulation is proportional to the number of test patterns. Therefore, the real time required for simulation increases exponentially as the number of patterns increases exponentially. For example, if a test circuit has ten primary inputs, then, experiments have shown the real time required will be eight hours. So, the real time simulation becomes an important issue in the VHDS analysis. The issue is especially significant for function hazard detection. To solve the problem, a pattern reducing mechanism is inserted after the three tuples set is created. Figure 24 shows the general block diagram of the pattern reducing mechanism. The purpose of this step is to reduce the number of test patterns for the function hazard detection. From the function NATURAL_TO_BIT_PATTERN, the given constant value (WORDLENGTH) in the input-specification file has generated a range of naturals from {0, 1, 2, ..., $2^{WORDLENGTH}$ -1}; and each integer in the range has been converted into its BIT_VECTOR form. The BIT_VECTOR values represent all the possible "input logic values" of the given test circuit. Each logic state value will then be injected into the test circuit. The time gap between each input state value injection is the maximum propagation delay (max) of the circuit. The procedure PRE_PROCESS_INJECTION implements the above process. The procedure specification of the PRE_PROCESS_INJECTION is listed below:

```
procedure PRE_PROCESS_INJECTION(
signal TIME_DELAY: PROP_DELAY_SPEC;
signal OUTPUT: out MVL9_VECTOR);
```

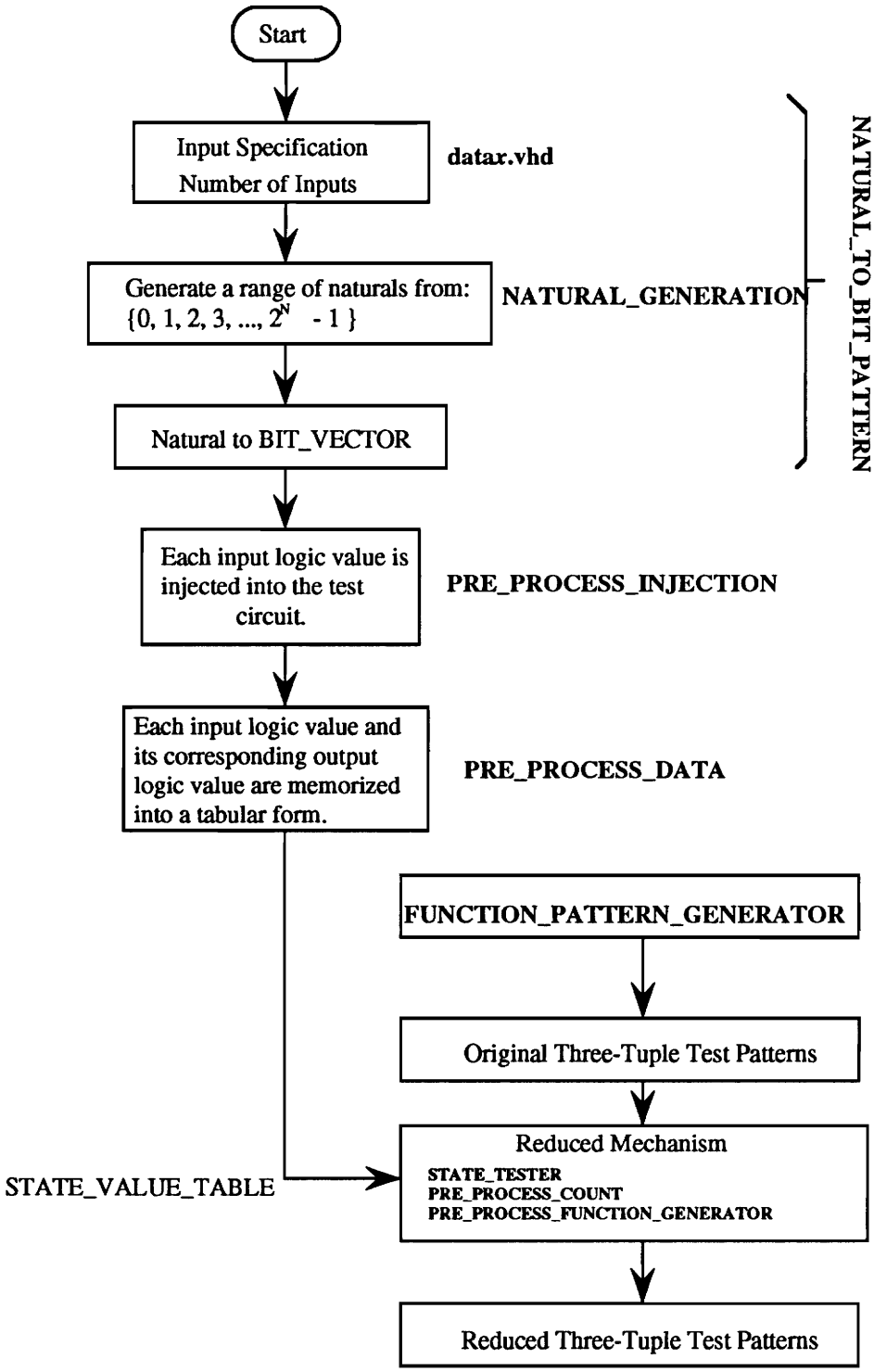


Figure 24. Flow Chart of the Three-Tuple Test Patterns Reducing Mechanism

TIME_DELAY is the timing signal that contains the min and max values of the circuit. The signal OUTPUT represents all the input logic values of the circuit. After each input logic value is injected into the test circuit, the input logic value and the corresponding stable output logic value are then memorized into a tabular form by a procedure called PRE_PROCESS_DATA. The procedure specification of the PRE_PROCESS_DATA is listed below:

```
procedure PRE_PROCESS_DATA(
                signal INPUT1: MVL9_VECTOR; signal INPUT2: MVL9;
                TIME_DELAY: PROP_DELAY_SPEC;
                DATA: inout INTEGER;
                signal STATE_VALUE_TABLE: out MVL9_STATE_LOGIC_VECTOR);
```

The signal INPUT1 is the logic input value of the cirucit. The signal INPUT2 is the corresponding stable output logic value of the circuit. The value TIME_DELAY is the min and max values of the circuit. The value DATA is an integer value which memorizes the index number of input logic values in the procedure PRE_PROCESS_DATA. The output signal STATE_VALUE_TABLE is of type MVL9_STATE_LOGIC_VECTOR. It memorizes all the input logic values and the output logic values together into a tabular format.

Three functions are used to reduce the original three-tuple test patterns that are generated from the function FUNCTION_PATTERN_GENERATOR to a reduced set of three-tuple test patterns based on the information in the STATE_VALUE_TABLE. They are the functions STATE_TESTER, PRE_PROCESS_COUNT and PRE_PROCESS_FUNCTION_GENERATOR. The function STATE_TESTER returns the output logic value from the specified input logic value. The specification of the function STATE_TESTER is defined as follows:

```
function STATE_TESTER(      INPUT: MVL9_VECTOR;
                            STATE_VALUE_TABLE: MVL9_STATE_LOGIC_VECTOR)
return MVL9;
```

The value INPUT is the input logic value. The value STATE_VALUE_TABLE is the array table that contains all the input logic values and the output values of the given test circuit. The function PRE_PROCESS_COUNT computes the total number of the reduced three-tuple test patterns for a given test circuit. The final return natural value is the total number of reduced potential function hazard test patterns. The function specification of the PRE_PROCESS_COUNT is defined as follows:

```
function PRE_PROCESS_COUNT(
INPUT: MVL9_VECTOR_TRIPLE_VECTOR
        (0 to ((2**WORDLENGTH)*WORDLENGTH*(WORDLEGNTH - 1)) - 1);
STATE_VALUE_TABLE: MVL9_STATE_LOGIC_VECTOR) return NATURAL;
```

The value INPUT is the original three-tuple test patterns that were generated from the function FUNCTION_PATTERN_GENERATOR.

The function PRE_PROCESS_FUNCTION_GENERATOR is similar to the function PRE_PROCESS_COUNT. Instead of finding the total number of reduced potential function hazard test patterns, the function PRE_PROCESS_FUNCTION_GENERATOR finds all the three-tuples that have the output logic value "1-0-1" or "0-1-0". That is, the function records all the reduced potential function hazard test patterns.

The specification of the function PRE_PROCESS_FUNCTION_GENERATOR is defined as follows:

```
function PRE_PROCESS_FUNCTION_GENERATOR(
INPUT: MVL9_VECTOR_TRIPLE_VECTOR
```

```
        (0 to ((2**WORDLENGTH)*WORDLENGTH*(WORDLENGTH - 1)) -1);
STATE_VALUE_TABLE: MVL9_STATE_LOGIC_VECTOR;
constant NUMBER: NATURAL) return MVL9_VECTOR_TRIPLE_VECTOR;
```

The value INPUT is the original three-tuple test patterns that were generated from the function FUNCTION_PATTERN_GENERATOR. The constant natural value NUMBER is the total number of reduced potential function hazard test patterns. The final return value is the set of reduced potential function hazard test patterns.

The reducing mechanism first scans **two** consecutive original three-tuple (original bit vector pattern, first neighbor bit vector pattern, second neighbor bit vector pattern) patterns. Then, each input pattern in the two original three-tuples will find its corresponding output logic value by looking at the look-up table STATE_VALUE_TABLE. If both three-tuple test patterns have the output logic values "1-0-1" or "0-1-0", then only one three-tuple test pattern will be chosen as the reduced potential function hazard test pattern. This is because the function FUNCTION_HAZARD_DISTINGUISHER has re-organized all the three tuples by grouping all three-tuples which have the same original bit patterns and the same second neighbor bit patterns adjacently. So, each two consecutive original three-tuples has the same original bit (first) pattern and the same second neighbor bit (third) pattern. However, for the two-input change (function) hazard analysis, two input variables are required to change simultaneously. Hence, in this situation, only the original bit (first) patterns and the second neighbor (third) bit patterns are useful. Therefore, even if both original three-tuples have the output logic values "1-0-1" or "0-1-0", only one of them is chosen. On the other hand, if any of the two original three-tuple test patterns has the output logic values "1-0-1" or "0-1-0", then this three-tuple test pattern will be chosen as the reduced potential function hazard test pattern. This process repeats until all the original three-tuple test patterns have been examined. Other three-tuple patterns that do not have

the output logic values "1-0-1" or "0-1-0" will not be chosen for the set of reduced potential function hazard test patterns. Figure 25 illustrates the reducing mechanism.

From these functions, the reduced set of three-tuple test patterns is found. The final step in the two-input pattern generator is to inject the reduced potential function hazard test patterns into the test circuit. Similar to the original single-input change hazard analysis, the injection task is done by the functions PRE_FIRST_FUNCTION_PATTERN, PRE_SECOND_FUNCTION_PATTERN, and PRE_FUNCTION_PATTERN_TESTER. The function PRE_FIRST_FUNCTION_PATTERN collects the first set (original bit vector pattern) of the reduced three-tuples set. The function PRE_SECOND_FUNCTION_PATTERN collects the **third** set (second neighbor bit vector pattern) of the reduced three-tuples set. The procedure PRE_FUNCTION_PATTERN_TESTER injects the reduced test patterns by controlling the functions PRE_FIRST_FUNCTION_PATTERN and PRE_SECOND_FUNCTION_PATTERN. Each pattern is injected into the test circuit in a particular order with a time gap of maximum propagation delay (max) to ensure stable state transitions. The specification of the procedure PRE_FUNCTION_PATTERN_TESTER is defined as follows:

```
procedure PRE_FUNCTION_PATTERN_TESTER(
INPUT_NUM: in MVL9_VECTOR_TRIPLE_VECTOR;
signal TIME_DELAY: in PROP_DELAY_SPEC;
constant NUMBER: NATURAL;
signal PATTERN1, PATTERN2, OUT_PATTERN: out MVL9_VECTOR;
MODE: TEST_ORDER);
```

The input signal INPUT_NUM is the reduced set of three-tuple test patterns. The input signal TIME_DELAY contains the min and max values of the circuit. The constant natural value NUMBER is the total number of reduced three-tuples that are used for function hazard analysis. The output signals PATTERN1 and PATTERN2 represent the original bit vector

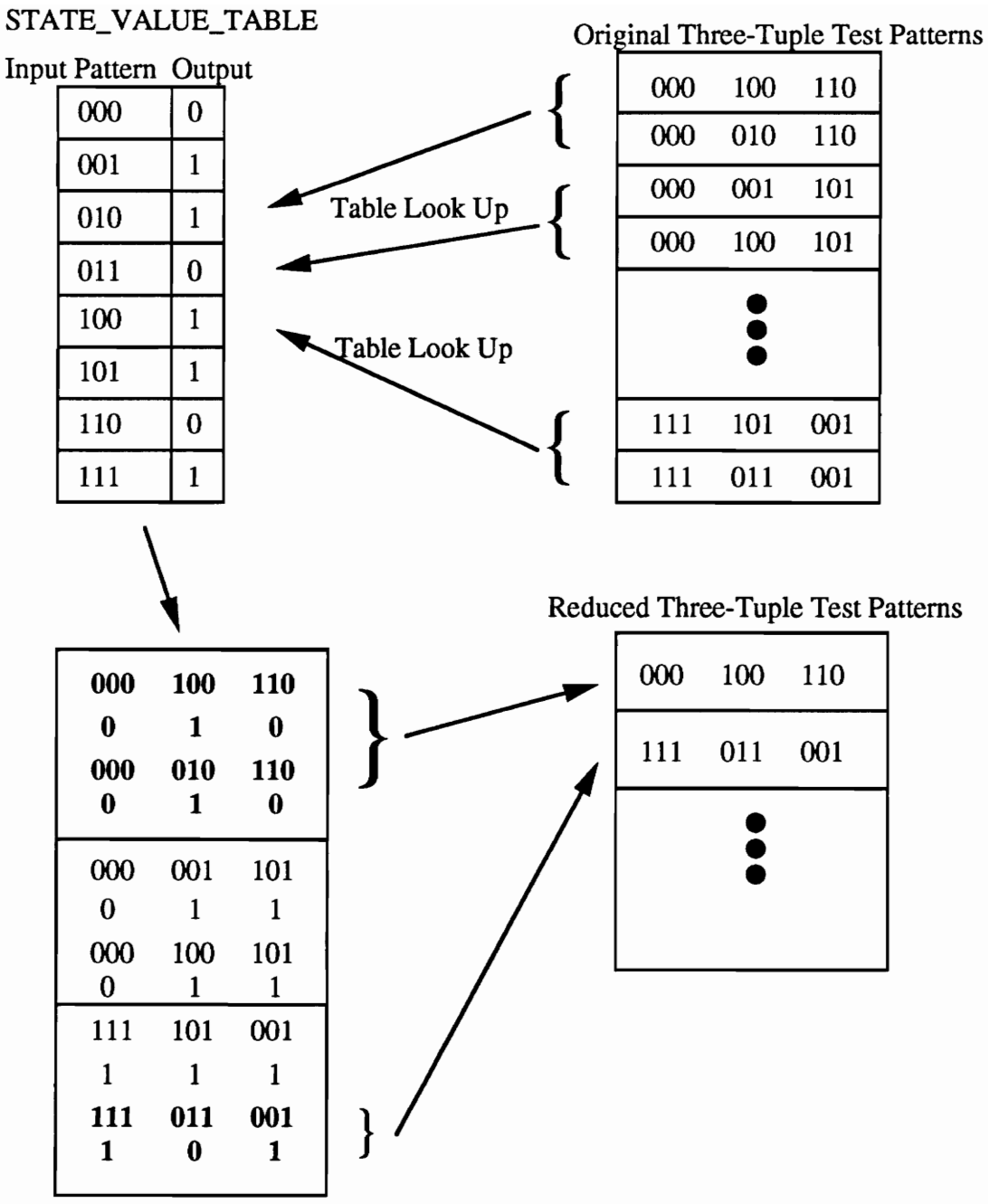


Figure 25. Illustration of the Reducing Mechanism

pattern and the second neighbor bit vector pattern from the HPG, respectively. The output signal OUT_PATTERN describes the current injected input pattern in the test circuit. The enumeration variable MODE describes which current simulation test mode is currently in use. By default, the test mode is the two-input change hazard detection (FUNCT). After each reduced three-tuple is injected into the test circuit and its corresponding output transition(s) are recorded, then all the previous input and output information will reset to 'X' by injecting a reset pattern to clear the circuit. Then, the test circuit will wait for the next reduced three-tuple injection.

The last function in the two-input pattern generator is called PRE_TEST_BEGIN_TIME. This function computes the starting time for the reduced three-tuples injection to the test circuit. This starting time value also indicates when the hazard detection procedure will start. The specification of the function PRE_TEST_BEGIN_TIME is listed below:

```
function PRE_TEST_BEGIN_TIME(TIME_DELAY: PROP_DELAY_SPEC) return TIME;
```

3.4.3. Other Features in the HPG

Beside the two major test pattern generators described above, the HPG also has some supporting features which are essential for hazard detection. They are the functions BIT_PATTERN_TO_NATURAL, RESET_PATTERN, DC_PATTERN, TOTAL_STATIC_PATTERNS, TOTAL_FUNCTION_PATTERNS, ONE_BIT_DIFF, TWO_BIT_DIFF, and MEAN_DELAY_TIME. Their specifications are listed below:

```
function BIT_PATTERN_TO_NATURAL(INPUT: BIT_VECTOR) return NATURAL;
```

```
function RESET_PATTERN(INPUT: NATURAL := WORDLENGTH) return MVL9_VECTOR;

function DC_PATTERN(INPUT: NATURAL := WORDLENGTH) return MVL9_VECTOR;

function TOTAL_STATIC_PATTERNS(INPUT: NATURAL) return NATURAL;

function TOTAL_FUNCTION_PATTERNS(INPUT: NATURAL) return NATURAL;

function ONE_BIT_DIFF(INPUT1, INPUT2: MVL9_VECTOR(0 to WORDLENGTH -1) return
BOOLEAN;

function TWO_BIT_DIFF(INPUT1, INPUT2: MVL9_VECTOR(0 to WORDLENGTH -1) return
BOOLEAN;

function MEAN_DELAY_TIME(TIME_DELAY: PROP_DELAY_SPEC) return TIME;
```

The function BIT_PATTERN_TO_NATURAL converts any bit pattern (BIT_VECTOR) into a natural number (NATURAL). The function RESET_PATTERN resets the circuit by injecting the logic value 'X'. If the circuit has the constant value WORDLENGTH = N, then the function RESET_PATTERN will produce a reset pattern of logic value 'X' of length N (i.e. "XXX ... X"). The function DC_PATTERN generates the don't care logic value '-' of any length. If the circuit has the constant value WORDLENGTH = N, then the function DC_PATTERN will produce a vector of the don't care logic value '-' of length N (i.e. "--- ... ---"). The function TOTAL_STATIC_PATTERNS computes the total number of single-input change test patterns required for a given circuit (WORDLENGTH). The function TOTAL_FUNCTION_PATTERNS computes the total number of the original un-reduced two-input change test patterns required for a given circuit (WORDLENGTH). The function ONE_BIT_DIFF compares the bit difference between the two input vectors. If the two input vectors differ by one bit, then the function returns a BOOLEAN value TRUE. The function TWO_BIT_DIFF compares the bit difference between the two input vectors. If the two input vectors differ by two bits, then the function returns a BOOLEAN value TRUE. The function MEAN_DELAY_TIME computes the average (mean) value of the min and max of the test circuit. The value TIME_DELAY is a record type value and

it holds the min and max of the whole VHDL test circuit. A listing of all the subprograms are enclosed in the appendix.

3.5 VHDS Hazard Detection Package

This section discusses the VHDS hazard detection package. The hazard detection package serves two main purposes in the VHDS. Firstly, the package contains important checks that are able to distinguish what type of hazard exists in a logic circuit. Secondly, the package generates a VHDS report summary for all the hazards found in the test circuit. Three major hazards are detected by the package. They are the static hazard, the dynamic hazard, and the function hazard. The hazard detection algorithm design is "*definition based.*" That is, the detection design is based on the unique *definitions* and the *characteristics* of each individual hazard.

This section starts with the problems posed and the general methodology in the hazard detection. Then, the methodology for the static hazard detection will be described. Next, the methodology for the dynamic hazard detection will be presented. Finally, the methodology for the function hazard detection will be explained. The filename for the VHDS hazards detection package is called *phaz1_test.vhd.* The package identifier is called PRE_HAZARD_TEST. There is only one procedure in the detection package. The procedure identifier is called HAZARDS_PATTERNS_TEST.

3.5.1 General Problems Posed in the Hazard Detection Algorithm

Timing hazards are a fundamental concept in digital design and testing. However, various similarities and differences in the timing hazards have made the implementation task difficult to

achieve. This is due to the similarity in the output waveforma between various hazards. As shown previously in the Figures 5 and 7 of chapter 2, the output waveforms of the static hazard and the function hazard are nearly identical. If the output event(s) is only considered, then both the two hazards produce a spike in the output. Similarity in the output waveform has made the hazard detection algorithm not only consider the output event(s) but also the input event(s).

Looking back to Figure 8b, the state transitions of XYZ from "111" to "110" will cause a static hazard. However, if we consider the state transitions from "011" to "111" and to "110", a function 1 hazard seems to occur in the state transition since the output transitions are "1-0-1" and two input events are involved in the transitions. In fact, only a static 1 hazard exists in the state transition from "111" to "110". The state transition "011" to "111" does not cause any effect in the output transition.

From Figure 8b, let's consider another problem on the state transitions of XYZ from "101" to "111" to "110". The output transitions are "0-1-0-1". Since the state transitions involve a two-input change, the designer may be confused that either a function 0 hazard ("0-1-0") or a function 1 hazard ("1-0-1") occurs on the above state transitions. In fact, only a static 1 hazard exists in the state transitions from "111" to "110". State transitions of XYZ from "101" to "111" do not affect the existence of a hazard. All the above phenomena are known as "*false hazard detection.*" This kind of improper hazard detection commonly confuses most designers and therefore, the actual effect of a hazard on the logic circuit cannot be rectified properly. In this thesis, the VHDS hazard detection package has been designed in order to avoid false hazard detection.

One way to predict a hazard accurately in a logic circuit and to avoid the false hazard detection problem is to implement each individual hazards' definition and characteristics into VHDL.

Based on their unique definitions and characteristics, various types of hazards can be distinguished. This kind of implementation is called the "*definition based implementation.*" The VHDS hazard detection package uses the above implementation technique.

A timing hazard is a kind of hazard that is related to physical circuit realization and device propagation delays. Hence, different hardware realizations and timing can affect the existence of hazards in a logic circuit. To detect all the common hazards existing in a logic circuit, the information regarding the input events, the output events, and the circuit timing become the necessary information in designing a hazard detection package.

3.5.2 General Design Methodology of the VHDS Hazard Detection Package.

Before the discussion of the general design methodology of the VHDS hazard detection package, one definition must be given.

An *Input pattern pair* is defined to be a pair of bit vector input patterns that are injected into the test circuit. For the single-input change hazard analysis, the input pattern pair refers to the two-tuple, i.e. the original bit vector pattern and the neighbor bit vector pattern. For the two-input change hazard analysis, the input pattern pair refers to the first pattern and the third pattern of each three-tuple, i.e. the original bit vector pattern, and the second neighbor bit vector pattern.

The general hazard detection algorithm is divided into two major phases. The purpose of the first phase is to collect the input and the output information after applying the input pattern pair. The second phase uses the information from the first phase and performs checking for the defined hazard conditions. If all the collected information matches with one of the hazard conditions,

then the assertion message will be printed. Figure 26 shows a general block diagram of the first phase of operation. Phase one operation is divided into two parts. Part one is the declaration region while part two is the data processing main body. There are two major declarations. The first one is the record type declaration. A record type called HAZARD_VAR is defined in the hazard detection package PRE_HAZARD_TEST. This record type contains a series of enumeration type and physical type declarations. All those declarations represent those data and control variables that need to be memorized in the hazard detection package. By using the record type, only one composite variable is needed to be declared. Hence, one subprogram input parameter for the composite variable is needed. The composite variable name in the hazard detection package is called HAZARD_SCHEDULE1. The initial value of the composite variable is called HAZARD_CONSTANT. The specification of the hazard detection procedure HAZARDS_PATTERNS_TEST is defined as follows:

```
procedure HAZARDS_PATTERNS_TEST(
signal OUTPUT: MVL9; NAME:STRING; HAZARD_SCHEDULE1: inout HAZARD_VAR;
signal PATTERNS: in MVL9_VECTOR;
signal PATT1, PATT2: in MVL9_VECTOR;
signal TIME_OUT: in PROP_DELAY_SPEC;
INFO: STATIC_INFO_VECTOR :=
        (( DC_PATTERN(WORDLENGTH), DC_PATTERN(WORDLENGTH), 0 ns, 0 ns),
         ( DC_PATTERN(WORDLENGTH), DC_PATTERN(WORDLENGTH), 0 ns, 0 ns)));
```

The signal OUTPUT represents the output signal which is under hazard analysis. The variable NAME describes the name of the output signal of type STRING. The record type variable HAZARD_SCHEDULE1 memorizes the data and the control variables in the hazard detection package. The input signal PATTERNS describes which pattern is currently being injected into the test cirucit. The signal PATT1 and PATT2 represents the current input pattern pair from the HPG. The record type value TIME_OUT holds the min and max values of the test circuit. The

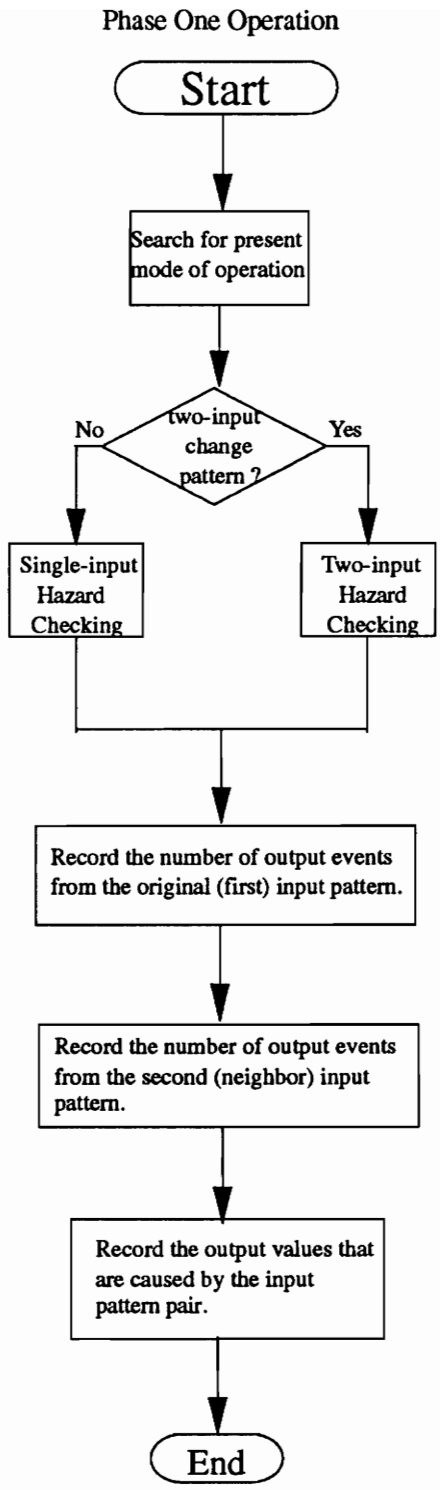


Figure 26. General Methodology Design in Hazard Detection Package

record type input variable INFO decribes the static hazard information of the test circuit. The default value of INFO is (DC_PATTERN(WORDLENGTH), DC_PATTERN(WORDLENGTH), 0 ns, 0 ns) where DC_PATTERN(WORDLENGTH) produces a vectorized array of don't care logic value '-' of length WORDLENGTH.

The second declaration region is called File and Text I/O Declarations. This region defines the structure of the VHDS report summary. It contains the external output writing file declaration, report summary framework declarations, and all the hazard assertion message declarations. The file I/O declaration is defined as follows:

```
file OUTFILE: TEXT is out "outfile.11";
```

The filename for the VHDS report summary is called "*outfile.11*" and will be generated in the same directory as the designer is currently working from. The summary framework declarations contain the test circuit statistics. These include the minimum and the maximum propagation delays of the test circuit, the selected output signal name, and the number of primary inputs of the test circuit. The hazard assertion messages for the TEXT I/O are the static 0 hazard assertion, the static 1 hazard assertion, the dynamic 0 hazard assertion, the dynamic 1 hazard assertion, the function 0 hazard assertion, and the function 1 hazard assertion. Each assertion message provides information on the type of hazard, the output signal which has found the hazard, and the time the hazard occurs.

Part two of phase one is the data processing main body. It is divided into four major processing units. The first processing unit determines the current mode of operation. If the two-input change

test patterns are used in the hazard analysis, then, the present mode of operation will switch to two-input change hazard detection. Otherwise, the present mode of operation will switch to single-input change hazard detection.

The next two processing units record the number of output events caused by the the first input pattern and the second input patterns respectively. For example, when the first input pattern (original input pattern) is injected into the test circuit, the time of event of the first input pattern is defined as HAZARD_SCHEDULE1.INPUT1_EVENT_TIME. During the time period from HAZARD_SCHEDULE1.INPUT1_EVENT_TIME + minimum propagation delay of the test circuit (min) to HAZARD_SCHEDULE1.INPUT1_EVENT_TIME + maximum propagation delay of the test cirucit (max), the number of output events will be counted and the value will be written to a memory variable called HAZARD_SCHEDULE1.OUTPUT_COUNT1. The procedures are the same for the second input pattern.

The last processing unit saves the output values in the form of a vector. It saves all the output values that are caused by the input pattern pair. For the single-input change hazard analysis, all the output logic values that are caused by the original input pattern and the neighbor input pattern will be recorded. For the two-input change (function hazard) analysis, all the output logic values that are caused by the original input pattern, and the second neighbor input pattern will be recorded. The output values will be written to a MVL9_VECTOR variable called HAZARD_SCHEDULE1.OUTPUT_PATTERNS in the order of the *first-in-first-write* format.

The second phase performs a series of checks between each hazard conditions with the input stimulus. Figure 27 shows the block diagram of phase two operation. From the first phase, the

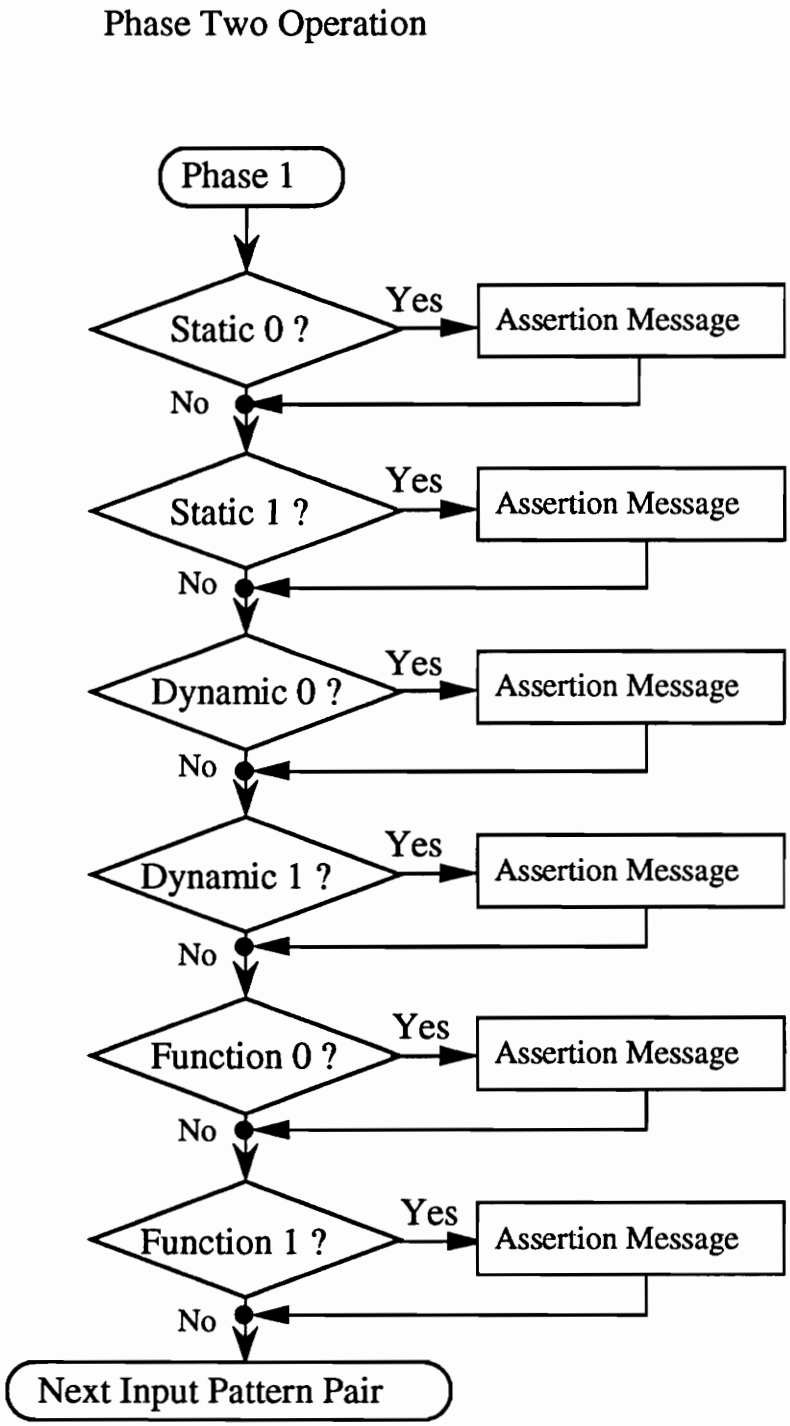


Figure 27. Flow Chart for the Hazard Detection Algorithm

input, the output, and the timing information have been collected. This data is checked against each hazard condition. If the input stimulus and the output conditions match with one of the hazard conditions, hazard assertion message will be flagged and the hazard information will be written to the VHDS report summary. If none of the hazard conditions are met, no assertion message will be flagged. After each input pattern pair has been tested by the detection package, the circuit will be reset by the reset pattern and it will then wait for the next input pattern series.

3.5.3 Methodology for Static Hazard Detection

The previous section has discussed the general design methodology for the hazard detection package. In this section, a more detailed description of the static hazard detection will be presented. Static hazard detection is divided into two steps. The first step is the potential static hazard detection. The second step is the confirmation of the potential static hazard detection. The conditions of the static 0 hazard in the hazard detection package are defined as follows:

Under the single-input change hazard detection mode, if all the following conditions are fulfilled, then a static 0 hazard is detected.

- *1) Output Conditions Initial Matching and the Output Event(s) Counting from the input test patterns:*

The present output (OUTPUT) is '0', the last output value (OUTPUT'last_value) is '1', and the next to last output value (HAZARD_SCHEDULE1.OUTPUT_LAST_TWO_VALUE) is '0'. While the original input pattern creates the output value '0' and the neighbor input pattern creates the output values "10", the number of output events that are caused from the the original input

pattern is one and the number of the output events that are caused from the neighbor input pattern is two. When all the above conditions are fulfilled, step 2 will not proceed until the present output is 'X' (i.e. the reset output). The two step processes distinguish the dynamic hazards and the static hazards. For example, if a single input is changed and the output values are "0-1-0-1". The package may be confused that either a static 0 hazard ("0-1-0") or a static 1 hazard ("1-0-1") occurs on the above single input change.

- *2) Confirmation of the Static 0 Hazard Detection:*

The present output (OUTPUT) is 'X' (i.e. the reset output), the immediately previous output value (OUTPUT'last_value) is not equal to 'X', and the output events pattern (HAZARD_SCHEDULE1.OUTPUT_PATTERNS) that is caused by the input pattern pair is "010Z". The logic value 'Z' is the high-impedance state and it will only be substituted into the HAZARD_SCHEDULE1.OUTPUT_PATTERNS when no further output event is found in the static hazard detection.

Similarly, the conditions of the static 1 hazard in the hazard detection package are defined as follows:

Under the single-input change hazard detection mode, if all the following conditions are fulfilled, then a static 1 hazard is detected.

- *1) Output Conditions Initial Matching and the Output Event(s) Counting from the input test patterns:*

The present output (OUTPUT) is '1', the last output value (OUTPUT'last_value) is '0', and the next to last output value (HAZARD_SCHEDULE1.OUTPUT_LAST_TWO_VALUE) is '1'. While the original input pattern creates the output value '1' and the neighbor input pattern creates the output values "01", the number of output events that are caused from the the original input pattern is one and the number of the output events that are caused from the neighbor input pattern is two. When all the above conditions are fulfilled, step 2 will not proceed until the present output is 'X' (i.e. the reset output).

- *2) Confirmation of the Static 1 Hazard Detection:*

The present output (OUTPUT) is 'X' (i.e. the reset output), the immediately previous output value (OUTPUT'last_value) is not equal to 'X', and the output events pattern (HAZARD_SCHEDULE1.OUTPUT_PATTERNS) that is caused by the input pattern pair is "101Z". The logic value 'Z' is the high-impedance state and it will only be substituted into the HAZARD_SCHEDULE1.OUTPUT_PATTERNS when no further output event is found in the static hazard detection.

3.5.4. Methodology for Dynamic Hazard Detection

In this section, a more detailed description of the dynamic hazard detection will be presented. From the definition of the dynamic 0 hazard in chapter 2, the dynamic hazard definition based implementation in VHDL is described as follows.

The conditions of the dynamic 0 hazard in the hazard detection package are defined as follows:

Under the single-input change hazard detection mode, if all the following conditions are fulfilled, then a dynamic 0 hazard is detected.

- *1) Output (OUTPUT) Conditions Matching:*

The present output (OUTPUT) is '0', the last output value (OUTPUT'last_value) is '1', the next to last output value (HAZARD_SCHEDULE1.OUTPUT_LAST_TWO_VALUE) is '0', and the output events pattern (HAZARD_SCHEDULE1.OUTPUT_PATTERNS) that is caused by the two-tuple is "1010".

- *2) Output Event(s) Counting from the input test patterns:*

While the original input pattern creates the output value '1' and the neighbor input pattern creates the output values "010", the number of output events that are caused from the the original input pattern is one and the number of the output events that are caused from the neighbor input pattern is three.

Similarly, the conditions of the dynamic 1 hazard in the hazard detection package are defined as follows:

Under the single-input change hazard detection mode, if all the following conditions are fulfilled, then a dynamic 1 hazard is detected.

- *1) Output (OUTPUT) Conditions Matching:*

The present output (OUTPUT) is '1', the last output value (OUTPUT'last_value) is '0', the last last output value (HAZARD_SCHEDULE1.OUTPUT_LAST_TWO_VALUE) is '1', and the output events pattern (HAZARD_SCHEDULE1.OUTPUT_PATTERNS) that is caused by the two-tuple is "0101".

- *2) Output Event(s) Counting from the input test patterns:*

While the original input pattern creates the output value '0' and the neighbor input pattern creates the output values "101", the number of output events that are caused from the the original input pattern is one and the number of the output events that are caused from the neighbor input pattern is three.

3.5.5. Methodology for Function Hazard Detection

In this section, a more detailed description of function hazard detection will be presented. Function hazard is different from the static hazard and the dynamic hazard. For example, the existence of the function hazard is dependent on the timing between input events. Also, as mentioned in section 3.5.1, function hazard detection is more difficult due to the confusion resulting from false hazard detection. Therefore, the methodology of function hazard detection is more complicated. It is divided into two steps. The first step is the initial potential function hazard checking. The second step is the confirmation of the potential function hazard checking. From the definition of the function 0 hazard in chapter 2, the function hazard detection implementation in VHDL is described as follows.

Under the two-input change hazard detection mode, if all the following conditions are fulfilled, then a function 0 hazard is detected.

- *Step 1) Output Conditions Initial Matching and the Ouput Event(s) Counting from the input test patterns:*

The present output (OUTPUT) is '0', the last output value (OUTPUT'last_value) is '1', and the next to last output value (HAZARD_SCHEDULE1.OUTPUT_LAST_TWO_VALUE is '0'. In the function hazard detection, the original input pattern creates the output value '0', and the second neighbor input pattern creates the output value "10". Hence, the number of output events that are caused from the the original input pattern is one, and the number of the output events that are caused from the second neighbor input pattern is two. When all the above conditions are fulfilled, step 2 will not proceed until the present output (OUTPUT) is 'X' (i.e. the reset output). The main purpose of step 2 is to avoid the false hazard detection and to ensure the package detects function 0 hazard correctly. For example, if two inputs are changed and the output values are "0-1-0-1". The package may be confused that either a function 0 hazard ("0-1-0") or a function 1 hazard ("1-0-1") occurs on the above two input change.

- *Step 2) Confirmation of the Function 0 Hazard Checking*:

The present output (OUTPUT) is 'X' (i.e. the reset output), the last output value (OUTPUT'last_value) is not equal to 'X', and the output events pattern (HAZARD_SCHEDULE1.OUTPUT_PATTERNS) that is caused by the three-tuple is "010-". The logic value '-' is the don't care logic value and it will only be substituted into the HAZARD_SCHEDULE1.OUTPUT_PATTERNS when no further output event is found in the

function hazard detection. Even if all the above conditions are fulfilled, a potential function 0 hazard is not confirmed if the hazard information matches with the static hazard information described in the record type constant INFO. During the function hazard analysis, the designer must read all the static hazards found in single-input change hazard analysis to a record type constant INFO in the test circuit source file. If a potential 0 function hazard matches with the static hazard information, then this potential 0 function hazard is not considered as the function 0 hazard. However, if a potential function 0 hazard does not match with the static hazard information, then this potential function 0 hazard is the function 0 hazard and the assertion message is printed to the simulation output file.

Similarly, the conditions of the function 1 hazard in the detection package are defined as follows:

Under the two-input change hazard detection mode, if all the following conditions are fulfilled, then a function 1 hazard is detected.

- *Step 1) Output Conditions Initial Matching and the Ouput Event(s) Counting from the input test patterns:*

The present output (OUTPUT) is '1', the last output value (OUTPUT'last_value) is '0', and the next to last output value (HAZARD_SCHEDULE1.OUTPUT_LAST_TWO_VALUE is '1'. In the function hazard detection, the original input pattern creates the output value '1', and the second neighbor input pattern creates the output value "01". Hence, the number of output events that are caused from the the original input pattern is one, and the number of the output events that are caused from the second neighbor input pattern is two. When all the above conditions are fulfilled, step 2 will not proceed until the present output (OUTPUT) is 'X' (i.e. the reset output).

The main purpose of step 2 is to avoid the false hazard detection and to ensure the package detects function 1 hazard correctly.

- *Step 2) Confirmation of the Function 1 Hazard Checking*:

The present output (OUTPUT) is 'X' (i.e. the reset output), the last output value (OUTPUT'last_value) is not equal to 'X', and the and the output events pattern (HAZARD_SCHEDULE1.OUTPUT_PATTERNS) that is caused by the three-tuple is "101-". The logic value '-' is the don't care logic value and it will only be substituted into the HAZARD_SCHEDULE1.OUTPUT_PATTERNS when no further output event is found in the function hazard detection. Even if all the above conditions are fulfilled, a function 1 hazard is not confirmed if the hazard information matches with the static hazard information described in the record type variable INFO. During the function hazard analysis, the designer must read all the static hazards found in single-input change hazard analysis to a record type constant INFO in the test circuit source file. If a potential 1 function hazard matches with the static hazard information, then this potential 1 function hazard is not considered as the function 1 hazard. However, if a potential function 1 hazard does not match with the static hazard information, then this potential function 1 hazard is the function 1 hazard and the assertion message is printed to the simulation output file.

3.6. VHDS Components Library

The last design unit in the VHDS design library is the VHDS components library. The VHDS components library collects some common gate models. Those models include AND gates, OR gates, and inverters. All of them use the nine-valued logic system. Therefore, they are compatible

with the nine-valued logic package and the nine-valued timing package. With the components library, the designer is able to implement any kind of gate level models easily. In addition, the components library is expandable. The designer can build his/her own models and add them to the existing VHDS design library. The next section discusses how to build a custom VHDS component primitive.

3.6.1. Component Primitives Design in the VHDS

A design primitive consists of different sections. From Figure 28, a primitive starts with the library and use clauses. Then comes the entity description, the generic description, and the port description of the model. The architectural body encloses the functionality of the model. Each function of the model is developed inside a VHDL process. Three processes are implemented in the architectural body of the model primitive. The first process describes the behavioral characteristic of the model. It also includes the nine-valued timing properties of the model. The undelayed output variable of the behavioral model is defined to be OUTPUT_SIG. The timing procedure called TIME_MODEL_SELECTION_1 should be inserted right after the behavioral description of the model.

The second process encloses a time unit. This time unit is an operator which updates the new timing output signal and computes the min and max of the VHDL model primitive. The function MAX_MIN_DELAY_COMPUTE represents the time unit.

The third process contains a spike detection procedure in the component primitive. The spike detection procedure is called SPIKE_DETECTION. By specifying the generic values of the pulse width margin of the 0-spike (GLITCH_0_TIME) and the 1-spike (GLITCH_1_TIME), the spike

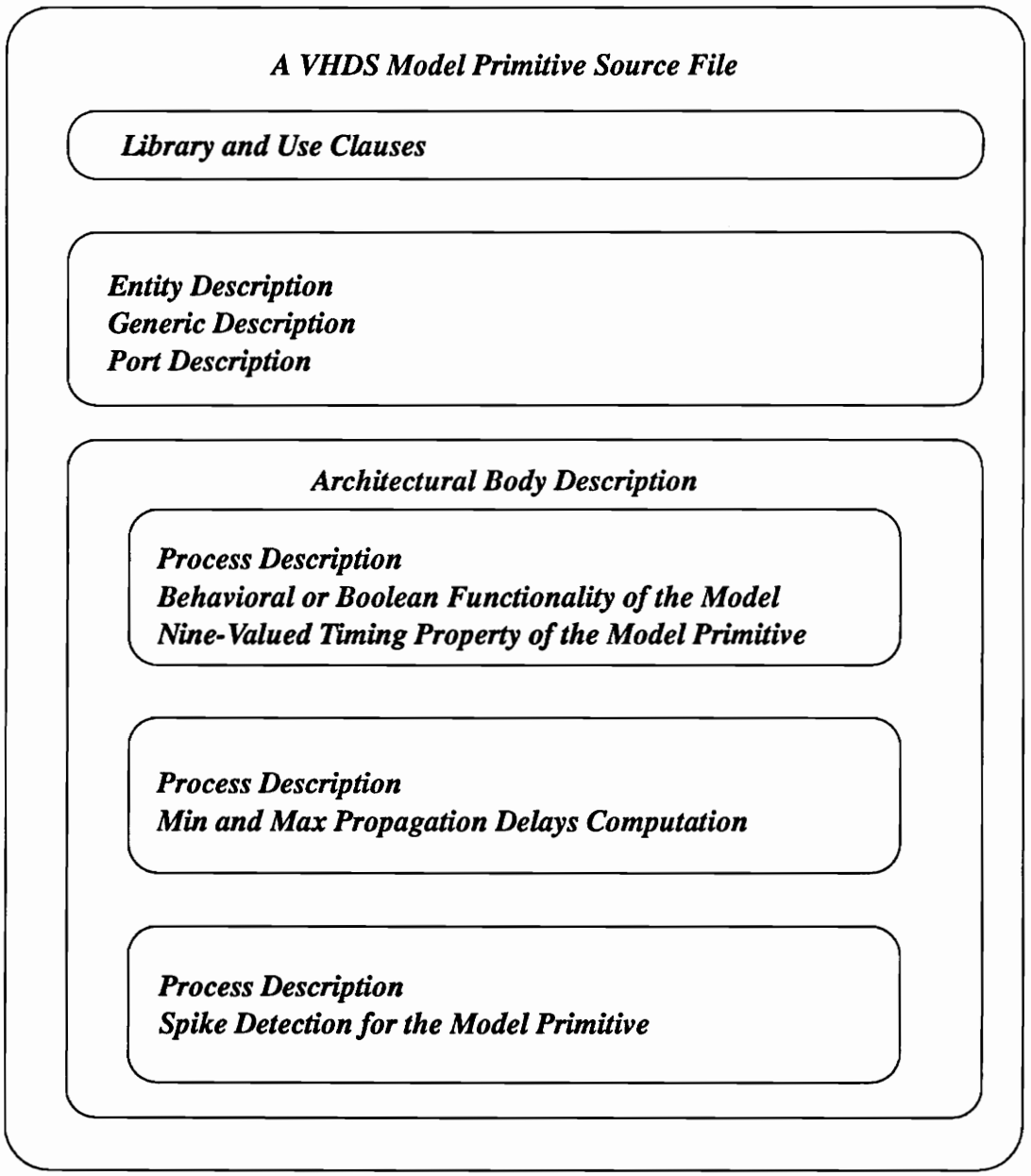


Figure 28. Block Diagram of a VHDS Component Primitive

detection routine will detect the existence of spikes in the component. As illustrated in Figure 17, this routine may help in locating the starting component which has hazard. Figure 29 shows a sample VHDL description of a model primitive. The main variation between different model primitives is the behavioral characteristic of the model as well as the generic and the port descriptions.

```
library VHDS_LIB;
use VHDS_LIB.MVL9_SYSTEM.all;
use VHDS_LIB.TIME_PACKAGE.all;

entity MODEL_PRIMITIVE is
generic(TP_01, TP_10: TIME; TP_1Z, TP_Z1, TP_0Z, TP_Z0, CONN_DELAY, GLITCH_0_TIME,
        GLITCH_1_TIME: TIME := 0 ns; OPTION: STRING := "OFF";
        CHOICE: TIME_MODEL_CHOICE := TRANSPORT DELAY);

port(INPUT: in MVL9 | MVL9_VECTOR; OUTPUT: out MVL9 | MVL9_VECTOR;
-- Input and Ouput can be defined as MVL9 or MVL9_VECTOR
     TIME_SPEC_IN: in PROP_DELAY_SPEC := (0 ns, 0 ns); -- Input Timing Signal
     TIME_SPEC_OUT: out PROP_DELAY_SPEC := (0 ns, 0 ns)); -- Output Timing Signal
end MODEL_PRIMITIVE;

architecture PURE_BEHAVIOR of MODEL_PRIMITIVE is
signal OUPUT_LAST_VALUE: MVL9;  -- This signal memorizes the most recent output value
begin

process(INPUT)
variable OUTPUT_SIG, OUTPUT_SIG_LAST_VALUE: MVL9;
variable LAST_DELAY_TIME: TIME := 0 ns; -- This variable memorizes the most recent gate delay value
begin

-- OUTPUT_SIG := BOOLEAN FUNCTIONALITY OF THE MODEL PRIMITIVE
-- (USER_SPECIFIED)

-- The VHDL nine-valued timing model.
TIME_MODEL_SELECTION_1(OUTPUT_SIG, OUTPUT, CHOICE, LAST_DELAY_TIME,
                       OUTPUT_SIG_LAST_VALUE, OUTPUT_LAST_VALUE,
                       TP_01, TP_10, TP_1Z, TP_Z1, TP_0Z, TP_Z0, CONN_DELAY);
end process;


-- This process is to insert a time unit that used to updates the output tming signal and to
-- compute the min and max of the VHDL model.

process (TIME_SPEC_IN)
begin
TIME_SPEC_OUT <= MAX_MIN_DELAY_COMPUTE(TP_01, TP_10, TP_1Z, TP_Z1, TP_0Z, TP_Z0,
                                       CONN_DELAY, OPTION, TIME_SPEC_IN);

end process;
```

```
-- This process is to detect the existence of spikes in the model primitive
process(OUTPUT_LAST_VALUE)
variable OUTPUT_LAST_EVENT: TIME := 0 ns; -- This variable holds the time of last output event.
begin
SPIKE_DETECTION(OUPUT_LAST_VALUE, OUTPUT_LAST_EVENT, GLITCH_0_TIME,
                        GLITCH_1_TIME);
end process;
end PURE_BEHAVIOR;
```

Figure 29. Sample VHDL Description of a VHDS Component Primitive

Chapter 4. VHDS Implementation in Logic Circuits

4.1 User-Specification Design in the VHDS

This chapter describes how to design the user-specification program so that it can interface properly with the VHDS. As in Figure 10, user-specification program divides into three parts. Part one is the input-specification file which is mentioned in section 3.4.1. Part two is the VHDL source file design for the VHDS. Part three is the test bench shell design for the VHDS. The source file design includes the VHDL description of the test circuit, the min and max timing computation design, and the hazard detection subroutines call. Test bench shell file includes the component description of the test circuit and the hazard patterns generator subroutines call. No designer-supplied arbitrary test vectors are required in the test bench shell.

Two test modes (STATIC and FUNCT) are allowed in the VHDS hazard analysis. Both test modes cannot be combined together to analyze a test circuit in one simulation task. In fact, two simulation tasks are required to analyze a test circuit with the single-input change hazard analysis and the two-input change (function) hazard analysis. The first simulation task should be the single-input change hazard analysis. After the comlpetion of the single-input change hazard analysis, the second simulation task is the function hazard analysis. During the function hazard analysis, the designer must read all the static hazards found in first simulation task to a record type constant INFO in the test circuit source file. This constant INFO will provide all the static

hazards information of the test circuit to the hazard detection package during the function hazard analysis.

4.1.1 VHDS Source File Design

Figure 30 is a black box model that illustrates the VHDS user-specification design. The VHDS source file is divided into six parts. The header part, the declaration part, the internal signals and types declaration part, the structural description part, the subroutine call part, and the configuration part. Figure 31 shows a general block diagram for the VHDS source file design.

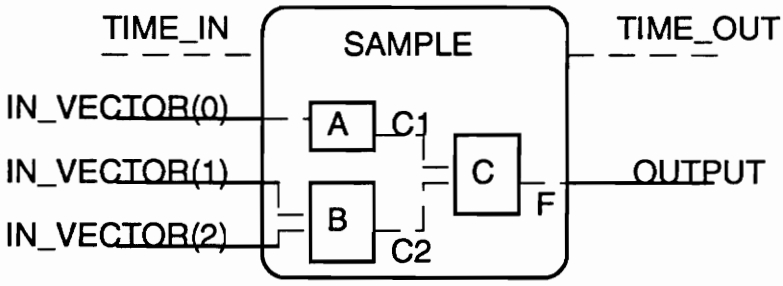


Figure 30. Illustration of the VHDS User-Specification Design From the Model SAMPLE

The header part contains all the library and use clause statements. They are described as follows:

```
library VHDS_LIB;
use VHDS_LIB.MVL9_SYSTEM.all;
use VHDS_LIB.TIME_PACKAGE.all;
use VHDS_LIB.PRE_HAZARD_TEST.all;
use VHDS_LIB.GATE_LIBRARY.all;
use VHDS_LIB.INPUT_DATA.all;
use VHDS_LIB.TEST_GENERATOR.all;
```

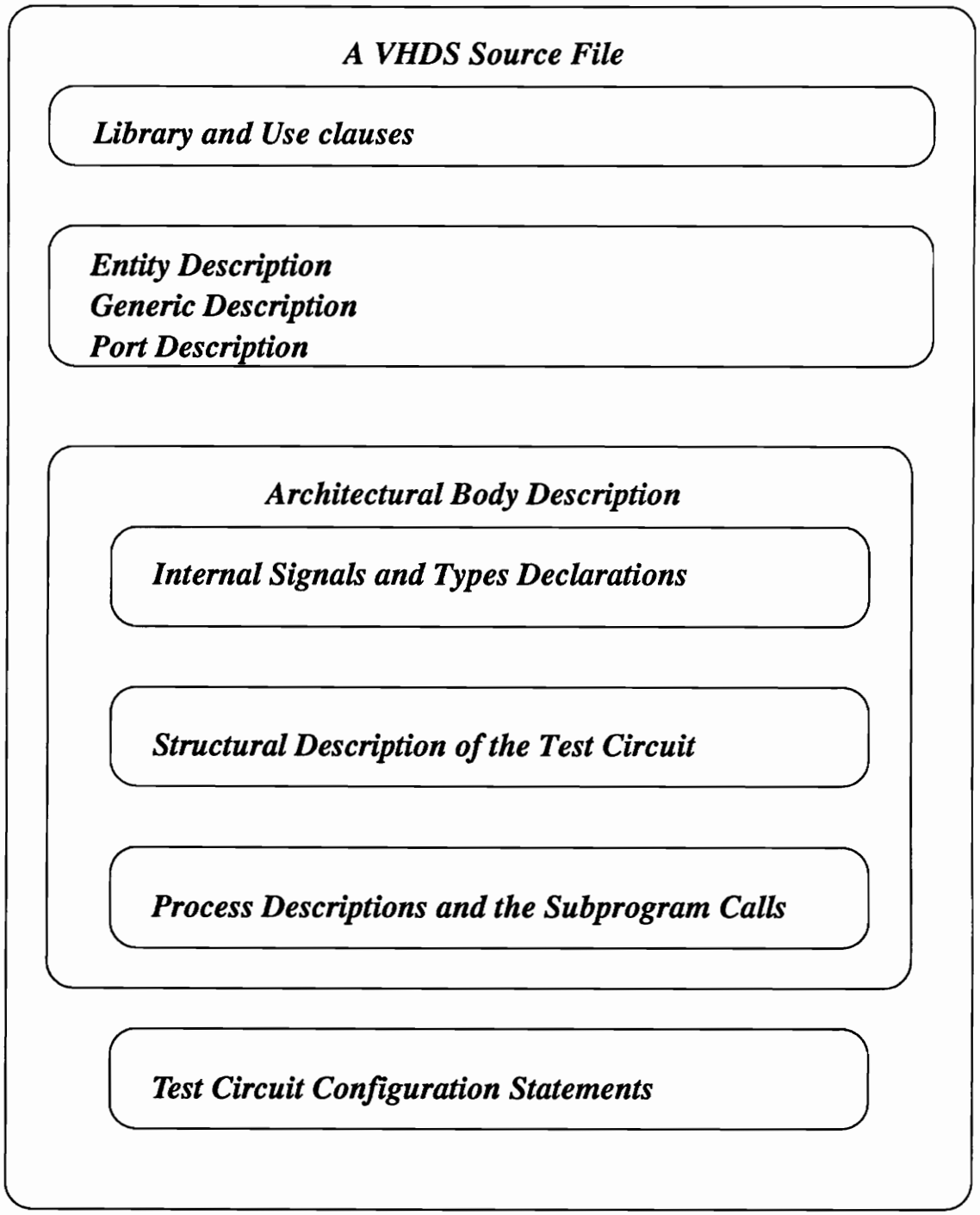


Figure 31. Block Diagram of the VHDS Source File Design

The statement "library VHDS_LIB" points to the design library VHDS_LIB. The statement "use VHDS_LIB.MVL9_SYSTEM.all" calls the nine-valued logic system from the design library unit VHDS_LIB. The statement "use VHDS_LIB.TIME_PACAKGE.all" calls the nine-valued timing package from the design library. The statement "use VHDS_LIB.PRE_HAZARD_TEST.all" calls the VHDS hazard detection package from the design unit. The statement "use VHDS_LIB.GATE_LIBRARY.all" calls the gate level component descriptions from the design unit. The statement "use VHDS_LIB.INPUT_DATA.all" calls the information contains in the input-specification file. That is, the constant value WORDLENGTH. Finally, the statement "use VHDS_LIB.TEST_GENERATOR.all" calls the VHDS hazard patterns generator from the design library.

The second part is the declaration region. It contains the entity description, the generic description, and the port declaration of the test circuit. For the single-input change hazard analysis, the entity SAMPLE is described as follows:

```
entity SAMPLE is
generic (GLITCH_0_TIME, GLITCH_1_TIME: TIME);
port(INPUT_VECTOR,PATT1,PATT2: in MVL9_VECTOR(0 to 2);
     OUTPUT: out MVL9_VECTOR;
     TIME_IN: in PROP_DELAY_SPEC := (0 ns, 0 ns);
     TIME_OUT: inout PROP_DELAY_SPEC := (0 ns, 0 ns));
end SAMPLE;
```

For the function hazard analysis, the entity SAMPLE is described as follows:

```
entity SAMPLE is
generic (GLITCH_0_TIME, GLITCH_1_TIME: TIME);
port(INPUT_VECTOR,PATT1,PATT2: in MVL9_VECTOR(0 to 2);
     OUTPUT: out MVL9_VECTOR;
     STATE_VALUE_TABLE: inout MVL9_STATE_LOGIC_VECTOR(0 to 2**WORDLENGTH -
     1);
     TIME_IN: in PROP_DELAY_SPEC := (0 ns, 0 ns);
```

```
    TIME_OUT: inout PROP_DELAY_SPEC := (0 ns, 0 ns));
end SAMPLE;
```

The entity declaration describes the overall context of the test model. Two standard generic values are declared in the generic statement. They are the pulse width margin for a 0-spike existing in any component (GLITCH_0_TIME), and the pulse width margin for a 1-spike existing in any component (GLITCH_1_TIME).

The port description describes all the inputs and outputs that are associated with the entity declaration. Signal IN_VECTOR describes all the primary inputs in vector form. Signals PATT1 and PATT2 are the standard VHDS declaration and they represent the original input test patterns and the second input test patterns from the HPG, respectively. The length of the vector signals IN_VECTOR, PATT1, and PATT2 must be equal to the constant value WORDLENGTH defined in the input-specification file. Otherwise, inaccurate and rare results will be produced. The signal OUTPUT describes the output behavior of the test model. In the function hazard analysis, an additional inout signal STATE_VALUE_TABLE is added. This inout signal holds all the input logic values and the corresponding output logic values of the test circuit. The signals TIME_IN and TIME_OUT describe the input and output timing behavior of the model respectively. That is, they hold the min and max propagation delay of the model. Signals STATE_VALUE_TABLE and TIME_OUT are of mode *inout* because they are required to be read back in as an input parameter in the VHDS.

The next step is the architectural body of the declared entity. The architectural body is divided into three regions, the internal signals and types declaration, the structural description of the model, and the VHDS hazard detection process. The internal signals are used to interconnect the

individual sub-models, components, or the output pins. For example, signal C1, C2, and F are the internal signals of the model SAMPLE.

Types declaration in the architectural body declares the vector subtype of the PROP_DELAY_SPEC. For the components in the test model that have more than one timing input port, a vector subtype of the PROP_DELAY_SPEC is required. One example is the model B and model C in the model SAMPLE. Internal signals BR and CR define the internal timing signals that are associated with model B and model C respectively. The VHDL description for the internal signals and types declarations for the model SAMPLE are listed below.

```
subtype PROP_DELAY_SPEC_VECTOR_2 is PROP_DELAY_SPEC_VECTOR(1 to 2);
signal BR, CR: PROP_DELAY_SPEC_VECTOR_2 := ((0 ns, 0 ns), (0 ns, 0 ns));
signal C1, C2, F: MVL9;
```

The main part in the architectural body is the VHDL structural description of the test model. In VHDS, the structural body description describes the structural logic connection of the model as well as the structural timing connection of the model. Hence, the VHDL structural description for the model SAMPLE is listed as below.

```
OUTPUT <= F;
BR <= TIME_IN & TIME_IN;

G1: A
generic map(2 ns, 2 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                GLITCH_1_TIME => GLITCH_1_TIME);
port map(INPUT => IN_VECTOR(0), OUTPUT => C1;
          TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => CR(1));

G2: B
generic map(4 ns, 4 ns, GLITCH_0_TIME => GLITCH_0_TIME,
```

```
            GLITCH_1_TIME => GLITCH_1_TIME);
port map(INPUT(1) => IN_VECTOR(1), INPUT(2) => IN_VECTOR(2), OUTPUT => C2;
        TIME_SPEC_IN => TIME_BUS_FUN(BR), TIME_SPEC_OUT => CR(2));

G3: C
generic map(5 ns, 5 ns,  GLITCH_0_TIME => GLITCH_0_TIME,
            GLITCH_1_TIME => GLITCH_1_TIME);
port map(INPUT(1) => C1, INPUT(2) => C2, OUTPUT => F;
        TIME_SPEC_IN => TIME_BUS_FUN(CR),
        TIME_SPEC_OUT => TIME_OUT);
```

The signal TIME_IN is the input timing signal and has the initial value (0 ns, 0 ns). The signal TIME_OUT is the output timing signal and always holds the updated min and max of the model SAMPLE. The function TIME_BUS_FUN is a time bus resolved unit that resolves multiple timing input signals.

The next part in the source file design is the process(es) description and subroutine call. For the single-input change hazard analysis, a process which contains the hazard detection package is created. This process is placed after the structural description of the model. The process sensitivity list contains the input signal (IN_VECTOR) and the output signal (F). Also, a variable called HAZARD_SCHEDULE1 of a composite type HAZARD_VAR must be declared in the process so that the logic and the timing values can always be memorized in the hazard detection procedure. The variable HAZARD_SCHEDULE1 has the initial value HAZARD_CONSTANT. The VHDL process description for the single-input change hazard analysis for the model SAMPLE is listed as follows.

```
HAZARD_DETECTION1: process(F, IN_VECTOR)
variable HAZARD_SCHEDULE1: HAZARD_VAR := HAZARD_CONSTANT;
begin

HAZARDS_PATTERNS_TEST(F,"OUTPUT", HAZARD_SCHEDULE1, IN_VECTOR,
                                PATT1, PATT2, TIME_OUT);
end process HAZARD_DETECTION1;
```

For the two-input change (function) hazard analysis, two processes are created. The first process has no sensitivity list. Hence, a wait statement is required. The first process contains a procedure PRE_PROCESS_DATA which is used to write all the input logic values and the output logic values into the inout signal STATE_VALUE_TABLE when the procedure PRE_PROCESS_INJECTION in the test bench shell injects the input logic values to the test circuit. A variable DATA is also defined in the process so that the index number of input logic values can be memorized. The second process is similar to the single-input change hazard detection process except the hazard detection procedure will be activated after the inout signal STATE_VALUE_TABLE completely memorizes all the input logic values and the output logic values. Thus, the starting time for the hazard detection procedure in the two-input change hazard analysis is equal to PRE_TEST_BEGIN_TIME(TIME_OUT). In addition, the constant value INFO, which describes all the static hazards found in the test circuit, needs to be defined when static hazards are found in the single-input change hazard analysis. The process descriptions for the two-input change hazard analysis for the model SAMPLE are listed below:

```
HAZARD_DETECTION: process
variable DATA: INTEGER := 0;
begin
PRE_PROCESS_DATA(IN_VECTOR, F, TIME_OUT, DATA, STATE_VALUE_TABLE);
wait on F, IN_VECTOR;
end process HAZARD_DETECTION;

HAZARD_DETECTION1: process(F, IN_VECTOR)
variable HAZARD_SCHEDULE1: HAZARD_VAR:= HAZARD_CONSTANT;
constant INFO: STATIC_INFO_VECTOR(0 to 1) := .....;
begin
 if (now >= PRE_TEST_BEGIN_TIME(TIME_OUT)) then
HAZARDS_PATTERNS_TEST(F, "OUTPUT", HAZARD_SCHEUDLE1, IN_VECTOR, PATT1,
PATT2, TIME_OUT, INFO);
end if;
end process HAZARD_DETECTION1;
```

The last part in the source file design is the configuration statements association. The configuration statements for the model SAMPLE is listed below.

```
configuration CFG_SAMPLE of SAMPLE is
for STRUCTURAL
        for G1: A
        use entity VHDS_LIB.A(PURE_BEHAVIOR);
        end for;

        for G2: B
        use entity VHDS_LIB.B(PURE_BEHAVIOR);
        end for;

        for G3: C
        use entity VHDS_LIB.C(PURE_BEHAVIOR);
        end for;

end for;
end CFG_SAMPLE;
```

The design of the source files for the single-input change hazard analysis and the two-input change hazard analysis has already presented in detail. The next section will extend the VHDS implementation to the test bench shell design.

4.1.2 VHDS Test Bench Shell Design

The test bench shell design is similar to the source file design. It is divided into six parts. The header part, the declaration part, the component instantiation part, the signals description part, the process(es) description and the subroutine call part, and the configuration part. Figure 32 shows a general block diagram for the VHDS test bench shell design.

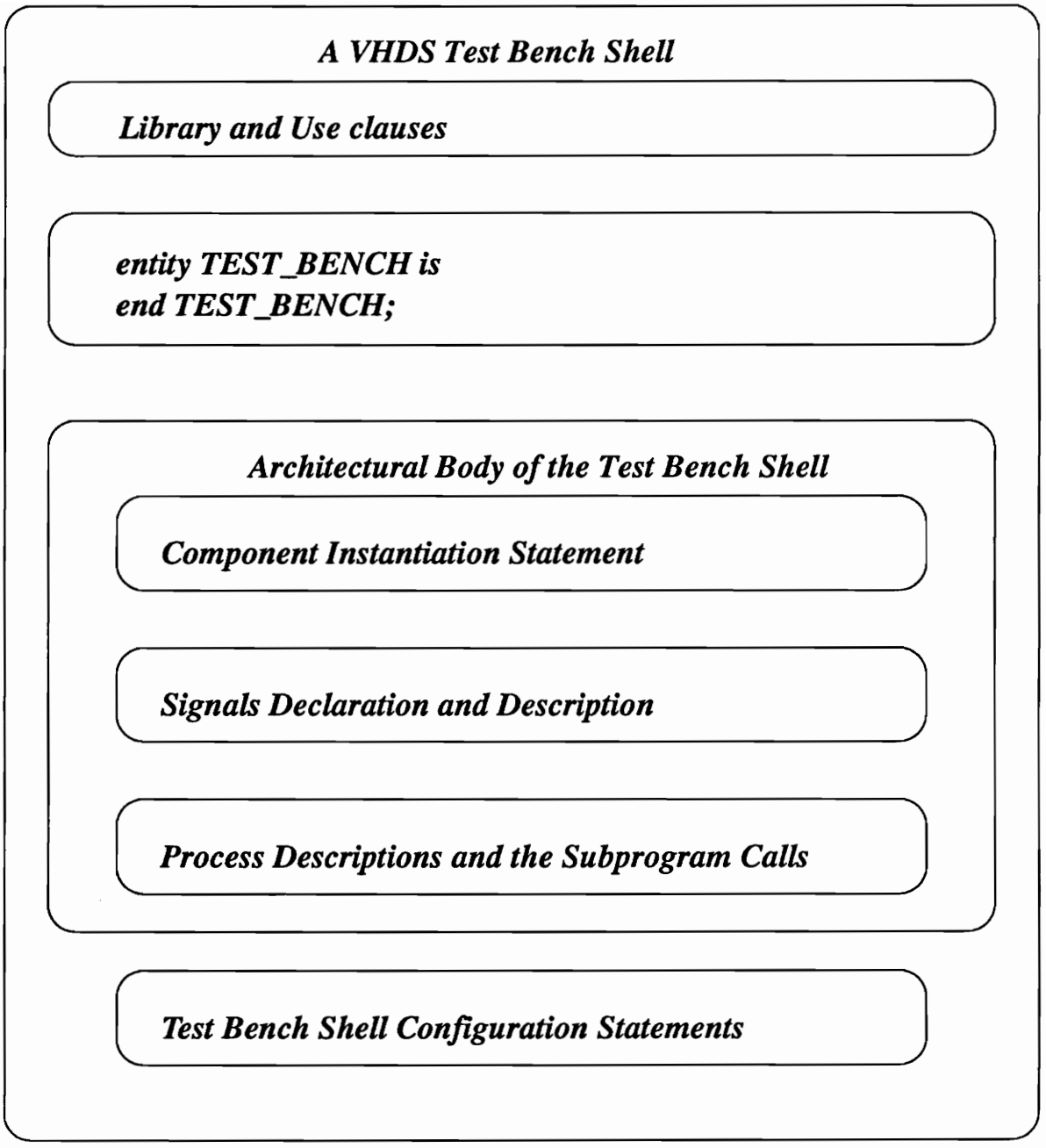


Figure 32. Block Diagram of the VHDS Test Bench Shell Design

The library and the use clauses for the test bench shell are described as follows:

```
library VHDS_LIB;
use VHDS_LIB.MVL9_SYSTEM.all;
use VHDS_LIB.TIME_PACKAGE.all;
use VHDS_LIB.INPUT_DATA.all;
use VHDS_LIB.TEST_GENERATOR.all;
```

The declaration region in the test bench shell contains the statement "***entity*** *TEST_BENCH* ***is end*** *TEST_BENCH;*" Inside the architectural body, they are the component instantiation unit, the signals declaration and description unit, and the VHDS hazard pattern generator unit. The component description for the model SAMPLE under the single-input change hazard analysis is listed as follows:

```
component SAMPLE
generic (GLITCH_0_TIME, GLITCH_1_TIME: TIME);
port (IN_VECTOR, PATT1, PATT2: in MVL9_VECTOR(0 to 2);
      OUTPUT: out MVL9;
      TIME_IN: in PROP_DELAY_SPEC;
      TIME_OUT: inout PROP_DELAY_SPEC);
end component;
```

Similarly, the component description for the model SAMPLE under the two-input change hazard analysis is listed as follows:

```
component SAMPLE
generic (GLITCH_0_TIME, GLITCH_1_TIME: TIME);
port (IN_VECTOR, PATT1, PATT2: in MVL9_VECTOR(0 to 2);
      OUTPUT: out MVL9;
      STATE_VALUE_TABLE: inout MVL9_STATE_LOGIC_VECTOR(
      0 to 2**WORDLENGTH - 1);
      TIME_IN: in PROP_DELAY_SPEC;
      TIME_OUT: inout PROP_DELAY_SPEC);
end component;
```

The next unit is the signals description. In this unit, the signals described in the component, *the starting signals* INIT and INIT1, and other optional signals and constants are defined in this region. The VHDL description for the model SAMPLE is listed as follows:

```
signal IN_VECTOR, PATT1, PATT2: MVL9_VECTOR(0 to 2);
signal OUTPUT: MVL9;
signal TIME_IN, TIME_OUT: PROP_DELAY_SPEC:= ((0 ns, 0 ns), (0 ns, 0 ns));
```

For the single-input change hazard anlysis, the starting signal is declared as follows:

```
signal INIT: BIT := '0';
```

For the two-input change hazard analysis, three additional signal declarations are needed. They are defined as follows:

```
signal INIT, INIT1: BIT := '0';
signal STATE_VALUE_TABLE: MVL9_STATE_LOGIC_VECTOR(
        0 to 2**WORDLENGTH - 1);
signal IN_VECTOR1, IN_VECTOR2: MVL9_VECTOR(0 to WORDLENGTH -1);
```

The signals IN_VECTOR1 and IN_VECTOR2 are the time-multiplexing signals of the input signal IN_VECTOR. They are listed as follows:

```
IN_VECTOR <=    IN_VECTOR1 when not IN_VECTOR1'quiet else
                IN_VECTOR2 when not IN_VECTOR2'quiet else
                IN_VECTOR;
```

If the designer wants to know the total number of patterns for the single-input change hazard analysis and the total number of original un-reduced three-tuple test patterns for the two-input change hazard analysis, two constant statements can be added in this region. They are respectively defined as follows:

```
constant TOTAL_PATTERN_STATIC : INTEGER :=
        TOTAL_STATIC_PATTERNS(WORDLENGTH);
constant TOTAL_PATTERN_FUNCTION : INTEGER :=
        TOTAL_FUNCTION_PATTERNS(WORDLENGTH);
```

After all the signals have been defined, the next step is the generic map and the port map associations. The VHDL description for the model SAMPLE under the single-input change hazard analysis is presented below:

```
L1: SAMPLE generic map (2 ns, 2 ns)
port map(IN_VECTOR, PATT1, PATT2, OUTPUT, TIME_IN, TIME_OUT);
```

The VHDL description for the model SAMPLE under the function hazard analysis is presented below:

```
L1: SAMPLE generic map (2 ns, 2 ns)
port map(IN_VECTOR, PATT1, PATT2, OUTPUT, STATE_VALUE_TABLE, TIME_IN,
TIME_OUT);
```

Since the simulation time 0 ns has been used for circuit initialization and the min/max propagation delays computation, the generator process which is controlled by the signal INIT, should begin at 1 ns.

```
INIT <= '1' after 1 ns;
```

The most important part in the test bench shell is the hazard pattern subroutine call process. For the single-input change hazard analysis, a VHDL process with signal INIT as the sensitivity list is created. Inside this process, a pair of subprograms is required and they must be placed in order. The first one is the pattern generator subprogram. The second one is the pattern injection subprogram. The VHDL hazard detection process for the model SAMPLE under the single-input change hazard analysis (STATIC) is listed as below:

```
HAZARD_INJECT1:  process (INIT)
variable TEST_PATT1: MVL9_VECTOR_TUPLE_VECTOR
                        (0 to TOTAL_PATTERN_STATIC - 1);
begin

TEST_PATT1 := STATIC_PATTERN_GENERATOR(WORDLENGTH);

STATIC_PATTERN_TESTER(TEST_PATT1, TIME_OUT, PATTERN1 => PATT1,
                                  PATTERN2 => PATT2,
                                  OUT_PATTERN => IN_VECTOR,
                                  MODE => STATIC);
end process HAZARD_INJECT1;
```

The value TEST_PATT1 holds the patterns that are generated from the single-input change hazard pattern generator (STATIC_PATTERN_GENERATOR). The procedure STATIC_PATTERN_TESTER injects all the possible single-input change patterns one by one to the test circuit for the potential single-input change hazard detection.

For the two-input change (function) hazard analysis, two processes are required in the test bench shell. The first process is called INJECTION. The sensitivity list of this process is the starting signal INIT. This process activates the procedure PRE_PROCESS_INJECTION and monitors the starting signal INIT1. The second process is called TESTING. This sensitivity list of this process is the starting signal INIT1. Inside this process, three subprograms are required and must be placed in order. The first subprogram is the patterns generator subprogram. The second subprogram computes the number of reduced two-input change hazard test patterns. The third subprogram is called PRE_FUNCTION_PATTERN_TESTER. This procedure injects the reduced two-input change test patterns into the test circuit.

The VHDL process descriptions for the model SAMPLE under the two-input change hazard detection are listed as follows:

```
INJECTION: process(INIT)
begin
PRE_PROCESS_INJECTION(TIME_OUT, IN_VECTOR1);

if (now = 1 ns) then
INIT1 <= transport '1' after (PRE_TEST_BEGIN_TIME(TIME_OUT) - 1 ns);
end if;
end process INJECTION;

TESTING: process(INIT1)
variable TEST_PATT2: MVL9_VECTOR_TRIPLE_VECTOR(
                        0 to TOTAL_PATTERN_FUNCTION(WORDLENGTH) - 1);
variable NUMBER : NATURAL := 0;
begin

TEST_PATT2 := FUNCTION_PATTERN_GENERATOR(WORDLENGTH);
NUMBER := PRE_PROCESS_COUNT(TEST_PATT2, STATE_VALUE_TABLE);

PRE_FUNCTION_PATTERN_TESTER(
PRE_PROCESS_FUNCTION_GENERATOR(TEST_PATT2,  STATE_VALUE_TABLE,
NUMBER), TIME_OUT, NUMBER, PATTER1 => PATT1, PATTERN2 => PATT2,
OUT_PATTERN => IN_VECTOR2, MODE => FUNCT);
end process TESTING;
```

The variable NUMBER declared in the process TESTING contains the total number of reduced two-input change hazard test patterns for the given test circuit. The values TEST_PATT2 holds all the original three-tuple test patterns that are generated from the two-input change hazard pattern generator (FUNCTION_PATTERN_GENERATOR).

The last part in the test bench shell design is the configuration statements description. Similar to the configuration statements defined in the source file, the configuration statements in the test bench shell is listed below.

```
use WORK.all;
configuration CON_SAMPLE of TEST_BENCH is
for SAMPLETB
        for all: SAMPLE
```

```
        use entity WORK.SAMPLE(STRUCTURAL);
        end for;
end for;
end CON_SAMPLE;
```

4.1.3 Synopsys Commands File for Running the VHDS.

Up to this point, the designer should able to design his/her own input-specification file, the VHDS source file, and the test bench shell file. Since the whole VHDS is running under the Synopsys V2.1 environment, the following is the listing of the Synopsys commands sequence that makes the whole implementation working properly.

```
vhdlan -w vhds_dir data3.vhd
vhdlan -w vhds_dir tpgred.vhd
vhdlan -w vhds_dir phaz1_test.vhd
vhdlan std_cir1.vhd
vhdlan std_test1.vhd
vhdlsim -i synvhdl.com configuration_name_in_the_test_bench_shell
```

The content of the simulation include file "synvhdl.com" is listed as follows:

```
open out.11
ls -t -v test_bench > out.11
cd test_bench
monitor active *'signal
set BASE BINARY
trace *'signal
logtime -e out.11
list > out.11
run > out.11
quit
```

The file "data3.vhd" is the input-specification file that contains the constant value WORDLENGTH (WORDLENGTH = 3). The file "tpgred.vhd" is the VHDS hazard pattern

generator package (HPG). The file "phaz1_test.vhd" is the VHDS hazard detection package. The file "std_cir1.vhd" is the VHDS source file (or test circuit file). The file "std_test1.vhd" is the test bench shell for the test circuit. The file "synvhdl.com" is a simulation include file that contains the Synopsys simulation commands sequence for the VHDL simulation. The Synopsys output file is called "out.11". More details about the Synopsys commands file can be found in reference [16].

4.1.4. VHDS Results Observation

As mentioned in section 3.1, the VHDS produces two files during the VHDS analyzing process. One is the Synopsys output file and the other one is the VHDS report summary. The Synopsys output file provides detailed information about every simulation step while the VHDS report summary contains the information that is related to the existence of hazards inside the test circuit. More details about the VHDS report summary can be found in the next chapter.

Chapter 5. Design Examples with VHDS

5.1 Introduction

This chapter will present five design examples that were analyzed under the VHDS. Each section will present one example test circuit. Section 5.2 discusses a three-input circuit example. Section 5.3 examines an example test circuit with four primary inputs. Section 5.4 presents an example circuit with five primary inputs. Section 5.5 discusses an example with six primary inputs. Finally, section 5.6 reveals an example that illustrates the dynamic hazard. In each section, detailed descriptions of the source files and the test bench shell will be discussed. Also, the VHDS report summary for each test circuit will be presented. Section 5.7 concludes the results of the test circuit and discusses the overall performance of the VHDS.

5.2 A Test Circuit with Three Primary Inputs

Figure 33 shows a test circuit with three primary inputs. This test circuit is extracted from **[12]**. The input of the test circuit is a three-bit MVL9 vector called TMP_IN_VECTOR. The output of the test circuit is a MVL9 logic signal F. From the VHDS analysis, a hazard report summary is generated. From the summary, the min and max values of the test circuit are 4 ns and 6 ns, respectively. Also, as in the single-input change hazard report summary and the two-input change hazard report summary, the VHDS found one static 1 hazard and one function 0 hazard existed in the test circuit. The static 1 hazard exists in the test circuit when the input combinations of

TMP_IN_VECTOR(0 to 2) changes from "111" to "110". The static hazard pulse has a duration of 2 ns. A function 0 hazard exists when the input combinations of TMP_IN_VECTOR(0 to 2) change from "000" to "101". The function hazard has a duration of 2 ns.

The VHDS source files, the VHDS test bench shells, and the VHDS report summaries for the single-input change and the two-input change hazard analysis are enclosed in the following pages.

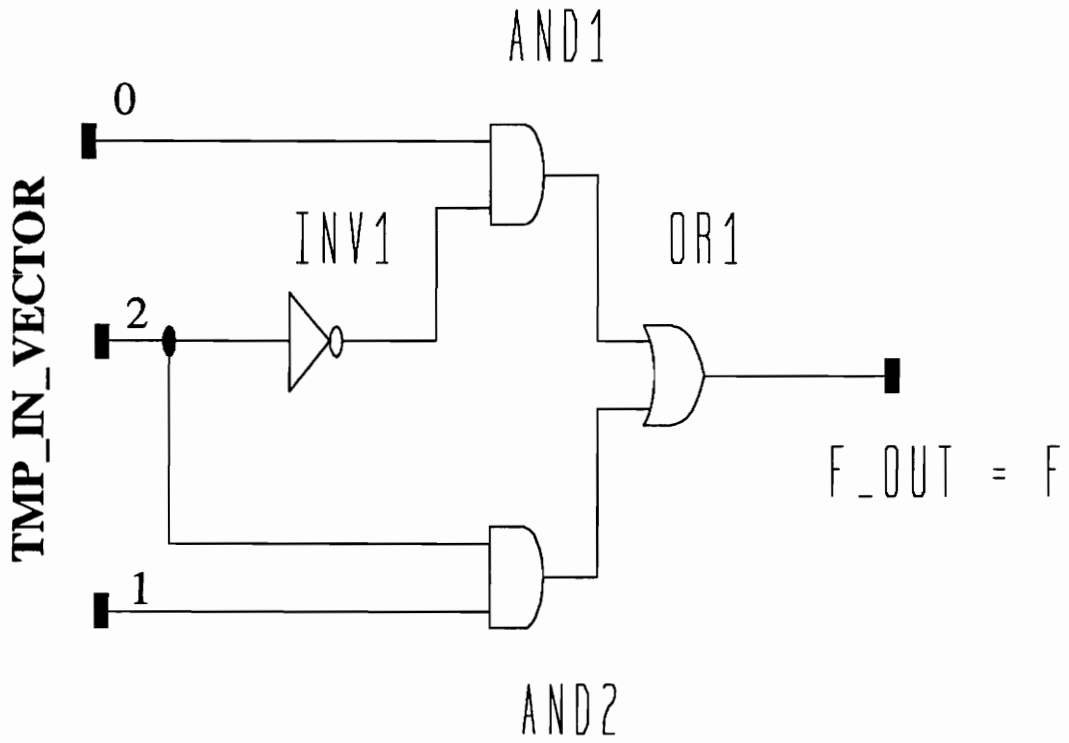


Figure 33. A Test Circuit with Three Primary Inputs

-- VHDS Source File design for Single-Input Change Hazard Analysis

```
library VHDS_LIB;

use VHDS_LIB.MVL9_SYSTEM.all;
use VHDS_LIB.TIME_PACKAGE.all;
use VHDS_LIB.PRE_HAZARD_TEST.all;
use VHDS_LIB.GATE_LIBRARY.all;
use VHDS_LIB.INPUT_DATA.all;
use VHDS_LIB.TEST_GENERATOR.all;

entity HAZ_1 is

generic (GLITCH_0_TIME, GLITCH_1_TIME: TIME);

port(TMP_IN_VECTOR: in MVL9_VECTOR(0 to 2);
     PATT1, PATT2: in MVL9_VECTOR(0 to 2);
     F: out MVL9;
    TIME_IN: in PROP_DELAY_SPEC;
    TIME_OUT: inout PROP_DELAY_SPEC);
end HAZ_1;


architecture STRUCTURAL of HAZ_1 is

--*****
subtype PROP_DELAY_SPEC_VECTOR_2 is PROP_DELAY_SPEC_VECTOR(1 to 2);
signal AR: PROP_DELAY_SPEC_VECTOR_2 := ((0 ns, 0 ns),(0 ns, 0 ns));
signal BR: PROP_DELAY_SPEC_VECTOR_2 := ((0 ns, 0 ns),(0 ns, 0 ns));
signal CR: PROP_DELAY_SPEC_VECTOR_2 := ((0 ns, 0 ns),(0 ns, 0 ns));
signal TIME_1: PROP_DELAY_SPEC := (0 ns, 0 ns);
--*****

signal C1, C2, C3, F_OUT: MVL9;

begin

F <= F_OUT;


INV1: INVGATE
        generic map(2 ns, 2 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => TMP_IN_VECTOR(2), OUTPUT => C1,
                    TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => BR(2));


AND1: AND2GATE
        generic map(2 ns, 2 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => C1,  INPUT(2) => TMP_IN_VECTOR(0), OUTPUT => C2,
                    TIME_SPEC_IN => TIME_BUS_FUN(BR),   TIME_SPEC_OUT => AR(1));
```

```
AND2: AND2GATE
        generic map(2 ns, 2 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => TMP_IN_VECTOR(1), INPUT(2) => TMP_IN_VECTOR(2),
                    OUTPUT => C3,
                    TIME_SPEC_IN => TIME_BUS_FUN(CR), TIME_SPEC_OUT => AR(2));

OR1: OR2GATE
        generic map(2 ns, 2 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => C2, INPUT(2) => C3, OUTPUT => F_OUT,
                    TIME_SPEC_IN => TIME_BUS_FUN(AR),
                    TIME_SPEC_OUT => TIME_OUT);


HAZARD_DETECTION:process(F_OUT,TMP_IN_VECTOR)
variable HAZARD_SCHEDULE1 : HAZARD_VAR := HAZARD_CONSTANT;

begin

HAZARDS_PATTERNS_TEST(OUTPUT => F_OUT, NAME => "F",
HAZARD_SCHEDULE1 => HAZARD_SCHEDULE1,
PATTERNS => TMP_IN_VECTOR, PATT1 => PATT1,
PATT2 => PATT2, TIME_DELAY => TIME_OUT);


end process HAZARD_DETECTION;

end STRUCTURAL;

configuration CFG_HAZ_1 of HAZ_1 is
        for STRUCTURAL

        for INV1: INVGATE
        use entity VHDS_LIB.INVGATE(PURE_BEHAVIOR);
        end for;

        for AND1,AND2: AND2GATE
        use entity VHDS_LIB.AND2GATE(PURE_BEHAVIOR);
        end for;

        for OR1: OR2GATE
        use entity VHDS_LIB.OR2GATE(PURE_BEHAVIOR);
        end for;

end for;
end CFG_HAZ_1;
```

-- VHDS Test Bench Shell Design for Single-Input Change Hazard Analysis

```
library VHDS_LIB;

use VHDS_LIB.MVL9_SYSTEM.all;
use VHDS_LIB.TIME_PACKAGE.all;
use VHDS_LIB.TEST_GENERATOR.all;
use VHDS_LIB.INPUT_DATA.all;

entity TEST_BENCH is
end TEST_BENCH;

architecture HAZ1TB of TEST_BENCH is

component HAZ_1
generic  (GLITCH_0_TIME, GLITCH_1_TIME: TIME);

port(TMP_IN_VECTOR: in MVL9_VECTOR(0 to 2);
     PATT1, PATT2: in MVL9_VECTOR(0 to 2);
     F: out MVL9;
     TIME_IN: in PROP_DELAY_SPEC;
     TIME_OUT: inout PROP_DELAY_SPEC);
end component;

signal TMP_IN_VECTOR: MVL9_VECTOR(0 to 2);
signal PATT1, PATT2: MVL9_VECTOR(0 to 2);
signal F: MVL9;
signal TIME_IN: PROP_DELAY_SPEC := (0 ns, 0 ns);
signal TIME_OUT: PROP_DELAY_SPEC:= (0 ns, 0 ns);
constant TOTAL_PATTERN_STATIC: INTEGER := TOTAL_STATIC_PATTERNS(WORDLENGTH);

signal INIT: BIT := '0';


begin

-----------------------------------------------
-- Generic Map Association
-----------------------------------------------

L1: HAZ_1 generic map(2 ns,2 ns)
port map(TMP_IN_VECTOR,PATT1, PATT2, F,TIME_IN,TIME_OUT);


INIT <= '1' after 1 ns;

TEST:process(INIT)
variable RESULT1: MVL9_VECTOR_TUPLE_VECTOR(0 to TOTAL_PATTERN_STATIC - 1);

begin

RESULT1 := STATIC_PATTERN_GENERATOR(WORDLENGTH);
```

```
STATIC_PATTERN_TESTER(RESULT1, TIME_OUT, PATTERN1 => PATT1, PATTERN2 => PATT2,
                                    OUT_PATTERN => TMP_IN_VECTOR, MODE => STATIC);

end process TEST;

end HAZ1TB;


use WORK.all;
configuration CON_HAZ1 of TEST_BENCH is
for HAZ1TB
        for all: HAZ_1
        use entity WORK.HAZ_1(STRUCTURAL);
        end for;
end for;
end CON_HAZ1;
```

-- VHDS Report Summary for the Single-Input Change Hazard Analysis

```
        Test Circuit Statistics:
        Number of Primary Inputs: 3
Minimum Propagation Delay of the test circuit with respect to F: 4 NS
Maximum Propagation Delay of the test circuit with respect to F: 6 NS

************************** VHDS Report Summary ************************

=======================================
Static 1 hazard is detected at     571 ns on F
559 NS        111
565 NS        110
563 NS         1      F
569 NS         0      F
571 NS         1      F
```

-- VHDS Source File design for Two-Input Change Hazard Analysis

```
library VHDS_LIB;

use VHDS_LIB.MVL9_SYSTEM.all;
use VHDS_LIB.TIME_PACKAGE.all;
use VHDS_LIB.PRE_HAZARD_TEST.all;
use VHDS_LIB.GATE_LIBRARY.all;
use VHDS_LIB.INPUT_DATA.all;
use VHDS_LIB.TEST_GENERATOR.all;

entity HAZ_1 is
generic (GLITCH_0_TIME, GLITCH_1_TIME: TIME);

port(TMP_IN_VECTOR: in MVL9_VECTOR(0 to 2);
   PATT1, PATT2: in MVL9_VECTOR(0 to 2);
   F: out MVL9;
   STATE_VALUE_TABLE: inout MVL9_STATE_LOGIC_VECTOR(0 to 2**WORDLENGTH - 1);
   TIME_IN: in PROP_DELAY_SPEC;
   TIME_OUT: inout PROP_DELAY_SPEC);
end HAZ_1;


architecture STRUCTURAL of HAZ_1 is

--*****
subtype PROP_DELAY_SPEC_VECTOR_2 is PROP_DELAY_SPEC_VECTOR(1 to 2);
signal AR: PROP_DELAY_SPEC_VECTOR_2 := ((0 ns, 0 ns),(0 ns, 0 ns));
signal BR: PROP_DELAY_SPEC_VECTOR_2 := ((0 ns, 0 ns),(0 ns, 0 ns));
signal CR: PROP_DELAY_SPEC_VECTOR_2 := ((0 ns, 0 ns),(0 ns, 0 ns));
--*****

signal C1, C2, C3, F_OUT: MVL9;

begin

F <= F_OUT;



INV1: INVGATE
        generic map(2 ns, 2 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => TMP_IN_VECTOR(2), OUTPUT => C1,
                TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => BR(2));

AND1: AND2GATE
        generic map(2 ns, 2 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => C1,  INPUT(2) => TMP_IN_VECTOR(0), OUTPUT => C2,
                TIME_SPEC_IN => TIME_BUS_FUN(BR), TIME_SPEC_OUT => AR(1));

AND2: AND2GATE
        generic map(2 ns, 2 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
```

```
        port map(INPUT(1) => TMP_IN_VECTOR(1),  INPUT(2) => TMP_IN_VECTOR(2),
                 OUTPUT => C3,
                 TIME_SPEC_IN => TIME_BUS_FUN(CR), TIME_SPEC_OUT => AR(2));

OR1: OR2GATE
        generic map(2 ns, 2 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                    GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => C2, INPUT(2) => C3, OUTPUT => F_OUT,
                 TIME_SPEC_IN => TIME_BUS_FUN(AR),  TIME_SPEC_OUT => TIME_OUT);


HAZARD_DETECTION:process
variable DATA: INTEGER := 0;
begin

PRE_PROCESS_DATA(TMP_IN_VECTOR, F_OUT, TIME_OUT, DATA, STATE_VALUE_TABLE);

wait on F_OUT, TMP_IN_VECTOR;
end process HAZARD_DETECTION;

HAZARD_DETECTION1:process(F_OUT, TMP_IN_VECTOR)
variable HAZARD_SCHEDULE1 : HAZARD_VAR := HAZARD_CONSTANT;
constant INFO : STATIC_INFO_VECTOR(0 to 1)  := (("---", "---", 0 ns, 0 ns), ("111", "110", 4 ns, 2 ns));
begin


if (now >= PRE_TEST_BEGIN_TIME(TIME_OUT)) then
HAZARDS_PATTERNS_TEST(F_OUT, "F", HAZARD_SCHEDULE1, TMP_IN_VECTOR, PATT1, PATT2,
TIME_OUT, INFO);
end if;

end process HAZARD_DETECTION1;

end STRUCTURAL;

configuration CFG_HAZ_1 of HAZ_1 is
        for STRUCTURAL

        for INV1: INVGATE
        use entity VHDS_LIB.INVGATE(PURE_BEHAVIOR);
        end for;

        for AND1,AND2: AND2GATE
        use entity VHDS_LIB.AND2GATE(PURE_BEHAVIOR);
        end for;

        for OR1: OR2GATE
        use entity VHDS_LIB.OR2GATE(PURE_BEHAVIOR);
        end for;

end for;
end CFG_HAZ_1;
```

-- VHDS Test Bench Shell design for Two-Input Change Hazard Analysis

```
library VHDS_LIB;

use VHDS_LIB.MVL9_SYSTEM.all;
use VHDS_LIB.TIME_PACKAGE.all;
use VHDS_LIB.TEST_GENERATOR.all;
use VHDS_LIB.INPUT_DATA.all;

entity TEST_BENCH is
end TEST_BENCH;

architecture HAZ1TB of TEST_BENCH is

component HAZ_1
generic  (GLITCH_0_TIME, GLITCH_1_TIME: TIME);

port(TMP_IN_VECTOR: in MVL9_VECTOR(0 to 2);
   PATT1, PATT2: in MVL9_VECTOR(0 to 2);
   F: out MVL9;
   STATE_VALUE_TABLE: inout MVL9_STATE_LOGIC_VECTOR(0 to 2**WORDLENGTH - 1);
   TIME_IN: in PROP_DELAY_SPEC;
   TIME_OUT: inout PROP_DELAY_SPEC);
end component;

signal TMP_IN_VECTOR: MVL9_VECTOR(0 to 2);
signal PATT1, PATT2: MVL9_VECTOR(0 to 2);
signal F: MVL9;
signal STATE_VALUE_TABLE: MVL9_STATE_LOGIC_VECTOR(0 to 2**WORDLENGTH - 1);
signal TIME_IN: PROP_DELAY_SPEC := (0 ns, 0 ns);
signal TIME_OUT: PROP_DELAY_SPEC:= (0 ns, 0 ns);
constant TOTAL_PATTERN_FUNCTION: INTEGER :=
TOTAL_FUNCTION_PATTERNS(WORDLENGTH);
signal NUM1: INTEGER := 0;
signal INIT,INIT1: BIT := '0';
signal TMP_IN_VECTOR1, TMP_IN_VECTOR2: MVL9_VECTOR(0 to 2);
begin

TMP_IN_VECTOR <= TMP_IN_VECTOR1 when not TMP_IN_VECTOR1'quiet else
                 TMP_IN_VECTOR2 when not TMP_IN_VECTOR2'quiet else
                 TMP_IN_VECTOR;

-------------------------------------------------
-- Generic Map Association
-------------------------------------------------
L1: HAZ_1 generic map(2 ns,2 ns)
port map(TMP_IN_VECTOR, PATT1, PATT2, F, STATE_VALUE_TABLE, TIME_IN, TIME_OUT);



INIT <= '1' after 1 ns;

INJECTION:process(INIT)
```

```
begin
PRE_PROCESS_INJECTION(TIME_OUT, TMP_IN_VECTOR1);

if (now = 1 ns) then
INIT1 <= transport '1' after (PRE_TEST_BEGIN_TIME(TIME_OUT) - 1 ns);
end if;
end process INJECTION;

TESTING: process (INIT1)
variable RESULT2: MVL9_VECTOR_TRIPLE_VECTOR(0 to TOTAL_PATTERN_FUNCTION - 1);
variable NUMBER: NATURAL := 0;
begin

RESULT2 := FUNCTION_PATTERN_GENERATOR(WORDLENGTH);

NUMBER := PRE_PROCESS_COUNT(RESULT2, STATE_VALUE_TABLE);
NUM1 <= NUMBER;

PRE_FUNCTION_PATTERN_TESTER(PRE_PROCESS_FUNCTION_GENERATOR(RESULT2,
                    STATE_VALUE_TABLE,NUMBER),
                    TIME_OUT, NUMBER,
                    PATTERN1 => PATT1, PATTERN2 => PATT2,
                    OUT_PATTERN => TMP_IN_VECTOR2, MODE => FUNCT);

end process TESTING;
end HAZ1TB;

use WORK.all;
configuration CON_HAZ1 of TEST_BENCH is
for HAZ1TB
        for all: HAZ_1
        use entity WORK.HAZ_1(STRUCTURAL);
        end for;
end for;
end CON_HAZ1;
```

-- VHDS Report Summary for the Two-Input Change Hazard Analysis

```
        Test Circuit Statistics:
        Number of Primary Inputs: 3
Minimum Propagation Delay of the test circuit with respect to F: 4 NS
Maximum Propagation Delay of the test circuit with respect to F: 6 NS

************************** VHDS Report Summary ************************

=======================================
Function 0 hazard is detected at      74 ns on F
62 NS       000
68 NS       101
66 NS        0      F
72 NS        1      F
74 NS        0      F
```

5.3 A Test Circuit with Four Primary Inputs

Figure 34 shows a test circuit with four primary inputs. This test circuit is realized from **[10]**. The input signal is a four bit MVL9 vector called TMP_IN_VECTOR. The output of the test circuit is a MVL9 signal called F. From the VHDS analysis, the hazard report summaries for the single-input change analysis and the two-input change hazard analysis were generated. From the summaries, the min and max values of the test circuit with respect to output signal F are 10 ns and 16 ns, respectively. In the report summaries, the VHDS found two static 1 hazard, two function 0 hazards which have a pulse width duration of 3 ns, three function 0 hazards which have a pulse width duration of 4 ns, two function 0 hazards which have a pulse width duration of 5 ns, four function 1 hazards which have a pulse width duration of 1 ns, four function 1 hazards which have a pulse width duration of 2 ns, one function 1 hazard which has a pulse width duration of 3 ns, three function 1 hazards which have a pulse width duration of 4 ns, and two function 1 hazards which have a pulse width duration of 5 ns existed in the test circuit. The static 1 hazard exists in the test circuit when the input TMP_IN_VECTOR(0 to 3) changes from "0110" to "0010" and "0111" to "0011". The static hazard pulses have a duration of 2 ns and 3 ns, respectively. For example, a function 1 hazard is found in the test circuit when the inputs change from "1110" to "1101". This is also true when the inputs change from "1110" to "1011", and "1101" to "1110", etc.

The VHDS source files, the VHDS test bench shells, and the VHDS report summaries for the single-input change and the two-input change hazard analysis are enclosed in the following pages.

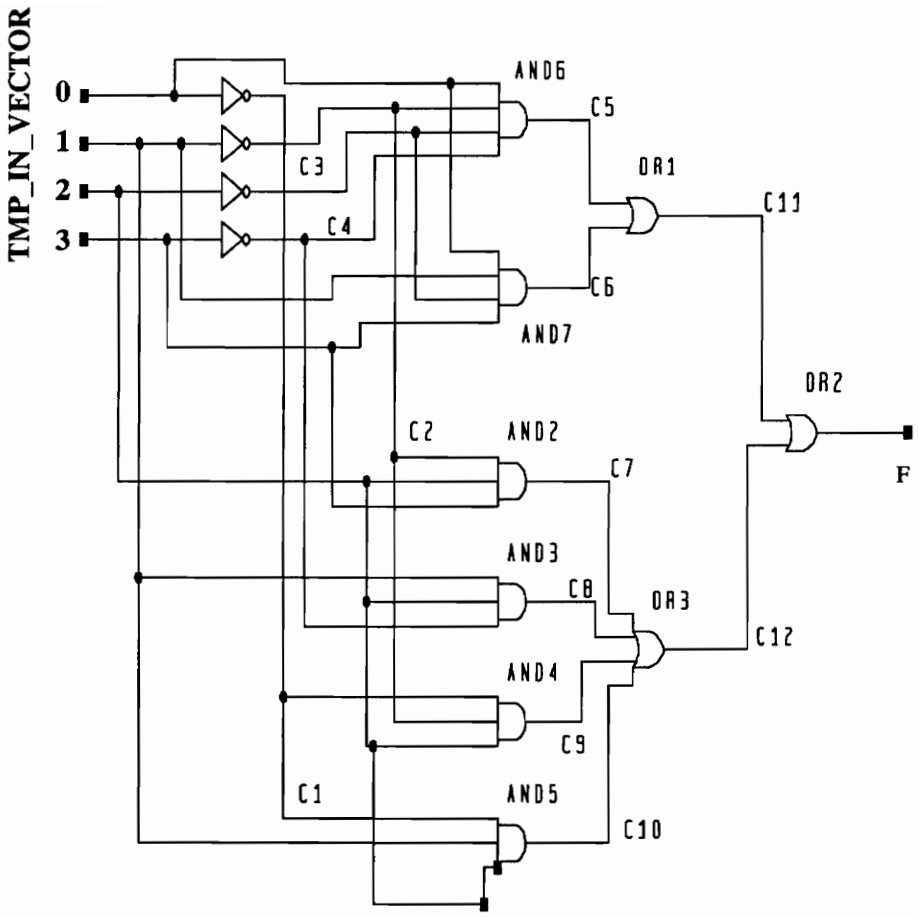


Figure 34. A Test Circuit with Four Primary Inputs

-- VHDS Source File Design for the Single-Input Change Hazard Analysis

```
library VHDS_LIB;

use VHDS_LIB.MVL9_SYSTEM.all;
use VHDS_LIB.TIME_PACKAGE.all;
use VHDS_LIB.PRE_HAZARD_TEST.all;
use VHDS_LIB.GATE_LIBRARY.all;
use VHDS_LIB.INPUT_DATA.all;
use VHDS_LIB.TEST_GENERATOR.all;


entity HAZ_1 is
generic (GLITCH_0_TIME, GLITCH_1_TIME: TIME);

port(TMP_IN_VECTOR, PATT1, PATT2: in MVL9_VECTOR(0 to 3); F: out MVL9;
     TIME_IN: in PROP_DELAY_SPEC := (0 ns, 0 ns);
     TIME_OUT: inout PROP_DELAY_SPEC := (0 ns, 0 ns));
end HAZ_1;

architecture STRUCTURAL of HAZ_1 is

subtype PROP_DELAY_SPEC_VECTOR4 is PROP_DELAY_SPEC_VECTOR(1 to 4);
subtype PROP_DELAY_SPEC_VECTOR3 is PROP_DELAY_SPEC_VECTOR(1 to 3);
subtype PROP_DELAY_SPEC_VECTOR2 is PROP_DELAY_SPEC_VECTOR(1 to 2);

signal C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,C12,F_OUT: MVL9;
signal TIME_OUT_A, TIME_OUT_B, TIME_OUT_C, TIME_OUT_D: PROP_DELAY_SPEC :=(0 ns, 0 ns);

signal L7,L9: PROP_DELAY_SPEC_VECTOR2 := ((0 ns, 0 ns), (0 ns, 0 ns));
signal L3,L4,L5,L6: PROP_DELAY_SPEC_VECTOR3 := ((0 ns, 0 ns), (0 ns, 0 ns), (0 ns, 0 ns));
signal L1,L2,L8: PROP_DELAY_SPEC_VECTOR4 := ((0 ns, 0 ns), (0 ns, 0 ns), (0 ns, 0 ns), (0 ns, 0 ns));


begin

-------------------------------------------------------------------
-- A timing resolve process to compute the max. and min. delay
-- of the whole structural component.
-------------------------------------------------------------------
L1(2) <= TIME_OUT_B;
L1(3) <= TIME_OUT_C;
L1(4) <= TIME_OUT_D;

L2(3) <= TIME_OUT_C;

L3(1) <= TIME_OUT_B;

L4(3) <= TIME_OUT_D;

L5(1) <= TIME_OUT_A;
L5(2) <= TIME_OUT_B;

L6(1) <= TIME_OUT_A;
```

```
INV1: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => TMP_IN_VECTOR(0), OUTPUT => C1,
                    TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_A);

INV2: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT => TMP_IN_VECTOR(1), OUTPUT => C2,
                    TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_B);

INV3: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => TMP_IN_VECTOR(2), OUTPUT => C3,
                    TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_C);

INV4: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => TMP_IN_VECTOR(3), OUTPUT => C4,
                    TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_D);

AND2: AND3GATE
        generic map(4 ns, 4 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => C2,  INPUT(2) => TMP_IN_VECTOR(2),
                    INPUT(3) => TMP_IN_VECTOR(3), OUTPUT => C7,
                    TIME_SPEC_IN => TIME_BUS_FUN(L3), TIME_SPEC_OUT => L8(1));

AND3: AND3GATE
        generic map(5 ns, 5 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => TMP_IN_VECTOR(1),  INPUT(2) => TMP_IN_VECTOR(2),
                    INPUT(3) => C4, OUTPUT => C8,
                    TIME_SPEC_IN => TIME_BUS_FUN(L4), TIME_SPEC_OUT => L8(2));

AND4: AND3GATE
        generic map(4 ns, 4 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => C1,  INPUT(2) => C2,
                    INPUT(3) => TMP_IN_VECTOR(2), OUTPUT => C9,
                    TIME_SPEC_IN => TIME_BUS_FUN(L5), TIME_SPEC_OUT => L8(3));

AND5: AND3GATE
        generic map(4 ns, 4 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => C1,  INPUT(2) => TMP_IN_VECTOR(1),
                    INPUT(3) => TMP_IN_VECTOR(2), OUTPUT => C10,
                    TIME_SPEC_IN => TIME_BUS_FUN(L6), TIME_SPEC_OUT => L8(4));
```

```
AND6: AND4GATE
        generic map(4 ns, 4 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => TMP_IN_VECTOR(0),  INPUT(2) => C2,
                    INPUT(3) => C3, INPUT(4) => C4, OUTPUT => C5,
                    TIME_SPEC_IN => TIME_BUS_FUN(L1), TIME_SPEC_OUT => L7(1));


AND7: AND4GATE
        generic map(5 ns, 5 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => TMP_IN_VECTOR(0),  INPUT(2) => TMP_IN_VECTOR(1),
                    INPUT(3) => C3, INPUT(4) => TMP_IN_VECTOR(3), OUTPUT => C6,
                    TIME_SPEC_IN => TIME_BUS_FUN(L2), TIME_SPEC_OUT => L7(2));


OR1: OR2GATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => C5, INPUT(2) => C6, OUTPUT => C11,
                    TIME_SPEC_IN => TIME_BUS_FUN(L7), TIME_SPEC_OUT => L9(1));

OR2: OR2GATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => C11, INPUT(2) => C12, OUTPUT => F_OUT,
                    TIME_SPEC_IN => TIME_BUS_FUN(L9), TIME_SPEC_OUT => TIME_OUT);


OR3: OR4GATE
        generic map(5 ns, 5 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => C7,  INPUT(2) => C8, INPUT(3) => C9, INPUT(4) => C10,
                    OUTPUT => C12,
                    TIME_SPEC_IN => TIME_BUS_FUN(L8), TIME_SPEC_OUT => L9(2));


F <= F_OUT;


HAZARD_DETECTION: process(F_OUT,TMP_IN_VECTOR)
variable HAZARD_SCHEDULE1: HAZARD_VAR := HAZARD_CONSTANT;

begin
HAZARDS_PATTERNS_TEST(OUTPUT => F_OUT, NAME => "F",
HAZARD_SCHEDULE1 => HAZARD_SCHEDULE1,
PATTERNS => TMP_IN_VECTOR, PATT1 => PATT1, PATT2 => PATT2, TIME_DELAY => TIME_OUT);

end process HAZARD_DETECTION;

end STRUCTURAL;

configuration CFG_HAZ_1 of HAZ_1 is
        for STRUCTURAL
```

```
        for INV1,INV2,INV3,INV4: INVGATE
        use entity VHDS_LIB.INVGATE(PURE_BEHAVIOR);
        end for;

        for AND2,AND3,AND4,AND5: AND3GATE
        use entity VHDS_LIB.AND3GATE(PURE_BEHAVIOR);
        end for;

        for AND6,AND7: AND4GATE
        use entity VHDS_LIB.AND4GATE(PURE_BEHAVIOR);
        end for;

        for OR1,OR2: OR2GATE
        use entity VHDS_LIB.OR2GATE(PURE_BEHAVIOR);
        end for;

        for OR3: OR4GATE
        use entity VHDS_LIB.OR4GATE(PURE_BEHAVIOR);
        end for;

end for;
end CFG_HAZ_1;
```

-- VHDS Test Bench Shell Design for the Single-Input Change Hazard Analysis

```
library VHDS_LIB;

use VHDS_LIB.MVL9_SYSTEM.all;
use VHDS_LIB.TIME_PACKAGE.all;
use VHDS_LIB.TEST_GENERATOR.all;
use VHDS_LIB.INPUT_DATA.all;

entity TEST_BENCH is
end TEST_BENCH;

architecture HAZ1TB of TEST_BENCH is

component HAZ_1
generic  (GLITCH_0_TIME, GLITCH_1_TIME: TIME);

port(TMP_IN_VECTOR: in MVL9_VECTOR(0 to 3);
   PATT1, PATT2: in MVL9_VECTOR(0 to 3);
   F: out MVL9;
   TIME_IN: in PROP_DELAY_SPEC;
   TIME_OUT: inout PROP_DELAY_SPEC);
end component;

signal TMP_IN_VECTOR: MVL9_VECTOR(0 to 3);
signal PATT1, PATT2: MVL9_VECTOR(0 to 3);
signal F: MVL9;
signal TIME_IN: PROP_DELAY_SPEC := (0 ns, 0 ns);
signal TIME_OUT: PROP_DELAY_SPEC:= (0 ns, 0 ns);
constant TOTAL_PATTERN_STATIC: INTEGER := TOTAL_STATIC_PATTERNS(WORDLENGTH);
signal INIT: BIT := '0';
begin

-----------------------------------------------
-- Generic Map Association
-----------------------------------------------
L1: HAZ_1 generic map(3 ns, 3 ns)
port map(TMP_IN_VECTOR,PATT1, PATT2, F,TIME_IN,TIME_OUT);


INIT <= '1' after 1 ns;

TEST1:process (INIT)
variable RESULT1: MVL9_VECTOR_TUPLE_VECTOR(0 to TOTAL_PATTERN_STATIC - 1);
begin


RESULT1 := STATIC_PATTERN_GENERATOR(WORDLENGTH);
STATIC_PATTERN_TESTER(RESULT1, TIME_OUT, PATTERN1 => PATT1, PATTERN2 => PATT2,
                                    OUT_PATTERN => TMP_IN_VECTOR, MODE => STATIC);


end process TEST1;
```

```
end HAZ1TB;


use WORK.all;
configuration CON_HAZ1 of TEST_BENCH is
for HAZ1TB
        for all: HAZ_1
        use entity WORK.HAZ_1(STRUCTURAL);
        end for;
end for;
end CON_HAZ1;
```

-- VHDS Report Summary for the Single-Input Change Hazard Analysis

```
        Test Circuit Statistics:
        Number of Primary Inputs: 4
Minimum Propagation Delay of the test circuit with respect to F: 10 NS
Maximum Propagation Delay of the test circuit with respect to F: 16 NS

************************* VHDS Report Summary ************************

=======================================
Static 1 hazard is detected at    1648 ns on F
1617 NS      0110
1633 NS      0010
1632 NS        1      F
1646 NS        0      F
1648 NS        1      F
=======================================
Static 1 hazard is detected at    1904 ns on F
1873 NS      0111
1889 NS      0011
1888 NS        1      F
1901 NS        0      F
1904 NS        1      F
```

-- VHDS Source File Design for the Two-Input Change Hazard Analysis

```
library VHDS_LIB;

use VHDS_LIB.MVL9_SYSTEM.all;
use VHDS_LIB.TIME_PACKAGE.all;
use VHDS_LIB.PRE_HAZARD_TEST.all;
use VHDS_LIB.GATE_LIBRARY.all;
use VHDS_LIB.INPUT_DATA.all;
use VHDS_LIB.TEST_GENERATOR.all;


entity HAZ_1 is
generic (GLITCH_0_TIME, GLITCH_1_TIME: TIME);

port(TMP_IN_VECTOR, PATT1, PATT2: in MVL9_VECTOR(0 to 3); F: out MVL9;
   STATE_VALUE_TABLE: inout MVL9_STATE_LOGIC_VECTOR(0 to 2**WORDLENGTH - 1);
   TIME_IN: in PROP_DELAY_SPEC := (0 ns, 0 ns);
   TIME_OUT: inout PROP_DELAY_SPEC := (0 ns, 0 ns));
end HAZ_1;

architecture STRUCTURAL of HAZ_1 is

subtype PROP_DELAY_SPEC_VECTOR4 is PROP_DELAY_SPEC_VECTOR(1 to 4);
subtype PROP_DELAY_SPEC_VECTOR3 is PROP_DELAY_SPEC_VECTOR(1 to 3);
subtype PROP_DELAY_SPEC_VECTOR2 is PROP_DELAY_SPEC_VECTOR(1 to 2);

signal C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,C12,F_OUT: MVL9;
signal TIME_OUT_A, TIME_OUT_B, TIME_OUT_C, TIME_OUT_D: PROP_DELAY_SPEC :=(0 ns, 0 ns);

signal L7,L9: PROP_DELAY_SPEC_VECTOR2 := ((0 ns, 0 ns), (0 ns, 0 ns));
signal L3,L4,L5,L6: PROP_DELAY_SPEC_VECTOR3 := ((0 ns, 0 ns), (0 ns, 0 ns), (0 ns, 0 ns));
signal L1,L2,L8: PROP_DELAY_SPEC_VECTOR4 := ((0 ns, 0 ns), (0 ns, 0 ns), (0 ns, 0 ns), (0 ns, 0 ns));



begin

------------------------------------------------------------------------
-- A timing resolve process to compute the max. and min. delay
-- of the whole structural component.
------------------------------------------------------------------------
L1(2) <= TIME_OUT_B;
L1(3) <= TIME_OUT_C;
L1(4) <= TIME_OUT_D;

L2(3) <= TIME_OUT_C;

L3(1) <= TIME_OUT_B;

L4(3) <= TIME_OUT_D;

L5(1) <= TIME_OUT_A;
L5(2) <= TIME_OUT_B;
```

```
L6(1) <= TIME_OUT_A;

INV1: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => TMP_IN_VECTOR(0), OUTPUT => C1,
                    TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_A);

INV2: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT => TMP_IN_VECTOR(1), OUTPUT => C2,
                    TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_B);

INV3: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => TMP_IN_VECTOR(2), OUTPUT => C3,
                    TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_C);

INV4: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => TMP_IN_VECTOR(3), OUTPUT => C4,
                    TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_D);

AND2: AND3GATE
        generic map(4 ns, 4 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => C2,  INPUT(2) => TMP_IN_VECTOR(2),
                    INPUT(3) => TMP_IN_VECTOR(3), OUTPUT => C7,
                    TIME_SPEC_IN => TIME_BUS_FUN(L3), TIME_SPEC_OUT => L8(1));

AND3: AND3GATE
        generic map(5 ns, 5 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => TMP_IN_VECTOR(1),  INPUT(2) => TMP_IN_VECTOR(2),
                    INPUT(3) => C4, OUTPUT => C8,
                    TIME_SPEC_IN => TIME_BUS_FUN(L4), TIME_SPEC_OUT => L8(2));

AND4: AND3GATE
        generic map(4 ns, 4 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => C1,  INPUT(2) => C2,
                    INPUT(3) => TMP_IN_VECTOR(2), OUTPUT => C9,
                    TIME_SPEC_IN => TIME_BUS_FUN(L5), TIME_SPEC_OUT => L8(3));

AND5: AND3GATE
        generic map(4 ns, 4 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => C1,  INPUT(2) => TMP_IN_VECTOR(1),
                    INPUT(3) => TMP_IN_VECTOR(2), OUTPUT => C10,
                    TIME_SPEC_IN => TIME_BUS_FUN(L6), TIME_SPEC_OUT => L8(4));
```

```
AND6: AND4GATE
        generic map(4 ns, 4 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => TMP_IN_VECTOR(0),  INPUT(2) => C2,
                    INPUT(3) => C3, INPUT(4) => C4, OUTPUT => C5,
                    TIME_SPEC_IN => TIME_BUS_FUN(L1), TIME_SPEC_OUT => L7(1));


AND7: AND4GATE
        generic map(5 ns, 5 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => TMP_IN_VECTOR(0),  INPUT(2) => TMP_IN_VECTOR(1),
                    INPUT(3) => C3, INPUT(4) => TMP_IN_VECTOR(3), OUTPUT => C6,
                    TIME_SPEC_IN => TIME_BUS_FUN(L2), TIME_SPEC_OUT => L7(2));


OR1: OR2GATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => C5, INPUT(2) => C6, OUTPUT => C11,
                    TIME_SPEC_IN => TIME_BUS_FUN(L7), TIME_SPEC_OUT => L9(1));

OR2: OR2GATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => C11, INPUT(2) => C12, OUTPUT => F_OUT,
                    TIME_SPEC_IN => TIME_BUS_FUN(L9), TIME_SPEC_OUT => TIME_OUT);


OR3: OR4GATE
        generic map(5 ns, 5 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => C7,  INPUT(2) => C8, INPUT(3) => C9, INPUT(4) => C10,
                    OUTPUT => C12,
                    TIME_SPEC_IN => TIME_BUS_FUN(L8), TIME_SPEC_OUT => L9(2));


F <= F_OUT;


HAZARD_DETECTION:process
variable DATA: INTEGER := 0;
begin

PRE_PROCESS_DATA(TMP_IN_VECTOR, F_OUT, TIME_OUT, DATA, STATE_VALUE_TABLE);

wait on F_OUT, TMP_IN_VECTOR;
end process HAZARD_DETECTION;


HAZARD_DETECTION1:process(F_OUT, TMP_IN_VECTOR)
variable HAZARD_SCHEDULE1 : HAZARD_VAR := HAZARD_CONSTANT;
constant INFO : STATIC_INFO_VECTOR(0 to 1)
        := (("0110", "0010", 13 ns, 2 ns), ("0111", "0011", 12 ns, 3 ns));
```

```
begin
if (now >= PRE_TEST_BEGIN_TIME(TIME_OUT)) then
HAZARDS_PATTERNS_TEST(F_OUT, "F", HAZARD_SCHEDULE1, TMP_IN_VECTOR, PATT1, PATT2,
TIME_OUT, INFO);

end if;

end process HAZARD_DETECTION1;

end STRUCTURAL;

configuration CFG_HAZ_1 of HAZ_1 is
        for STRUCTURAL

        for INV1,INV2,INV3,INV4: INVGATE
        use entity VHDS_LIB.INVGATE(PURE_BEHAVIOR);
        end for;

        for AND2,AND3,AND4,AND5: AND3GATE
        use entity VHDS_LIB.AND3GATE(PURE_BEHAVIOR);
        end for;

        for AND6,AND7: AND4GATE
        use entity VHDS_LIB.AND4GATE(PURE_BEHAVIOR);
        end for;

        for OR1,OR2: OR2GATE
        use entity VHDS_LIB.OR2GATE(PURE_BEHAVIOR);
        end for;

        for OR3: OR4GATE
        use entity VHDS_LIB.OR4GATE(PURE_BEHAVIOR);
        end for;

end for;
end CFG_HAZ_1;
```

-- VHDS Test Bench Shell Design for the Two-Input Change Hazard Analysis

```
library VHDS_LIB;

use VHDS_LIB.MVL9_SYSTEM.all;
use VHDS_LIB.TIME_PACKAGE.all;
use VHDS_LIB.TEST_GENERATOR.all;
use VHDS_LIB.INPUT_DATA.all;

entity TEST_BENCH is
end TEST_BENCH;

architecture HAZ1TB of TEST_BENCH is

component HAZ_1
generic  (GLITCH_0_TIME, GLITCH_1_TIME: TIME);

port(TMP_IN_VECTOR: in MVL9_VECTOR(0 to 3);
   PATT1, PATT2: in MVL9_VECTOR(0 to 3);
   F: out MVL9;
   STATE_VALUE_TABLE: inout MVL9_STATE_LOGIC_VECTOR(0 to 2**WORDLENGTH - 1);
   TIME_IN: in PROP_DELAY_SPEC;
   TIME_OUT: inout PROP_DELAY_SPEC);
end component;

signal TMP_IN_VECTOR: MVL9_VECTOR(0 to 3);
signal PATT1, PATT2: MVL9_VECTOR(0 to 3);
signal F: MVL9;
signal STATE_VALUE_TABLE: MVL9_STATE_LOGIC_VECTOR(0 to 2**WORDLENGTH - 1);
signal TIME_IN: PROP_DELAY_SPEC := (0 ns, 0 ns);
signal TIME_OUT: PROP_DELAY_SPEC:= (0 ns, 0 ns);
constant TOTAL_PATTERN_FUNCTION: INTEGER :=
TOTAL_FUNCTION_PATTERNS(WORDLENGTH);
signal NUM1: INTEGER := 0;
signal INIT, INIT1: BIT := '0';
signal TMP_IN_VECTOR1, TMP_IN_VECTOR2: MVL9_VECTOR(0 to 3);
begin

TMP_IN_VECTOR <= TMP_IN_VECTOR1 when not TMP_IN_VECTOR1'quiet else
                 TMP_IN_VECTOR2 when not TMP_IN_VECTOR2'quiet else
                 TMP_IN_VECTOR;

-------------------------------------------------
-- Generic Map Association
-------------------------------------------------
L1: HAZ_1 generic map(3 ns, 3 ns)
port map(TMP_IN_VECTOR,PATT1, PATT2,F,STATE_VALUE_TABLE,TIME_IN,TIME_OUT);


INIT <= '1' after 1 ns;

INJECTION:process(INIT)
begin
PRE_PROCESS_INJECTION(TIME_OUT, TMP_IN_VECTOR1);
```

```
if (now = 1 ns) then
INIT1 <= transport '1' after (PRE_TEST_BEGIN_TIME(TIME_OUT) - 1 ns);
end if;

end process INJECTION;


TESTING: process (INIT1)
variable RESULT2: MVL9_VECTOR_TRIPLE_VECTOR(0 to TOTAL_PATTERN_FUNCTION - 1);

variable NUMBER: NATURAL := 0;

begin

RESULT2 := FUNCTION_PATTERN_GENERATOR(WORDLENGTH);

NUMBER := PRE_PROCESS_COUNT(RESULT2, STATE_VALUE_TABLE);
NUM1 <= NUMBER;

PRE_FUNCTION_PATTERN_TESTER(PRE_PROCESS_FUNCTION_GENERATOR(RESULT2,
                    STATE_VALUE_TABLE,NUMBER),
                    TIME_OUT, NUMBER,
                    PATTERN1 => PATT1, PATTERN2 => PATT2,
                    OUT_PATTERN => TMP_IN_VECTOR2, MODE => FUNCT);

end process TESTING;

end HAZ1TB;


use WORK.all;
configuration CON_HAZ1 of TEST_BENCH is
for HAZ1TB
        for all: HAZ_1
        use entity WORK.HAZ_1(STRUCTURAL);
        end for;
end for;
end CON_HAZ1;
```

-- VHDS Report Summary for the Two-Input Change Hazard Analysis

```
        Test Circuit Statistics:
        Number of Primary Inputs: 4
Minimum Propagation Delay of the test circuit with respect to F: 10 NS
Maximum Propagation Delay of the test circuit with respect to F: 16 NS

************************** VHDS Report Summary ************************

=======================================
Function 0 hazard is detected at     319 ns on F
290 NS      0000
306 NS      1100
303 NS        0     F
316 NS        1     F
319 NS        0     F
=======================================
Function 0 hazard is detected at     385 ns on F
354 NS      0000
370 NS      1010
367 NS        0     F
380 NS        1     F
385 NS        0     F
=======================================
Function 0 hazard is detected at     447 ns on F
418 NS      0000
434 NS      1001
431 NS        0     F
444 NS        1     F
447 NS        0     F
=======================================
Function 1 hazard is detected at     575 ns on F
546 NS      0010
562 NS      1000
561 NS        1     F
574 NS        0     F
575 NS        1     F
=======================================
Function 0 hazard is detected at     769 ns on F
738 NS      0101
754 NS      1111
751 NS        0     F
765 NS        1     F
769 NS        0     F
=======================================
Function 1 hazard is detected at     960 ns on F
930 NS      0111
946 NS      1101
945 NS        1     F
958 NS        0     F
960 NS        1     F
=======================================
Function 1 hazard is detected at     1026 ns on F
994 NS      0111
```

```
1010 NS        1110
1009 NS          1       F
1025 NS          0       F
1026 NS          1       F
=======================================
Function 1 hazard is detected at    1089 ns on F
1058 NS        1000
1074 NS        0010
1071 NS          1       F
1084 NS          0       F
1089 NS          1       F
=======================================
Function 0 hazard is detected at    1473 ns on F
1442 NS        1001
1458 NS        1111
1455 NS          0       F
1469 NS          1       F
1473 NS          0       F
=======================================
Function 0 hazard is detected at    1794 ns on F
1762 NS        1010
1778 NS        1111
1777 NS          0       F
1790 NS          1       F
1794 NS          0       F
=======================================
Function 1 hazard is detected at    1985 ns on F
1954 NS        1011
1970 NS        0010
1969 NS          1       F
1982 NS          0       F
1985 NS          1       F
=======================================
Function 1 hazard is detected at    2048 ns on F
2018 NS        1011
2034 NS        1101
2033 NS          1       F
2046 NS          0       F
2048 NS          1       F
=======================================
Function 1 hazard is detected at    2114 ns on F
2082 NS        1011
2098 NS        1110
2097 NS          1       F
2110 NS          0       F
2114 NS          1       F
=======================================
Function 1 hazard is detected at    2175 ns on F
2146 NS        1011
2162 NS        1000
2161 NS          1       F
2174 NS          0       F
2175 NS          1       F
=======================================
Function 0 hazard is detected at    2498 ns on F
```

```
2466 NS        1100
2482 NS        1111
2479 NS          0       F
2493 NS          1       F
2498 NS          0       F
=======================================
Function 1 hazard is detected at     2561 ns on F
2530 NS        1101
2546 NS        0111
2544 NS          1       F
2557 NS          0       F
2561 NS          1       F
=======================================
Function 1 hazard is detected at     2625 ns on F
2594 NS        1101
2610 NS        1011
2608 NS          1       F
2621 NS          0       F
2625 NS          1       F
=======================================
Function 1 hazard is detected at     2687 ns on F
2658 NS        1101
2674 NS        1000
2672 NS          1       F
2685 NS          0       F
2687 NS          1       F
=======================================
Function 1 hazard is detected at     2754 ns on F
2722 NS        1101
2738 NS        1110
2736 NS          1       F
2749 NS          0       F
2754 NS          1       F
=======================================
Function 1 hazard is detected at     3009 ns on F
2978 NS        1110
2994 NS        1011
2994 NS          1       F
3007 NS          0       F
3009 NS          1       F
=======================================
Function 1 hazard is detected at     3072 ns on F
3042 NS        1110
3058 NS        1101
3058 NS          1       F
3071 NS          0       F
3072 NS          1       F
```

5.4 A Test Circuit with Five Primary Inputs

Figure 35 shows a test circuit with four primary inputs. This test circuit is designed from [12]. The input signals of the test circuit are V, W, X, Y, and Z. The output of the test circuit is a MVL9 signal called F_OUTPUT. From the VHDS analysis, the hazard report summaries for the single-input change hazard analysis and the two-input change hazard analysis were generated. From the summaries, the min and max values of the test circuit with respect to output signal F_OUTPUT are 7 ns and 12 ns, respectively. In the report summaries, the VHDS found two static 1 hazards, twelve function 0 hazards which have a pulse duration of 3 ns, one function 1 hazard which has a pulse width duration of 4 ns, and one function 1 hazards which has a pulse width duration of 5 ns. The static 1 hazard exists in the test circuit when the inputs VWXYZ change from "01000" to "00000" (i.e. W changes from '1' to '0') and "01111" to "01101" (i.e. Y changes form '1' to '0'). The static hazard pulse has a duration of 4 ns.

The VHDS source files, the VHDS test bench shells, and the VHDS report summaries are enclosed in the following pages.

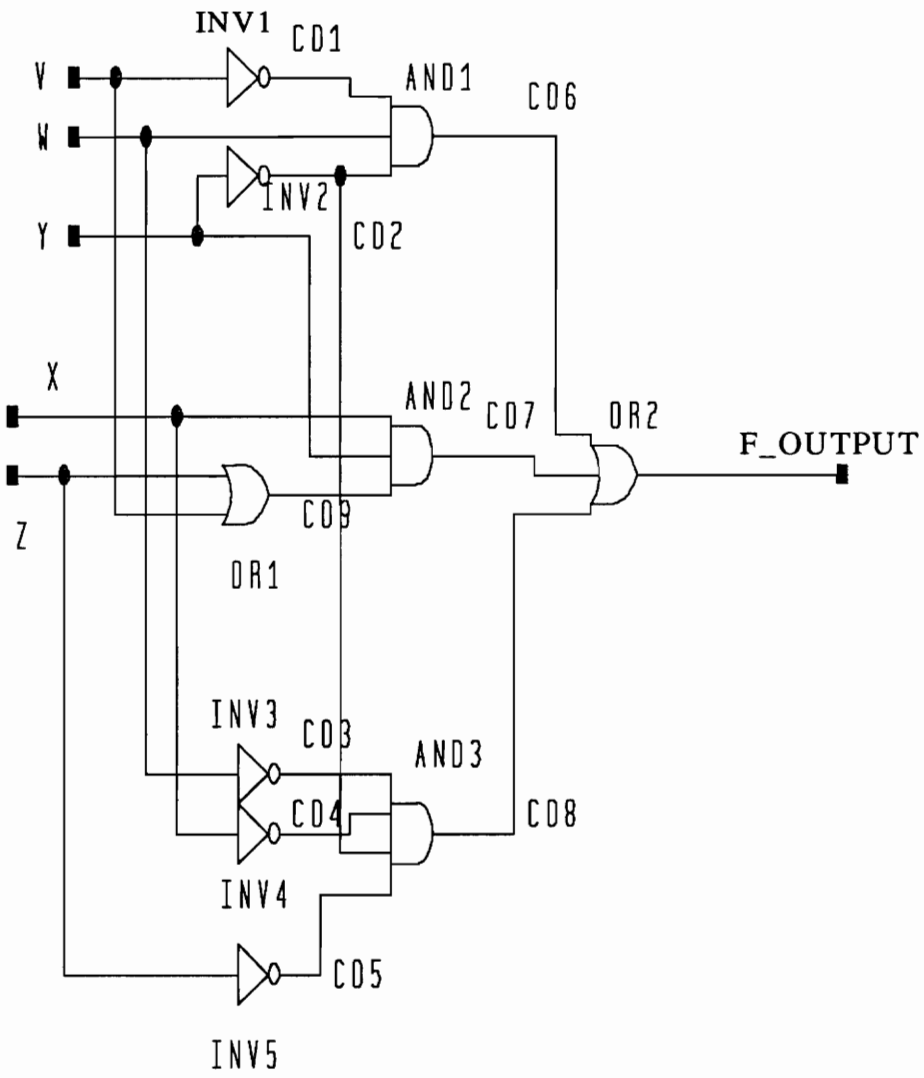


Figure 35. A Test Circuit with Five Primary Inputs

-- VHDS Source File Design for the Single-Input Change Hazard Analysis

```
library VHDS_LIB;

use VHDS_LIB.MVL9_SYSTEM.all;
use VHDS_LIB.TIME_PACKAGE.all;
use VHDS_LIB.PRE_HAZARD_TEST.all;
use VHDS_LIB.GATE_LIBRARY.all;
use VHDS_LIB.INPUT_DATA.all;
use VHDS_LIB.TEST_GENERATOR.all;


entity HAZ_1 is
generic (GLITCH_0_TIME, GLITCH_1_TIME: TIME);

port(PATT1, PATT2: in MVL9_VECTOR(0 to 4); V, W, X, Y, Z: in MVL9; F_OUTPUT: out MVL9;
     TIME_IN: in PROP_DELAY_SPEC := (0 ns, 0 ns);
     TIME_OUT: inout PROP_DELAY_SPEC := (0 ns, 0 ns));
end HAZ_1;

architecture STRUCTURAL of HAZ_1 is

subtype PROP_DELAY_SPEC_VECTOR4 is PROP_DELAY_SPEC_VECTOR(1 to 4);
subtype PROP_DELAY_SPEC_VECTOR3 is PROP_DELAY_SPEC_VECTOR(1 to 3);
subtype PROP_DELAY_SPEC_VECTOR2 is PROP_DELAY_SPEC_VECTOR(1 to 2);

signal CO1,CO2,CO3,CO4,CO5,CO6,CO7,CO8,CO9,F_OUT: MVL9;
signal IN_VECTOR: MVL9_VECTOR(0 to 4);
signal TIME_OUT_A, TIME_OUT_B, TIME_OUT_C, TIME_OUT_D, TIME_OUT_E: PROP_DELAY_SPEC
:=(0 ns, 0 ns);
signal BR: PROP_DELAY_SPEC_VECTOR2 := ((0 ns, 0 ns), (0 ns, 0 ns));
signal AR, ER, CR: PROP_DELAY_SPEC_VECTOR3 := ((0 ns, 0 ns), (0 ns, 0 ns), (0 ns, 0 ns));
signal DR: PROP_DELAY_SPEC_VECTOR4 := ((0 ns, 0 ns), (0 ns, 0 ns), (0 ns, 0 ns), (0 ns, 0 ns));


begin

-------------------------------------------------------------------
-- A timing resolve process to compute the max. and min. delay
-- of the whole structural component.
-------------------------------------------------------------------
AR(1) <= TIME_OUT_A;
AR(3) <= TIME_OUT_B;

DR(1) <= TIME_OUT_C;
DR(2) <= TIME_OUT_D;
DR(3) <= TIME_OUT_B;
DR(4) <= TIME_OUT_E;


IN_VECTOR <= V & W & X & Y & Z;


INV1: INVGATE
```

```
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => V, OUTPUT => CO1,
                TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_A);

INV2: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => Y, OUTPUT => CO2,
                TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_B);

INV3: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => W, OUTPUT => CO3,
                TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_C);

INV4: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => X, OUTPUT => CO4,
                 TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_D);

INV5: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => Z, OUTPUT => CO5,
                TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_E);

AND1: AND3GATE
        generic map(4 ns, 4 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => CO1,  INPUT(2) => W, INPUT(3) => CO2, OUTPUT => CO6,
                TIME_SPEC_IN => TIME_BUS_FUN(AR), TIME_SPEC_OUT => ER(1));

AND2: AND3GATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => X,  INPUT(2) => Y, INPUT(3) => CO9, OUTPUT => CO7,
                TIME_SPEC_IN => TIME_BUS_FUN(CR), TIME_SPEC_OUT => ER(2));

AND3: AND4GATE
        generic map(5 ns, 5 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => CO3,  INPUT(2) => CO4,
                INPUT(3) => CO2, INPUT(4) => CO5, OUTPUT => CO8,
                TIME_SPEC_IN => TIME_BUS_FUN(DR), TIME_SPEC_OUT => ER(3));


OR1: OR2GATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => Z, INPUT(2) => V, OUTPUT => CO9,
                TIME_SPEC_IN => TIME_BUS_FUN(BR), TIME_SPEC_OUT => CR(3));
```

```
OR2: OR3GATE
        generic map(4 ns, 4 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => CO6,  INPUT(2) => CO7, INPUT(3) => CO8, OUTPUT => F_OUT,
                    TIME_SPEC_IN => TIME_BUS_FUN(ER), TIME_SPEC_OUT => TIME_OUT);


F_OUTPUT <= F_OUT;



HAZARD_DETECTION: process(F_OUT,IN_VECTOR)
variable HAZARD_SCHEDULE1: HAZARD_VAR := HAZARD_CONSTANT;
begin

HAZARDS_PATTERNS_TEST(OUTPUT => F_OUT, NAME => "F_OUTPUT",
HAZARD_SCHEDULE1 => HAZARD_SCHEDULE1,
PATTERNS => IN_VECTOR, PATT1 => PATT1,
PATT2 => PATT2, TIME_DELAY => TIME_OUT);


end process HAZARD_DETECTION;

end STRUCTURAL;

configuration CFG_HAZ_1 of HAZ_1 is
        for STRUCTURAL

        for INV1,INV2,INV3,INV4,INV5: INVGATE
        use entity VHDS_LIB.INVGATE(PURE_BEHAVIOR);
        end for;

        for AND1,AND2: AND3GATE
        use entity VHDS_LIB.AND3GATE(PURE_BEHAVIOR);
        end for;

        for AND3: AND4GATE
        use entity VHDS_LIB.AND4GATE(PURE_BEHAVIOR);
        end for;

        for OR1: OR2GATE
        use entity VHDS_LIB.OR2GATE(PURE_BEHAVIOR);
        end for;

        for OR2: OR3GATE
        use entity VHDS_LIB.OR3GATE(PURE_BEHAVIOR);
        end for;

end for;
end CFG_HAZ_1;
```

-- VHDS Test Bench Shell Design for the Single-Input Change Hazard Analysis

```
library VHDS_LIB;

use VHDS_LIB.MVL9_SYSTEM.all;
use VHDS_LIB.TIME_PACKAGE.all;
use VHDS_LIB.TEST_GENERATOR.all;
use VHDS_LIB.INPUT_DATA.all;

entity TEST_BENCH is
end TEST_BENCH;

architecture HAZ1TB of TEST_BENCH is

component HAZ_1
generic (GLITCH_0_TIME, GLITCH_1_TIME: TIME);
port(PATT1,PATT2: in MVL9_VECTOR(0 to 4); V, W, X, Y, Z: in MVL9;
      F_OUTPUT: out MVL9;
      TIME_IN: in PROP_DELAY_SPEC := (0 ns, 0 ns);
      TIME_OUT: inout PROP_DELAY_SPEC := (0 ns, 0 ns));

end component;

signal PATT1, PATT2: MVL9_VECTOR(0 to 4);
signal V, W, X, Y, Z: MVL9;
signal F_OUTPUT: MVL9;
signal TIME_IN: PROP_DELAY_SPEC := (0 ns, 0 ns);
signal TIME_OUT: PROP_DELAY_SPEC:= (0 ns, 0 ns);
constant TOTAL_PATTERN_STATIC: INTEGER := TOTAL_STATIC_PATTERNS(WORDLENGTH);
signal IN_VECTOR: MVL9_VECTOR(0 to 4);
signal INIT: BIT := '0';
begin

-----------------------------------------------
-- Generic Map Association
-----------------------------------------------
L1: HAZ_1 generic map(5 ns,5 ns)
port map(PATT1,PATT2, V, W, X, Y, Z, F_OUTPUT,TIME_IN,TIME_OUT);

V <= transport IN_VECTOR(0);
W <= transport IN_VECTOR(1);
X <= transport IN_VECTOR(2);
Y <= transport IN_VECTOR(3);
Z <= transport IN_VECTOR(4);


INIT <= '1' after 1 ns;


TEST:process (INIT)
variable RESULT1: MVL9_VECTOR_TUPLE_VECTOR(0 to TOTAL_PATTERN_STATIC - 1);

begin
```

```
RESULT1 := STATIC_PATTERN_GENERATOR(WORDLENGTH);

STATIC_PATTERN_TESTER(RESULT1, TIME_OUT, PATTERN1 => PATT1, PATTERN2 => PATT2,
                                    OUT_PATTERN => IN_VECTOR, MODE => STATIC);


end process TEST;

end HAZ1TB;


use WORK.all;
configuration CON_HAZ1 of TEST_BENCH is
for HAZ1TB
        for all: HAZ_1
        use entity WORK.HAZ_1(STRUCTURAL);
        end for;
end for;
end CON_HAZ1;
```

-- VHDS Report Summary for the Single-Input Change Hazard Analysis

```
        Test Circuit Statistics:
        Number of Primary Inputs: 5
Minimum Propagation Delay of the test circuit with respect to F_OUTPUT: 7 NS
Maximum Propagation Delay of the test circuit with respect to F_OUTPUT: 12 NS

************************** VHDS Report Summary ************************

=======================================
Static 1 hazard is detected at    2005 ns on F_OUTPUT
1981 NS      01000
1993 NS      00000
1992 NS        1  F_OUTPUT
2001 NS        0  F_OUTPUT
2005 NS        1  F_OUTPUT
=======================================
Static 1 hazard is detected at    3780 ns on F_OUTPUT
3757 NS      01111
3769 NS      01101
3767 NS        1  F_OUTPUT
3776 NS        0  F_OUTPUT
3780 NS        1  F_OUTPUT
```

-- VHDS Source File Design for the Two-Input Change Hazard Analysis

```
library VHDS_LIB;

use VHDS_LIB.MVL9_SYSTEM.all;
use VHDS_LIB.TIME_PACKAGE.all;
use VHDS_LIB.PRE_HAZARD_TEST.all;
use VHDS_LIB.GATE_LIBRARY.all;
use VHDS_LIB.INPUT_DATA.all;
use VHDS_LIB.TEST_GENERATOR.all;


entity HAZ_1 is
generic(GLITCH_0_TIME, GLITCH_1_TIME:TIME);

port(PATT1, PATT2: in MVL9_VECTOR(0 to 4); V, W, X, Y, Z: in MVL9;  F_OUTPUT: out MVL9;
   STATE_VALUE_TABLE: inout MVL9_STATE_LOGIC_VECTOR(0 to 2**WORDLENGTH - 1);
   TIME_IN: in PROP_DELAY_SPEC := (0 ns, 0 ns);
   TIME_OUT: inout PROP_DELAY_SPEC := (0 ns, 0 ns));
end HAZ_1;

architecture STRUCTURAL of HAZ_1 is

subtype PROP_DELAY_SPEC_VECTOR4 is PROP_DELAY_SPEC_VECTOR(1 to 4);
subtype PROP_DELAY_SPEC_VECTOR3 is PROP_DELAY_SPEC_VECTOR(1 to 3);
subtype PROP_DELAY_SPEC_VECTOR2 is PROP_DELAY_SPEC_VECTOR(1 to 2);

signal CO1,CO2,CO3,CO4,CO5,CO6,CO7,CO8,CO9,F_OUT: MVL9;
signal IN_VECTOR: MVL9_VECTOR(0 to 4);
signal TIME_OUT_A, TIME_OUT_B, TIME_OUT_C, TIME_OUT_D, TIME_OUT_E: PROP_DELAY_SPEC
:=(0 ns, 0 ns);
signal BR: PROP_DELAY_SPEC_VECTOR2 := ((0 ns, 0 ns), (0 ns, 0 ns));
signal AR, ER, CR: PROP_DELAY_SPEC_VECTOR3 := ((0 ns, 0 ns), (0 ns, 0 ns), (0 ns, 0 ns));
signal DR: PROP_DELAY_SPEC_VECTOR4 := ((0 ns, 0 ns), (0 ns, 0 ns), (0 ns, 0 ns), (0 ns, 0 ns));
begin

-------------------------------------------------------------------
-- A timing resolve process to compute the max. and min. delay
-- of the whole structural component.
-------------------------------------------------------------------
AR(1) <= TIME_OUT_A;
AR(3) <= TIME_OUT_B;

DR(1) <= TIME_OUT_C;
DR(2) <= TIME_OUT_D;
DR(3) <= TIME_OUT_B;
DR(4) <= TIME_OUT_E;


IN_VECTOR <= V & W & X & Y & Z;


INV1: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
```

```
                    GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => V, OUTPUT => CO1,
                    TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_A);

INV2: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                    GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => Y, OUTPUT => CO2,
                   TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_B);

INV3: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                    GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => W, OUTPUT => CO3,
                   TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_C);

INV4: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                    GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT => X, OUTPUT => CO4,
                    TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_D);

INV5: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                    GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => Z, OUTPUT => CO5,
                    TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_E);

AND1: AND3GATE
        generic map(4 ns, 4 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                    GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => CO1,  INPUT(2) => W, INPUT(3) => CO2, OUTPUT => CO6,
                   TIME_SPEC_IN => TIME_BUS_FUN(AR), TIME_SPEC_OUT => ER(1));

AND2: AND3GATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                    GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => X,  INPUT(2) => Y, INPUT(3) => CO9, OUTPUT => CO7,
                   TIME_SPEC_IN => TIME_BUS_FUN(CR), TIME_SPEC_OUT => ER(2));

AND3: AND4GATE
        generic map(5 ns, 5 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                    GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => CO3,  INPUT(2) => CO4,
                    INPUT(3) => CO2, INPUT(4) => CO5, OUTPUT => CO8,
                    TIME_SPEC_IN => TIME_BUS_FUN(DR), TIME_SPEC_OUT => ER(3));


OR1: OR2GATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                    GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => Z, INPUT(2) => V, OUTPUT => CO9,
                   TIME_SPEC_IN => TIME_BUS_FUN(BR), TIME_SPEC_OUT => CR(3));

OR2: OR3GATE
```

```
        generic map(4 ns, 4 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => CO6,  INPUT(2) => CO7, INPUT(3) => CO8, OUTPUT => F_OUT,
                    TIME_SPEC_IN => TIME_BUS_FUN(ER), TIME_SPEC_OUT => TIME_OUT);


F_OUTPUT <= F_OUT;

HAZARD_DETECTION1:process
variable DATA: INTEGER := 0;
begin

PRE_PROCESS_DATA(IN_VECTOR, F_OUT, TIME_OUT, DATA, STATE_VALUE_TABLE);

wait on F_OUT, IN_VECTOR;
end process HAZARD_DETECTION1;



HAZARD_DETECTION2:process(F_OUT, IN_VECTOR)
variable HAZARD_SCHEDULE1 : HAZARD_VAR := HAZARD_CONSTANT;
constant INFO: STATIC_INFO_VECTOR(0 to 1) :=
(("01000", "00000",  8 ns, 4 ns), ("01111", "01101", 7 ns, 4 ns));
begin


if (now >= PRE_TEST_BEGIN_TIME(TIME_OUT)) then
HAZARDS_PATTERNS_TEST(F_OUT, "F", HAZARD_SCHEDULE1, IN_VECTOR, PATT1, PATT2,
TIME_OUT, INFO);

end if;


end process HAZARD_DETECTION2;



end STRUCTURAL;

configuration CFG_HAZ_1 of HAZ_1 is
        for STRUCTURAL

        for INV1,INV2,INV3,INV4,INV5: INVGATE
        use entity VHDS_LIB.INVGATE(PURE_BEHAVIOR);
        end for;

        for AND1,AND2: AND3GATE
        use entity VHDS_LIB.AND3GATE(PURE_BEHAVIOR);
        end for;

        for AND3: AND4GATE
        use entity VHDS_LIB.AND4GATE(PURE_BEHAVIOR);
        end for;

        for OR1: OR2GATE
```

```
        use entity VHDS_LIB.OR2GATE(PURE_BEHAVIOR);
        end for;

        for OR2: OR3GATE
        use entity VHDS_LIB.OR3GATE(PURE_BEHAVIOR);
        end for;

end for;
end CFG_HAZ_1;
```

-- VHDS Test Bench Shell Design for the Two-Input Change Hazard Analysis

```
library VHDS_LIB;

use VHDS_LIB.MVL9_SYSTEM.all;
use VHDS_LIB.TIME_PACKAGE.all;
use VHDS_LIB.TEST_GENERATOR.all;
use VHDS_LIB.INPUT_DATA.all;

entity TEST_BENCH is
end TEST_BENCH;

architecture HAZ1TB of TEST_BENCH is

component HAZ_1
generic(GLITCH_0_TIME, GLITCH_1_TIME:TIME);

port(PATT1,PATT2: in MVL9_VECTOR(0 to 4); V, W, X, Y, Z: in MVL9;
   F_OUTPUT: out MVL9;
   STATE_VALUE_TABLE: inout MVL9_STATE_LOGIC_VECTOR(0 to 2**WORDLENGTH - 1);
   TIME_IN: in PROP_DELAY_SPEC := (0 ns, 0 ns);
   TIME_OUT: inout PROP_DELAY_SPEC := (0 ns, 0 ns));

end component;

signal PATT1, PATT2: MVL9_VECTOR(0 to 4);
signal V, W, X, Y, Z: MVL9;
signal F_OUTPUT: MVL9;
signal STATE_VALUE_TABLE: MVL9_STATE_LOGIC_VECTOR(0 to 2**WORDLENGTH - 1);
signal NUM1: INTEGER := 0;
signal TIME_IN: PROP_DELAY_SPEC := (0 ns, 0 ns);
signal TIME_OUT: PROP_DELAY_SPEC:= (0 ns, 0 ns);
constant TOTAL_PATTERN_FUNCTION: INTEGER :=
TOTAL_FUNCTION_PATTERNS(WORDLENGTH);
signal IN_VECTOR: MVL9_VECTOR(0 to 4);
signal INIT,INIT1: BIT := '0';
signal IN_VECTOR1, IN_VECTOR2: MVL9_VECTOR(0 to WORDLENGTH - 1);
begin

IN_VECTOR <= IN_VECTOR1 when not IN_VECTOR1'quiet else
                IN_VECTOR2 when not IN_VECTOR2'quiet else
                IN_VECTOR;


-------------------------------------------------
-- Generic Map Association
-------------------------------------------------
L1: HAZ_1 generic map(5 ns,5 ns)
port map(PATT1,PATT2,V, W, X, Y, Z, F_OUTPUT,STATE_VALUE_TABLE,TIME_IN,TIME_OUT);

V <= transport IN_VECTOR(0);
W <= transport IN_VECTOR(1);
X <= transport IN_VECTOR(2);
Y <= transport IN_VECTOR(3);
```

```
Z <= transport IN_VECTOR(4);


INIT <= '1' after 1 ns;


INJECTION:process(INIT)
begin
PRE_PROCESS_INJECTION(TIME_OUT, IN_VECTOR1);

if (now = 1 ns) then
INIT1 <= transport '1' after (PRE_TEST_BEGIN_TIME(TIME_OUT) - 1 ns);
end if;

end process INJECTION;



TESTING: process (INIT1)
variable RESULT2: MVL9_VECTOR_TRIPLE_VECTOR(0 to TOTAL_PATTERN_FUNCTION - 1);
variable NUMBER: NATURAL := 0;

begin

RESULT2 := FUNCTION_PATTERN_GENERATOR(WORDLENGTH);

NUMBER := PRE_PROCESS_COUNT(RESULT2, STATE_VALUE_TABLE);
NUM1 <= NUMBER;


PRE_FUNCTION_PATTERN_TESTER(PRE_PROCESS_FUNCTION_GENERATOR(RESULT2,
                STATE_VALUE_TABLE,NUMBER),
                TIME_OUT, NUMBER,
                PATTERN1 => PATT1, PATTERN2 => PATT2,
                OUT_PATTERN => IN_VECTOR2, MODE => FUNCT);


end process TESTING;

end HAZ1TB;


use WORK.all;
configuration CON_HAZ1 of TEST_BENCH is
for HAZ1TB
        for all: HAZ_1
        use entity WORK.HAZ_1(STRUCTURAL);
        end for;
end for;
end CON_HAZ1;
```

-- VHDS Report Summary for the Two-Input Change Hazard Analysis

```
        Test Circuit Statistics:
        Number of Primary Inputs: 5
Minimum Propagation Delay of the test circuit with respect to F: 7 NS
Maximum Propagation Delay of the test circuit with respect to F: 12 NS

********** VHDS Report Summary **********

=======================================
Function 0 hazard is detected at    529 ns on F
506 NS      00001
518 NS      11001
518 NS        0     F
526 NS        1     F
529 NS        0     F
=======================================
Function 0 hazard is detected at    577 ns on F
554 NS      00001
566 NS      01011
566 NS        0     F
574 NS        1     F
577 NS        0     F
=======================================
Function 0 hazard is detected at    864 ns on F
842 NS      00011
854 NS      00110
854 NS        0     F
861 NS        1     F
864 NS        0     F
=======================================
Function 0 hazard is detected at    913 ns on F
890 NS      00100
902 NS      11100
902 NS        0     F
910 NS        1     F
913 NS        0     F
=======================================
Function 0 hazard is detected at    961 ns on F
938 NS      00100
950 NS      01110
950 NS        0     F
958 NS        1     F
961 NS        0     F
=======================================
Function 0 hazard is detected at    1105 ns on F
1082 NS     00101
1094 NS     11101
1094 NS       0     F
1102 NS       1     F
1105 NS       0     F
=======================================
Function 0 hazard is detected at    1200 ns on F
1178 NS     00101
```

```
1190 NS        00110
1190 NS          0       F
1197 NS          1       F
1200 NS          0       F
=======================================
Function 0 hazard is detected at     1824 ns on F
1802 NS        01011
1814 NS        01110
1814 NS          0       F
1821 NS          1       F
1824 NS          0       F
=======================================
Function 0 hazard is detected at     2688 ns on F
2666 NS        10010
2678 NS        00110
2678 NS          0       F
2685 NS          1       F
2688 NS          0       F
=======================================
Function 0 hazard is detected at     2928 ns on F
2906 NS        10100
2918 NS        00110
2918 NS          0       F
2925 NS          1       F
2928 NS          0       F
=======================================
Function 1 hazard is detected at     3218 ns on F
3194 NS        10110
3206 NS        10000
3204 NS          1       F
3213 NS          0       F
3218 NS          1       F
=======================================
Function 0 hazard is detected at     3552 ns on F
3530 NS        11010
3542 NS        01110
3542 NS          0       F
3549 NS          1       F
3552 NS          0       F
=======================================
Function 0 hazard is detected at     3744 ns on F
3722 NS        11100
3734 NS        01110
3734 NS          0       F
3741 NS          1       F
3744 NS          0       F
=======================================
Function 1 hazard is detected at     3937 ns on F
3914 NS        11110
3926 NS        01100
3924 NS          1       F
3933 NS          0       F
3937 NS          1       F
```

5.5 A Test Circuit with Six Primary Inputs

Figure 36 shows a test circuit with six primary inputs. This test circuit is designed from [14]. The input signals of the test circuit are A, B, C, D, E, and F. The output of the test circuit is a MVL9 signal called F_OUTPUT. From the VHDS analysis, the min and max values of the test circuit with respect to output signal F_OUTPUT are 8 ns and 20 ns, respectively. In the report summaries, the VHDS found no static hazard, twenty-three function 0 hazards which have a pulse width duration of 3 ns, three function 0 hazards which have a pulse width duration of 4 ns, three function 0 hazards which have a pulse width duration of 5 ns, six function 0 hazards which have a pulse width duration of 7 ns, three function 0 hazards which have a pulse width duration of 8 ns, three function 1 hazards which have a pulse width duration of 2 ns, three function 1 hazards which have a pulse width duration of 3 ns, four function 1 hazards which have a pulse width duration of 4 ns, one function 1 hazard which has a pulse width duration of 8 ns, and one function 1 hazard which has a pulse width duration of 11 ns.

The VHDS source file, the VHDS test bench shell, and the VHDS report summary are enclosed in the following pages.

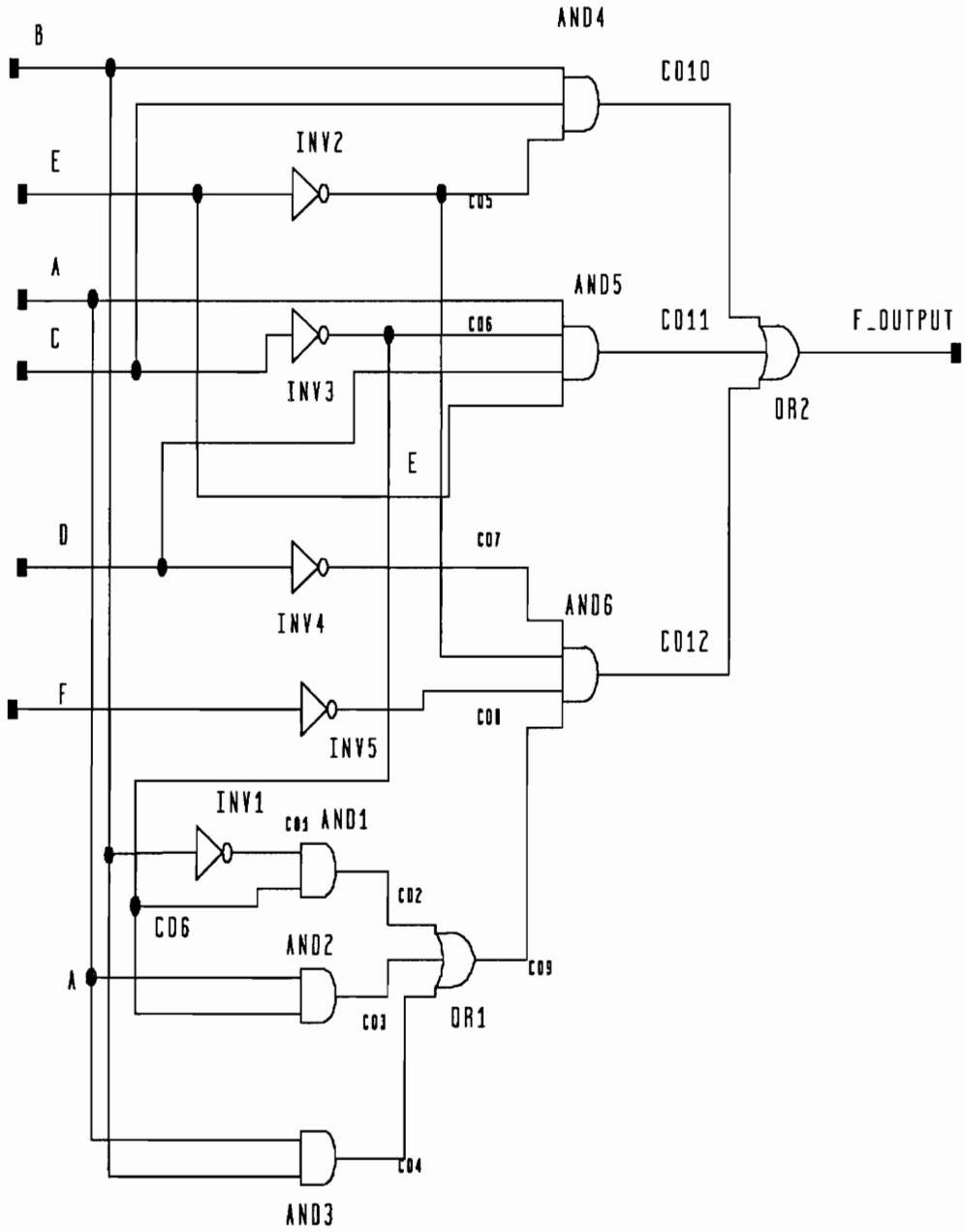


Figure 36. A Test Circuit with Six Primary Inputs

-- VHDS Source File Design for the Two-Input Change Hazard Analysis

```
library VHDS_LIB;

use VHDS_LIB.MVL9_SYSTEM.all;
use VHDS_LIB.TIME_PACKAGE.all;
use VHDS_LIB.PRE_HAZARD_TEST.all;
use VHDS_LIB.GATE_LIBRARY.all;
use VHDS_LIB.INPUT_DATA.all;
use VHDS_LIB.TEST_GENERATOR.all;


entity HAZ_1 is
generic(GLITCH_0_TIME, GLITCH_1_TIME:TIME);

port(PATT1, PATT2: in MVL9_VECTOR(0 to 5); A, B, C, D, E, F: in MVL9;  F_OUTPUT: out MVL9;
   STATE_VALUE_TABLE: inout MVL9_STATE_LOGIC_VECTOR(0 to 2**WORDLENGTH - 1);
   TIME_IN: in PROP_DELAY_SPEC := (0 ns, 0 ns);
   TIME_OUT: inout PROP_DELAY_SPEC := (0 ns, 0 ns));
end HAZ_1;

architecture STRUCTURAL of HAZ_1 is

subtype PROP_DELAY_SPEC_VECTOR4 is PROP_DELAY_SPEC_VECTOR(1 to 4);
subtype PROP_DELAY_SPEC_VECTOR3 is PROP_DELAY_SPEC_VECTOR(1 to 3);
subtype PROP_DELAY_SPEC_VECTOR2 is PROP_DELAY_SPEC_VECTOR(1 to 2);

signal CO1,CO2,CO3,CO4,CO5,CO6,CO7,CO8,CO9,CO10,CO11,CO12,F_OUT: MVL9;
signal IN_VECTOR: MVL9_VECTOR(0 to 5);
signal TIME_OUT_A, TIME_OUT_B, TIME_OUT_C, TIME_OUT_D, TIME_OUT_E: PROP_DELAY_SPEC
:=(0 ns, 0 ns);
signal AR,BR,CR: PROP_DELAY_SPEC_VECTOR2 := ((0 ns, 0 ns), (0 ns, 0 ns));
signal DR,ER,HR: PROP_DELAY_SPEC_VECTOR3 := ((0 ns, 0 ns), (0 ns, 0 ns), (0 ns, 0 ns));
signal FR,GR: PROP_DELAY_SPEC_VECTOR4 := ((0 ns, 0 ns), (0 ns, 0 ns), (0 ns, 0 ns), (0 ns, 0 ns));
begin

-------------------------------------------------------------------------
-- A timing resolve process to compute the max. and min. delay
-- of the whole structural component.
-------------------------------------------------------------------------
AR(1) <= TIME_OUT_A;

ER(3) <= TIME_OUT_B;
GR(2) <= TIME_OUT_B;

FR(2) <= TIME_OUT_C;
AR(2) <= TIME_OUT_C;
BR(2) <= TIME_OUT_C;

GR(1) <= TIME_OUT_D;

GR(3) <= TIME_OUT_E;
```

```
IN_VECTOR <= A & B & C & D & E & F;


INV1: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => B, OUTPUT => CO1,
                TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_A);

INV2: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => E, OUTPUT => CO5,
                 TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_B);

INV3: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => C, OUTPUT => CO6,
                TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_C);

INV4: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                         GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT => D, OUTPUT => CO7,
                TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_D);

INV5: INVGATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => F, OUTPUT => CO8,
                 TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_E);

AND1: AND2GATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => CO1,  INPUT(2) => CO6, OUTPUT => CO2,
                 TIME_SPEC_IN => TIME_BUS_FUN(AR), TIME_SPEC_OUT => DR(1));

AND2: AND2GATE
        generic map(4 ns, 4 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                         GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => A,  INPUT(2) => CO6, OUTPUT => CO3,
                 TIME_SPEC_IN => TIME_BUS_FUN(BR), TIME_SPEC_OUT => DR(2));

AND3: AND2GATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => A,  INPUT(2) => B, OUTPUT => CO4,
                 TIME_SPEC_IN => TIME_BUS_FUN(CR), TIME_SPEC_OUT => DR(3));

AND4: AND3GATE
        generic map(4 ns, 4 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                         GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => B,  INPUT(2) => C, INPUT(3) => CO5, OUTPUT => CO10,
```

```
                    TIME_SPEC_IN => TIME_BUS_FUN(ER), TIME_SPEC_OUT => HR(1));

AND5: AND4GATE
        generic map(5 ns, 5 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => A,  INPUT(2) => CO6,
                    INPUT(3) => D, INPUT(4) => E, OUTPUT => CO11,
                    TIME_SPEC_IN => TIME_BUS_FUN(FR), TIME_SPEC_OUT => HR(2));

AND6: AND4GATE
        generic map(5 ns, 5 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => CO7,  INPUT(2) => CO5,
                    INPUT(3) => CO8, INPUT(4) => CO9, OUTPUT => CO12,
                    TIME_SPEC_IN => TIME_BUS_FUN(GR), TIME_SPEC_OUT => HR(3));


OR1: OR3GATE
        generic map(4 ns, 4 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => CO2,  INPUT(2) => CO3, INPUT(3) => CO4, OUTPUT => CO9,
                    TIME_SPEC_IN => TIME_BUS_FUN(DR), TIME_SPEC_OUT => GR(4));

OR2: OR3GATE
        generic map(4 ns, 4 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => CO10,  INPUT(2) => CO11, INPUT(3) => CO12, OUTPUT => F_OUT,
                    TIME_SPEC_IN => TIME_BUS_FUN(HR), TIME_SPEC_OUT => TIME_OUT);


F_OUTPUT <= F_OUT;


HAZARD_DETECTION:process
variable DATA: INTEGER := 0;
begin

PRE_PROCESS_DATA(IN_VECTOR, F_OUT, TIME_OUT, DATA, STATE_VALUE_TABLE);

wait on F_OUT, IN_VECTOR;
end process HAZARD_DETECTION;



HAZARD_DETECTION1:process(F_OUT, IN_VECTOR)
variable HAZARD_SCHEDULE1 : HAZARD_VAR := HAZARD_CONSTANT;
begin

if (now >= PRE_TEST_BEGIN_TIME(TIME_OUT)) then
HAZARDS_PATTERNS_TEST(F_OUT, "F_OUTPUT", HAZARD_SCHEDULE1, IN_VECTOR, PATT1,
PATT2, TIME_OUT);

end if;
end process HAZARD_DETECTION1;
end STRUCTURAL;
```

```
configuration CFG_HAZ_1 of HAZ_1 is
        for STRUCTURAL

        for INV1,INV2,INV3,INV4,INV5: INVGATE
        use entity VHDS_LIB.INVGATE(PURE_BEHAVIOR);
        end for;

        for AND1,AND2,AND3: AND2GATE
        use entity VHDS_LIB.AND2GATE(PURE_BEHAVIOR);
        end for;

        for AND4: AND3GATE
        use entity VHDS_LIB.AND3GATE(PURE_BEHAVIOR);
        end for;

        for AND5,AND6: AND4GATE
        use entity VHDS_LIB.AND4GATE(PURE_BEHAVIOR);
        end for;

        for OR1,OR2: OR3GATE
        use entity VHDS_LIB.OR3GATE(PURE_BEHAVIOR);
        end for;

end for;
end CFG_HAZ_1;
```

-- VHDS Test Bench Shell for the Two-Input Change Hazard Analysis

```
library VHDS_LIB;

use VHDS_LIB.MVL9_SYSTEM.all;
use VHDS_LIB.TIME_PACKAGE.all;
use VHDS_LIB.TEST_GENERATOR.all;
use VHDS_LIB.INPUT_DATA.all;

entity TEST_BENCH is
end TEST_BENCH;

architecture HAZ1TB of TEST_BENCH is

component HAZ_1
generic(GLITCH_0_TIME, GLITCH_1_TIME:TIME);

port(PATT1,PATT2: in MVL9_VECTOR(0 to 5); A,B,C,D,E,F: in MVL9;
   F_OUTPUT: out MVL9;
   STATE_VALUE_TABLE: inout MVL9_STATE_LOGIC_VECTOR(0 to 2**WORDLENGTH - 1);
   TIME_IN: in PROP_DELAY_SPEC := (0 ns, 0 ns);
   TIME_OUT: inout PROP_DELAY_SPEC := (0 ns, 0 ns));

end component;

signal PATT1, PATT2: MVL9_VECTOR(0 to 5);
signal A,B,C,D,E,F: MVL9;
signal F_OUTPUT: MVL9;
signal STATE_VALUE_TABLE: MVL9_STATE_LOGIC_VECTOR(0 to 2**WORDLENGTH - 1);
signal TIME_IN: PROP_DELAY_SPEC := (0 ns, 0 ns);
signal TIME_OUT: PROP_DELAY_SPEC:= (0 ns, 0 ns);
constant TOTAL_PATTERN_FUNCTION: INTEGER :=
TOTAL_FUNCTION_PATTERNS(WORDLENGTH);
signal NUM1: INTEGER := 0;
signal IN_VECTOR: MVL9_VECTOR(0 to WORDLENGTH - 1);
signal INIT,INIT1: BIT := '0';
signal IN_VECTOR1, IN_VECTOR2: MVL9_VECTOR(0 to WORDLENGTH - 1);
begin


IN_VECTOR <= IN_VECTOR1 when not IN_VECTOR1'quiet else
                IN_VECTOR2 when not IN_VECTOR2'quiet else
                IN_VECTOR;

-----------------------------------------------
-- Generic Map Association
-----------------------------------------------
L1: HAZ_1 generic map(8 ns,8 ns)
port map(PATT1,PATT2,A,B,C,D,E,F,F_OUTPUT,STATE_VALUE_TABLE,TIME_IN,TIME_OUT);

A <= transport IN_VECTOR(0);
B <= transport IN_VECTOR(1);
C <= transport IN_VECTOR(2);
```

```
D <= transport IN_VECTOR(3);
E <= transport IN_VECTOR(4);
F <= transport IN_VECTOR(5);


INIT <= '1' after 1 ns;

INJECTION:process(INIT)
begin

PRE_PROCESS_INJECTION(TIME_OUT, IN_VECTOR1);

if (now = 1 ns) then
INIT1 <= transport '1' after (PRE_TEST_BEGIN_TIME(TIME_OUT) - 1 ns);
end if;

end process INJECTION;

TESTING: process (INIT1)
variable RESULT2: MVL9_VECTOR_TRIPLE_VECTOR(0 to TOTAL_PATTERN_FUNCTION - 1);

variable NUMBER: NATURAL := 0;

begin

RESULT2 := FUNCTION_PATTERN_GENERATOR(WORDLENGTH);

NUMBER := PRE_PROCESS_COUNT(RESULT2, STATE_VALUE_TABLE);
NUM1 <= NUMBER;

PRE_FUNCTION_PATTERN_TESTER(PRE_PROCESS_FUNCTION_GENERATOR(RESULT2,
                STATE_VALUE_TABLE,NUMBER),
                TIME_OUT, NUMBER,
                PATTERN1 => PATT1, PATTERN2 => PATT2,
                OUT_PATTERN => IN_VECTOR2, MODE => FUNCT);

end process TESTING;

end HAZ1TB;


use WORK.all;
configuration CON_HAZ1 of TEST_BENCH is
for HAZ1TB
        for all: HAZ_1
        use entity WORK.HAZ_1(STRUCTURAL);
        end for;
end for;
end CON_HAZ1;
```

-- VHDS Report Summary for the Two-Input Change Hazard Analysis

```
        Test Circuit Statistics:
        Number of Primary Inputs: 6
Minimum Propagation Delay of the test circuit with respect to F_OUTPUT: 8 NS
Maximum Propagation Delay of the test circuit with respect to F_OUTPUT: 20 NS

************************* VHDS Report Summary ************************

=======================================
Function 0 hazard is detected at    1521 ns on F_OUTPUT
1482 NS      000001
1502 NS      010000
1494 NS         0 F_OUTPUT
1514 NS         1 F_OUTPUT
1521 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    1601 ns on F_OUTPUT
1562 NS      000001
1582 NS      001000
1574 NS         0 F_OUTPUT
1594 NS         1 F_OUTPUT
1601 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    1841 ns on F_OUTPUT
1802 NS      000010
1822 NS      010000
1814 NS         0 F_OUTPUT
1834 NS         1 F_OUTPUT
1841 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    1921 ns on F_OUTPUT
1882 NS      000010
1902 NS      001000
1894 NS         0 F_OUTPUT
1914 NS         1 F_OUTPUT
1921 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    2161 ns on F_OUTPUT
2122 NS      000100
2142 NS      010000
2134 NS         0 F_OUTPUT
2154 NS         1 F_OUTPUT
2161 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    2241 ns on F_OUTPUT
2202 NS      000100
2222 NS      001000
2214 NS         0 F_OUTPUT
2234 NS         1 F_OUTPUT
2241 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    2474 ns on F_OUTPUT
```

```
2442 NS      000110
2462 NS      101110
2454 NS         0 F_OUTPUT
2471 NS         1 F_OUTPUT
2474 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    2714 ns on F_OUTPUT
2682 NS      000111
2702 NS      101111
2694 NS         0 F_OUTPUT
2711 NS         1 F_OUTPUT
2714 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    3033 ns on F_OUTPUT
3002 NS      001000
3022 NS      011010
3021 NS         0 F_OUTPUT
3030 NS         1 F_OUTPUT
3033 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    3433 ns on F_OUTPUT
3402 NS      001001
3422 NS      011011
3414 NS         0 F_OUTPUT
3430 NS         1 F_OUTPUT
3433 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    3593 ns on F_OUTPUT
3562 NS      001100
3582 NS      011110
3574 NS         0 F_OUTPUT
3590 NS         1 F_OUTPUT
3593 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    3753 ns on F_OUTPUT
3722 NS      001101
3742 NS      011111
3734 NS         0 F_OUTPUT
3750 NS         1 F_OUTPUT
3753 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    4393 ns on F_OUTPUT
4362 NS      010000
4382 NS      011010
4381 NS         0 F_OUTPUT
4390 NS         1 F_OUTPUT
4393 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    4553 ns on F_OUTPUT
4522 NS      010001
4542 NS      011011
4534 NS         0 F_OUTPUT
4550 NS         1 F_OUTPUT
4553 NS         0 F_OUTPUT
=======================================
```

```
Function 0 hazard is detected at    4713 ns on F_OUTPUT
4682 NS      010100
4702 NS      011110
4694 NS         0 F_OUTPUT
4710 NS         1 F_OUTPUT
4713 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    4873 ns on F_OUTPUT
4842 NS      010101
4862 NS      011111
4854 NS         0 F_OUTPUT
4870 NS         1 F_OUTPUT
4873 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    4954 ns on F_OUTPUT
4922 NS      010110
4942 NS      111110
4934 NS         0 F_OUTPUT
4951 NS         1 F_OUTPUT
4954 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    5194 ns on F_OUTPUT
5162 NS      010111
5182 NS      111111
5174 NS         0 F_OUTPUT
5191 NS         1 F_OUTPUT
5194 NS         0 F_OUTPUT
=======================================
Function 1 hazard is detected at    5438 ns on F_OUTPUT
5402 NS      011000
5422 NS      110000
5413 NS         1 F_OUTPUT
5430 NS         0 F_OUTPUT
5438 NS         1 F_OUTPUT
=======================================
Function 1 hazard is detected at    5521 ns on F_OUTPUT
5482 NS      011000
5502 NS      000000
5493 NS         1 F_OUTPUT
5510 NS         0 F_OUTPUT
5521 NS         1 F_OUTPUT
=======================================
Function 0 hazard is detected at    6402 ns on F_OUTPUT
6362 NS      100001
6382 NS      101000
6374 NS         0 F_OUTPUT
6394 NS         1 F_OUTPUT
6402 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    6714 ns on F_OUTPUT
6682 NS      100010
6702 NS      101110
6694 NS         0 F_OUTPUT
6711 NS         1 F_OUTPUT
6714 NS         0 F_OUTPUT
```

```
=======================================
Function 0 hazard is detected at    6802 ns on F_OUTPUT
6762 NS      100010
6782 NS      101000
6774 NS         0 F_OUTPUT
6794 NS         1 F_OUTPUT
6802 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    7114 ns on F_OUTPUT
7082 NS      100011
7102 NS      101111
7094 NS         0 F_OUTPUT
7111 NS         1 F_OUTPUT
7114 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    7362 ns on F_OUTPUT
7322 NS      100100
7342 NS      101000
7334 NS         0 F_OUTPUT
7354 NS         1 F_OUTPUT
7362 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    7434 ns on F_OUTPUT
7402 NS      100100
7422 NS      101110
7414 NS         0 F_OUTPUT
7431 NS         1 F_OUTPUT
7434 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    7754 ns on F_OUTPUT
7722 NS      100101
7742 NS      101111
7734 NS         0 F_OUTPUT
7751 NS         1 F_OUTPUT
7754 NS         0 F_OUTPUT
=======================================
Function 1 hazard is detected at    7914 ns on F_OUTPUT
7882 NS      100110
7902 NS      100000
7894 NS         1 F_OUTPUT
7911 NS         0 F_OUTPUT
7914 NS         1 F_OUTPUT
=======================================
Function 0 hazard is detected at    7993 ns on F_OUTPUT
7962 NS      101000
7982 NS      111010
7982 NS         0 F_OUTPUT
7990 NS         1 F_OUTPUT
7993 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at    8393 ns on F_OUTPUT
8362 NS      101001
8382 NS      111011
8374 NS         0 F_OUTPUT
8390 NS         1 F_OUTPUT
```

```
8393 NS          0 F_OUTPUT
=======================================
Function 0 hazard is detected at    8553 ns on F_OUTPUT
8522 NS      101100
8542 NS      111110
8534 NS          0 F_OUTPUT
8550 NS          1 F_OUTPUT
8553 NS          0 F_OUTPUT
=======================================
Function 0 hazard is detected at    8713 ns on F_OUTPUT
8682 NS      101101
8702 NS      111111
8694 NS          0 F_OUTPUT
8710 NS          1 F_OUTPUT
8713 NS          0 F_OUTPUT
=======================================
Function 1 hazard is detected at    9281 ns on F_OUTPUT
9242 NS      110000
9262 NS      000000
9258 NS          1 F_OUTPUT
9279 NS          0 F_OUTPUT
9281 NS          1 F_OUTPUT
=======================================
Function 0 hazard is detected at    9679 ns on F_OUTPUT
9642 NS      110001
9662 NS      010000
9654 NS          0 F_OUTPUT
9674 NS          1 F_OUTPUT
9679 NS          0 F_OUTPUT
=======================================
Function 0 hazard is detected at    9833 ns on F_OUTPUT
9802 NS      110001
9822 NS      111011
9814 NS          0 F_OUTPUT
9830 NS          1 F_OUTPUT
9833 NS          0 F_OUTPUT
=======================================
Function 0 hazard is detected at   10159 ns on F_OUTPUT
10122 NS     110010
10142 NS     010000
10134 NS         0 F_OUTPUT
10154 NS         1 F_OUTPUT
10159 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at   10234 ns on F_OUTPUT
10202 NS     110010
10222 NS     111110
10214 NS         0 F_OUTPUT
10231 NS         1 F_OUTPUT
10234 NS         0 F_OUTPUT
=======================================
Function 0 hazard is detected at   10554 ns on F_OUTPUT
10522 NS     110011
10542 NS     111111
10534 NS         0 F_OUTPUT
```

```
10551 NS          1 F_OUTPUT
10554 NS          0 F_OUTPUT
=======================================
Function 0 hazard is detected at   10719 ns on F_OUTPUT
10682 NS       110100
10702 NS       010000
10694 NS          0 F_OUTPUT
10714 NS          1 F_OUTPUT
10719 NS          0 F_OUTPUT
=======================================
Function 0 hazard is detected at   10954 ns on F_OUTPUT
10922 NS       110100
10942 NS       111110
10934 NS          0 F_OUTPUT
10950 NS          1 F_OUTPUT
10954 NS          0 F_OUTPUT
=======================================
Function 0 hazard is detected at   11354 ns on F_OUTPUT
11322 NS       110101
11342 NS       111111
11334 NS          0 F_OUTPUT
11350 NS          1 F_OUTPUT
11354 NS          0 F_OUTPUT
=======================================
Function 1 hazard is detected at   11513 ns on F_OUTPUT
11482 NS       110110
11502 NS       111100
11494 NS          1 F_OUTPUT
11511 NS          0 F_OUTPUT
11513 NS          1 F_OUTPUT
=======================================
Function 1 hazard is detected at   11594 ns on F_OUTPUT
11562 NS       110110
11582 NS       110000
11574 NS          1 F_OUTPUT
11591 NS          0 F_OUTPUT
11594 NS          1 F_OUTPUT
=======================================
Function 1 hazard is detected at   11673 ns on F_OUTPUT
11642 NS       110111
11662 NS       111101
11654 NS          1 F_OUTPUT
11671 NS          0 F_OUTPUT
11673 NS          1 F_OUTPUT
=======================================
Function 1 hazard is detected at   11761 ns on F_OUTPUT
11722 NS       111000
11742 NS       100000
11733 NS          1 F_OUTPUT
11758 NS          0 F_OUTPUT
11761 NS          1 F_OUTPUT
=======================================
Function 1 hazard is detected at   11834 ns on F_OUTPUT
11802 NS       111001
11822 NS       110000
```

```
11813 NS          1 F_OUTPUT
11830 NS          0 F_OUTPUT
11834 NS          1 F_OUTPUT
=======================================
Function 0 hazard is detected at   11918 ns on F_OUTPUT
11882 NS       111010
11902 NS       101000
11894 NS          0 F_OUTPUT
11914 NS          1 F_OUTPUT
11918 NS          0 F_OUTPUT
=======================================
Function 1 hazard is detected at   12154 ns on F_OUTPUT
12122 NS       111100
12142 NS       110000
12133 NS          1 F_OUTPUT
12150 NS          0 F_OUTPUT
12154 NS          1 F_OUTPUT
=======================================
Function 1 hazard is detected at   12234 ns on F_OUTPUT
12202 NS       111100
12222 NS       110110
12213 NS          1 F_OUTPUT
12230 NS          0 F_OUTPUT
12234 NS          1 F_OUTPUT
=======================================
Function 1 hazard is detected at   12314 ns on F_OUTPUT
12282 NS       111101
12302 NS       110111
12293 NS          1 F_OUTPUT
12310 NS          0 F_OUTPUT
12314 NS          1 F_OUTPUT
```

5.6 A Test Circuit with Four Primary Inputs (Dynamic Hazard Illustration)

Figure 37 shows a test circuit with four primary inputs. This test circuit is extracted from **[12]**. The input signals of the test circuit are W, X, Y, and Z. The output of the test circuit is a MVL9 signal called F. From the VHDS analysis, a hazard report summary is generated. From the summary, the min and max values of the test circuit with respect to output signal F are 8 ns and 14 ns, respectively. In the report summary, the VHDS found one static 0 hazard, three static 1 hazards, and one dynamic 0 hazard. The static 0 hazard exists in the test circuit when the inputs WXYZ change from "0000" to "0100" (i.e. X changes from '0' to '1'). The static 1 hazard exists in the test circuit when the inputs WXYZ changes from "0001" to "1001" (i.e. W changes from '0' to '1'), "0011" to "1011" (i.e. W changes from '0' to '1'), and "0011" to "0111" (i.e. X changes form '0' to '1'). The static hazard pulse has a duration of 2 ns. The dynamic 0 hazard exists in the test circuit when the inputs change from "0001" to "0101" (i.e. X changes from '0' to '1').

The VHDS source file, the VHDS test bench shell, and the VHDS report summary are enclosed in the following pages.

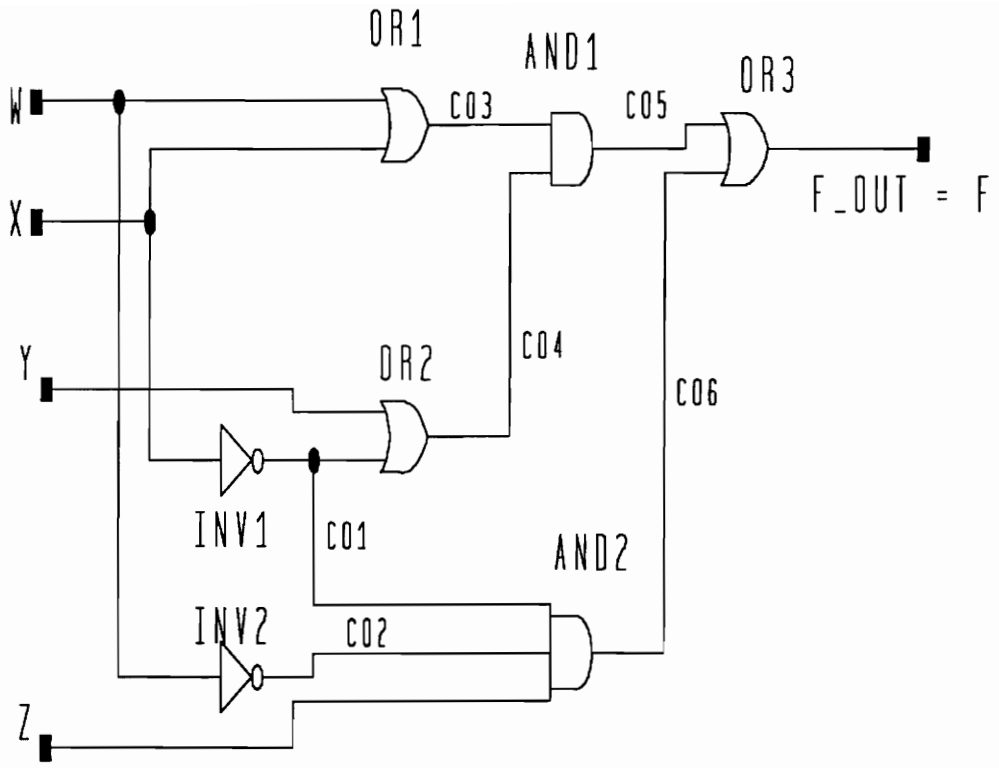


Figure 37. A Test Circuit with Four Primary Inputs (Dynamic Hazard Illustration)

-- VHDS Source File Design for the Single-Input Change Hazard Analysis

```
library VHDS_LIB;

use VHDS_LIB.MVL9_SYSTEM.all;
use VHDS_LIB.TIME_PACKAGE.all;
use VHDS_LIB.PRE_HAZARD_TEST.all;
use VHDS_LIB.GATE_LIBRARY.all;
use VHDS_LIB.INPUT_DATA.all;
use VHDS_LIB.TEST_GENERATOR.all;

entity HAZ_1 is
generic(GLITCH_0_TIME, GLITCH_1_TIME:TIME);


port(PATT1, PATT2: in MVL9_VECTOR(0 to 3); W,X,Y,Z: in MVL9; F: out MVL9;
     TIME_IN: in PROP_DELAY_SPEC := (0 ns, 0 ns);
     TIME_OUT: inout PROP_DELAY_SPEC := (0 ns, 0 ns));
end HAZ_1;

architecture STRUCTURAL of HAZ_1 is

subtype PROP_DELAY_SPEC_VECTOR3 is PROP_DELAY_SPEC_VECTOR(1 to 3);
subtype PROP_DELAY_SPEC_VECTOR2 is PROP_DELAY_SPEC_VECTOR(1 to 2);

signal IN_VECTOR: MVL9_VECTOR(0 to 3);
signal CO1, CO2, CO3, CO4, CO5, CO6, F_OUT: MVL9;
signal TIME_OUT_A, TIME_OUT_B: PROP_DELAY_SPEC :=(0 ns, 0 ns);

signal AR,BR,CR,ER: PROP_DELAY_SPEC_VECTOR2 := ((0 ns, 0 ns), (0 ns, 0 ns));
signal DR: PROP_DELAY_SPEC_VECTOR3 := ((0 ns, 0 ns), (0 ns, 0 ns), (0 ns, 0 ns));
begin

-------------------------------------------------------------------
-- A timing resolve process to compute the max. and min. delay
-- of the whole structural component.
-------------------------------------------------------------------
IN_VECTOR <= W & X & Y & Z;

BR(2) <= TIME_OUT_A;
DR(2) <= TIME_OUT_B;
DR(1) <= TIME_OUT_A;

INV1: INVGATE
        generic map(2 ns, 2 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT => X, OUTPUT => CO1,
                TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_A);

INV2: INVGATE
        generic map(2 ns, 2 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT => W, OUTPUT => CO2,
                TIME_SPEC_IN => TIME_IN, TIME_SPEC_OUT => TIME_OUT_B);
```

```
OR1: OR2GATE
        generic map(6 ns, 6 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME, CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => W, INPUT(2) => X, OUTPUT => CO3,
                    TIME_SPEC_IN => TIME_BUS_FUN(AR), TIME_SPEC_OUT => CR(1));

OR2: OR2GATE
        generic map(6 ns, 6 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => Y, INPUT(2) => CO1, OUTPUT => CO4,
                    TIME_SPEC_IN => TIME_BUS_FUN(BR), TIME_SPEC_OUT => CR(2));

OR3: OR2GATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => CO5, INPUT(2) => CO6, OUTPUT => F_OUT,
                    TIME_SPEC_IN => TIME_BUS_FUN(ER), TIME_SPEC_OUT => TIME_OUT);


AND1: AND2GATE
        generic map(3 ns, 3 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => CO3,  INPUT(2) => CO4, OUTPUT => CO5,
                    TIME_SPEC_IN => TIME_BUS_FUN(CR), TIME_SPEC_OUT => ER(1));

AND2: AND3GATE
        generic map(5 ns, 5 ns, GLITCH_0_TIME => GLITCH_0_TIME,
                        GLITCH_1_TIME => GLITCH_1_TIME,  CHOICE => TRANSPORT_DELAY)
        port map(INPUT(1) => CO1,  INPUT(2) => CO2,
                     INPUT(3) => Z, OUTPUT => CO6,
                     TIME_SPEC_IN => TIME_BUS_FUN(DR), TIME_SPEC_OUT => ER(2));

F <= transport F_OUT;


HAZARD_DETECTION: process(F_OUT,IN_VECTOR)
variable HAZARD_SCHEDULE1: HAZARD_VAR := HAZARD_CONSTANT;
begin

HAZARDS_PATTERNS_TEST(OUTPUT => F_OUT, NAME => "F",
HAZARD_SCHEDULE1 => HAZARD_SCHEDULE1,
PATTERNS => IN_VECTOR, PATT1 => PATT1,
PATT2 => PATT2, TIME_DELAY => TIME_OUT);


end process HAZARD_DETECTION;

end STRUCTURAL;

configuration CFG_HAZ_1 of HAZ_1 is
        for STRUCTURAL

        for INV1,INV2: INVGATE
        use entity VHDS_LIB.INVGATE(PURE_BEHAVIOR);
```

```
        end for;

        for AND1: AND2GATE
        use entity VHDS_LIB.AND2GATE(PURE_BEHAVIOR);
        end for;

        for AND2: AND3GATE
        use entity VHDS_LIB.AND3GATE(PURE_BEHAVIOR);
        end for;

        for OR1,OR2,OR3: OR2GATE
        use entity VHDS_LIB.OR2GATE(PURE_BEHAVIOR);
        end for;

end for;
end CFG_HAZ_1;
```

-- VHDS Test Bench Shell Design for the Single-Input Change Hazard Analysis

```
library VHDS_LIB;

use VHDS_LIB.MVL9_SYSTEM.all;
use VHDS_LIB.TIME_PACKAGE.all;
use VHDS_LIB.TEST_GENERATOR.all;
use VHDS_LIB.INPUT_DATA.all;


entity TEST_BENCH is
end TEST_BENCH;

architecture HAZ1TB of TEST_BENCH is

component HAZ_1
generic (GLITCH_0_TIME, GLITCH_1_TIME: TIME);

port(PATT1, PATT2: in MVL9_VECTOR(0 to 3);
     W,X,Y,Z: in MVL9;
     F: out MVL9;
     TIME_IN: in PROP_DELAY_SPEC;
     TIME_OUT: inout PROP_DELAY_SPEC);
end component;

signal PATT1, PATT2: MVL9_VECTOR(0 to 3);
signal W,X,Y,Z: MVL9;
signal F: MVL9;
signal TIME_IN: PROP_DELAY_SPEC := (0 ns, 0 ns);
signal TIME_OUT: PROP_DELAY_SPEC:= (0 ns, 0 ns);
constant TOTAL_PATTERN_STATIC: INTEGER := TOTAL_STATIC_PATTERNS(WORDLENGTH);
signal INIT: BIT := '0';
signal IN_VECTOR: MVL9_VECTOR(0 to 3);
begin

-----------------------------------------------
-- Generic Map Association
-----------------------------------------------
L1: HAZ_1 generic map(7 ns, 7 ns)
port map(PATT1, PATT2, W,X,Y,Z,F,TIME_IN,TIME_OUT);


INIT <= '1' after 1 ns;

W <= transport IN_VECTOR(0);
X <= transport IN_VECTOR(1);
Y <= transport IN_VECTOR(2);
Z <= transport IN_VECTOR(3);

TEST1:process (INIT)
variable RESULT1: MVL9_VECTOR_TUPLE_VECTOR(0 to TOTAL_PATTERN_STATIC - 1);

begin
```

```
RESULT1 := STATIC_PATTERN_GENERATOR(WORDLENGTH);

STATIC_PATTERN_TESTER(RESULT1, TIME_OUT, PATTERN1 => PATT1, PATTERN2 => PATT2,
                                    OUT_PATTERN => IN_VECTOR, MODE => STATIC);



end process TEST1;

end HAZ1TB;


use WORK.all;
configuration CON_HAZ1 of TEST_BENCH is
for HAZ1TB
        for all: HAZ_1
        use entity WORK.HAZ_1(STRUCTURAL);
        end for;
end for;
end CON_HAZ1;
```

-- VHDS Report Summary for the Single-Input Change Hazard Analysis

```
        Test Circuit Statistics:
        Number of Primary Inputs: 4
Minimum Propagation Delay of the test circuit with respect to F: 8 NS
Maximum Propagation Delay of the test circuit with respect to F: 14 NS

************************* VHDS Report Summary *************************

=======================================
Static 0 hazard is detected at     99 ns on F
71 NS       0000
85 NS       0100
83 NS         0      F
97 NS         1      F
99 NS         0      F
=======================================
Static 1 hazard is detected at     265 ns on F
239 NS      0001
253 NS      1001
249 NS        1      F
263 NS        0      F
265 NS        1      F
=======================================
Dynamic 0 hazard is detected at     323 ns on F
295 NS      0001
309 NS      0101
305 NS        1      F
319 NS        0      F
321 NS        1      F
323 NS        0      F
=======================================
Static 1 hazard is detected at     713 ns on F
687 NS      0011
701 NS      1011
697 NS        1      F
711 NS        0      F
713 NS        1      F
=======================================
Static 1 hazard is detected at     769 ns on F
743 NS      0011
757 NS      0111
753 NS        1      F
767 NS        0      F
769 NS        1      F
```

5.7 Overall Performance of the VHDS

The performance of the VHDS is determined by the real time required in finding all the potential hazards existed in the test circuit. Table 2 shows the summary of the real time required in finding all the hazards under the specific simulation order. As mentioned before, two simlation test modes are allowed in the VHDS simulation. They are the single-input change hazard detection (STATIC), and the two-input change hazard detection (FUNCT).

Table 2. Summary of the Real Time Required for the Sample Test Circuits (Unit in Minutes)

#	Circuit Model	STATIC	FUNCT	STATIC + FUNCT
1	3-INPUT CIRCUIT	1.19	1.14	2.33
2	4-INPUT CIRCUIT	2.19	2.63	4.82
3	5-INPUT CIRCUIT	3.21	3.79	7.00
4	6-INPUT CIRCUIT	8.41	9.15	18.56
5	4-INPUT[2]	1.99	1.71	3.70

The results of the first four circuit models in Table 2 have been plotted in Figure 38. Table 3 below shows the number of test patterns required to analyze the above sample circuits.

[2] This 4-input test circuit is used to illustrate the dynamic hazard

Table 3. Number of Test Patterns Required for Analyzing the Sample Test Circuits

Circuit Model	# of Single-Input Change Patterns	# of Original Test Patterns for Two-Input Change Hazard Analysis	# of Reduced Test Patterns for Two-Input Change Hazard Analysis
3-Input Circuit	24	48/2 = 24	8
4-Input Circuit	64	192/2 = 96	48
5-Input Circuit	160	640/2 = 320	76
6-Input Circuit	384	1920/2 = 960	146
4-Input Circuit[2]	64	192/2 = 96	20

As seen in Table 3, a large number of patterns have been reduced by using the reducing mechanism in the two-input pattern generator. Hence, a great amount of time for simulation is saved. Hence, the reducing mechanism is proved to be successful in the two-input change hazard detection.

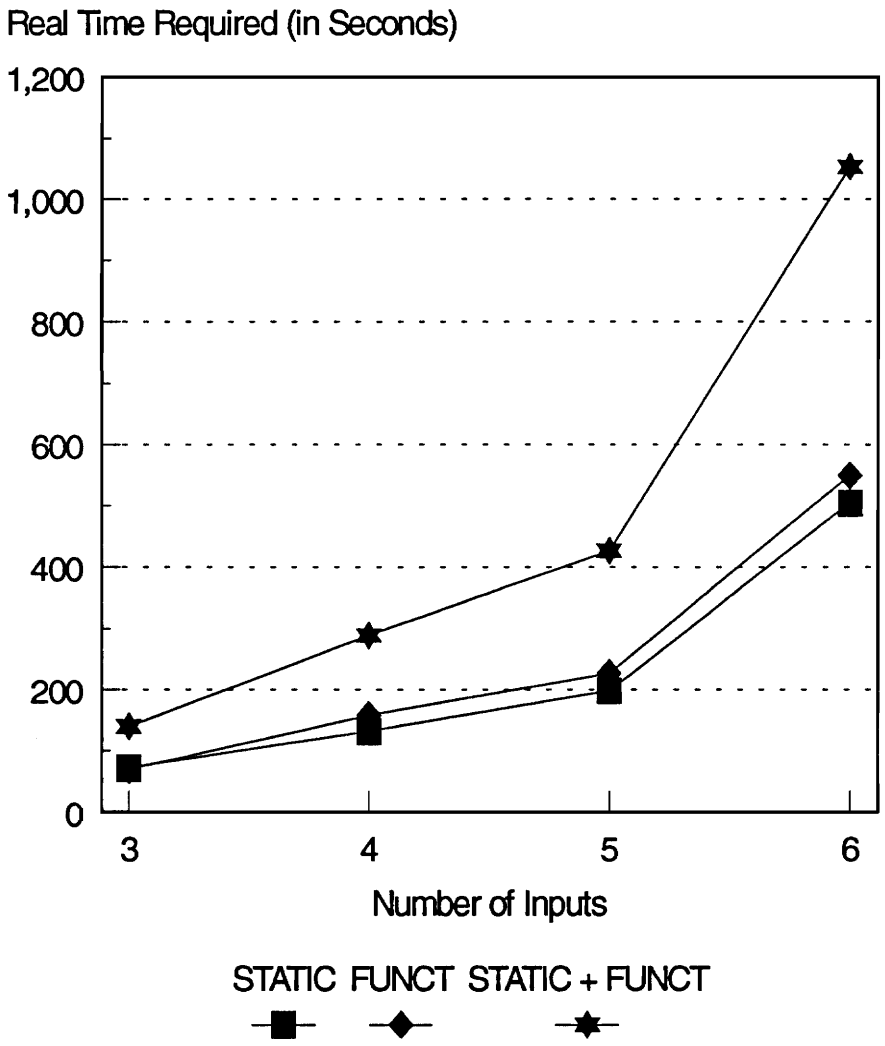


Figure 38. Real Time Required For Sample Test Circuits

Chapter 6. Conclusion

A new integrated hazard detection system (HDS), which is implemented in VHDL, has been developed. This system detects the static, the dynamic, and the function hazards in any logic circuit that is described structurally in VHDL. The system uses a new technique to compute the minimum propagation delay and the maximum propagation delay of any circuit that is described structurally in VHDL. The system evaluated five sample test circuits with hazard analysis. The real time required for the sample test circuits were satisfactory and the results were successful.

References

[1] E. B. Eichelberger, "Hazard Detection in Combinational and Sequential Switching Circuits," *IBM J. Res. Develop.* Vol. 9, No. 2, pp. 90-99, March 1965.

[2] D. W. Lewis, "Hazard Detection by Quinary Simulation of Logic Device with Bounded Propagation Delay," *Proc. Design Automation Workshop*, pp. 157-164, June 1972.

[3] M.A. Breuer and L. Harrison, "Procedures for Eliminating Static and Dynamic Hazards in Test Generation," *IEEE Trans. Computers*, Vol.C-23, pp. 1069-1078, Oct. 1974.

[4] Jonah Mcleod, "Catching Glitches and Delays in Dense ASIC Designs," *Electronics*, Febrary 18, 1988.

[5] *IEEE Standard VHDL Language Reference Manual*, IEEE, New York, NY, March 1988.

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

[7] Douglas L. Perry, *VHDL*, McGraw-Hill, Inc., New York, 1991.

[8] James R. Armstrong, F. Gail Gray, *Structured Logic Design with VHDL*, Book Draft.

[9] David R. Coelho, *The VHDL Handbook*, Kluwer Academic Press, Norwell, MA, 1989.

[10] Stephen H. Unger, *Asynchronous Sequential Switching Circuits*, Wiley-Interscience, New York, 1969.

[11] Edward J. McCluskey, *Logic Design Principles*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[12] John F. Wakerly, *Digital Design Principles and Practices*, Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[13] Arthur D. Friedman, Premachandran R. Menon, *Theory & Design of Switching Circuits*, Computer Science Press, Rockville, Maryland, 1975.

[14] Fredrick J. Hill, Gerald R. Peterson, *Introduction to Switching Theory and Logical Design*, John Wiley & Sons, New York, 1982.

[15] *VHDL Tech Notes*, The VHDL Consulting Group, Bethlehem, PA, July & August 1990.

[16] *VHDL System Simulator Reference Manual Part 1*, Synopsys Inc., Mountain View, CA, Nov. 1990.

[17] *VantageSpreadsheet User's Guide*, Vantage Analysis Systems, Fremont, CA, Mar. 1990.

[18] *STD_LOGIC_1164: Technical Overview*, IEEE VHDL Model Standards Group, July 1991.

Appendix A. The Nine-Valued MVL Package

```
library VHDS_LIB;
PACKAGE mvl9_system is

  ------------------------------------------------------------------
  -- Logic State System  (unresolved)
  ------------------------------------------------------------------
  TYPE MVL9 is ( 'U',  -- Unitialized
                 'X',  -- Forcing  0 or 1
                 '0',  -- Forcing  0
                 '1',  -- Forcing  1
                 'Z',  -- High Impedance
                 'W',  -- Weak    0 or 1
                 'L',  -- Weak    0      ( for ECL open emitter )
                 'H',  -- Weak    1      ( for open Drain or Collector )
                 '-'   -- don't care
                );
  ------------------------------------------------------------------
  -- Unconstrained array of std_ulogic for use with the resolution function
  ------------------------------------------------------------------
  TYPE MVL9_VECTOR IS ARRAY ( NATURAL RANGE <> ) of MVL9;

  ------------------------------------------------------------------
  -- Resolution function
  ------------------------------------------------------------------
  FUNCTION resolved ( s : MVL9_VECTOR ) RETURN MVL9;

  ------------------------------------------------------------------
  -- Three basic states
  ------------------------------------------------------------------
  SUBTYPE X01 is MVL9 RANGE 'X' to '1'; -- ('X','0','1')

  ------------------------------------------------------------------
  -- Unconstrained array of state for use in declaring registers
  ------------------------------------------------------------------
  TYPE X01_vector IS ARRAY ( NATURAL RANGE <> ) of X01;

  ------------------------------------------------------------------
  -- Overloaded Logical Operators
  ------------------------------------------------------------------

  FUNCTION "and"  ( l : MVL9; r : MVL9 ) RETURN MVL9;
  FUNCTION "nand" ( l : MVL9; r : MVL9 ) RETURN MVL9;
  FUNCTION "or"   ( l : MVL9; r : MVL9 ) RETURN MVL9;
  FUNCTION "nor"  ( l : MVL9; r : MVL9 ) RETURN MVL9;
  FUNCTION "xor"  ( l : MVL9; r : MVL9 ) RETURN MVL9;
  FUNCTION "not"  ( l : MVL9           ) RETURN MVL9;
```

```
-------------------------------------------------------------------
-- Vectorized Overloaded Logical Operators
-------------------------------------------------------------------
FUNCTION "and"  ( l, r : MVL9_VECTOR ) RETURN MVL9_VECTOR;
FUNCTION "nand" ( l, r : MVL9_VECTOR ) RETURN MVL9_VECTOR;
FUNCTION "or"   ( l, r : MVL9_VECTOR ) RETURN MVL9_VECTOR;
FUNCTION "nor"  ( l, r : MVL9_VECTOR ) RETURN MVL9_VECTOR;
FUNCTION "xor"  ( l, r : MVL9_VECTOR ) RETURN MVL9_VECTOR;
FUNCTION "not"  ( l    : MVL9_VECTOR ) RETURN MVL9_VECTOR;


-------------------------------------------------------------------
-- conversion functions
-------------------------------------------------------------------
FUNCTION To_bit      ( s : MVL9;       xmap : BIT := '0') RETURN BIT;
FUNCTION To_bitvector ( s : MVL9_VECTOR; xmap : BIT := '0') RETURN BIT_VECTOR;

-------------------------------------------------------------------
-- Types
-------------------------------------------------------------------
TYPE logic_X01_table is array (MVL9'low to MVL9'high) of X01;

-------------------------------------------------------------------
-- Tables
-------------------------------------------------------------------
-------------------------------------------------------
-- Table name : convert_to_X01
--
-- Parameters :
--       in  :: MVL9  -- some logic value
-- Returns    :  X01        -- state value of logic value
-- Purpose    :  to convert state-strength to state only
--
-- Example    : if (convert_to_X01 (input_signal) = '1' ) then ...
--
-------------------------------------------------------
CONSTANT convert_to_X01 : logic_X01_table := (
                'X',  -- 'U'
                'X',  -- 'X'
                '0',  -- '0'
                '1',  -- '1'
                'X',  -- 'Z'
                'X',  -- 'W'
                '0',  -- 'L'
                '1',  -- 'H'
                'X'   -- '-'
               );

END mvl9_system;
```

```
PACKAGE BODY mvl9_system is
  ------------------------------------------------------------
  -- Local Types
  ------------------------------------------------------------
  TYPE stdlogic_1D is array (MVL9) of MVL9;
  TYPE stdlogic_table is array(MVL9, MVL9) of MVL9;

  ------------------------------------------------------------
  -- Resolution function
  ------------------------------------------------------------
  CONSTANT resolution_table : stdlogic_table := (
        --    ---------------------------------------------------
        --    | U  X  0  1  Z  W  L  H  -              |  |
        --    ---------------------------------------------------
              ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ),   -- | U |
              ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ),   -- | X |
              ( 'U', 'X', '0', 'X', '0', '0', '0', '0', '0' ),   -- | 0 |
              ( 'U', 'X', 'X', '1', '1', '1', '1', '1', '1' ),   -- | 1 |
              ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-' ),   -- | Z |
              ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'W' ),-- | W |
              ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'L' ),   -- | L |
              ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'H' ),   -- | H |
              ( 'U', 'X', '0', '1', '-', 'W', 'L', 'H', '-' )    -- | - |
    );

  FUNCTION resolved ( s : MVL9_VECTOR ) RETURN MVL9 IS
    VARIABLE result : MVL9 := 'Z'; -- weakest state default
  BEGIN
    IF   (s'LENGTH = 1) THEN   RETURN s(s'LOW);
    ELSE
      -- Iterate through all inputs
      FOR i IN s'RANGE LOOP
        result := resolution_table(result, s(i));
      END LOOP;
      -- Return the resultant value
      RETURN result;
    END IF;
  END resolved;
```

```
-------------------------------------------------------------------
-- Tables for Logical Operations
-------------------------------------------------------------------

        -- truth table for "and" function
CONSTANT and_table : stdlogic_table := (
        --    ---------------------------------------------------
        --    | U   X   0   1   Z   W   L   H   -       |   |
        --    ---------------------------------------------------
              ( 'X', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | U |
              ( 'X', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | X |
              ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | 0 |
              ( 'X', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | 1 |
             ( 'X', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | Z |
             ( 'X', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | W |
             ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | L |
             ( 'X', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | H |
             ( 'X', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' )  -- | - |
);

        -- truth table for "or" function
        CONSTANT or_table : stdlogic_table := (
        --    ---------------------------------------------------
        --    | U   X   0   1   Z   W   L   H   -       |   |
        --    ---------------------------------------------------
              ( 'X', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ), -- | U |
             ( 'X', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ), -- | X |
             ( 'X', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | 0 |
             ( '1', '1', '1', '1', '1', '1', '1', '1', '1' ), -- | 1 |
             ( 'X', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ), -- | Z |
             ( 'X', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ), -- | W |
             ( 'X', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | L |
             ( '1', '1', '1', '1', '1', '1', '1', '1', '1' ), -- | H |
             ( 'X', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' )  -- | - |
);


        -- truth table for "xor" function
-- NOTE SPECIAL CASE FOR 'U' XOR 'U'
        CONSTANT xor_table : stdlogic_table := (
        --    ---------------------------------------------------
        --    | U   X   0   1   Z   W   L   H   -       |   |
        --    ---------------------------------------------------
              ( '0', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | U |
             ( 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
             ( 'X', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | 0 |
             ( 'X', 'X', '1', '0', 'X', 'X', '1', '0', 'X' ), -- | 1 |
```

```
        ( 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | Z |
        ( 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | W |
        ( 'X', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | L |
        ( 'X', 'X', '1', '0', 'X', 'X', '1', '0', 'X' ), -- | H |
        ( 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
);

    -- truth table for "not" function
    CONSTANT not_table: stdlogic_1D :=
    -- ---------------------------------------------
    -- |  U  X  0  1  Z  W  L  H  -  |
    -- ---------------------------------------------
       ( 'X', 'X', '1', '0', 'X', 'X', '1', '0', 'X' );

-------------------------------------------------------------------
-- Overloaded Logical Operators ( with optimizing hints )
-------------------------------------------------------------------

FUNCTION "and"  ( l : MVL9; r : MVL9 ) RETURN MVL9 IS
BEGIN
  RETURN (and_table(L, R));
END "and";

FUNCTION "nand" ( l : MVL9; r : MVL9 ) RETURN MVL9 IS
BEGIN
  RETURN  (not_table ( and_table(L, R)));
END "nand";

FUNCTION "or"   ( l : MVL9; r : MVL9 ) RETURN MVL9 IS
BEGIN
  RETURN (or_table(L, R));
END "or";

FUNCTION "nor"  ( l : MVL9; r : MVL9 ) RETURN MVL9 IS
BEGIN
  RETURN  (not ( or_table( L, R )));
END "nor";

FUNCTION "xor"  ( l : MVL9; r : MVL9 ) RETURN MVL9 IS
BEGIN
  RETURN (xor_table(L, R));
END "xor";

FUNCTION "not"  ( l : MVL9 ) RETURN MVL9 IS
BEGIN
  RETURN (not_table(L));
END "not";

-------------------------------------------------------------------
-- Vectorized Overloaded Logical Operators
```

```
--------------------------------------------------------------------
FUNCTION "and"  ( L,R : MVL9_VECTOR ) RETURN MVL9_VECTOR IS
  -- Note : Implementations may use aliases instead of the variables
  VARIABLE LV : MVL9_VECTOR ( 1 to L'length ) := L;
  VARIABLE RV : MVL9_VECTOR ( 1 to R'length ) := R;
  VARIABLE result : MVL9_VECTOR ( 1 to L'length ) := (Others => 'X');
begin
  if ( L'length /= R'length ) then
    assert false
    report "Arguments of overloaded 'and' operator are not of the same length"
    severity FAILURE;
  else
    for i in result'range loop
      result(i) := and_table (LV(i), RV(i));
    end loop;
  end if;
  return result;
end "and";

--------------------------------------------------------------------

FUNCTION "nand" ( L,R : MVL9_VECTOR ) RETURN MVL9_VECTOR IS
  -- Note : Implementations may use aliases instead of the variables
  VARIABLE LV : MVL9_VECTOR ( 1 to L'length ) := L;
  VARIABLE RV : MVL9_VECTOR ( 1 to R'length ) := R;
  VARIABLE result : MVL9_VECTOR ( 1 to L'length ) := (Others => 'X');
begin
  if ( L'length /= R'length ) then
    assert false
    report "Arguments of overloaded 'nand' operator are not of the same length"
    severity FAILURE;
  else
    for i in result'range loop
      result(i) := not_table(and_table (LV(i), RV(i)));
    end loop;
  end if;
  return result;
end "nand";

--------------------------------------------------------------------

FUNCTION "or"  ( L,R : MVL9_VECTOR ) RETURN MVL9_VECTOR IS
  -- Note : Implementations may use aliases instead of the variables
  VARIABLE LV : MVL9_VECTOR ( 1 to L'length ) := L;
  VARIABLE RV : MVL9_VECTOR ( 1 to R'length ) := R;
  VARIABLE result : MVL9_VECTOR ( 1 to L'length ) := (Others => 'X');
begin
  if ( L'length /= R'length ) then
    assert false
    report "Arguments of overloaded 'or' operator are not of the same length"
```

```
    severity FAILURE;
  else
    for i in result'range loop
      result(i) := or_table (LV(i), RV(i));
    end loop;
  end if;
  return result;
end "or";

-------------------------------------------------------------------

FUNCTION "nor"  ( L,R : MVL9_VECTOR ) RETURN MVL9_VECTOR IS
  -- Note : Implementations may use aliases instead of the variables
  VARIABLE LV : MVL9_VECTOR ( 1 to L'length ) := L;
  VARIABLE RV : MVL9_VECTOR ( 1 to R'length ) := R;
  VARIABLE result : MVL9_VECTOR ( 1 to L'length ) := (Others => 'X');
begin
  if ( l'length /= r'length ) then
    assert false
    report "Arguments of overloaded 'nor' operator are not of the same length"
    severity FAILURE;
  else
    for i in result'range loop
      result(i) := not_table(or_table (LV(i), RV(i)));
    end loop;
  end if;
  return result;
end "nor";

-------------------------------------------------------------------

FUNCTION "xor"  ( L,R : MVL9_VECTOR ) RETURN MVL9_VECTOR IS
  -- Note : Implementations may use aliases instead of the variables
  VARIABLE LV : MVL9_VECTOR ( 1 to L'length ) := L;
  VARIABLE RV : MVL9_VECTOR ( 1 to R'length ) := R;
  VARIABLE result : MVL9_VECTOR ( 1 to L'length ) := (Others => 'X');
begin
  if ( l'length /= r'length ) then
    assert false
    report "Arguments of overloaded 'xor' operator are not of the same length"
    severity FAILURE;
  else
    for i in result'range loop
      result(i) := xor_table (LV(i), RV(i));
    end loop;
  end if;
  return result;
end "xor";

-------------------------------------------------------------------
```

```
  FUNCTION "not"  ( l : MVL9_VECTOR ) RETURN MVL9_VECTOR IS
    -- Note : Implementations may use aliases instead of the variables
    VARIABLE LV : MVL9_VECTOR ( 1 to L'length ) := L;
    VARIABLE result : MVL9_VECTOR ( 1 to L'length ) := (Others => 'X');
  begin
    for i in result'range loop
      result(i) := not_table( LV(i) );
    end loop;
    return result;
  end;

  ---------------------------------------------------------------------
  -- conversion functions
  ---------------------------------------------------------------------
  FUNCTION To_bit      ( s : MVL9;      xmap : BIT := '0') RETURN BIT IS
  BEGIN
       CASE s IS
         WHEN '0' | 'L' => RETURN ('0');
         WHEN '1' | 'H' => RETURN ('1');
         WHEN OTHERS => RETURN xmap;
       END CASE;
  END;
  ---------------------------------------------------------------------
  FUNCTION To_bitvector ( s : MVL9_VECTOR; xmap : BIT := '0') RETURN BIT_VECTOR IS
    ALIAS sv : MVL9_VECTOR( s'LENGTH-1 DOWNTO 0 ) IS s;
    VARIABLE result : BIT_VECTOR ( s'LENGTH-1 DOWNTO 0 );
  BEGIN
    FOR i IN result'RANGE LOOP
      CASE sv(i) IS
        WHEN '0' | 'L' => result(i) := '0';
        WHEN '1' | 'H' => result(i) := '1';
        WHEN OTHERS => result(i) := xmap;
      END CASE;
    END LOOP;
    RETURN result;
  END;
  ---------------------------------------------------------------------


END mvl9_system;
```

Appendix B. The Nine-Valued Timing Package

```
library VHDS_LIB;
use VHDS_LIB.MVL9_SYSTEM.all;

package TIME_PACKAGE is


--=====================================================
-- Define necessary type:
--=====================================================
type TIME_MODEL_CHOICE is (INERTIAL_DELAY, TRANSPORT_DELAY,
DELTA_DELAY);
type TIME_VECTOR_6 is array (1 to 6) of TIME;

-- Define a record type
type PROP_DELAY_SPEC is
record
        MIN_PROP_DELAY: TIME;
        MAX_PROP_DELAY: TIME;
end record;

-- Define an vector array of a record type.
type PROP_DELAY_SPEC_VECTOR is array(POSITIVE RANGE <>) of PROP_DELAY_SPEC;

type TIME_VECTOR is array(NATURAL RANGE <>) of TIME;
type NATURAL_VECTOR is array(NATURAL RANGE <>) of NATURAL;


-- Type declaration that change from natural to character /natural to strings
subtype NUMERIC is NATURAL range 0 to 9;
type NUMERIC_VECTOR is array (NATURAL RANGE <>) of NUMERIC;
subtype NEW_STRING_10 is STRING(1 to 10);


--=================================================
-- FUNCTION DECLARATION:
--=================================================
-- Functions that used to determine the min and max values propagation delay of any circuit:

function MAX_TIME(TP_01,TP_10,TP_1Z,TP_Z1,TP_0Z,TP_Z0: TIME; TRI_STATE_OPTION:
STRING:= "OFF") return TIME;

function MIN_TIME(TP_01,TP_10,TP_1Z,TP_Z1,TP_0Z,TP_Z0: TIME; TRI_STATE_OPTION:
STRING:= "OFF") return TIME;

function MAX_TIME_ALL(INPUT_TIME_VECTOR: in TIME_VECTOR) return TIME;
```

```
function MIN_TIME_ALL(INPUT_TIME_VECTOR: in TIME_VECTOR) return TIME;

function MAX_MIN_DELAY_COMPUTE(TP_01,TP_10,TP_1Z,TP_Z1,TP_0Z,TP_Z0,
CONN_DELAY: TIME; OPTION: STRING := "OFF"; TIME_SPEC_IN: PROP_DELAY_SPEC:=(0
ns, 0 ns))
return PROP_DELAY_SPEC;


-- Bus resolution function in time domain.
function TIME_BUS_FUN(MAX_MIN_TIME_SET: PROP_DELAY_SPEC_VECTOR)
return PROP_DELAY_SPEC;

-----------------------------------------------------------------------------------
-- This package contains the nine-valued rise/fall delays model:
--        1) Inertial/Transport/Delta delay option
--        2) Rise/Fall time selections on binary logic
--          (TP_01, TP_10). And, rise/fall delay options
--          on tri-state logic (TP_1Z, TP_Z1, TP_0Z, TP_Z0)
--        3) Propagation delay + interconnection wire delay
-----------------------------------------------------------------------------------

procedure TIME_MODEL_SELECTION_1(OUTPUT_SIG: in MVL9; signal OUTPUT: out MVL9;
CHOICE: TIME_MODEL_CHOICE; LAST_DELAY_TIME: inout TIME;
OUTPUT_SIG_LAST_VALUE: inout MVL9;
signal OUTPUT_LAST_VALUE: inout MVL9;
TP_01, TP_10, TP_1Z, TP_Z1, TP_0Z, TP_Z0, CONN_DELAY: TIME);


procedure SPIKE_DETECTION(signal OUTPUT_SIGNAL: in MVL9; OUTPUT_LAST_EVENT:
inout TIME; GLITCH_0_TIME, GLITCH_1_TIME: in TIME);

function TIME_TO_NATURAL(TIME_VALUE: in TIME) return NATURAL;

function TIME_TO_NATURAL(TIME_VALUE_VECTOR: in TIME_VECTOR) return
NATURAL_VECTOR;

function NATURAL_TO_TIME(NATURAL_VALUE: in NATURAL) return TIME;

function NATURAL_TO_TIME(NATURAL_VALUE_VECTOR: in NATURAL_VECTOR)
return TIME_VECTOR;

function NATURAL_TO_STRING(NUMBER: in NATURAL) return NEW_STRING_10;

function NATURAL_TO_REAL(INPUT: NATURAL) return REAL;

end TIME_PACKAGE;
```

```
---------------------------------
-- Package Body Description
---------------------------------
package body TIME_PACKAGE is

function MAX_TIME(TP_01,TP_10,TP_1Z,TP_Z1,TP_0Z,TP_Z0: TIME; TRI_STATE_OPTION:
STRING := "OFF") return TIME is

variable RESULT : TIME_VECTOR_6 := (0 ns, 0 ns, 0 ns, 0 ns, 0 ns, 0 ns);
variable TIME_OUT: TIME := 0 ns;
variable I : NATURAL := 1;
begin

if (TRI_STATE_OPTION = "ON") then
        RESULT := TP_01 & TP_10 & TP_1Z & TP_Z1 & TP_0Z & TP_Z0;

        for I in 1 to 5 loop
        if RESULT(I + 1) >= RESULT(I) then
        RESULT(I + 1) := RESULT(I + 1);
        else
        RESULT(I + 1) := RESULT(I);
        end if;
        TIME_OUT := RESULT(I + 1);
        end loop;

elsif(TRI_STATE_OPTION = "OFF") then
        if (TP_01 >= TP_10) then
        TIME_OUT := TP_01;
        else
        TIME_OUT := TP_10;
        end if;

else
        assert (FALSE)
        report "Incorrect tri_state_option, please check!"
        severity WARNING;

end if;

return TIME_OUT;
end MAX_TIME;


function MIN_TIME(TP_01,TP_10,TP_1Z,TP_Z1,TP_0Z,TP_Z0: TIME; TRI_STATE_OPTION:
STRING:= "OFF")
return TIME is

variable RESULT : TIME_VECTOR_6 := (0 ns, 0 ns, 0 ns, 0 ns, 0 ns, 0 ns);
variable TIME_OUT: TIME := 0 ns;
variable I : NATURAL := 1;
```

```
begin
if (TRI_STATE_OPTION = "ON") then
        RESULT := TP_01 & TP_10 & TP_1Z & TP_Z1 & TP_0Z & TP_Z0;

        for I in 1 to 5 loop
        if RESULT(I + 1) <= RESULT(I) then
        RESULT(I + 1) := RESULT(I + 1);
        else
        RESULT(I + 1) := RESULT(I);
        end if;
        TIME_OUT := RESULT(I + 1);
        end loop;

elsif(TRI_STATE_OPTION = "OFF") then
        if (TP_01 >= TP_10) then
        TIME_OUT := TP_10;
        else
        TIME_OUT := TP_01;
        end if;

else
        assert (FALSE)
        report "Incorrect tri_state_option, please check!"
        severity WARNING;


end if;

return TIME_OUT;
end MIN_TIME;


_*****
function MAX_MIN_DELAY_COMPUTE(TP_01,TP_10,TP_1Z,TP_Z1,TP_0Z,TP_Z0,
CONN_DELAY: TIME; OPTION: STRING :="OFF"; TIME_SPEC_IN: PROP_DELAY_SPEC:=(0
ns, 0 ns))
return PROP_DELAY_SPEC is

variable TIME_SPEC_OUT: PROP_DELAY_SPEC;
begin

TIME_SPEC_OUT := ((MIN_TIME(TP_01,TP_10,TP_1Z,TP_Z1,TP_0Z,TP_Z0, OPTION) +
                        TIME_SPEC_IN.MIN_PROP_DELAY + CONN_DELAY),
                       (MAX_TIME(TP_01,TP_10,TP_1Z,TP_Z1,TP_0Z,TP_Z0, OPTION) +
                        TIME_SPEC_IN.MAX_PROP_DELAY + CONN_DELAY));


return TIME_SPEC_OUT;
end MAX_MIN_DELAY_COMPUTE;
```

```
function MAX_TIME_ALL(INPUT_TIME_VECTOR: in TIME_VECTOR) return TIME is
variable RESULT: TIME := 0 ns;
variable I : NATURAL := 1;
variable INPUT_RANGE: TIME_VECTOR(1 to INPUT_TIME_VECTOR'length - 1);
begin
-- Iterate through all time inputs
        for I in INPUT_RANGE'range loop

        if (I = 1) then
        RESULT := INPUT_TIME_VECTOR(I);
        end if;
        if (I /= 1 and INPUT_TIME_VECTOR(I) >= RESULT) then
        RESULT := INPUT_TIME_VECTOR(I);
        end if;

        if (INPUT_TIME_VECTOR(I + 1) >=  RESULT) then
        RESULT := INPUT_TIME_VECTOR(I + 1);
        end if;
        end loop;

return RESULT;
end MAX_TIME_ALL;



function MIN_TIME_ALL(INPUT_TIME_VECTOR: in TIME_VECTOR) return TIME is
variable RESULT: TIME := 0 ns;
variable I : NATURAL := 1;
variable INPUT_RANGE: TIME_VECTOR(1 to INPUT_TIME_VECTOR'length - 1);
begin
-- Iterate through all time inputs
        for I in INPUT_RANGE'range loop
        if (I = 1) then
        RESULT := INPUT_TIME_VECTOR(I);
        end if;
        if (I /= 1 and INPUT_TIME_VECTOR(I) <= RESULT) then
        RESULT := INPUT_TIME_VECTOR(I);
        end if;

        if (INPUT_TIME_VECTOR(I + 1) <= RESULT) then
        RESULT := INPUT_TIME_VECTOR(I + 1);
        end if;
        end loop;

return RESULT;
end MIN_TIME_ALL;


function TIME_BUS_FUN(MAX_MIN_TIME_SET: PROP_DELAY_SPEC_VECTOR)
```

```
return PROP_DELAY_SPEC is

variable MAX_TIME_SET, MIN_TIME_SET: TIME_VECTOR(1 to
MAX_MIN_TIME_SET'length);
variable INPUT_RANGE: PROP_DELAY_SPEC_VECTOR(1 to MAX_MIN_TIME_SET'length);
begin

-- Iterate for all inputs
for I in INPUT_RANGE'range loop
MAX_TIME_SET(I) := MAX_MIN_TIME_SET(I).MAX_PROP_DELAY;
MIN_TIME_SET(I) := MAX_MIN_TIME_SET(I).MIN_PROP_DELAY;
end loop;


return (MIN_TIME_ALL(MIN_TIME_SET),  MAX_TIME_ALL(MAX_TIME_SET));
end TIME_BUS_FUN;

_*****

-------------------------------------------------------------------------------
-- The procedure is the TIME_MODEL_SELECTION_1, i.e. the nine-valued rise/fall delays
-- model. It computes the propagation delay and selects the delay model type.
-------------------------------------------------------------------------------
procedure TIME_MODEL_SELECTION_1(OUTPUT_SIG: in MVL9; signal OUTPUT: out MVL9;
CHOICE: TIME_MODEL_CHOICE; LAST_DELAY_TIME: inout TIME;
OUTPUT_SIG_LAST_VALUE: inout MVL9;
signal OUTPUT_LAST_VALUE: inout MVL9;
TP_01, TP_10, TP_1Z, TP_Z1, TP_0Z, TP_Z0, CONN_DELAY: TIME) is

variable OUTPUT_CURR: MVL9;
variable OUTPUT_LAST_CURR: MVL9_VECTOR(0 to 1);
variable GATE_DELAY : TIME := 0 ns;
begin

OUTPUT_CURR:= OUTPUT_SIG;
OUTPUT_LAST_CURR := OUTPUT_LAST_VALUE & OUTPUT_CURR;


if (OUTPUT_SIG_LAST_VALUE /= OUTPUT_CURR) then
case OUTPUT_LAST_CURR is

        when  "XX" | "XU" | "X-" | "XW" |
              "UX" | "UU" | "U-" | "UW" |
              "-X" | "-U" | "--" | "-W" |
              "WX" | "WU" | "W-" | "WW"  =>    GATE_DELAY := 0 ns;  -- No delay

        when  "11" | "1H" | "H1" | "HH" |
              "X1" | "XH" | "U1" | "UH" | "-1" | "-H" | "W1" | "WH" |
               "01" | "0H" | "L1" | "LH" | "0X" | "LX" | "0U" | "LU" |
                "0-" | "L-" | "0W" | "LW"  =>             GATE_DELAY:= TP_01; --binary rise delay
```

```
        when "1Z" | "HZ"        =>          GATE_DELAY := TP_1Z; -- tri-state rise delay

        when "Z1" | "ZH"        =>          GATE_DELAY := TP_Z1; -- tri-state fall delay

        when "0Z" | "LZ"        =>          GATE_DELAY := TP_0Z; -- tri-state rise delay

        when "Z0" | "ZL"        =>          GATE_DELAY := TP_Z0; -- tri-state fall delay

        when others             =>          GATE_DELAY := TP_10; -- binary fall delay

end case;
else
        GATE_DELAY := LAST_DELAY_TIME - CONN_DELAY;
end if;

assert not ((GATE_DELAY /= 0 ns or CONN_DELAY /= 0 ns) and CHOICE =
DELTA_DELAY)
        report "WARNING! DELTA_DELAY is not properly implmented!"
                severity WARNING;

if (GATE_DELAY = 0 ns and CONN_DELAY = 0 ns and CHOICE = DELTA_DELAY) then
OUTPUT <= OUTPUT_SIG;
OUTPUT_LAST_VALUE <= OUTPUT_SIG;

elsif (GATE_DELAY = 0 ns and (CHOICE = INERTIAL_DELAY or CHOICE =
TRANSPORT_DELAY)) then
OUTPUT <= OUTPUT_SIG after CONN_DELAY;
OUTPUT_LAST_VALUE <= OUTPUT_SIG after CONN_DELAY;

elsif(GATE_DELAY /= 0 ns and CHOICE = INERTIAL_DELAY) then
OUTPUT <= OUTPUT_SIG after (GATE_DELAY + CONN_DELAY);
OUTPUT_LAST_VALUE <= OUTPUT_SIG after (GATE_DELAY + CONN_DELAY);

elsif(GATE_DELAY /= 0 ns and CHOICE = TRANSPORT_DELAY) then
OUTPUT <= transport OUTPUT_SIG after (GATE_DELAY + CONN_DELAY);
OUTPUT_LAST_VALUE <= transport OUTPUT_SIG after (GATE_DELAY + CONN_DELAY);

end if;

OUTPUT_SIG_LAST_VALUE := OUTPUT_SIG;
LAST_DELAY_TIME := GATE_DELAY + CONN_DELAY;

end TIME_MODEL_SELECTION_1;
```

```
-- This procedure detects spike(s) on the output.
-- It is not only suitable in detecting a spike in a
-- VHDL component but also, it may trace the flow of
-- hazard, especially in a multi-level complex circuit.
-------------------------------------------------------------
procedure SPIKE_DETECTION(signal OUTPUT_SIGNAL: in MVL9; OUTPUT_LAST_EVENT:
inout TIME; GLITCH_0_TIME, GLITCH_1_TIME: in TIME) is
begin

if (OUTPUT_SIGNAL = '0' and OUTPUT_SIGNAL'event and OUTPUT_SIGNAL'last_value =
'1') then
assert ((now = 0 ns) or ((now - OUTPUT_LAST_EVENT) = 0 ns) or
(now - OUTPUT_LAST_EVENT) > GLITCH_0_TIME)
report "A 0 glitch is detected on the output signal."
severity WARNING;
end if;

if (OUTPUT_SIGNAL = '1' and OUTPUT_SIGNAL'event and OUTPUT_SIGNAL'last_value =
'0') then
assert ((now = 0 ns) or ((now - OUTPUT_LAST_EVENT) = 0 ns) or
(now - OUTPUT_LAST_EVENT) > GLITCH_1_TIME)
report "A 1 glitch is detected on the output signal."
severity WARNING;
end if;

if (OUTPUT_SIGNAL'active) then
OUTPUT_LAST_EVENT := now;
end if;

end SPIKE_DETECTION;

-------------------------------------------------------------------------
-- This function converts the time value into natural value.
-- The time value cannot be negative.
-------------------------------------------------------------------------
function TIME_TO_NATURAL(TIME_VALUE: in TIME) return NATURAL is
variable NATURAL_VALUE: NATURAL := 0;
begin

NATURAL_VALUE := TIME_VALUE/(1 ns);

if (TIME_VALUE >= 0 ns) then
return NATURAL_VALUE;
else
assert FALSE
report "The value of time must be positive integer with ns time base!"
severity ERROR;
end if;
end TIME_TO_NATURAL;
```

```
-------------------------------------------------------------------------------
-- A vectorized overloaded function that converts the time_vector values
-- into natural_vector value.
-- Any value in the time_vector cannot be negative.
-------------------------------------------------------------------------------
function TIME_TO_NATURAL(TIME_VALUE_VECTOR: in TIME_VECTOR) return
NATURAL_VECTOR is
variable NATURAL_VALUE_VECTOR: NATURAL_VECTOR(1 to
TIME_VALUE_VECTOR'length);
variable I: NATURAL := 1;
begin

for I in TIME_VALUE_VECTOR'range loop
        NATURAL_VALUE_VECTOR(I) := TIME_VALUE_VECTOR(I)/(1 ns);

        if (TIME_VALUE_VECTOR(I) >= 0 ns) then
        return NATURAL_VALUE_VECTOR;
        else
        assert FALSE
        report "The value of time must be positive integer with ns time base!"
        severity ERROR;
        end if;
end loop;
end TIME_TO_NATURAL;


-------------------------------------------------------------------
-- This function converts the natural value to the time value.
-------------------------------------------------------------------
function NATURAL_TO_TIME(NATURAL_VALUE: in NATURAL) return TIME is
variable TIME_VALUE: TIME;
begin

TIME_VALUE := NATURAL_VALUE * (1 ns);

if (NATURAL_VALUE >= 0) then
return TIME_VALUE;
else
assert FALSE
report "The integer value must be positive!"
severity ERROR;
end if;

end NATURAL_TO_TIME;




-------------------------------------------------------------------------------
```

```
-- A vectorized overloaded function converts the natural_vector values
-- into time_vector value.
-- Any value in the integer_vector cannot be negative.
-------------------------------------------------------------------------------
function NATURAL_TO_TIME(NATURAL_VALUE_VECTOR: in NATURAL_VECTOR)
                    return TIME_VECTOR is
variable TIME_VALUE_VECTOR: TIME_VECTOR(1 to NATURAL_VALUE_VECTOR'length);
variable I: NATURAL := 1;
begin

for I in NATURAL_VALUE_VECTOR'range loop
        TIME_VALUE_VECTOR(I) := NATURAL_VALUE_VECTOR(I) * (1 ns);

        if (NATURAL_VALUE_VECTOR(I) >= 0) then
        return TIME_VALUE_VECTOR;
        else
        assert FALSE
        report "The integer value must be positive!"
        severity ERROR;
        end if;
end loop;
end NATURAL_TO_TIME;



-------------------------------------------------------------------------
-- This function converts any natural into character/string type.
-- The integer is valid from 0 to 1000000000.
-------------------------------------------------------------------------
function NATURAL_TO_STRING(NUMBER: in NATURAL) return NEW_STRING_10 is
variable I,J,RESULT,DIGIT_10,TMP_DIGIT, REDUCE: NATURAL := 0;
variable ANSWER: NUMERIC_VECTOR(0 to 9);
variable OUTPUT, OUTPUT_NEW : NEW_STRING_10;
begin

assert (NUMBER <= 1000000000)
report " The integer value must be in the range from 0 to 1000000000"
severity ERROR;

-- 1) First step is to test how big the integer is:
--   This algorithm can change the integer range from 0 to 1000000000

if (NUMBER = 0) then
OUTPUT_NEW := "         0";
elsif (NUMBER = 1000000000) then
OUTPUT_NEW := "1000000000";
else
for I in 0 to 9 loop
DIGIT_10 := 10 ** I;
TMP_DIGIT := I;
```

```
exit when (NUMBER < DIGIT_10);
end loop;

REDUCE := TMP_DIGIT - 1;
RESULT := NUMBER;
for J in 0 to REDUCE loop
ANSWER(J) := RESULT rem (10);
RESULT := RESULT/10;
-- Change the whole integer into bit-slice digit
case ANSWER(J) is
        when 0 => OUTPUT(J + 1) := '0';
        when 1 => OUTPUT(J + 1) := '1';
        when 2 => OUTPUT(J + 1) := '2';
        when 3 => OUTPUT(J + 1) := '3';
        when 4 => OUTPUT(J + 1) := '4';
        when 5 => OUTPUT(J + 1) := '5';
        when 6 => OUTPUT(J + 1) := '6';
        when 7 => OUTPUT(J + 1) := '7';
        when 8 => OUTPUT(J + 1) := '8';
        when 9 => OUTPUT(J + 1) := '9';
end case;
end loop;

I := 0;
for J in 10 downto 10 - REDUCE loop
OUTPUT_NEW(J) := OUTPUT(I + 1);
I := I + 1;
end loop;

for J in 1 to 9 - REDUCE loop
OUTPUT_NEW(J) := ' ';
end loop;
end if;

return OUTPUT_NEW;
end NATURAL_TO_STRING;

-------------------------------------------------------------
-- This function returns natural value to floating
-- value.
-------------------------------------------------------------
function NATURAL_TO_REAL(INPUT: NATURAL) return REAL is
begin
return real(INPUT);
end NATURAL_TO_REAL;


end TIME_PACKAGE;
```

Appendix C. The VHDS Hazard Pattern Generator

```
library VHDS_LIB;
use VHDS_LIB.MVL9_SYSTEM.all;
use VHDS_LIB.TIME_PACKAGE.all;
use VHDS_LIB.INPUT_DATA.all;


package TEST_GENERATOR is

-----------------------
-- type declaration
-----------------------
type MVL9_VECTOR_ARRAY is array(NATURAL RANGE <>) of
MVL9_VECTOR(0 to WORDLENGTH - 1);

type BOOLEAN_ARRAY is array (NATURAL range <>) of BOOLEAN;

type NATURAL_VECTOR is array(NATURAL range <>) of NATURAL;

type MVL9_VECTOR_TUPLE is
record
        ORG_CELL:    MVL9_VECTOR(WORDLENGTH - 1 downto 0);
        NEG_CELL:    MVL9_VECTOR(WORDLENGTH - 1 downto 0);
end record;


type MVL9_VECTOR_TRIPLE is
record
        ORG_CELL:           MVL9_VECTOR(WORDLENGTH - 1 downto 0);
        NEG_CELL:           MVL9_VECTOR(WORDLENGTH - 1 downto 0);
        NEG_NEG_CELL:       MVL9_VECTOR(WORDLENGTH - 1 downto 0);
end record;

type MVL9_STATE_LOGIC is
record
        STATE:              MVL9_VECTOR(WORDLENGTH - 1 downto 0);
        LOGIC_VALUE:        MVL9;
end record;

type TEST_ORDER is (STATIC, FUNCT);

type MVL9_STATE_LOGIC_VECTOR is array (NATURAL range <>) of MVL9_STATE_LOGIC;

type STATIC_INFO is
record
        STATE_PATT1:        MVL9_VECTOR(WORDLENGTH - 1 downto 0);
        STATE_PATT2:        MVL9_VECTOR(WORDLENGTH - 1 downto 0);
```

```
        STATIC_EVENT_TIME:      TIME;     -- The time between the 2nd input
                                          -- event and the first output event
                                          -- due to the 2nd input event.
        PULSE_DURATION:         TIME;     -- The hazard pulse width.
end record;

type STATIC_INFO_VECTOR is array (NATURAL range <>) of STATIC_INFO;

-------------------------------------------------------------------------------
type MVL9_VECTOR_TUPLE_VECTOR is array(NATURAL range <>) of
MVL9_VECTOR_TUPLE;
subtype MVL9_VECTOR_TUPLE_LENGTH is MVL9_VECTOR_TUPLE_VECTOR(0 to
WORDLENGTH - 1);
type MVL9_VECTOR_TUPLE_LENGTH_VECTOR is array (NATURAL RANGE <>) of
MVL9_VECTOR_TUPLE_LENGTH;
-------------------------------------------------------------------------------
type MVL9_VECTOR_TRIPLE_VECTOR is array(NATURAL range <>) of
MVL9_VECTOR_TRIPLE;
subtype MVL9_VECTOR_TRIPLE_LENGTH is MVL9_VECTOR_TRIPLE_VECTOR(0 to
(WORDLENGTH*(WORDLENGTH -1) - 1));
type MVL9_VECTOR_TRIPLE_LENGTH_VECTOR is array (NATURAL RANGE <>) of
MVL9_VECTOR_TRIPLE_LENGTH;
-------------------------------------------------------------------------------


-----------------------------------------------
-- Functions Declaration
-----------------------------------------------

-----------------------------------------------------------------
-- Functions required for single-input change hazard detection.
-----------------------------------------------------------------

function NATURAL_GENERATION(INPUT:NATURAL) return NATURAL_VECTOR;

function NATURAL_TO_BIT_PATTERN(INPUT:NATURAL) return MVL9_VECTOR_ARRAY;

function BIT_PATTERN_TO_NATURAL(INPUT: BIT_VECTOR) return NATURAL;

function STATIC_NEIGHBOR(INPUT: MVL9_VECTOR(0 to WORDLENGTH - 1)) return
MVL9_VECTOR_TUPLE_LENGTH;

function STATIC_VECTOR_FORMATTER(INPUT:
MVL9_VECTOR_TUPLE_LENGTH_VECTOR)
return MVL9_VECTOR_TUPLE_VECTOR;

function STATIC_PATTERN_GENERATOR(INPUT: NATURAL) return
MVL9_VECTOR_TUPLE_VECTOR;

function TOTAL_STATIC_PATTERNS(INPUT: NATURAL) return NATURAL;
```

```
function FIRST_STATIC_PATTERN(INPUT: MVL9_VECTOR_TUPLE_VECTOR(0 to
(2**WORDLENGTH)*WORDLENGTH - 1); ITERATION: NATURAL) return MVL9_VECTOR;

function SECOND_STATIC_PATTERN(INPUT: MVL9_VECTOR_TUPLE_VECTOR(0 to
(2**WORDLENGTH)*WORDLENGTH - 1); ITERATION: NATURAL) return MVL9_VECTOR;

procedure STATIC_PATTERN_TESTER(INPUT_NUM: in MVL9_VECTOR_TUPLE_VECTOR(0
to (2**WORDLENGTH)*WORDLENGTH - 1);  signal TIME_DELAY: in PROP_DELAY_SPEC;
signal PATTERN1, PATTERN2, OUT_PATTERN: out MVL9_VECTOR;
MODE: TEST_ORDER);

function RESET_PATTERN(INPUT: NATURAL:= WORDLENGTH) return MVL9_VECTOR;

function U_PATTERN(INPUT: NATURAL:= WORDLENGTH) return MVL9_VECTOR;

function DC_PATTERN(INPUT: NATURAL:= WORDLENGTH) return MVL9_VECTOR;

function TIME_RESET(INPUT: NATURAL:= WORDLENGTH) return TIME_VECTOR;

function BOOLEAN_PATTERN_FALSE(INPUT: NATURAL := WORDLENGTH) return
BOOLEAN_ARRAY;

function BOOLEAN_PATTERN_TRUE(INPUT: NATURAL := WORDLENGTH) return
BOOLEAN_ARRAY;

function ONE_BIT_DIFF(INPUT1, INPUT2: MVL9_VECTOR(0 to WORDLENGTH - 1)) return
BOOLEAN;

function TWO_BIT_DIFF(INPUT1, INPUT2: MVL9_VECTOR(0 to WORDLENGTH - 1)) return
BOOLEAN;


------------------------------------------------------------------
-- Functions required for two-input change hazard detection.
------------------------------------------------------------------
function FUNCTION_NEIGHBOR(INPUT: MVL9_VECTOR(0 to WORDLENGTH - 1))
return MVL9_VECTOR_TRIPLE_LENGTH;

function FUNCTION_PATTERN_GENERATOR(INPUT: NATURAL) return
MVL9_VECTOR_TRIPLE_VECTOR;

function FUNCTION_VECTOR_FORMATTER(INPUT:
MVL9_VECTOR_TRIPLE_LENGTH_VECTOR)
return MVL9_VECTOR_TRIPLE_VECTOR;

function FUNCTION_HAZARD_DISTINGUISHER(INPUT:
MVL9_VECTOR_TRIPLE_LENGTH)
return MVL9_VECTOR_TRIPLE_LENGTH;
```

```
function TOTAL_FUNCTION_PATTERNS(INPUT: NATURAL) return NATURAL;

function MEAN_DELAY_TIME(TIME_DELAY: PROP_DELAY_SPEC) return TIME;


-----------------------------------------------------------------
-- Functions required for the three-tuple reducing mechanism.
-----------------------------------------------------------------
procedure PRE_PROCESS_INJECTION(signal TIME_DELAY: in PROP_DELAY_SPEC;
signal OUTPUT: out MVL9_VECTOR);

procedure PRE_PROCESS_DATA(signal INPUT1: MVL9_VECTOR; signal INPUT2: MVL9;
TIME_DELAY: PROP_DELAY_SPEC; DATA: inout INTEGER; signal ANSWER: out
MVL9_STATE_LOGIC_VECTOR);

function STATE_TESTER(INPUT: MVL9_VECTOR; STATE_VALUE_TABLE:
MVL9_STATE_LOGIC_VECTOR) return MVL9;

function PRE_TEST_BEGIN_TIME(TIME_DELAY: PROP_DELAY_SPEC) return TIME;

function PRE_PROCESS_FUNCTION_GENERATOR(
INPUT: MVL9_VECTOR_TRIPLE_VECTOR(0 to
((2**WORDLENGTH)*WORDLENGTH*(WORDLENGTH - 1)) - 1);
STATE_VALUE_TABLE: MVL9_STATE_LOGIC_VECTOR; constant NUMBER: NATURAL)
return MVL9_VECTOR_TRIPLE_VECTOR;

function PRE_PROCESS_COUNT(
INPUT: MVL9_VECTOR_TRIPLE_VECTOR(0 to
((2**WORDLENGTH)*WORDLENGTH*(WORDLENGTH - 1)) - 1);
STATE_VALUE_TABLE: MVL9_STATE_LOGIC_VECTOR)
return NATURAL;

function PRE_FIRST_FUNCTION_PATTERN(INPUT: MVL9_VECTOR_TRIPLE_VECTOR;
ITERATION: NATURAL) return MVL9_VECTOR;

function PRE_SECOND_FUNCTION_PATTERN(INPUT: MVL9_VECTOR_TRIPLE_VECTOR;
ITERATION: NATURAL) return MVL9_VECTOR;


procedure PRE_FUNCTION_PATTERN_TESTER(
INPUT_NUM: in MVL9_VECTOR_TRIPLE_VECTOR;
signal TIME_DELAY: in PROP_DELAY_SPEC;
constant NUMBER: NATURAL;
signal PATTERN1, PATTERN2, OUT_PATTERN: out MVL9_VECTOR;
MODE: TEST_ORDER);

end TEST_GENERATOR;

package body TEST_GENERATOR is
```

```
--------------------------------------------------------------------------------
--          Single-Input Change Hazard Detection Scheme
--------------------------------------------------------------------------------

-------------------------------------------------------------------------
-- This function generates a range of integers from 0 to 2**WORDLENGTH - 1.
-------------------------------------------------------------------------
function NATURAL_GENERATION(INPUT:NATURAL) return NATURAL_VECTOR is
variable I: NATURAL:= 0;
variable RESULT: NATURAL_VECTOR(0 to 2**WORDLENGTH - 1);
begin
for I in 0 to 2**WORDLENGTH - 1 loop
RESULT(I) := I;
end loop;

return RESULT;

end NATURAL_GENERATION;


-------------------------------------------------------------------------
-- This function generates a range of integers and
-- converts each of these integers into its bit format.
-------------------------------------------------------------------------
function NATURAL_TO_BIT_PATTERN(INPUT:NATURAL) return MVL9_VECTOR_ARRAY
is
variable RESULT_TMP: NATURAL;
variable K,N: NATURAL := 0;
variable BIT_PATTERN : MVL9_VECTOR(WORDLENGTH - 1 downto 0);
variable ANSWER: MVL9_VECTOR_ARRAY(0 to 2**WORDLENGTH  - 1);
begin

RESULT_TMP := INPUT;
for K in 0 to 2**WORDLENGTH - 1 loop
RESULT_TMP := K;
for N in 0 to WORDLENGTH - 1 loop
if (RESULT_TMP rem 2 /= 0) then
BIT_PATTERN(N) := '1';
else
BIT_PATTERN(N) := '0';
end if;

RESULT_TMP := RESULT_TMP / 2;
end loop;

ANSWER(K) := BIT_PATTERN;

end loop;
return ANSWER;
end NATURAL_TO_BIT_PATTERN;
```

```
-------------------------------------------------------------------------
-- This function converts any BIT_VECTOR pattern into its natural value.
-------------------------------------------------------------------------
function BIT_PATTERN_TO_NATURAL(INPUT: BIT_VECTOR) return NATURAL is
variable RESULT: NATURAL := 0;
begin
for I in INPUT'range loop
if (INPUT(I) = '1') then
RESULT := RESULT + (2 ** I);
end if;
end loop;
return RESULT;

end BIT_PATTERN_TO_NATURAL;


-----------------------------------------------------------------
-- This function uses the original bit pattern to
-- generate all its possible neighbor bit patterns for the
-- single input change hazard detection.
-----------------------------------------------------------------
function STATIC_NEIGHBOR(INPUT: MVL9_VECTOR(0 to WORDLENGTH - 1)) return
MVL9_VECTOR_TUPLE_LENGTH is
variable I: NATURAL := 0;
variable BIT_PATTERN_GENERATOR : MVL9_VECTOR(0 to WORDLENGTH - 1);
variable ANSWER: MVL9_VECTOR_TUPLE_LENGTH;
begin


for I in 0 to WORDLENGTH - 1 loop

BIT_PATTERN_GENERATOR := INPUT;

if (INPUT(I) = '1') then
BIT_PATTERN_GENERATOR(I) := '0';
elsif (INPUT(I) = '0') then
BIT_PATTERN_GENERATOR(I) := '1';
end if;

ANSWER(I).ORG_CELL := INPUT;
ANSWER(I).NEG_CELL := BIT_PATTERN_GENERATOR;

end loop;
return ANSWER;

end STATIC_NEIGHBOR;
```

```
-------------------------------------------------------------------------------
-- This function re-orgainizes the bit patterns into two-tuples format for the
-- single-input change hazard detection.
-------------------------------------------------------------------------------
function STATIC_VECTOR_FORMATTER(INPUT:
MVL9_VECTOR_TUPLE_LENGTH_VECTOR)
return MVL9_VECTOR_TUPLE_VECTOR is
variable RESULT: MVL9_VECTOR_TUPLE_VECTOR(0 to
(2**WORDLENGTH)*WORDLENGTH - 1);
variable TMP: MVL9_VECTOR_TUPLE_LENGTH;
variable I,J,K: NATURAL:= 0;
begin

K := 0;
for J in 0 to 2**WORDLENGTH - 1 loop
TMP := INPUT(J);

for I in 0 to WORDLENGTH - 1 loop
RESULT(K) := TMP(I);
K := K + 1;
end loop;
end loop;

return RESULT;
end STATIC_VECTOR_FORMATTER;

-------------------------------------------------------------------------
-- This function generates the two-tuples for the
-- single-input change hazard detection.
-------------------------------------------------------------------------
function STATIC_PATTERN_GENERATOR(INPUT: NATURAL) return
MVL9_VECTOR_TUPLE_VECTOR is
variable I: NATURAL:= 0;
variable TMP : MVL9_VECTOR_ARRAY(0 to 2**WORDLENGTH -1);
variable RESULT: MVL9_VECTOR_TUPLE_LENGTH_VECTOR(0 to 2**WORDLENGTH - 1);
begin

TMP :=  NATURAL_TO_BIT_PATTERN(INPUT);

for I in 0 to 2**WORDLENGTH - 1 loop
RESULT(I) := (STATIC_NEIGHBOR(TMP(I)));
end loop;

return (STATIC_VECTOR_FORMATTER(RESULT));
end STATIC_PATTERN_GENERATOR;
```

```
-------------------------------------------------------------
-- This function computes the total numbers of patterns
-- required for the single-input change hazard analysis.
-------------------------------------------------------------
function TOTAL_STATIC_PATTERNS(INPUT: NATURAL) return NATURAL is
begin
return ((2**INPUT)*INPUT);
end TOTAL_STATIC_PATTERNS;



-------------------------------------------------------------------------
-- This function retrieves the first input pattern (original bit
-- pattern) for the single-input change hazard detection .
-------------------------------------------------------------------------
function FIRST_STATIC_PATTERN(INPUT: MVL9_VECTOR_TUPLE_VECTOR(0 to
(2**WORDLENGTH)*WORDLENGTH - 1); ITERATION: NATURAL) return MVL9_VECTOR is

variable ANSWER: MVL9_VECTOR(0 to WORDLENGTH - 1);
begin


ANSWER := INPUT(ITERATION).ORG_CELL;

return ANSWER;
end FIRST_STATIC_PATTERN;



-------------------------------------------------------------------------
-- This function retrieves the second input pattern (neighbor bit
-- pattern) for the single-input change hazard detection .
-------------------------------------------------------------------------
function SECOND_STATIC_PATTERN(INPUT: MVL9_VECTOR_TUPLE_VECTOR(0 to
(2**WORDLENGTH)*WORDLENGTH - 1); ITERATION: NATURAL) return MVL9_VECTOR is

variable ANSWER: MVL9_VECTOR(0 to WORDLENGTH - 1);
begin


ANSWER := INPUT(ITERATION).NEG_CELL;

return ANSWER;
end SECOND_STATIC_PATTERN;
```

```
-------------------------------------------------------------------------------
-- This procedure injects the original bit patterns, the neighbor bit patterns,
-- and the resetting patterns into the test circuit for the detection of the
-- single-input change hazard.
-------------------------------------------------------------------------------
procedure STATIC_PATTERN_TESTER(INPUT_NUM: in MVL9_VECTOR_TUPLE_VECTOR(0
to (2**WORDLENGTH)*WORDLENGTH - 1);
signal TIME_DELAY: in PROP_DELAY_SPEC;
signal PATTERN1, PATTERN2, OUT_PATTERN: out MVL9_VECTOR;
MODE: TEST_ORDER) is
variable I: NATURAL:= 0;
constant RESET_SEQUENCE: MVL9_VECTOR(0 to WORDLENGTH - 1) :=
RESET_PATTERN(WORDLENGTH);
begin

assert not (MODE = FUNCT)
report "WRONG command is used. It should not be the Two-input change hazard analysis
(FUNCT)."
severity ERROR;


for I in 0 to TOTAL_STATIC_PATTERNS(WORDLENGTH) - 1 loop
if (MODE = STATIC) then

-----------------------------------------------------------------
-- Display the injected patterns for the single-input
-- change hazard analysis.
-----------------------------------------------------------------
PATTERN1 <= transport FIRST_STATIC_PATTERN(INPUT_NUM, I)
        after (4*I + 1)*TIME_DELAY.MAX_PROP_DELAY;

PATTERN2 <=  transport SECOND_STATIC_PATTERN(INPUT_NUM, I)
        after (4*I + 1)*TIME_DELAY.MAX_PROP_DELAY;
-----------------------------------------------------------------

OUT_PATTERN <= transport FIRST_STATIC_PATTERN(INPUT_NUM, I)
        after (4*I + 1)*TIME_DELAY.MAX_PROP_DELAY;

OUT_PATTERN <= transport SECOND_STATIC_PATTERN(INPUT_NUM, I)
        after (4*I + 2)*TIME_DELAY.MAX_PROP_DELAY;

OUT_PATTERN <= transport RESET_SEQUENCE --RESET_PATTERN(WORDLENGTH)
        after (4*I + 3)*TIME_DELAY.MAX_PROP_DELAY;
--=================================================================
end if;
end loop;

end STATIC_PATTERN_TESTER;
```

```
------------------------------------------------------------
-- This function generates the reset pattern 'X' for the
-- VHDS in any specific length.
------------------------------------------------------------
function RESET_PATTERN(INPUT: NATURAL:= WORDLENGTH) return MVL9_VECTOR is
constant TMP : NATURAL := INPUT;
variable ANSWER: MVL9_VECTOR(0 to TMP - 1);
variable I: NATURAL := 0;
begin
for I in 0 to INPUT - 1 loop
ANSWER(I) := 'X';
end loop;

return ANSWER;
end RESET_PATTERN;


------------------------------------------------------------
-- This function generates the reset pattern 'U' for the
-- VHDS in any specific length.
------------------------------------------------------------
function U_PATTERN(INPUT: NATURAL:= WORDLENGTH) return MVL9_VECTOR is
constant TMP: NATURAL := INPUT;
variable ANSWER: MVL9_VECTOR(0 to TMP - 1);
variable I: NATURAL := 0;
begin
for I in 0 to INPUT - 1 loop
ANSWER(I) := 'U';
end loop;

return ANSWER;
end U_PATTERN;


------------------------------------------------------------
-- This function generates the don't care pattern '-' for the
-- VHDS in any specific length.
------------------------------------------------------------
function DC_PATTERN(INPUT: NATURAL:= WORDLENGTH) return MVL9_VECTOR is
constant TMP: NATURAL := INPUT;
variable ANSWER: MVL9_VECTOR(0 to TMP - 1);
variable I: NATURAL := 0;
begin
for I in 0 to INPUT - 1 loop
ANSWER(I) := '-';
end loop;

return ANSWER;
end DC_PATTERN;
```

```
-------------------------------------------------------------------
-- This function resets all the timing parameters into 0 ns.
-------------------------------------------------------------------
function TIME_RESET(INPUT: NATURAL:= WORDLENGTH) return TIME_VECTOR is
constant TMP: NATURAL := INPUT;
variable ANSWER: TIME_VECTOR(0 to TMP - 1);
variable I: NATURAL := 0;
begin
for I in 0 to INPUT - 1 loop
ANSWER(I) := 0 ns;
end loop;

return ANSWER;
end TIME_RESET;



-------------------------------------------------------------------
-- A function initializes the BOOLEAN_VECTOR into FALSE values.
-------------------------------------------------------------------
function BOOLEAN_PATTERN_FALSE(INPUT: NATURAL := WORDLENGTH) return
BOOLEAN_ARRAY is
constant TMP : NATURAL := INPUT;
variable ANSWER: BOOLEAN_ARRAY(1 to TMP);
variable I: NATURAL := 0;
begin
for I in 1 to TMP loop
ANSWER(I) := FALSE;
end loop;

return ANSWER;
end BOOLEAN_PATTERN_FALSE;

-------------------------------------------------------------------
-- A function initializes the BOOLEAN_VECTOR into TRUE values.
-------------------------------------------------------------------
function BOOLEAN_PATTERN_TRUE(INPUT: NATURAL := WORDLENGTH) return
BOOLEAN_ARRAY is
constant TMP : NATURAL := INPUT;
variable ANSWER: BOOLEAN_ARRAY(1 to TMP);
variable I: NATURAL := 0;
begin
for I in 1 to TMP loop
ANSWER(I) := TRUE;
end loop;

return ANSWER;
end BOOLEAN_PATTERN_TRUE;
```

```
------------------------------------------------------------
-- This function is to compute whether the two
-- input has a bit difference by one bit.
-- If they do, that is implies there is a single
-- input change exists.
------------------------------------------------------------
function ONE_BIT_DIFF(INPUT1, INPUT2: MVL9_VECTOR(0 to WORDLENGTH - 1)) return
BOOLEAN is
variable I, J: NATURAL := 0;
variable TMP : MVL9_VECTOR(0 to WORDLENGTH - 1);
begin

TMP := INPUT1 xor INPUT2;

for I in 0 to WORDLENGTH - 1 loop
if (TMP(I) = '1') then
J := J + 1;
end if;
end loop;

if (J = 1) THEN
return TRUE;
else
return FALSE;
end if;

end ONE_BIT_DIFF;

------------------------------------------------------------
-- This function is to compute whether the two
-- input has a bit difference by two bit.
-- If they do, that is implies there is a two
-- input change exists.
------------------------------------------------------------
function TWO_BIT_DIFF(INPUT1, INPUT2: MVL9_VECTOR(0 to WORDLENGTH - 1)) return
BOOLEAN is
variable I, J: NATURAL := 0;
variable TMP : MVL9_VECTOR(0 to WORDLENGTH - 1);
begin

TMP := INPUT1 xor INPUT2;

for I in 0 to WORDLENGTH - 1 loop
if (TMP(I) = '1') then
J := J + 1;
end if;
end loop;
```

```
if (J = 2) THEN
return TRUE;
else
return FALSE;
end if;
end TWO_BIT_DIFF;


--------------------------------------------------------------------------------
--              Two-Input Change Hazard Detection Scheme
--------------------------------------------------------------------------------

--------------------------------------------------------------------------
-- This function generates all the possible first and second neighbor
-- bit patterns for the two-input change hazard detection.
--------------------------------------------------------------------------
function FUNCTION_NEIGHBOR(INPUT: MVL9_VECTOR(0 to WORDLENGTH - 1))
return MVL9_VECTOR_TRIPLE_LENGTH is
variable I, J, K: NATURAL := 0;
variable FUNCTION_PATTERN_1, FUNCTION_PATTERN_2 : MVL9_VECTOR(0 to
WORDLENGTH - 1);
variable ANSWER: MVL9_VECTOR_TRIPLE_LENGTH;
variable TMP: MVL9_VECTOR_TRIPLE_VECTOR(0 to WORDLENGTH**2 - 1);
variable TMP1: MVL9_VECTOR_TRIPLE_LENGTH;
begin

-------------------------------------------
-- Generate all the possible patterns.
-------------------------------------------
K := 0;
for I in 0 to WORDLENGTH - 1 loop

FUNCTION_PATTERN_1 := INPUT;

if (INPUT(I) = '1') then
FUNCTION_PATTERN_1(I) := '0';
elsif (INPUT(I) = '0') then
FUNCTION_PATTERN_1(I) := '1';
end if;

ANSWER(I).ORG_CELL := INPUT;
ANSWER(I).NEG_CELL := FUNCTION_PATTERN_1;

for J in 0 to WORDLENGTH - 1 loop

FUNCTION_PATTERN_2 := FUNCTION_PATTERN_1;

if (FUNCTION_PATTERN_1(J) = '1') then
FUNCTION_PATTERN_2(J) := '0';
elsif (FUNCTION_PATTERN_1(J) = '0') then
```

```
FUNCTION_PATTERN_2(J) := '1';
end if;

ANSWER(J).ORG_CELL := INPUT;
ANSWER(J).NEG_CELL := FUNCTION_PATTERN_1;
ANSWER(J).NEG_NEG_CELL := FUNCTION_PATTERN_2;
TMP(K) := ANSWER(J);
K := K + 1;
end loop;
end loop;

-------------------------------------------
-- Cut out the return patterns.
-------------------------------------------
K := 0;
for I in 0 to WORDLENGTH**2 - 1 loop
if(TMP(I).ORG_CELL /= TMP(I).NEG_NEG_CELL) then
TMP1(K) := TMP(I);
K := K + 1;
end if;
end loop;

return TMP1;
end FUNCTION_NEIGHBOR;

-----------------------------------------------------------------------------
-- This function re-organizes the bit patterns into three-tuples format.
-----------------------------------------------------------------------------
function FUNCTION_VECTOR_FORMATTER(INPUT:
MVL9_VECTOR_TRIPLE_LENGTH_VECTOR)
return MVL9_VECTOR_TRIPLE_VECTOR is
variable RESULT: MVL9_VECTOR_TRIPLE_VECTOR(0 to
((2**WORDLENGTH)*WORDLENGTH*(WORDLENGTH - 1)) - 1);
variable TMP: MVL9_VECTOR_TRIPLE_LENGTH;
variable I,J,K: NATURAL:= 0;
begin

K := 0;
for J in 0 to 2**WORDLENGTH - 1 loop
TMP := INPUT(J);

for I in 0 to (WORDLENGTH*(WORDLENGTH - 1)) - 1 loop
RESULT(K) := TMP(I);
K := K + 1;
end loop;
end loop;

return RESULT;
end FUNCTION_VECTOR_FORMATTER;
```

```
-------------------------------------------------------------------------
-- This function generates the three-tuples test patterns for the
-- two-input change hazard analysis.
-------------------------------------------------------------------------
function FUNCTION_PATTERN_GENERATOR(INPUT: NATURAL) return
MVL9_VECTOR_TRIPLE_VECTOR is
variable I :NATURAL := 0;
variable TMP: MVL9_VECTOR_ARRAY(0 to 2**WORDLENGTH - 1);
variable RESULT: MVL9_VECTOR_TRIPLE_LENGTH_VECTOR(0 to
((2**WORDLENGTH)*WORDLENGTH*(WORDLENGTH - 1)) - 1);
begin

TMP := NATURAL_TO_BIT_PATTERN(INPUT);

for I in 0 to 2**WORDLENGTH - 1 loop
RESULT(I) := FUNCTION_HAZARD_DISTINGUISHER(FUNCTION_NEIGHBOR(TMP(I)));
end loop;

return (FUNCTION_VECTOR_FORMATTER(RESULT));
end FUNCTION_PATTERN_GENERATOR;



-------------------------------------------------------------------
-- This function re-organizes the three-tuples so that the
-- first patterns and the third patterns are the same.
-------------------------------------------------------------------
function FUNCTION_HAZARD_DISTINGUISHER(INPUT:
MVL9_VECTOR_TRIPLE_LENGTH)
return MVL9_VECTOR_TRIPLE_LENGTH is
variable ARRAY_SUB_ELEMENT: MVL9_VECTOR_ARRAY(0 to
WORDLENGTH*(WORDLENGTH - 1) - 1);
variable TMP: MVL9_VECTOR_TRIPLE_LENGTH;
variable INDEX: MVL9_VECTOR(0 to WORDLENGTH - 1);
variable RESULT: MVL9_VECTOR_TRIPLE_LENGTH;
variable I, J, K, L :NATURAL := 0;
begin

TMP := INPUT;
-----------------------------------------------
-- Re-organise the patterns
-----------------------------------------------

for I in 0 to WORDLENGTH*(WORDLENGTH - 1) - 1 loop
ARRAY_SUB_ELEMENT(I) := TMP(I).NEG_NEG_CELL;
end loop;

for J in 0 to WORDLENGTH*(WORDLENGTH - 1) - 1 loop
```

```
INDEX := ARRAY_SUB_ELEMENT(J);

for K in 0 to WORDLENGTH*(WORDLENGTH - 1) - 1 loop


if (TMP(K).NEG_NEG_CELL = INDEX) then

RESULT(L) :=  TMP(K);
TMP(K) := (RESET_PATTERN(WORDLENGTH), RESET_PATTERN(WORDLENGTH),
RESET_PATTERN(WORDLENGTH));
L := L + 1;
end if;

end loop;
end loop;

return RESULT;
end FUNCTION_HAZARD_DISTINGUISHER;



-------------------------------------------------------------------
-- This function computes the total numbers of patterns
-- required for the two-input change hazard analysis.
-------------------------------------------------------------------
function TOTAL_FUNCTION_PATTERNS(INPUT: NATURAL) return NATURAL is
begin
return((2**INPUT)*INPUT*(INPUT - 1));
end TOTAL_FUNCTION_PATTERNS;

-------------------------------------------------------------------
-- This function computes the average value of the min and max values
-- of the test circuit.
-------------------------------------------------------------------
function MEAN_DELAY_TIME(TIME_DELAY: PROP_DELAY_SPEC) return TIME is
variable TMP1: NATURAL := 0;
variable TMP: REAL := 0.0;
begin

TMP := REAL(TIME_TO_NATURAL(TIME_DELAY.MAX_PROP_DELAY +
TIME_DELAY.MIN_PROP_DELAY)) / 2.0;
TMP1 := TIME_TO_NATURAL(TIME_DELAY.MAX_PROP_DELAY +
TIME_DELAY.MIN_PROP_DELAY) / 2;

if (abs(TMP - REAL(TMP1)) /= 0.0) then
return (NATURAL_TO_TIME(TMP1 + 1));
else
return (NATURAL_TO_TIME(TMP1));
end if;
end MEAN_DELAY_TIME;
```

```
--------------------------------------------------------------------------------
--              Two-input change reducing mechanism
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
-- This procedure injects all the possible state values (bit_vector pattern)
-- to the test circuit to observe the output responses.
--------------------------------------------------------------------------------
procedure PRE_PROCESS_INJECTION(signal TIME_DELAY: in PROP_DELAY_SPEC;
signal OUTPUT: out MVL9_VECTOR) is
variable I: NATURAL := 0;
variable RESULT: MVL9_VECTOR_ARRAY(0 to 2**WORDLENGTH - 1);
begin

RESULT := NATURAL_TO_BIT_PATTERN(WORDLENGTH);

for I in 0 to 2**WORDLENGTH - 1 loop

OUTPUT <=  transport RESULT(I) after (I + 1)*TIME_DELAY.MAX_PROP_DELAY;
end loop;

end PRE_PROCESS_INJECTION;

--------------------------------------------------------------------------
-- This function memorizes all the state values and their corresponding
-- output logic values into a tabular form.
--------------------------------------------------------------------------
procedure PRE_PROCESS_DATA(signal INPUT1: MVL9_VECTOR; signal INPUT2: MVL9;
TIME_DELAY: PROP_DELAY_SPEC; DATA: inout INTEGER; signal ANSWER: out
MVL9_STATE_LOGIC_VECTOR) is

begin

if (INPUT1'event and now >= 1 ns and DATA <= 2**WORDLENGTH - 1 and
  (INPUT1 /= RESET_PATTERN(WORDLENGTH) or INPUT1 /=
U_PATTERN(WORDLENGTH))) then
ANSWER(DATA).STATE <= INPUT1;

if (DATA >= 1) then
ANSWER(DATA - 1).LOGIC_VALUE <= INPUT2;
end if;
DATA := DATA + 1;
end if;
if (DATA = 2**WORDLENGTH)  then
wait for TIME_DELAY.MAX_PROP_DELAY;
ANSWER(2**WORDLENGTH - 1).LOGIC_VALUE <= INPUT2;
end if;

end PRE_PROCESS_DATA;
```

```
--------------------------------------------------------------------------
-- This function returns the logic value from a specified state value.
--------------------------------------------------------------------------
function STATE_TESTER(INPUT: MVL9_VECTOR; STATE_VALUE_TABLE:
MVL9_STATE_LOGIC_VECTOR)
return MVL9 is

begin
return(STATE_VALUE_TABLE(BIT_PATTERN_TO_NATURAL(TO_BITVECTOR(INPUT))).LO
GIC_VALUE);
end STATE_TESTER;



--------------------------------------------------------------------------
-- This function computes the time that is required for the reduced test
-- patterns set injection as well as the starting time of the hazard
-- detection.
--------------------------------------------------------------------------
function PRE_TEST_BEGIN_TIME(TIME_DELAY: PROP_DELAY_SPEC) return TIME is
begin
return ((2**WORDLENGTH + 1)*TIME_DELAY.MAX_PROP_DELAY + 2 ns);

end PRE_TEST_BEGIN_TIME;



--------------------------------------------------------------------------
-- This function produces a reduced three-tuples test patterns set from
-- the original three-tuples test patterns set based on the specific
-- output events (i.e. "0-1-0" or "1-0-1").
--------------------------------------------------------------------------
function PRE_PROCESS_FUNCTION_GENERATOR(
INPUT: MVL9_VECTOR_TRIPLE_VECTOR(0 to
((2**WORDLENGTH)*WORDLENGTH*(WORDLENGTH - 1)) - 1);
STATE_VALUE_TABLE: MVL9_STATE_LOGIC_VECTOR; constant NUMBER: NATURAL)
return MVL9_VECTOR_TRIPLE_VECTOR is

variable I,K: NATURAL := 0;
variable TEST_LOGIC1, TEST_LOGIC2, TEST_LOGIC3: MVL9;
variable RESULT: MVL9_VECTOR_TRIPLE_VECTOR(0 to NUMBER);
variable TEST_LOGIC11, TEST_LOGIC21, TEST_LOGIC31: MVL9;
variable VECTOR1, VECTOR11: MVL9_VECTOR(1 to 3);
begin

for I in 0 to TOTAL_FUNCTION_PATTERNS(WORDLENGTH)/2 - 1 loop
if (I <= TOTAL_FUNCTION_PATTERNS(WORDLENGTH)/2 - 1) then
TEST_LOGIC1 := STATE_TESTER(INPUT(2 * I).ORG_CELL, STATE_VALUE_TABLE);
```

```
TEST_LOGIC2 := STATE_TESTER(INPUT(2 * I).NEG_CELL, STATE_VALUE_TABLE);
TEST_LOGIC3 := STATE_TESTER(INPUT(2 * I).NEG_NEG_CELL, STATE_VALUE_TABLE);

TEST_LOGIC11 := STATE_TESTER(INPUT(2 * I + 1).ORG_CELL, STATE_VALUE_TABLE);
TEST_LOGIC21 := STATE_TESTER(INPUT(2 * I + 1).NEG_CELL, STATE_VALUE_TABLE);
TEST_LOGIC31 := STATE_TESTER(INPUT(2 * I + 1).NEG_NEG_CELL,
STATE_VALUE_TABLE);
end if;

VECTOR1 := TEST_LOGIC1 & TEST_LOGIC2 & TEST_LOGIC3;
VECTOR11 := TEST_LOGIC11 & TEST_LOGIC21 & TEST_LOGIC31;

if (VECTOR1 = "101" and VECTOR11 = "101") or (VECTOR1 = "010" and VECTOR11 = "010")
then
RESULT(K) := INPUT(2 * I);
K := K + 1;
end if;

if (VECTOR1 = "101" and VECTOR11 /= "101") or (VECTOR1 = "010" and VECTOR11 /= "010")
then
RESULT(K) := INPUT(2 * I);
K := K + 1;
end if;

if (VECTOR11 = "101" and VECTOR1 /= "101") or (VECTOR11 = "010" and VECTOR1 /= "010")
then
RESULT(K) := INPUT(2 * I + 1);
K := K + 1;
end if;

end loop;

return RESULT;

end PRE_PROCESS_FUNCTION_GENERATOR;



------------------------------------------------------------------
-- This function counts the total numbers of three-tuples in the
-- reduced three-tuples test patterns set.
------------------------------------------------------------------
function PRE_PROCESS_COUNT(
INPUT: MVL9_VECTOR_TRIPLE_VECTOR(0 to
((2**WORDLENGTH)*WORDLENGTH*(WORDLENGTH - 1)) - 1);
STATE_VALUE_TABLE: MVL9_STATE_LOGIC_VECTOR)
return NATURAL is

variable I,K: NATURAL := 0;
variable TEST_LOGIC1, TEST_LOGIC2, TEST_LOGIC3: MVL9;
```

```
variable TEST_LOGIC11, TEST_LOGIC21, TEST_LOGIC31: MVL9;
variable VECTOR1, VECTOR11: MVL9_VECTOR(1 to 3);
begin

for I in 0 to TOTAL_FUNCTION_PATTERNS(WORDLENGTH)/2 - 1 loop
if (I <= TOTAL_FUNCTION_PATTERNS(WORDLENGTH)/2 - 1) then
TEST_LOGIC1 := STATE_TESTER(INPUT(2 * I).ORG_CELL, STATE_VALUE_TABLE);
TEST_LOGIC2 := STATE_TESTER(INPUT(2 * I).NEG_CELL, STATE_VALUE_TABLE);
TEST_LOGIC3 := STATE_TESTER(INPUT(2 * I).NEG_NEG_CELL, STATE_VALUE_TABLE);

TEST_LOGIC11 := STATE_TESTER(INPUT(2 * I + 1).ORG_CELL, STATE_VALUE_TABLE);
TEST_LOGIC21 := STATE_TESTER(INPUT(2 * I + 1).NEG_CELL, STATE_VALUE_TABLE);
TEST_LOGIC31 := STATE_TESTER(INPUT(2 * I + 1).NEG_NEG_CELL,
STATE_VALUE_TABLE);
end if;

VECTOR1 := TEST_LOGIC1 & TEST_LOGIC2 & TEST_LOGIC3;
VECTOR11 := TEST_LOGIC11 & TEST_LOGIC21 & TEST_LOGIC31;

if (VECTOR1 = "101" and VECTOR11 = "101") or (VECTOR1 = "010" and VECTOR11 = "010")
then
K := K + 1;
end if;

if (VECTOR1 = "101" and VECTOR11 /= "101") or (VECTOR1 = "010" and VECTOR11 /= "010")
then
K := K + 1;
end if;

if (VECTOR11 = "101" and VECTOR1 /= "101") or (VECTOR11 = "010" and VECTOR1 /= "010")
then
K := K + 1;
end if;

end loop;

return K;

end PRE_PROCESS_COUNT;

-------------------------------------------------------------------------------
-- This procedure injects the original test patterns, second neighbor test patterns,
-- and the reset test patterns into the test circuit
-- for the two-input reducing mechanism.
-------------------------------------------------------------------------------
procedure PRE_FUNCTION_PATTERN_TESTER(
INPUT_NUM: in MVL9_VECTOR_TRIPLE_VECTOR;
signal TIME_DELAY: in PROP_DELAY_SPEC;
constant NUMBER: NATURAL;
signal PATTERN1, PATTERN2, OUT_PATTERN: out MVL9_VECTOR;
```

```
MODE: TEST_ORDER) is

variable I: NATURAL := 0;
constant RESET_SEQUENCE: MVL9_VECTOR(0 to WORDLENGTH - 1) :=
RESET_PATTERN(WORDLENGTH);
begin

assert not (MODE = STATIC)
report "WRONG command is used. Only Two-input change hazard detection (FUNCT) is
allowed in the pre-processing mechanism"
severity ERROR;


if (now = PRE_TEST_BEGIN_TIME(TIME_DELAY)) then
OUT_PATTERN <= transport RESET_SEQUENCE;
end if;

for I in 0 to NUMBER - 1 loop

if (MODE = FUNCT) then
-----------------------------------------------------------------------
-- Display the injected patterns for the two-input
-- chnage hazard detection in the pre-processing mechanism.
-----------------------------------------------------------------------
PATTERN1 <=  transport PRE_FIRST_FUNCTION_PATTERN(INPUT_NUM, I)
          after (4*I + 1)*TIME_DELAY.MAX_PROP_DELAY;

PATTERN2 <=  transport PRE_SECOND_FUNCTION_PATTERN(INPUT_NUM, I)
          after (4*I + 1)*TIME_DELAY.MAX_PROP_DELAY;

-----------------------------------------------------------------------

OUT_PATTERN <= transport PRE_FIRST_FUNCTION_PATTERN(INPUT_NUM, I)
          after (4*I + 1)*TIME_DELAY.MAX_PROP_DELAY;

OUT_PATTERN <= transport PRE_SECOND_FUNCTION_PATTERN(INPUT_NUM, I)
          after ((4*I + 2)*TIME_DELAY.MAX_PROP_DELAY);

OUT_PATTERN <= transport RESET_SEQUENCE --RESET_PATTERN(WORDLENGTH)
          after (4*I + 3)*TIME_DELAY.MAX_PROP_DELAY;

--=================================================

end if;

end loop;
end PRE_FUNCTION_PATTERN_TESTER;
```

```
-------------------------------------------------------------------------------
-- This function collects the original bit patterns for the two-input change
-- hazard detection.
-------------------------------------------------------------------------------
function PRE_FIRST_FUNCTION_PATTERN(INPUT: MVL9_VECTOR_TRIPLE_VECTOR;
ITERATION: NATURAL) return MVL9_VECTOR is

variable ANSWER: MVL9_VECTOR(0 to WORDLENGTH - 1);
begin

ANSWER := INPUT(ITERATION).ORG_CELL;

return ANSWER;

end PRE_FIRST_FUNCTION_PATTERN;


-------------------------------------------------------------------------------
-- This function collects the second neighbor bit patterns for the two-input change
-- hazard detection.
-------------------------------------------------------------------------------
function PRE_SECOND_FUNCTION_PATTERN(INPUT: MVL9_VECTOR_TRIPLE_VECTOR;
ITERATION: NATURAL) return MVL9_VECTOR is

variable ANSWER: MVL9_VECTOR(0 to WORDLENGTH - 1);
begin

ANSWER := INPUT(ITERATION).NEG_NEG_CELL;

return ANSWER;

end PRE_SECOND_FUNCTION_PATTERN;


end TEST_GENERATOR;
```

Appendix D. The VHDS Hazard Detection Package

```
library VHDS_LIB;
use VHDS_LIB.MVL9_SYSTEM.all;
use VHDS_LIB.TIME_PACKAGE.all;
use VHDS_LIB.TEST_GENERATOR.all;
use VHDS_LIB.INPUT_DATA.all;

library STD;
use STD.TEXTIO.all;

package PRE_HAZARD_TEST is

------------------------------------------------------------
-- Type Declarations
------------------------------------------------------------
type HAZARD_VAR is
record
        OUTPUT_COUNT1:              NATURAL;
        OUTPUT_COUNT2:              NATURAL;
        LOOP_VAR:                   NATURAL;
        STAT_CONTROL:               BOOLEAN;
        FUN_CONTROL:                BOOLEAN;
        STAT_0_CHECK:               BOOLEAN;
        STAT_1_CHECK:               BOOLEAN;
        FUN_0_CHECK:                BOOLEAN;
        FUN_1_CHECK:                BOOLEAN;
        FUN_INFO_TIME:              TIME;
        INPUT1_EVENT_TIME:          TIME;
        INPUT2_EVENT_TIME:          TIME;
        STAT_0_CHECK_TIME:          TIME;
        STAT_1_CHECK_TIME:          TIME;
        FUN_0_CHECK_TIME:           TIME;
        FUN_1_CHECK_TIME:           TIME;
        OUTPUT_LAST_TWO_VALUE:          MVL9;
        OUTPUT_PATTERNS:            MVL9_VECTOR(0 to 3);
------------------------------------------------------------
-- File I/O Control Variables Declaration
------------------------------------------------------------
        STAT_MESSAGE:               BOOLEAN;
        DYN_MESSAGE:                BOOLEAN;
        FUN_MESSAGE:                BOOLEAN;
        OUTPUT_RECORD:              MVL9_VECTOR(0 to 2 ** WORDLENGTH - 1);
        OUT_TIME_RECORD:            TIME_VECTOR(0 to 2 ** WORDLENGTH - 1);
        ICOUNT:                     INTEGER;
end record;
```

```
-----------------------------------------------
-- File I/O Declarations
-----------------------------------------------
file OUTFILE: TEXT is out "outfile.11";


-----------------------------------------------
-- Signals and Variables Initializations
-----------------------------------------------
constant HAZARD_CONSTANT: HAZARD_VAR := (0, 0, 0, TRUE, FALSE, FALSE, FALSE,
FALSE, FALSE,  0 ns, 0 ns, 0 ns, 0 ns, 0 ns, 0 ns, 0 ns, 'U', RESET_PATTERN(4),
FALSE, FALSE, FALSE, RESET_PATTERN(2 ** WORDLENGTH),
TIME_RESET(2 ** WORDLENGTH), 0);


-----------------------------------------------
-- Subprogram Declaration
-----------------------------------------------
procedure HAZARDS_PATTERNS_TEST (signal OUTPUT:  MVL9; NAME: STRING;
HAZARD_SCHEDULE1:  inout HAZARD_VAR; signal PATTERNS: in MVL9_VECTOR;
signal PATT1,PATT2: in MVL9_VECTOR;
signal TIME_DELAY: in PROP_DELAY_SPEC;
INFO: STATIC_INFO_VECTOR := ((DC_PATTERN(WORDLENGTH),
DC_PATTERN(WORDLENGTH), 0 ns, 0 ns),
(DC_PATTERN(WORDLENGTH), DC_PATTERN(WORDLENGTH), 0 ns, 0 ns)));

end PRE_HAZARD_TEST;


package body PRE_HAZARD_TEST is

-------------------------------------------
-- Subprogram body
-------------------------------------------
-- Check for static 0 hazard, static 1 hazard, dynamic 0 hazard, dynamic 1 hazard,
-- function 0 hazard, and function 1 hazard.

procedure HAZARDS_PATTERNS_TEST (signal OUTPUT:  MVL9; NAME: STRING;
HAZARD_SCHEDULE1:  inout HAZARD_VAR; signal PATTERNS: in MVL9_VECTOR;
signal PATT1,PATT2: in MVL9_VECTOR;
signal TIME_DELAY: in PROP_DELAY_SPEC;
INFO: STATIC_INFO_VECTOR := ((DC_PATTERN(WORDLENGTH),
DC_PATTERN(WORDLENGTH), 0 ns, 0 ns),
(DC_PATTERN(WORDLENGTH), DC_PATTERN(WORDLENGTH), 0 ns, 0 ns))) is


alias INFO_VAR: STATIC_INFO_VECTOR(1 to INFO'length) is INFO;
variable STATIC_FUN_CONFLICT : BOOLEAN := FALSE;

constant OUT_NAME: STRING(1 to NAME'length) := NAME;
```

```
---------------------------------------
-- Text I/O Variables Declarations
---------------------------------------
variable OUTLINE: LINE;

-------------------------------------------------------------------------
-- Text I/O Constant Declarations (miscellaneous framework declarations and
-- hazard assertion messages declaration)
-------------------------------------------------------------------------
constant OUTSTRING1: STRING(1 to 47) :=
"Static 0 hazard is detected at" &
NATURAL_TO_STRING(TIME_TO_NATURAL(HAZARD_SCHEDULE1.STAT_0_CHECK_TI
ME)) & " ns on ";

constant OUTSTRING2: STRING(1 to 47) :=
"Static 1 hazard is detected at" &
NATURAL_TO_STRING(TIME_TO_NATURAL(HAZARD_SCHEDULE1.STAT_1_CHECK_TI
ME)) & " ns on ";

constant OUTSTRING3: STRING(1 to 48) :=
"Dynamic 0 hazard is detected at" &   NATURAL_TO_STRING(TIME_TO_NATURAL(now)) &
" ns on ";

constant OUTSTRING4: STRING(1 to 48) :=
"Dynamic 1 hazard is detected at" &  NATURAL_TO_STRING(TIME_TO_NATURAL(now)) &
" ns on ";

constant OUTSTRING5: STRING(1 to 49) :=
"Function 0 hazard is detected at" &
NATURAL_TO_STRING(TIME_TO_NATURAL(HAZARD_SCHEDULE1.FUN_0_CHECK_TI
ME)) & " ns on ";

constant OUTSTRING6: STRING(1 to 49) :=
"Function 1 hazard is detected at" &
NATURAL_TO_STRING(TIME_TO_NATURAL(HAZARD_SCHEDULE1.FUN_1_CHECK_TI
ME)) & " ns on ";

constant OUTSTRING10: STRING(1 to 25) :=
      HT & "Test Circuit Statistics:";


constant OUTSTRING11: STRING(1 to 27) :=
      HT & "Number of Primary Inputs: ";

constant OUTSTRING12: STRING(1 to 62) :=
      "Minimum Propagation Delay of the test circuit with respect to ";

constant OUTSTRING13: STRING(1 to 62) :=
      "Maximum Propagation Delay of the test circuit with respect to ";
```

```
constant OUTSTRING14: STRING(1 to 74) :=
"************************ VHDS Report Summary ************************";

constant SPACE: CHARACTER := ' ';
constant COLON: CHARACTER := ':';
constant OUTSPACE: STRING(1 to 40):=
"========================================";
begin

-----------------------------------------------------------------
-- Reset the output event counting for all the input patterns.
-----------------------------------------------------------------
if (PATTERNS'event and PATTERNS = PATT1) then
HAZARD_SCHEDULE1.OUTPUT_COUNT1 := 0;
HAZARD_SCHEDULE1.OUTPUT_COUNT2 := 0;
end if;



-------------------------------------------------------------------------------
-- Search for the present mode of simulation (single-input hazard analysis or
-- two-input hazard analysis).
-------------------------------------------------------------------------------
if (PATTERNS'event and PATTERNS = PATT2 and ONE_BIT_DIFF(PATT1, PATT2) = TRUE)
then
HAZARD_SCHEDULE1.STAT_CONTROL := TRUE;
end if;

if (PATTERNS'event and PATTERNS = PATT2 and TWO_BIT_DIFF(PATT1, PATT2) = TRUE)
then
HAZARD_SCHEDULE1.FUN_CONTROL := TRUE;
end if;



-------------------------------------------------------------------------------
-- Record the number of output events from the original input pattern.
-------------------------------------------------------------------------------
if (PATTERNS'event and PATTERNS /= RESET_PATTERN(WORDLENGTH) and PATTERNS
   = PATT1) then
HAZARD_SCHEDULE1.INPUT1_EVENT_TIME := now;
end if;

if (now >= HAZARD_SCHEDULE1.INPUT1_EVENT_TIME +
TIME_DELAY.MIN_PROP_DELAY and
now <= HAZARD_SCHEDULE1.INPUT1_EVENT_TIME +
TIME_DELAY.MAX_PROP_DELAY) then
if (OUTPUT'event and OUTPUT /= 'X') then
HAZARD_SCHEDULE1.OUTPUT_COUNT1 :=  HAZARD_SCHEDULE1.OUTPUT_COUNT1
+ 1;
```

```
end if;
end if;


-------------------------------------------------------------------------------
-- Record the number of output events from the (second) neighbor input pattern.
-------------------------------------------------------------------------------
if (PATTERNS'event and PATTERNS /= RESET_PATTERN(WORDLENGTH) and PATTERNS
   = PATT2) then
HAZARD_SCHEDULE1.INPUT2_EVENT_TIME := now;
end if;

if (now >= HAZARD_SCHEDULE1.INPUT2_EVENT_TIME +
TIME_DELAY.MIN_PROP_DELAY and
now <= HAZARD_SCHEDULE1.INPUT2_EVENT_TIME +
TIME_DELAY.MAX_PROP_DELAY) then
if (OUTPUT'event and OUTPUT /= 'X') then
HAZARD_SCHEDULE1.OUTPUT_COUNT2 :=  HAZARD_SCHEDULE1.OUTPUT_COUNT2
+ 1;
end if;
end if;



--=====================================================
-------------------------------------------------------------------------------
-- Trace the output event and record all the output event in a two-tuple or
-- three-tuple hazard analysis.
-------------------------------------------------------------------------------
if (OUTPUT'event and OUTPUT'last_value = 'X' and OUTPUT /= 'X') then
HAZARD_SCHEDULE1.OUTPUT_PATTERNS(0) := OUTPUT;
end if;


if (OUTPUT'event and OUTPUT'last_value /= 'X' and OUTPUT /= 'X'
   and HAZARD_SCHEDULE1.LOOP_VAR < 3) then
   HAZARD_SCHEDULE1.LOOP_VAR := HAZARD_SCHEDULE1.LOOP_VAR + 1;
   HAZARD_SCHEDULE1.OUTPUT_PATTERNS(HAZARD_SCHEDULE1.LOOP_VAR) :=
   OUTPUT;
end if;


-----------------------------------------------------
-- Specifically used for static hazards detection
-----------------------------------------------------
if (HAZARD_SCHEDULE1.LOOP_VAR = 2 and OUTPUT'event and OUTPUT /= 'X' and
OUTPUT'last_value /= 'X'  and HAZARD_SCHEDULE1.STAT_CONTROL = TRUE) then
HAZARD_SCHEDULE1.OUTPUT_PATTERNS(3) := 'Z';
end if;
```

```
-------------------------------------------------------
-- Specifically used for function hazard detection
-------------------------------------------------------
if (HAZARD_SCHEDULE1.LOOP_VAR = 2 and OUTPUT'event and OUTPUT /= 'X' and
OUTPUT'last_value /= 'X'  and HAZARD_SCHEDULE1.FUN_CONTROL = TRUE) then
HAZARD_SCHEDULE1.OUTPUT_PATTERNS(3) := '-';
end if;



-------------------------------------------------
-- Reset the output event record schedule
-------------------------------------------------
if (OUTPUT'event and OUTPUT = 'X' and OUTPUT'last_value /= 'X') then
HAZARD_SCHEDULE1.LOOP_VAR := 0;
end if;
if (PATTERNS = PATT1 and PATTERNS'event) then
HAZARD_SCHEDULE1.OUTPUT_PATTERNS := RESET_PATTERN(4);
end if;
--=========================================

-------------------------------------------------------
-- Detection of Possible Static Hazard(s).
-------------------------------------------------------
-- Check for static 0 hazard
if (OUTPUT = '0' and OUTPUT'last_value = '1' and OUTPUT'event and
   ( HAZARD_SCHEDULE1.OUTPUT_COUNT1 = 1 and
     HAZARD_SCHEDULE1.OUTPUT_COUNT2 = 2) and
     HAZARD_SCHEDULE1.OUTPUT_LAST_TWO_VALUE = '0' and
     HAZARD_SCHEDULE1.STAT_CONTROL = TRUE) then


HAZARD_SCHEDULE1.STAT_0_CHECK := TRUE;
HAZARD_SCHEDULE1.STAT_0_CHECK_TIME := now;
end if;

-- Check for static 1 hazard
if (OUTPUT = '1' and OUTPUT'last_value = '0' and OUTPUT'event and
   ( HAZARD_SCHEDULE1.OUTPUT_COUNT1 = 1 and
     HAZARD_SCHEDULE1.OUTPUT_COUNT2 = 2) and
     HAZARD_SCHEDULE1.OUTPUT_LAST_TWO_VALUE = '1' and
     HAZARD_SCHEDULE1.STAT_CONTROL = TRUE) then

HAZARD_SCHEDULE1.STAT_1_CHECK := TRUE;
HAZARD_SCHEDULE1.STAT_1_CHECK_TIME := now;
end if;
```

```
-----------------------------------------------
-- Confirmation of the Possible Static Hazard
-----------------------------------------------
-- Check for static 0 hazard
if (OUTPUT = 'X' and OUTPUT'last_value /= 'X' and OUTPUT'event and
   HAZARD_SCHEDULE1.OUTPUT_PATTERNS = "010Z") then
if (HAZARD_SCHEDULE1.STAT_0_CHECK = TRUE) then

assert FALSE
report "Static 0 hazard (1-variable logic 0 hazard) is detected on " & NAME & " at" &
NATURAL_TO_STRING(TIME_TO_NATURAL(HAZARD_SCHEDULE1.STAT_0_CHECK_TI
ME)) & " ns"
severity WARNING;

-- File I/O
        HAZARD_SCHEDULE1.STAT_MESSAGE := TRUE;
        WRITE(OUTLINE, OUTSPACE);
        WRITELINE(OUTFILE, OUTLINE);

        WRITE(OUTLINE, OUTSTRING1);
        WRITE(OUTLINE, OUT_NAME);
        WRITELINE(OUTFILE, OUTLINE);

end if;
end if;

-- Check for static 1 hazard

if (OUTPUT = 'X' and OUTPUT'last_value /= 'X' and OUTPUT'event and
   HAZARD_SCHEDULE1.OUTPUT_PATTERNS = "101Z") then
if (HAZARD_SCHEDULE1.STAT_1_CHECK = TRUE) then


assert FALSE
report "Static 1 hazard (1-variable logic 1 hazard) is detected on " & NAME & " at"  &
NATURAL_TO_STRING(TIME_TO_NATURAL(HAZARD_SCHEDULE1.STAT_1_CHECK_TI
ME)) & " ns"
severity WARNING;

-- File I/O
        HAZARD_SCHEDULE1.STAT_MESSAGE := TRUE;
        WRITE(OUTLINE, OUTSPACE);
        WRITELINE(OUTFILE, OUTLINE);

        WRITE(OUTLINE, OUTSTRING2);
        WRITE(OUTLINE, OUT_NAME);
        WRITELINE(OUTFILE, OUTLINE);

end if;
end if;
```

```
-------------------------------------------------------
-- Check for Possible Dynamic Hazard(s).
-------------------------------------------------------
-- Check for dynamic 0 hazard
if (OUTPUT = '0' and OUTPUT'last_value = '1' and OUTPUT'event and
   (HAZARD_SCHEDULE1.OUTPUT_COUNT1 = 1 and
    HAZARD_SCHEDULE1.OUTPUT_COUNT2 = 3) and
    HAZARD_SCHEDULE1.OUTPUT_LAST_TWO_VALUE = '0' and
    HAZARD_SCHEDULE1.OUTPUT_PATTERNS = "1010" and
    HAZARD_SCHEDULE1.STAT_CONTROL = TRUE) then

                assert FALSE
                report "Dynamic 0 hazard is detected on " & NAME & " at"
                 & NATURAL_TO_STRING(TIME_TO_NATURAL(now)) & " ns"
                severity WARNING;

-- File I/O
        HAZARD_SCHEDULE1.DYN_MESSAGE := TRUE;
        WRITE(OUTLINE, OUTSPACE);
        WRITELINE(OUTFILE, OUTLINE);

        WRITE(OUTLINE, OUTSTRING3);
        WRITE(OUTLINE, OUT_NAME);
        WRITELINE(OUTFILE, OUTLINE);

end if;


-- Check for dynamic 1 hazard
if (OUTPUT = '1' and OUTPUT'last_value = '0' and OUTPUT'event and
   (HAZARD_SCHEDULE1.OUTPUT_COUNT1 = 1 and
    HAZARD_SCHEDULE1.OUTPUT_COUNT2 = 3) and
    HAZARD_SCHEDULE1.OUTPUT_LAST_TWO_VALUE = '1' and
    HAZARD_SCHEDULE1.OUTPUT_PATTERNS = "0101" and
    HAZARD_SCHEDULE1.STAT_CONTROL = TRUE) then

                assert FALSE
                report "Dynamic 1 hazard is detected on " & NAME & " at"
                 & NATURAL_TO_STRING(TIME_TO_NATURAL(now)) & " ns"
                severity WARNING;

-- File I/O
        HAZARD_SCHEDULE1.DYN_MESSAGE := TRUE;
        WRITE(OUTLINE, OUTSPACE);
        WRITELINE(OUTFILE, OUTLINE);

        WRITE(OUTLINE, OUTSTRING4);
        WRITE(OUTLINE, OUT_NAME);
```

```
        WRITELINE(OUTFILE, OUTLINE);

end if;


-------------------------------------------------------
-- Detect the time of the first event when a
-- function hazard may be occur.
-------------------------------------------------------
if (OUTPUT'event and (OUTPUT = '0' or OUTPUT = '1') and
   HAZARD_SCHEDULE1.OUTPUT_LAST_TWO_VALUE = 'X'
   and HAZARD_SCHEDULE1.FUN_CONTROL = TRUE) then
   HAZARD_SCHEDULE1.FUN_INFO_TIME := now -
   HAZARD_SCHEDULE1.INPUT2_EVENT_TIME;
end if;


-------------------------------------------------------------
-- Check for Possible Function Hazard(s).
-------------------------------------------------------------
-- Check for function 0 hazard
if ((OUTPUT = '0' and OUTPUT'last_value = '1' and OUTPUT'event) and
     HAZARD_SCHEDULE1.FUN_CONTROL = TRUE and
   ((HAZARD_SCHEDULE1.OUTPUT_COUNT1 = 1 and
     HAZARD_SCHEDULE1.OUTPUT_COUNT2 = 2) and
     HAZARD_SCHEDULE1.OUTPUT_LAST_TWO_VALUE = '0')) then


HAZARD_SCHEDULE1.FUN_0_CHECK := TRUE;
HAZARD_SCHEDULE1.FUN_0_CHECK_TIME := now;
end if;


-- Check for function 1 hazard
if ((OUTPUT = '1' and OUTPUT'last_value = '0' and OUTPUT'event) and
     HAZARD_SCHEDULE1.FUN_CONTROL = TRUE and
   ((HAZARD_SCHEDULE1.OUTPUT_COUNT1 = 1 and
     HAZARD_SCHEDULE1.OUTPUT_COUNT2 = 2) and
     HAZARD_SCHEDULE1.OUTPUT_LAST_TWO_VALUE = '1')) then



HAZARD_SCHEDULE1.FUN_1_CHECK := TRUE;
HAZARD_SCHEDULE1.FUN_1_CHECK_TIME := now;
end if;
```

```
-------------------------------------------------------------
-- Confirmation of Possible Function Hazards.
-------------------------------------------------------------
-- Confirmation of function 0 hazard.
if (OUTPUT = 'X' and OUTPUT'last_value /= 'X' and OUTPUT'event and
   HAZARD_SCHEDULE1.OUTPUT_PATTERNS = "010-") then
if (HAZARD_SCHEDULE1.FUN_0_CHECK = TRUE)  then

for  I in INFO_VAR'range loop
if (((INFO_VAR(I).STATE_PATT1 = PATT1 and
     ONE_BIT_DIFF(INFO_VAR(I).STATE_PATT2, PATT2) = TRUE) or
    (INFO_VAR(I).STATE_PATT2 = PATT2 and
     ONE_BIT_DIFF(INFO_VAR(I).STATE_PATT1, PATT1) = TRUE)) and
     INFO_VAR(I).STATIC_EVENT_TIME = HAZARD_SCHEDULE1.FUN_INFO_TIME and
     INFO_VAR(I).PULSE_DURATION  = HAZARD_SCHEDULE1.FUN_0_CHECK_TIME -
     HAZARD_SCHEDULE1.INPUT2_EVENT_TIME -
     INFO_VAR(I).STATIC_EVENT_TIME) then
     STATIC_FUN_CONFLICT := TRUE;
end if;
exit when STATIC_FUN_CONFLICT = TRUE;
end loop;

                if (STATIC_FUN_CONFLICT = FALSE) then

assert FALSE
report "2-variable function 0 hazard is detected on " & NAME & " at"  &
NATURAL_TO_STRING(TIME_TO_NATURAL(HAZARD_SCHEDULE1.FUN_0_CHECK_TI
ME)) & " ns"
severity WARNING;

-- File I/O
        HAZARD_SCHEDULE1.FUN_MESSAGE := TRUE;
        WRITE(OUTLINE, OUTSPACE);
        WRITELINE(OUTFILE, OUTLINE);

        WRITE(OUTLINE, OUTSTRING5);
        WRITE(OUTLINE, OUT_NAME);
        WRITELINE(OUTFILE, OUTLINE);

                end if;

end if;


HAZARD_SCHEDULE1.FUN_0_CHECK := FALSE;
HAZARD_SCHEDULE1.FUN_0_CHECK_TIME := 0 ns;
end if;
```

```
-- Confirmation of function 1 hazard.
if (OUTPUT = 'X' and OUTPUT'last_value /= 'X' and OUTPUT'event and
   HAZARD_SCHEDULE1.OUTPUT_PATTERNS = "101-") then
if (HAZARD_SCHEDULE1.FUN_1_CHECK = TRUE)  then

for  I in INFO_VAR'range loop
if (((INFO_VAR(I).STATE_PATT1 = PATT1 and
     ONE_BIT_DIFF(INFO_VAR(I).STATE_PATT2, PATT2) = TRUE) or
    (INFO_VAR(I).STATE_PATT2 = PATT2 and
     ONE_BIT_DIFF(INFO_VAR(I).STATE_PATT1, PATT1) = TRUE)) and
     INFO_VAR(I).STATIC_EVENT_TIME = HAZARD_SCHEDULE1.FUN_INFO_TIME and
     INFO_VAR(I).PULSE_DURATION  = HAZARD_SCHEDULE1.FUN_1_CHECK_TIME -
     HAZARD_SCHEDULE1.INPUT2_EVENT_TIME -
     INFO_VAR(I).STATIC_EVENT_TIME) then

  STATIC_FUN_CONFLICT := TRUE;
end if;
exit when STATIC_FUN_CONFLICT = TRUE;
end loop;

                if (STATIC_FUN_CONFLICT = FALSE) then

assert FALSE
report "2-variable function 1 hazard is detected on " & NAME & " at"  &
NATURAL_TO_STRING(TIME_TO_NATURAL(HAZARD_SCHEDULE1.FUN_1_CHECK_TI
ME)) & " ns"
severity WARNING;

-- File I/O
        HAZARD_SCHEDULE1.FUN_MESSAGE := TRUE;
        WRITE(OUTLINE, OUTSPACE);
        WRITELINE(OUTFILE, OUTLINE);

        WRITE(OUTLINE, OUTSTRING6);
        WRITE(OUTLINE, OUT_NAME);
        WRITELINE(OUTFILE, OUTLINE);

                end if;

end if;

HAZARD_SCHEDULE1.FUN_1_CHECK := FALSE;
HAZARD_SCHEDULE1.FUN_1_CHECK_TIME := 0 ns;
end if;
```

```
-------------------------------------------------------------------
-- Memorize the current simulation time and logic value as the time
-- and values of the last event
-------------------------------------------------------------------
if (OUTPUT'event) then
HAZARD_SCHEDULE1.OUTPUT_LAST_TWO_VALUE := OUTPUT'last_value;
end if;


-------------------------------------------------------------------------------------
-- File I/O
-------------------------------------------------------------------------------------

if (OUTPUT'event and OUTPUT /= 'X') then
HAZARD_SCHEDULE1.OUT_TIME_RECORD(HAZARD_SCHEDULE1.ICOUNT) := now;
HAZARD_SCHEDULE1.OUTPUT_RECORD(HAZARD_SCHEDULE1.ICOUNT)  := OUTPUT;
HAZARD_SCHEDULE1.ICOUNT := HAZARD_SCHEDULE1.ICOUNT + 1;
end if;


-------------------------------------------------
-- Static hazard Summary
-------------------------------------------------
if (HAZARD_SCHEDULE1.STAT_MESSAGE = TRUE and
HAZARD_SCHEDULE1.DYN_MESSAGE = FALSE) then
WRITE(OUTLINE, HAZARD_SCHEDULE1.INPUT1_EVENT_TIME);
WRITE(OUTLINE, TO_BITVECTOR(PATT1), FIELD => 15);
WRITELINE(OUTFILE, OUTLINE);

WRITE(OUTLINE, HAZARD_SCHEDULE1.INPUT2_EVENT_TIME);
WRITE(OUTLINE, TO_BITVECTOR(PATT2), FIELD => 15);
WRITELINE(OUTFILE, OUTLINE);

end if;

-------------------------------------------------
-- Dynamic hazard Summary
-------------------------------------------------
if (HAZARD_SCHEDULE1.DYN_MESSAGE = TRUE and OUTPUT /= 'X' and
   OUTPUT'last_value /= 'X' and OUTPUT'event) then
WRITE(OUTLINE, HAZARD_SCHEDULE1.INPUT1_EVENT_TIME);
WRITE(OUTLINE, TO_BITVECTOR(PATT1), FIELD => 15);
WRITELINE(OUTFILE, OUTLINE);

WRITE(OUTLINE, HAZARD_SCHEDULE1.INPUT2_EVENT_TIME);
WRITE(OUTLINE, TO_BITVECTOR(PATT2), FIELD => 15);
WRITELINE(OUTFILE, OUTLINE);

end if;
```

```
-------------------------------------------------
-- function hazard Summary
-------------------------------------------------
if (HAZARD_SCHEDULE1.FUN_MESSAGE = TRUE) then
WRITE(OUTLINE, HAZARD_SCHEDULE1.INPUT1_EVENT_TIME);
WRITE(OUTLINE, TO_BITVECTOR(PATT1), FIELD => 15);
WRITELINE(OUTFILE, OUTLINE);

WRITE(OUTLINE, HAZARD_SCHEDULE1.INPUT2_EVENT_TIME);
WRITE(OUTLINE, TO_BITVECTOR(PATT2), FIELD => 15);
WRITELINE(OUTFILE, OUTLINE);

end if;

-------------------------------------------------
-- Reset all the file/IO variables as well as
-- print out a line break.
-------------------------------------------------
if (OUTPUT'event and OUTPUT = 'X' and (HAZARD_SCHEDULE1.STAT_MESSAGE = TRUE
or HAZARD_SCHEDULE1.DYN_MESSAGE = TRUE or
HAZARD_SCHEDULE1.FUN_MESSAGE = TRUE)) then

for I in 0 to HAZARD_SCHEDULE1.ICOUNT - 1 loop
WRITE(OUTLINE, HAZARD_SCHEDULE1.OUT_TIME_RECORD(I));
WRITE(OUTLINE, TO_BIT(HAZARD_SCHEDULE1.OUTPUT_RECORD(I)), FIELD => 15);
WRITE(OUTLINE, OUT_NAME, FIELD => 10);
WRITELINE(OUTFILE, OUTLINE);
end loop;

HAZARD_SCHEDULE1.STAT_MESSAGE := FALSE;
HAZARD_SCHEDULE1.DYN_MESSAGE := FALSE;
HAZARD_SCHEDULE1.FUN_MESSAGE := FALSE;
end if;

if (OUTPUT'event and OUTPUT = 'X') then
HAZARD_SCHEDULE1.ICOUNT := 0;
HAZARD_SCHEDULE1.OUT_TIME_RECORD := TIME_RESET(2 ** WORDLENGTH);
HAZARD_SCHEDULE1.OUTPUT_RECORD := RESET_PATTERN(2 ** WORDLENGTH);
end if;

-------------------------------------------------
-- Create a File I/O test circuit statistics
-------------------------------------------------
if ((PATTERNS'event and PATTERNS = PATT1) and
   ((now = TIME_DELAY.MAX_PROP_DELAY + 1 ns) or
    now = PRE_TEST_BEGIN_TIME(TIME_DELAY) +
    TIME_DELAY.MAX_PROP_DELAY)) then

WRITE(OUTLINE, OUTSTRING10);
```

```
WRITELINE(OUTFILE, OUTLINE);

WRITE(OUTLINE, OUTSTRING11);
WRITE(OUTLINE, WORDLENGTH);
WRITELINE(OUTFILE, OUTLINE);

WRITE(OUTLINE, OUTSTRING12);
WRITE(OUTLINE, OUT_NAME);
WRITE(OUTLINE, COLON);
WRITE(OUTLINE, SPACE);
WRITE(OUTLINE, TIME_DELAY.MIN_PROP_DELAY);
WRITELINE(OUTFILE, OUTLINE);

WRITE(OUTLINE, OUTSTRING13);
WRITE(OUTLINE, OUT_NAME);
WRITE(OUTLINE, COLON);
WRITE(OUTLINE, SPACE);
WRITE(OUTLINE, TIME_DELAY.MAX_PROP_DELAY);
WRITELINE(OUTFILE, OUTLINE);

WRITE(OUTLINE, SPACE);
WRITELINE(OUTFILE, OUTLINE);

WRITE(OUTLINE, OUTSTRING14);
WRITELINE(OUTFILE, OUTLINE);

WRITE(OUTLINE, SPACE);
WRITELINE(OUTFILE, OUTLINE);
end if;

-----------------------------------------------
-- Reset control variables
-----------------------------------------------
if (OUTPUT'event and OUTPUT = 'X') then
HAZARD_SCHEDULE1.FUN_CONTROL:= FALSE;
HAZARD_SCHEDULE1.STAT_CONTROL:= FALSE;
HAZARD_SCHEDULE1.STAT_0_CHECK := FALSE;
HAZARD_SCHEDULE1.STAT_0_CHECK_TIME := 0 ns;
HAZARD_SCHEDULE1.STAT_1_CHECK := FALSE;
HAZARD_SCHEDULE1.STAT_1_CHECK_TIME := 0 ns;
HAZARD_SCHEDULE1.FUN_0_CHECK := FALSE;
HAZARD_SCHEDULE1.FUN_0_CHECK_TIME := 0 ns;
HAZARD_SCHEDULE1.FUN_1_CHECK := FALSE;
HAZARD_SCHEDULE1.FUN_1_CHECK_TIME := 0 ns;
HAZARD_SCHEDULE1.FUN_INFO_TIME := 0 ns;
STATIC_FUN_CONFLICT := FALSE;
end if;

end HAZARDS_PATTERNS_TEST;
end PRE_HAZARD_TEST;
```

Vita

Ming-Cheung Chu was born in Hong Kong on November 22, 1967. After finishing high school in Hong Kong, he attended the University of South Alabama at Mobile on September 1986. After completing a B.S. degree in Electrical Engineering on June 1989, he decided to continue his education at Virginia Tech and pursue a M.S. degree in Electrical Engineering. Under the guidance of Dr. Armstrong, he found himself interested in VHDL and VLSI Design. He is currently a candidate for the M.S. degree in Electrical Engineering.

His research interests include VHDL Modeling, VLSI Design, and Fault Tolerant Computing.